

Sensor Fusion and Tracking Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2022b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Sensor Fusion and Tracking Toolbox™ User's Guide

© COPYRIGHT 2018–2022 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

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)
March 2021	Online only	Revised for Version 2.1 (Release 2021a)
September 2021	Online only	Revised for Version 2.2 (Release 2021b)
March 2022	Online only	Revised for Version 2.3 (Release 2022a)
September 2022	Online only	Revised for Version 2.4 (Release 2022b)

	Tracking Scenarios	
1		
	Tracking Simulation Overview	1-2
	Creating a Tracking Scenario	1-4
	Create Tracking Scenario with Two Platforms	1-6
	Model Platform Motion Using Trajectory Objects	1-9
	Introduction	1-9
	waypointTrajectory	1-9
	geoTrajectory	1-12
	kinematicTrajectory	1-14
	Summary	1-20

	Radar Detections	
2		
	Simulate Radar Detections	2-2
	Create Radar Sensor	2-2
	Detector Input	2-8
	Radar Sensor Coordinate Systems	2-10
	INS	2-12
	Detections	2-12
	Introduction to Statistical Radar Models for Object Tracking	2-15
	Sensor Overview	2-15
	Radar Detection Mode	2-15
	Mounting Radar on Platform	2-16
	Detection Ability and Quality	2-16
	Measurement and Detection Format	2-18

	Inertial Sensor and Sensor Fusion	
3		
	Choose Inertial Sensor Fusion Filters	3-2
	Fuse Inertial Sensor Data Using insEKF-Based Flexible Fusion Framework	3-7
	Object Properties	3-7

Object Functions	3-9
Example: Fuse Inertial Sensor Data Using insEKF	3-10

Multi-Object Tracking

4

Tracking and Tracking Filters	4-2
Multi-Object Tracking	4-2
Multi-Object Tracker Properties	4-3
Introduction to Estimation Filters	4-9
Background	4-9
Filter Design	4-10
Estimation Filters in Sensor Fusion and Tracking Toolbox	4-12
How to Choose a Tracking Filter	4-16
Introduction to Out-of-Sequence Measurement Handling	4-18
Introduction	4-18
Report an Error for OOSM	4-18
Neglect OOSM	4-19
Process OOSM Using Retrodiction	4-19
Motion Model, State, and Process Noise	4-22
Introduction	4-22
Constant Velocity Model	4-22
Constant Acceleration Model	4-23
Constant Turn Rate Model	4-24
Singer Model	4-25
Summary	4-26
Linear Kalman Filters	4-28
Motion Model	4-28
Measurement Models	4-29
Filter Loop	4-29
Built-In Motion Models in trackingKF	4-31
Example: Estimate 2-D Target States Using trackingKF	4-32
Extended Kalman Filters	4-36
State Update Model	4-36
Measurement Model	4-37
Extended Kalman Filter Loop	4-37
Predefined Extended Kalman Filter Functions	4-38
Example: Estimate 2-D Target States with Angle and Range Measurements Using trackingEKF	4-39
Introduction to Multiple Target Tracking	4-44
Background	4-44
Elements of an MTT System	4-44
Tracking Metrics	4-47
Non-Assignment-Based Trackers	4-48

Introduction to Assignment Methods in Tracking Systems	4-49
Background	4-49
2-D Assignment in Multiple Target Tracking	4-49
S-D Assignment in Multiple Target Tracking	4-53
Introduction to Track-To-Track Fusion	4-56
Track-To-Track Fusion Versus Central-Level Tracking	4-56
Benefits and Challenges of Track-To-Track Fusion	4-56
Track Fuser and Tracking Architecture	4-57
Multiple Extended Object Tracking	4-59
Configure Time Scope MATLAB Object	4-61
Signal Display	4-61
Multiple Signal Names and Colors	4-62
Configure Scope Settings	4-62
Use timescope Measurements and Triggers	4-63
Share or Save the Time Scope	4-79
Scale Axes	4-80

Code Generation

5

Generate Code with Strict Single-Precision and Non-Dynamic Memory Allocation	5-2
Introduction to Strict Single-Precision and Non-Dynamic Memory Allocation	5-2
Supported Trackers and Tracking Filters	5-3
Supported Assignment and Partition Functions	5-4
Supported Motion Model Functions	5-5
Supported Filter Initialization Functions	5-5

Featured Examples

6

Air Traffic Control	6-2
IMU and GPS Fusion for Inertial Navigation	6-17
Estimate Position and Orientation of a Ground Vehicle	6-25
Rotations, Orientation, and Quaternions	6-33
Lowpass Filter Orientation Using Quaternion SLERP	6-48
Introduction to Simulating IMU Measurements	6-52
Introduction to Using the Global Nearest Neighbor Tracker	6-64

Introduction to Track Logic	6-78
Introduction to Tracking Scenario and Simulating Sensor Detections .	6-88
Inertial Sensor Noise Analysis Using Allan Variance	6-97
Estimate Orientation Through Inertial Sensor Fusion	6-108
Estimate Orientation and Height Using IMU, Magnetometer, and Altimeter	6-118
Scanning Radar Mode Configuration	6-122
Extended Object Tracking of Highway Vehicles with Radar and Camera	6-148
Tracking Closely Spaced Targets Under Ambiguity	6-168
Visual-Inertial Odometry Using Synthetic Data	6-184
Tracking Maneuvering Targets	6-193
Multiplatform Radar Detection Fusion	6-203
Multiplatform Radar Detection Generation	6-215
Tracking Using Distributed Synchronous Passive Sensors	6-227
Passive Ranging Using a Single Maneuvering Sensor	6-241
Benchmark Trajectories for Multi-Object Tracking	6-259
Tracking with Range-Only Measurements	6-269
Adaptive Tracking of Maneuvering Targets with Managed Radar	6-279
How to Generate C Code for a Tracker	6-296
How to Efficiently Track Large Numbers of Objects	6-303
Tracking a Flock of Birds	6-317
Tracking Using Bistatic Range Detections	6-324
Pose Estimation From Asynchronous Sensors	6-338
Magnetometer Calibration	6-343
Track Vehicles Using Lidar: From Point Cloud to Track List	6-352
Extended Object Tracking With Radar For Marine Surveillance	6-370
Track Vehicles Using Lidar Data in Simulink	6-383

Track Closely Spaced Targets Under Ambiguity in Simulink	6-391
Design and Simulate Tracking Scenario with Tracking Scenario Designer 	6-403
Estimate Orientation with a Complementary Filter and IMU Data	6-416
Logged Sensor Data Alignment for Orientation Estimation	6-424
Remove Bias from Angular Velocity Measurement	6-431
Convert Detections to objectDetection Format	6-435
Estimating Orientation Using Inertial Sensor Fusion and MPU-9250 .	6-445
Track Simulated Vehicles Using GNN and JPDA Trackers in Simulink	6-456
Track-to-Track Fusion for Automotive Safety Applications in Simulink	6-463
Track Point Targets in Dense Clutter Using GM-PHD Tracker	6-466
Introduction to Tracking Metrics	6-478
Track-Level Fusion of Radar and Lidar Data	6-496
Tuning a Multi-Object Tracker	6-516
Generate Off-Centered IMU Readings	6-540
Detect Multipath GPS Reading Errors Using Residual Filtering in Inertial Sensor Fusion	6-545
IMU Sensor Fusion with Simulink	6-551
Track Space Debris Using a Keplerian Motion Model	6-553
Simulate and Track En-Route Aircraft in Earth-Centered Scenarios . .	6-564
Simulate, Detect, and Track Anomalies in a Landing Approach	6-578
Generate Code for a Track Fuser with Heterogeneous Source Tracks .	6-588
Extended Object Tracking with Lidar for Airport Ground Surveillance	6-598
Track Multiple Lane Boundaries with a Global Nearest Neighbor Tracker 	6-611
Grid-Based Tracking in Urban Environments Using Multiple Lidars . .	6-618
Automatic Tuning of the insfilterAsync Filter	6-630
Detect, Classify, and Track Vehicles Using Lidar	6-638
Use theaterPlot to Visualize Tracking Scenario	6-653

Custom Tuning of Fusion Filters	6-658
Wireless Data Streaming and Sensor Fusion Using BNO055	6-668
Binaural Audio Rendering Using Head Tracking	6-674
Motion Planning in Urban Environments Using Dynamic Occupancy Grid Map	6-679
Track-to-Track Fusion for Automotive Safety Applications	6-693
Detect and Track LEO Satellite Constellation with Ground Radars ...	6-708
Highway Vehicle Tracking with Multipath Radar Reflections	6-719
Track-Level Fusion of Radar and Lidar Data in Simulink	6-730
Track Point Targets in Dense Clutter Using GM-PHD Tracker in Simulink	6-740
Define and Test Tracking Architectures for System-of-Systems	6-745
Estimate Phone Orientation Using Sensor Fusion	6-755
Handle Out-of-Sequence Measurements in Multisensor Tracking Systems	6-761
Handle Out-of-Sequence Measurements with Filter Retrodiction	6-771
Extended Object Tracking of Highway Vehicles with Radar and Camera in Simulink	6-778
Grid-based Tracking in Urban Environments Using Multiple Lidars in Simulink	6-791
Simulate INS Block	6-795
Reconstruct Ground Truth Trajectory from Sampled Data Using Filtering, Smoothing, and Interpolation	6-797
Asynchronous Sensor Fusion and Tracking with Retrodiction	6-810
Object Tracking and Motion Planning Using Frenet Reference Path ..	6-813
Extended Target Tracking with Multipath Radar Reflections in Simulink	6-824
Lidar and Radar Fusion in Urban Air Mobility Scenario	6-834
Smooth Trajectory Estimation of trackingIMM Filter	6-853
Extended Object Tracking with Radar for Marine Surveillance in Simulink	6-867

How to Simulate Out-of-Sequence Measurements	6-874
Processor-in-the-Loop Verification of JPDA Tracker for Automotive Applications	6-886
Track Objects with Wrapping Azimuth Angles and Ambiguous Range and Range Rate Measurements	6-896
Export trackingArchitecture to Simulink	6-909
Define and Test Tracking Architectures for System-of-Systems in Simulink	6-914
Simulate and Track Targets with Terrain Occlusions	6-929
Tuning Kalman Filter to Improve State Estimation	6-939
Object Tracking Using Time Difference of Arrival (TDOA)	6-948
Design Fusion Filter for Custom Sensors	6-969
Estimate Orientation Using GPS-Derived Yaw Measurements	6-982
Ground Vehicle Pose Estimation for Tightly Coupled IMU and GNSS ..	6-988
Estimate Orientation Using AHRS Filter and IMU Data in Simulink ..	6-993
Angle and Position Measurement Fusion for Marine Surveillance ...	6-1002
Introduction to Class Fusion and Classification-Aided Tracking	6-1014
Air Traffic Control in Simulink	6-1022
Gesture Recognition Using Inertial Measurement Units	6-1037
Automatically Tune Tracking Filter for Multi-Object Tracker	6-1047
Asynchronous Angle-only Tracking with GM-PHD Tracker	6-1067
Analyze Track and Detection Association Using Analysis Info	6-1076

Tracking Scenarios

Tracking Simulation Overview

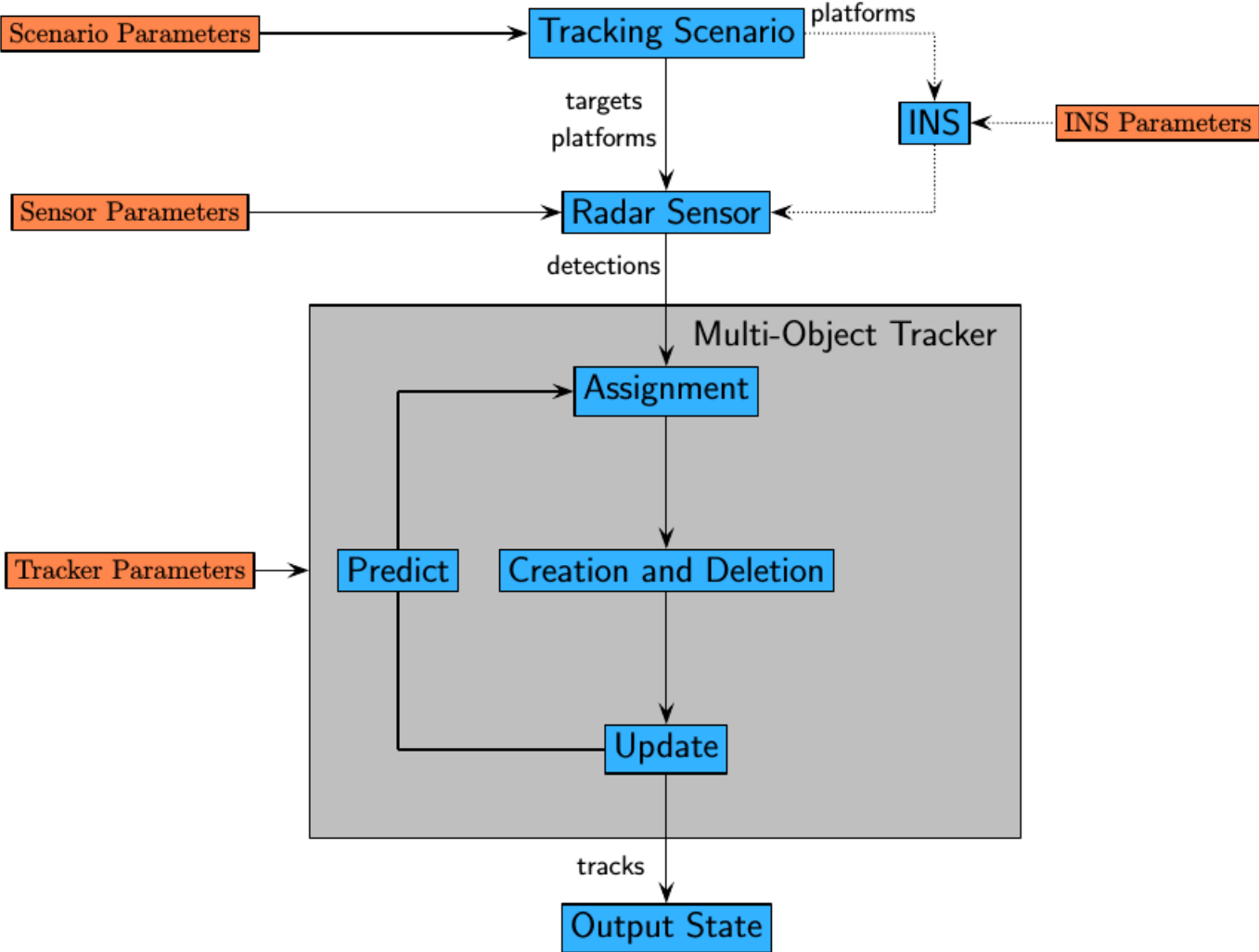
You can build a complete tracking simulation using the functions and objects supplied in this toolbox. The workflow for sensor fusion and tracking simulation consists of three (and optionally four) components. These components are

- 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 Convention” 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 `fusionRadarSensor` 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

<code>PlatformID</code>	Scenario-defined platform ID.
<code>ClassID</code>	User-specified platform classification ID.
<code>Dimensions</code>	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 <code>Dimensions</code> has all fields equal to zero, which corresponds to a point model.
<code>Trajectory</code>	Platform motion, specified by <code>kinematicTrajectory</code> or <code>waypointTrajectory</code> .
<code>Signatures</code>	Platform signatures, specified as a cell array of <code>irSignature</code> , <code>rscSignature</code> , and <code>tsSignature</code> objects. A signature represents the reflection or emission pattern of a platform.
<code>PoseEstimator</code>	A pose estimator, specified as a pose-estimator object such as <code>insSensor</code> (default).
<code>Emitter</code>	Emitters mounted on platform, specified as a cell array of emitter objects, such as <code>radarEmitter</code> or <code>sonarEmitter</code> .
<code>Sensors</code>	Sensors mounted on platform, specified as a cell array of sensor objects such as <code>irSensor</code> or <code>sonarSensor</code> .

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.

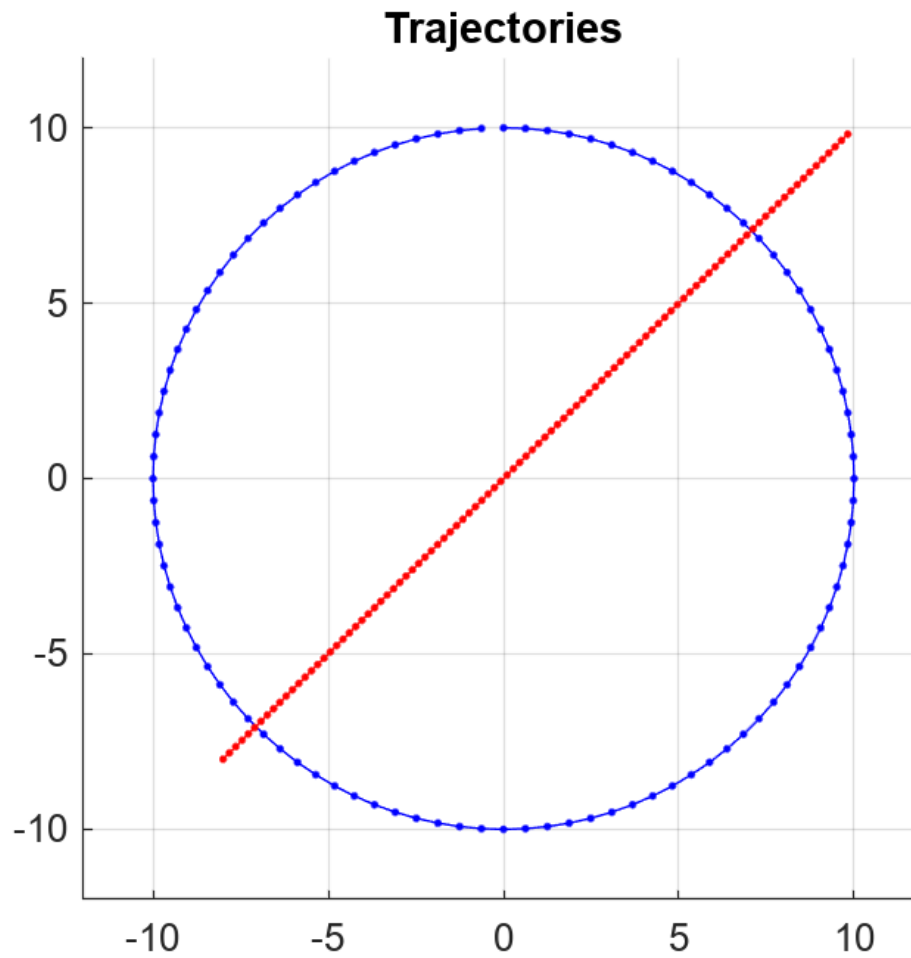
```
disp(sc.Platforms)

    {1x1 fusion.scenario.Platform}    {1x1 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.

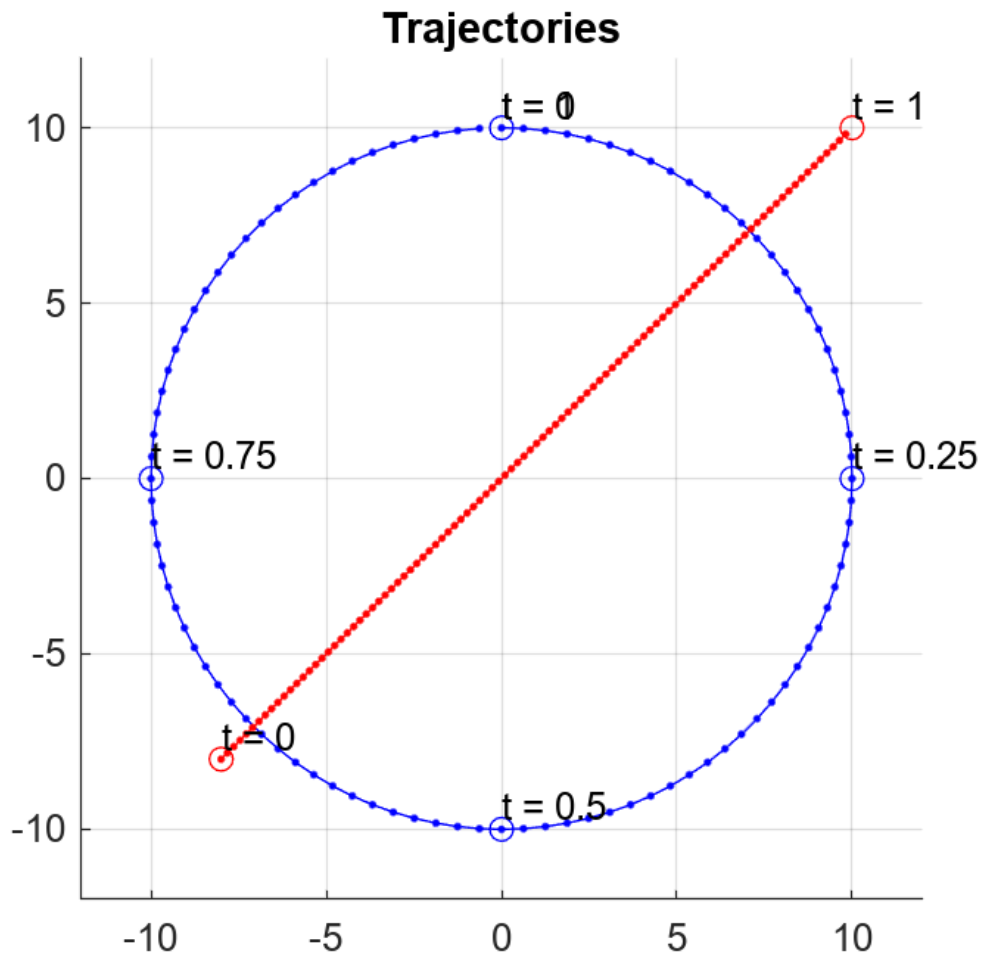
```
figure
grid
axis equal
axis([-12 12 -12 12])
line1 = animatedline('DisplayName','Trajectory 1','Color','b','Marker','.');
line2 = animatedline('DisplayName','Trajectory 2','Color','r','Marker','.');
title('Trajectories')
p1 = pose(platfm1);
p2 = pose(platfm2);
addpoints(line1,p1.Position(1),p1.Position(2));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

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

Plot the waypoints for both platforms.

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



Model Platform Motion Using Trajectory Objects

This topic introduces how to use three different trajectory objects to model platform trajectories, and how to choose between them.

Introduction

Sensor Fusion and Tracking Toolbox provides three System objects that you can use to model trajectories of platforms including ground vehicles, ships, aircraft, and spacecraft. You can choose between these trajectory objects based on the available trajectory information and the distance span of the trajectory.

- **waypointTrajectory** — Defines a trajectory using a few waypoints in Cartesian coordinates that the trajectory must pass through. The trajectory assumes the reference frame is a fixed North-East-Down (NED) or East-North-Up (ENU) frame. Since the trajectory interpolation assumes that the gravitational acceleration expressed in the trajectory reference frame is constant, **waypointTrajectory** is typically used for a trajectory defined within an area that spans only tens or hundreds of kilometers.
- **geoTrajectory** — Defines a trajectory using a few waypoints in geodetic coordinates (latitude, longitude, and altitude) that the trajectory must pass through. Since the waypoints are expressed in geodetic coordinates, **geoTrajectory** is typically used for a trajectory from hundreds to thousands of kilometers of distance span.
- **kinematicTrajectory** — Defines a trajectory using kinematic properties, such as acceleration and angular acceleration, expressed in the platform body frame. You can use **kinematicTrajectory** to generate a trajectory of any distance span as long as the kinematic information of the trajectory is available. The object assumes a Cartesian coordinate reference frame.

The two waypoint-based trajectory objects (**waypointTrajectory** and **geoTrajectory**) can automatically calculate the linear velocity information of the platform, but you can also explicitly specify the linear velocity using the **Velocity** property or a combination of the **Course**, **GroundSpeed**, and **ClimbRate** properties.

The trajectory of a platform is composed of rotational motion and translational motion. By default, the two waypoint-based trajectory objects (**waypointTrajectory** and **geoTrajectory**) automatically generate the orientation of the platform at each waypoint by aligning the yaw angle with the path of the trajectory, but you can explicitly specify the orientation using the **Orientation** property. Alternately, you can use the **AutoPitch** and **AutoBank** properties to enable automatic pitch and roll angles, respectively. For **kinematicTrajectory**, you need to use the **Orientation** property and the angular velocity input to specify the rotational motion of the trajectory.

waypointTrajectory

The **waypointTrajectory** System object defines a trajectory that smoothly passes through waypoints expressed in Cartesian coordinates. Generally, you can use **waypointTrajectory** to model vehicles travelling within hundreds of kilometers. These vehicles include automobiles, surface marine craft, and commercial aircraft (helicopters, planes, and quadcopters). You can choose the reference frame as a fixed ENU or NED frame using the **ReferenceFrame** property. For more details on how the object generates the trajectory, see the “Algorithms” section of **waypointTrajectory**.

waypointTrajectory Example for Aircraft Landing

Define the trajectory of a landing aircraft using a `waypointTrajectory` object.

```
waypoints = [-421 -384 2000;
            47 -294 1600;
            1368 174 1300;
            995 1037 900;
            -285 293 600;
            -1274 84 350;
            -2328 101 150;
            -3209 83 0];
timeOfArrival = [0; 16.71; 76.00; 121.8; 204.3; 280.31; 404.33; 624.6];
aircraftTraj = waypointTrajectory(waypoints,timeOfArrival);
```

Create a `theaterPlot` object to visualize the trajectory and the aircraft.

```
minCoords = min(waypoints);
maxCoords = max(waypoints);
tp = theaterPlot('XLimits',1.2*[minCoords(1) maxCoords(1)], ...
               'YLimits',1.2*[minCoords(2) maxCoords(2)], ...
               'ZLimits',1.2*[minCoords(3) maxCoords(3)]);
```

```
% Create a trajectory plotter and a platform plotter
tPlotter = trajectoryPlotter(tp,'DisplayName','Trajectory');
pPlotter = platformPlotter(tp,'DisplayName','Aircraft');
```

Obtain the Cartesian waypoints of the trajectory using the `lookupPose` object function.

```
sampleTimes = timeOfArrival(1):timeOfArrival(end);
numSteps = length(sampleTimes);
[positions,orientations] = lookupPose(aircraftTraj,sampleTimes);
plotTrajectory(tPlotter,{positions})
axis equal
```

Plot the platform motion using an airplane mesh object.

```
mesh = scale(rotate(tracking.scenario.airplaneMesh,[0 0 180]),15); % Exaggerated scale for better
view(20.545,-20.6978)
for i = 1:numSteps
    plotPlatform(pPlotter,positions(i,:),mesh,orientations(i))
    % Uncomment the next line to slow the aircraft motion animation
    % pause(1e-7)
end
```

In the animation, the yaw angle of the aircraft aligns with the trajectory by default.

Create a second aircraft trajectory with the same waypoints as the first aircraft trajectory, but set its `AutoPitch` and `AutoBank` properties to `true`. This generates a trajectory more representative of the possible aircraft maneuvers.

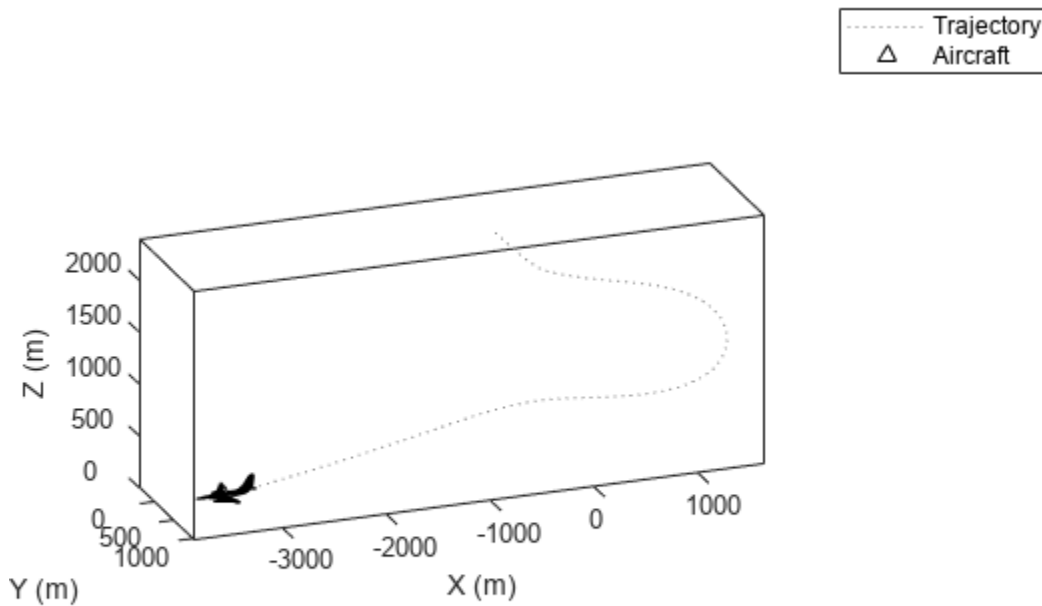
```
aircraftTraj2 = waypointTrajectory(waypoints,timeOfArrival, ...
                                   'AutoPitch',true, ...
                                   'AutoBank',true);
```

Plot the second trajectory and observe the change in aircraft orientation.

```

[positions2,orientations2] = lookupPose(aircraftTraj2,sampleTimes);
for i = 1:numSteps
    plotPlatform(pPlotter,positions2(i,:),mesh,orientations2(i));
    % Uncomment the next line to slow the aircraft motion animation
    % pause(1e-7)
end

```

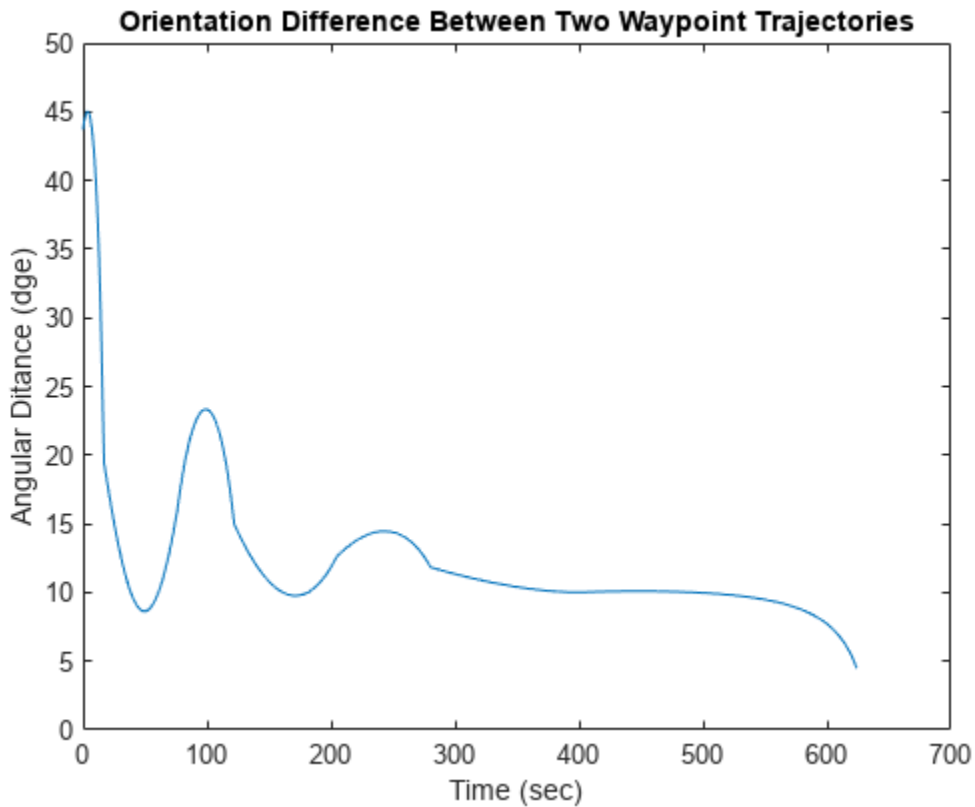


Visualize the orientation differences between the two trajectories in angles.

```

distance = dist(orientations2,orientations);
figure
plot(sampleTimes,distance*180/pi)
xlabel('Time (sec)')
ylabel('Angular Dittance (dge)')
title('Orientation Difference Between Two Waypoint Trajectories')

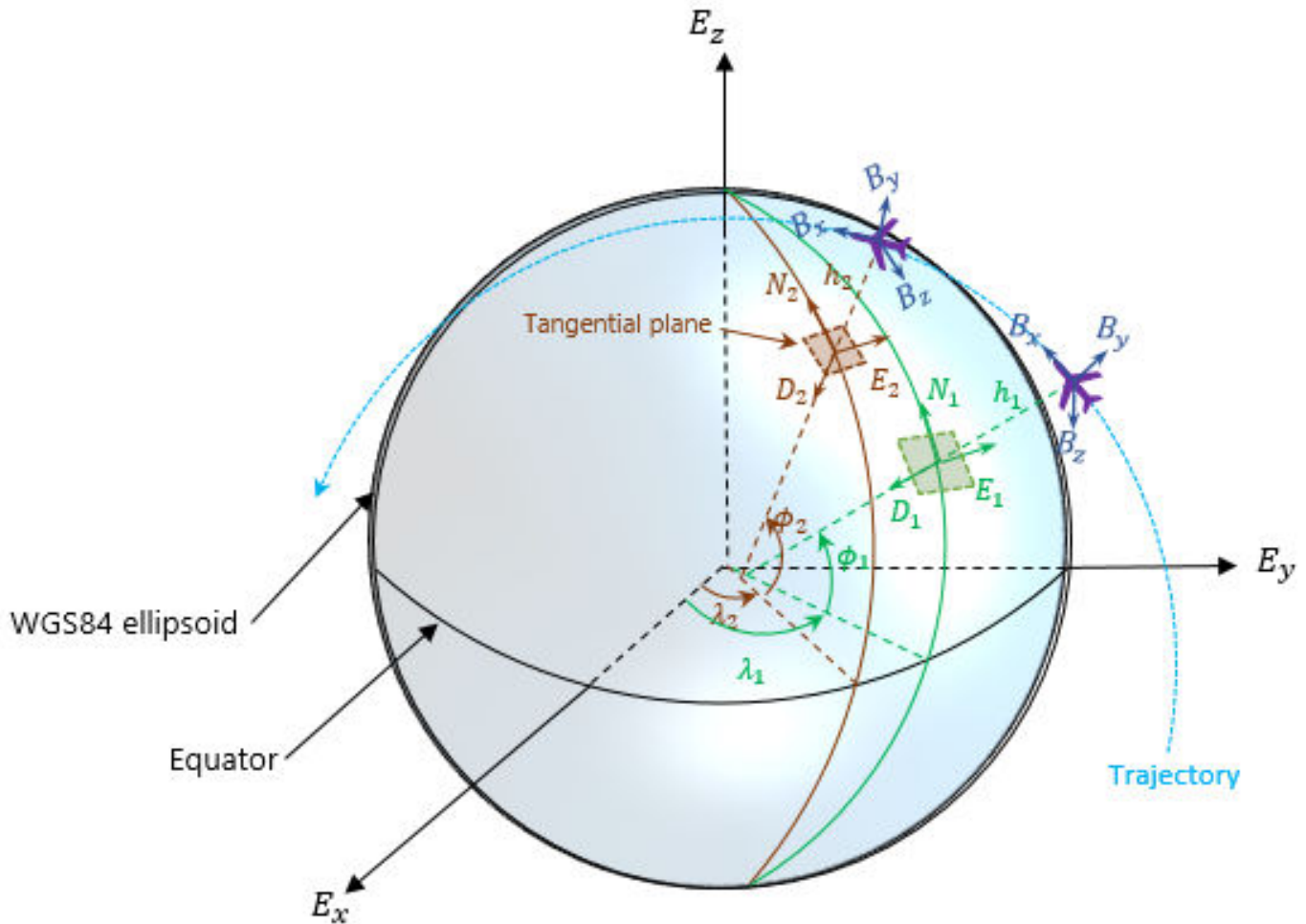
```



geoTrajectory

The `geoTrajectory` System object generates a trajectory using waypoints in a similar fashion as the `waypointTrajectory` object, but it has two major differences in how to specify waypoints and velocity inputs.

- When specifying waypoints for `geoTrajectory`, express each waypoint in the geodetic coordinates of latitude, longitude, and altitude above the WG84 ellipsoid model. Using geodetic coordinates, you can conveniently specify long-range trajectories, such as airplane flight trajectory on a realistic Earth model.
- When specifying the orientation and velocity information for each waypoint, the reference frame for orientation and velocity is the local NED or ENU frame defined under the current trajectory waypoint. For example, the $N_1-E_1-D_1$ frame shown in the figure is a local NED reference frame.



In the figure,

- E_x , E_y , and E_z are the three axes of the Earth-centered Earth-fixed (ECEF) frame, which is fixed on the Earth.
- (λ_1, ϕ_1, h_1) and (λ_2, ϕ_2, h_2) are the geodetic coordinates of the plane at the two waypoints.
- (N_1, E_1, D_1) and (N_2, E_2, D_2) are the two local NED frames corresponding to the two trajectory waypoints.
- B_x , B_y , and B_z are the three axes of the platform body frame, which is fixed on the platform.

geoTrajectory Example For Flight Trajectory

Load the flight data of a flight trajectory from Los Angeles to Boston. The data contains flight information including flight time, geodetic coordinates for each waypoint, course, and ground speed.

```
load flightData.mat
```

Create a geoTrajectory object based on the flight data.

```
planeTraj = geoTrajectory([latitudes longitudes heights],timeOfArrival, ...
    'Course',courses,'GroundSpeed',speeds);
```

Look up the Cartesian coordinates of the waypoints in the ECEF frame.

```
sampleTimes = 0:1000:3600*10;  
positionsCart = lookupPose(planeTraj,sampleTimes,'ECEF');
```

Show the trajectory using the helperGlobeView class, which approximates the Earth sphere.

```
viewer = helperGlobeView;  
plot3(positionsCart(:,1),positionsCart(:,2),positionsCart(:,3),'r')
```



You can further explore the trajectory by querying other outputs of the trajectory.

kinematicTrajectory

Unlike the two waypoint trajectory objects, the `kinematicTrajectory` System object uses kinematic attributes to specify a trajectory. Think of the trajectory as a numerical integration of the pose (position and orientation) and linear velocity of the platform, based on the linear acceleration and angular acceleration information. The pose and linear velocity are specified with respect to a chosen, fixed scenario frame, whereas the linear acceleration and angular velocity are specified with respect to the platform body frame.

kinematicTrajectory Example For UAV Path

Create a `kinematicTrajectory` object for simulating a UAV path. Specify the time span of the trajectory as 120 seconds.


```

traj = kinematicTrajectory('SampleRate',1, ...
    'AngularVelocitySource','Property');

tStart = 0;
tFinal = 120;
tspan = tStart:tFinal;

numSteps = length(tspan);
positions = NaN(3,numSteps);
velocities = NaN(3,numSteps);
vel = NaN(1,numSteps);

```

To form a square path covering a small region, separate the UAV trajectory into six segments:

- Taking off and ascending in the z-direction
- Moving in the positive x-direction
- Moving in the positive y-direction
- Moving in the negative x-direction
- Moving in the negative y-direction
- Descending in the z-direction and landing

In each segment, the UAV accelerates in one direction and then decelerates in that direction with the same acceleration magnitude. As a result, at the end of each segment, the velocity of the UAV is zero.

```

segSteps = floor(numSteps/12);
accelerations = zeros(3,numSteps);
acc = 1;
% Acceleration for taking off and ascending in the z-direction
accelerations(3,1:segSteps) = acc;
accelerations(3,segSteps+1:2*segSteps) = -acc;

% Acceleration for moving in the positive x-direction
accelerations(1,2*segSteps+1:3*segSteps) = acc;
accelerations(1,3*segSteps+1:4*segSteps) = -acc;

% Acceleration for moving in the positive y-direction
accelerations(2,4*segSteps+1:5*segSteps) = acc;
accelerations(2,5*segSteps+1:6*segSteps) = -acc;

% Acceleration for moving in the negative x-direction
accelerations(1,6*segSteps+1:7*segSteps) = -acc;
accelerations(1,7*segSteps+1:8*segSteps) = acc;

% Acceleration for moving in the negative y-direction
accelerations(2,8*segSteps+1:9*segSteps) = -acc;
accelerations(2,9*segSteps+1:10*segSteps) = acc;

% Descending in the z-direction and landing
accelerations(3,10*segSteps+1:11*segSteps) = -acc;
accelerations(3,11*segSteps+1:end) = acc;

```

Simulate the trajectory by calling the kinematicTrajectory object with the specified acceleration.

```

for i = 1:numSteps
    [positions(:,i),~,velocities(:,i)] = traj(accelerations(:,i)');
end

```

```

    vel(i) = norm(velocities(:,i));
end

```

Visualize the trajectory using theaterPlot.

```

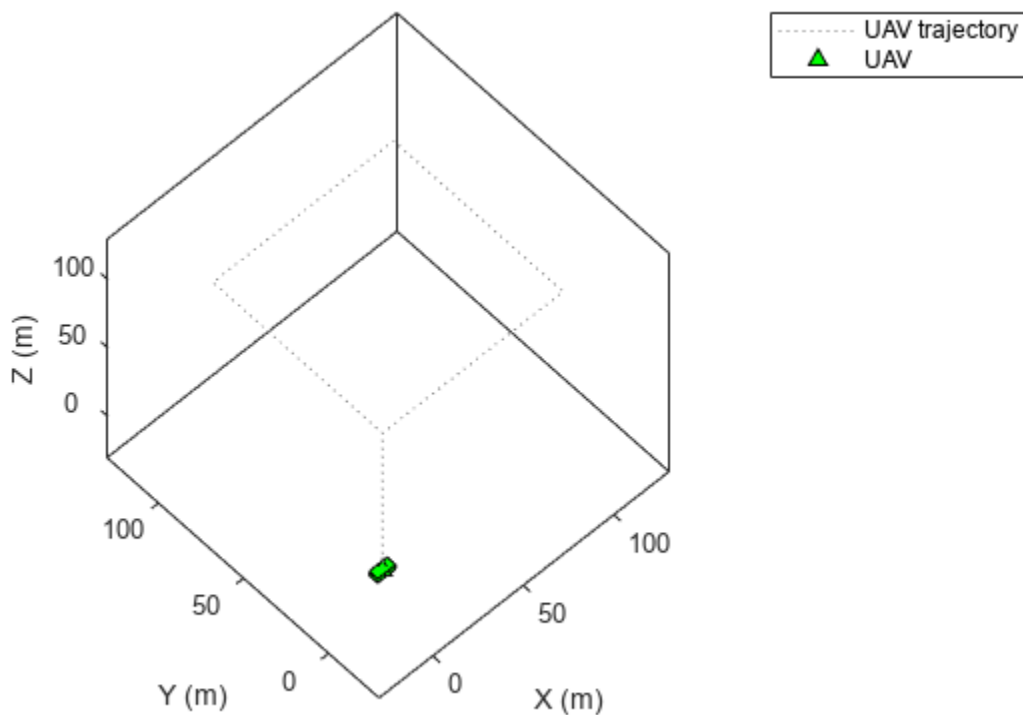
% Create a theaterPlot object and create plotters for the trajectory and the
% UAV Platform.
figure
tp = theaterPlot('XLimits',[-30 130],'YLimits',[-30 130],'ZLimits',[-30 130]);
tPlotter = trajectoryPlotter(tp,'DisplayName','UAV trajectory');
pPlotter = platformPlotter(tp,'DisplayName','UAV','MarkerFaceColor','g');

% Plot the trajectory.
plotTrajectory(tPlotter,{positions'})
view(-43.18,56.49)

% Use a cube to represent the UAV platform.
dims = struct('Length',10, ...
    'Width',5, ...
    'Height',3, ...
    'OriginOffset',[0 0 0]);

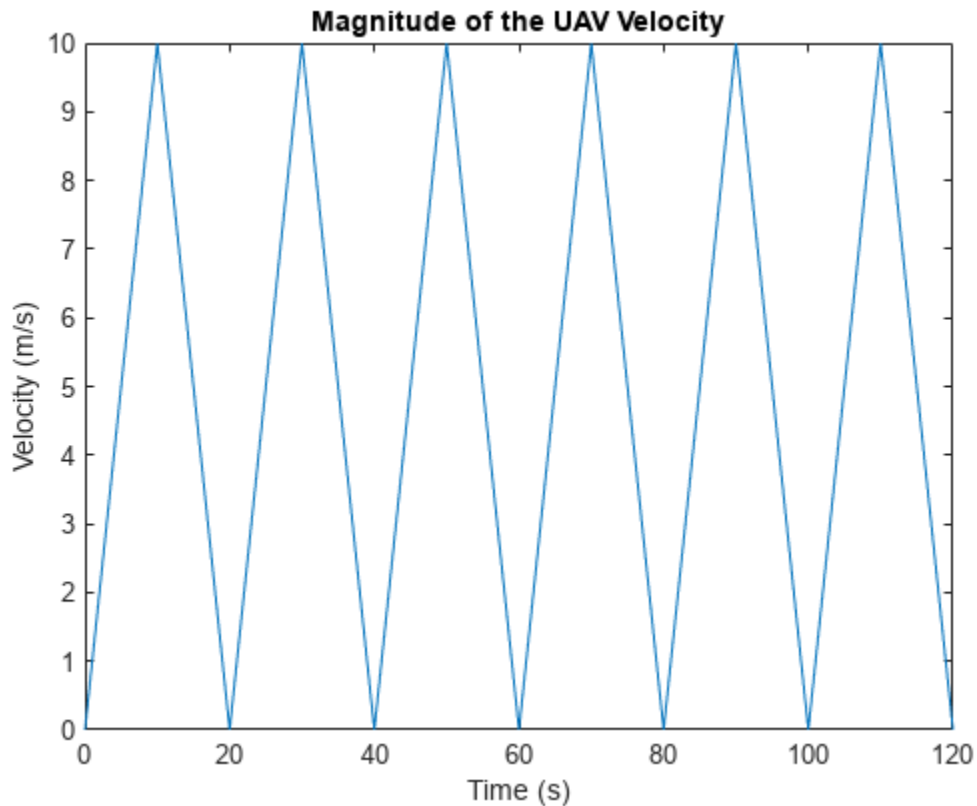
% Animate the UAV platform position.
for i = 1:numSteps
    plotPlatform(pPlotter,positions(:,i)',dims,eye(3))
    pause(0.01)
end

```



Show the velocity magnitude of the UAV platform.

```
figure
plot(tspan,vel)
xlabel('Time (s)')
ylabel('Velocity (m/s)')
title('Magnitude of the UAV Velocity')
```

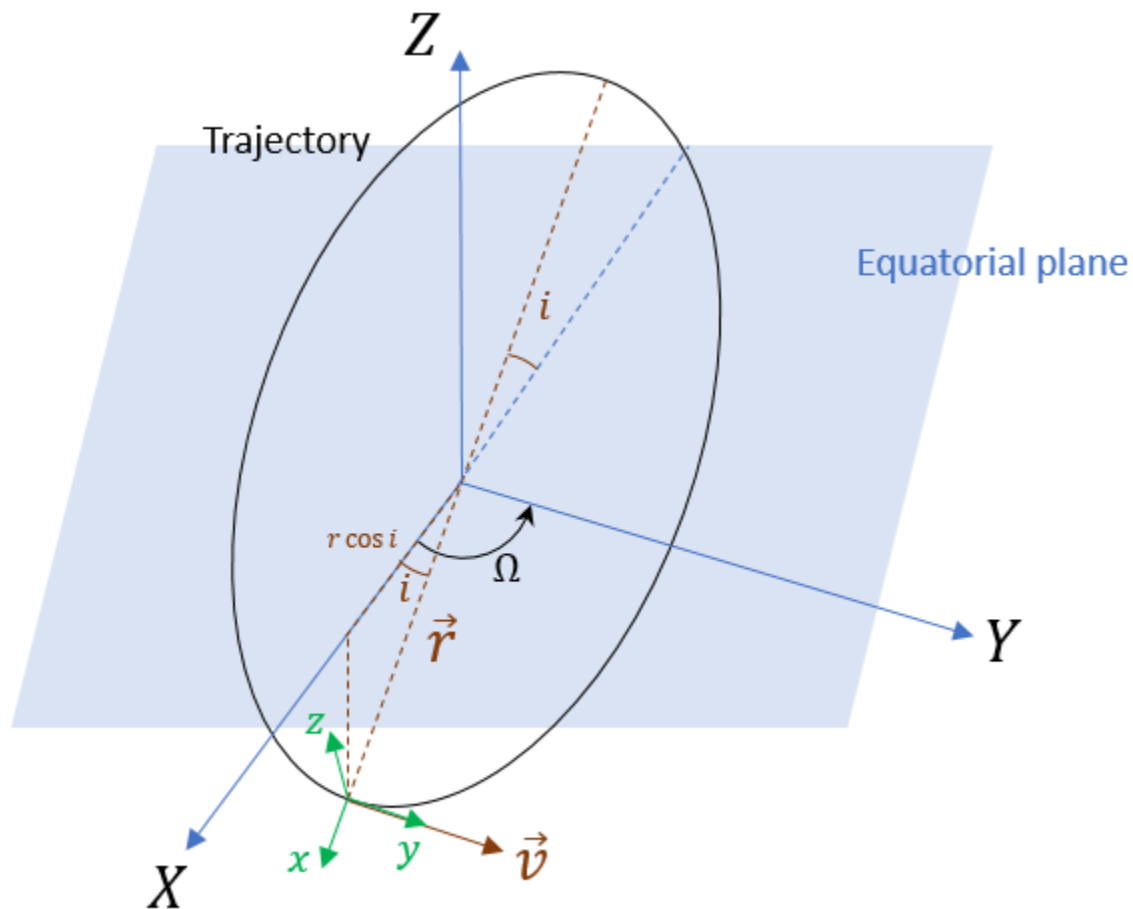


kinematicTrajectory Example For Spacecraft Trajectory

Use `kinematicTrajectory` to specify a circular spacecraft trajectory. The orbit has these elements:

- Orbital radius (r) — 7000 km
- Inclination (i) — 60 degrees
- Argument of ascending node (Ω) — 90 degrees. The ascending node direction is aligned with the Y-direction.
- True anomaly (ν) — -90 degrees

In the figure, X - Y - Z is the Earth-centered inertial (ECI) frame, which has a fixed position and orientation in space. x - y - z is the spacecraft body frame, fixed on the spacecraft. \vec{r} and \vec{v} are the initial position and velocity of the spacecraft.



To specify the circular orbit using `kinematicTrajectory`, you need to provide the initial position, initial velocity, and initial orientation of the spacecraft with respect to the ECI frame. For the chosen true anomaly ($v = -90^\circ$), the spacecraft velocity is aligned with the Y-direction.

```
inclination = 60; % degrees
mu = 3.986e14; % standard earth gravitational parameter
radius = 7000e3; % meters
v = sqrt(mu/radius); % speed
initialPosition = [radius*cosd(inclination),0,-radius*sind(inclination)]';
initialVelocity = [0 v 0]';
```

Assume the x-direction of the spacecraft body frame is the radial direction, the z-direction is the normal direction of the orbital plane, and the y-direction completes the right-hand rule. Use the assumptions to specify the orientation of the body frame at the initial position.

```
orientation = quaternion([0 inclination 0], 'eulerd', 'zyx', 'frame');
```

Express the angular velocity and the angular acceleration of the trajectory in the platform body frame.

```
omega = v/radius;
angularVelocity = [0 0 omega]';
```

```
a = v^2/radius;
acceleration = [-a 0 0]';
```

Set a simulation time of one orbital period. Specify a simulation step as 2 seconds.

```
tFinal = 2*pi/omega;
dt = 2;
sampleRate = 1/dt;
tspan = 0:dt:tFinal;
numSteps = length(tspan);
```

Create the spacecraft trajectory. Since the acceleration and angular velocity of the spacecraft remain constant with respect to the spacecraft body frame, specify them as constants. Generate position and orientation outputs along the trajectory by using the `kinematicTrajectory` object.

```
traj = kinematicTrajectory('SampleRate',sampleRate, ...
    'Position',initialPosition, ...
    'Velocity',initialVelocity, ...
    'Orientation',orientation, ...
    'AngularVelocity',angularVelocity, ...
    'Acceleration',acceleration, ...
    'AccelerationSource','Property', ...
    'AngularVelocitySource','Property');
```

```
% Generate position and orientation outputs.
positions = NaN(3,numSteps);
orientations = zeros(numSteps,1,'quaternion');
for i = 1:numSteps
    [positions(:,i),orientations(i)] = traj();
end
```

Use the `helperGlobeView` class and `theaterPlot` to show the trajectory.

```
viewer = helperGlobeView(0,[60 0]);
tp = theaterPlot('Parent',gca,...
    'XLimits',1.2*[-radius radius],...
    'YLimits',1.2*[-radius radius],...
    'ZLimits',1.2*[-radius radius]);
tPlotter = trajectoryPlotter(tp,'LineWidth',2);
pPlotter = platformPlotter(tp,'MarkerFaceColor','m');
legend(gca,'off')

plotTrajectory(tPlotter,{positions'})
% Use a cube with exaggerated dimensions to represent the spacecraft.
dims = struct('Length',8e5,'Width',4e5,'Height',2e5,'OriginOffset',[0 0 0]);

for i = 1:numSteps
    plotPlatform(pPlotter,positions(:,i)',dims,orientations(i))
    % Since the reference frame is the ECI frame, earth rotates with respect to it.
    rotate(viewer,dt)
end
```



Summary

In this topic, you learned how to use three trajectory objects to customize your own trajectories based on the available information. In addition, you learned the fundamental differences in applying them. This table highlights the main attributes of these trajectory objects.

Trajectory Object	Position Inputs	Linear Velocity Inputs	Orientation	Acceleration and Angular Velocity Inputs	Recommended Distance Span
waypointTrajectory	Cartesian waypoints expressed in a fixed frame (NED or ENU)	One of these options: <ul style="list-style-type: none"> • Automatically generate velocity for a smooth trajectory, by default • Specify velocity in the fixed frame at each waypoint • Specify course, ground speed, and climb rate in the fixed frame at each waypoint 	One of these options: <ul style="list-style-type: none"> • Auto yaw by default, auto pitch by selection, and auto bank by selection • Specify orientation in the fixed frame 	Cannot specify	From within tens to hundreds of kilometers

Trajectory Object	Position Inputs	Linear Velocity Inputs	Orientation	Acceleration and Angular Velocity Inputs	Recommended Distance Span
geoTrajectory	Geodetic waypoints in the ECEF frame	One of the these options: <ul style="list-style-type: none"> • Automatically generate velocity for a smooth trajectory, by default • Specify velocity in the local frame (NED or ENU) for each waypoint • Specify course, ground speed, and climb rate in the local frame (NED or ENU) for each waypoint 	One of these options: <ul style="list-style-type: none"> • Auto yaw by default, auto pitch by selection, and auto bank by selection • Specify orientation in the local frame 	Cannot specify	From hundreds to thousands of kilometers
kinematicTrajectory	Initial position expressed in a chosen, fixed frame	Only initial velocity in the fixed frame	Only initial orientation in the fixed frame	Specify acceleration and angular velocity in the platform body frame	Unlimited distance span

Radar Detections

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

Simulate Radar Detections

The `fusionRadarSensor` object simulates the detection of targets by a 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 `fusionRadarSensor` creates a sensor. The output of `fusionRadarSensor` 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 `fusionRadarSensor` object. Set the radar properties using name-value pairs and then execute the simulator. For example,

```
radar1 = fusionRadarSensor( ...
    'SensorIndex',1,...
    'UpdateRate',10, ...           % Hz
    'ReferenceRange', 111.0e3, ...  % m
    'ReferenceRCS', 0.0, ...        % dBsm
    '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 `fusionRadarSensor` that make it easier to specify the properties of commonly implemented radar scan modes.

- `sensor = fusionRadarSensor('Rotator')` creates a `fusionRadarSensor` 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 = fusionRadarSensor('Sector')` creates a `fusionRadarSensor` 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 = fusionRadarSensor('Raster')` returns a `fusionRadarSensor` 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 = fusionRadarSensor('No scanning')` returns a `fusionRadarSensor` 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 = fusionRadarSensor(1,'Raster','ScanMode','Electronic')
```

Radar Sensor Parameters

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

```
help fusionRadarSensor
```

at the command line.

Sensor Location Parameters

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.
MountingAngles	Yaw, pitch, and roll angles of the sensor mounting frame with respect to the platform frame.
DetectionCoordinates	<p>Specifies the coordinate system for detections reported in the "Detections" on page 2-12 output struct. The coordinate system can be one of:</p> <ul style="list-style-type: none"> • '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

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.

Resolution and Bias 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 rate 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

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 <code>MaxUnambiguousRange</code> property value are wrapped into the interval <code>[0 MaxUnambiguousRange]</code> . When false, targets are reported at their unwrapped range.
HasRangeRateAmbiguities	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 <code>MaxUnambiguousRadialSpeed</code> property value are wrapped into the interval <code>[0 MaxUnambiguousRadialSpeed]</code> . When false, targets are reported at their unwrapped range rates. This property only applies when the <code>HasRangeRate</code> property is set to <code>true</code> .
HasNoise	Specifies if noise is added to the sensor measurements. Set this property to <code>true</code> to report measurements with noise. Set this property to <code>false</code> to report measurements without noise. The reported measurement noise covariance matrix contained in the output <code>objectDetection</code> struct is always computed regardless of the setting of this property.
HasOcclusion	Enable occlusion from extended objects, specified as <code>true</code> or <code>false</code> . Set this property to <code>true</code> 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 <code>false</code> to disable occlusion of extended objects.
HasINS	Set this property to <code>true</code> 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 <code>MeasurementParameters</code> field of the reported detections. Then, the tracking and fusion algorithms can estimate the state of the target detections in scenario coordinates.

Range and Range Rate Parameters

MaxUnambiguousRange	<p>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 $[0 \text{ MaxUnambiguousRange}]$. This property only applies to true target detections when you set <code>HasRangeAmbiguities</code> property to <code>true</code>.</p> <p>This property also defines the maximum range at which false alarms are generated. This property only applies to false target detections when you set <code>HasFalseAlarms</code> property to <code>true</code>.</p>
MaxUnambiguousRadialSpeed	<p>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 $[-\text{MaxUnambiguousRadialSpeed} \text{ MaxUnambiguousRadialSpeed}]$. This property only applies to true target detections when you set both the <code>HasRangeRate</code> and <code>HasRangeRateAmbiguities</code> properties to <code>true</code>.</p> <p>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 <code>HasFalseAlarms</code> and <code>HasRangeRate</code> properties to <code>true</code>.</p>

Detector Input

Each sensor created by `fusionRadarSensor` 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 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. 0 is reserved for unclassified platform types and is the default value.
Position	Position of target in the platform body frame, 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 the platform body frame, 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 the platform body frame, specified as a 1-by-3 vector. Units are in meters per second-squared. The default is [0 0 0].
Orientation	Orientation of the target with respect to platform body frame, specified as a scalar quaternion or a 3-by-3 rotation matrix. Orientation defines the frame rotation from the platform coordinate system to the target body coordinate system. Units are dimensionless. The default is quaternion(1,0,0,0).
AngularVelocity	Angular velocity of the target in the platform body frame, specified as a real-valued, 1-by-3 vector. The magnitude of the vector defines the angular speed. 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

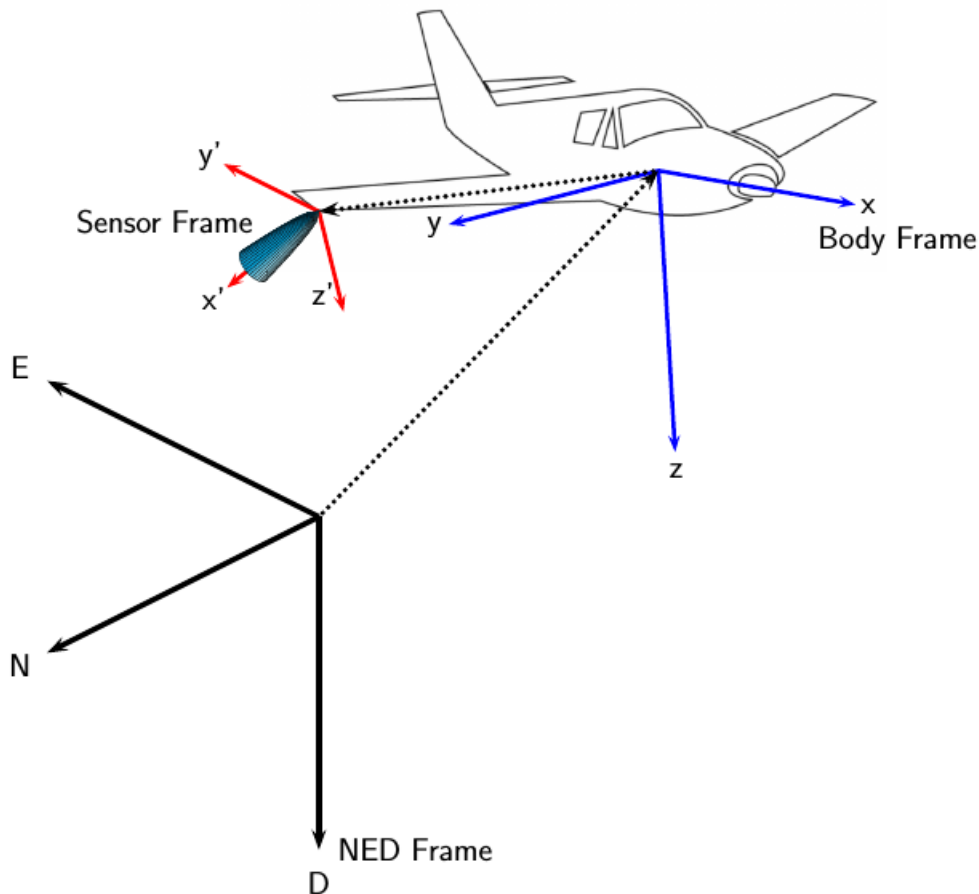
PlatformID
ClassID
RCSPattern
RCSAzimuthAngles
RCSElevationAngles

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 `fusionRadarSensor` 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, <code>Frame</code> is set to <code>'rectangular'</code> . When detections are reported in spherical coordinates, <code>Frame</code> is set <code>'spherical'</code> for the first <code>struct</code> .
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 <code>Orientation</code> performs a frame rotation from the parent coordinate frame to the child coordinate frame. If <code>false</code> , <code>Orientation</code> 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 <code>false</code> for the first <code>struct</code> when detections are reported in spherical coordinates and <code>HasElevation</code> is <code>false</code> , otherwise it is <code>true</code> .
HasVelocity	A logical scalar indicating if the reported detections include velocity measurements. <code>true</code> when <code>HasRangeRate</code> is enabled, otherwise <code>false</code> .

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 Statistical Radar Models 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 targets either emit or reflect. Sensors primarily use the intercepted energy to obtain information about the state and attributes of targets.

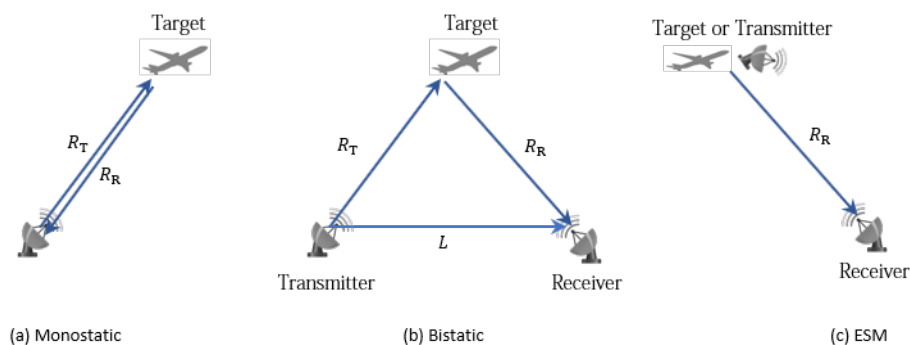
A sensor is an active sensor if the intercepted energy originates from itself, such as a monostatic radar or a monostatic sonar. A sensor is a passive sensor if the intercepted energy originates from an outside source, such as an infrared (IR) sensor, which receives radiated energy from a target.

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 signal). Therefore, the detection quality of sensors involves many factors such as accuracy, resolution, bias, and false alarms. Also, it is essential to consider the detectability of sensors, which relies on factors such as scanning limits, field of view, and sensor mounting in tracking system design.

This introduction mainly discusses radar (including the `fusionRadarSensor` and `radarEmitter` objects), but some of the following descriptions also apply to other types of sensors (including the `irSensor`, `sonarSensor`, and `sonarEmitter` objects).

Radar Detection Mode

Radar uses radio wave signals reflected or emitted from a target to detect the target. Given different transmitter and receiver configurations, a radar can have one of three detection modes: monostatic, bistatic, or electronic support measures (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. In addition to 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. The radar sensor obtains the bistatic range measurement as $R_b = c\Delta t$, where Δt is the time difference between the receiver intercepting the direct signal from the transmitter and intercepting the reflected signal from the target. In addition to the bistatic range measurement, a bistatic radar can optionally report the 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. Without additional information, a bistatic sensor can only report the measurements in a spherical coordinate system.

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

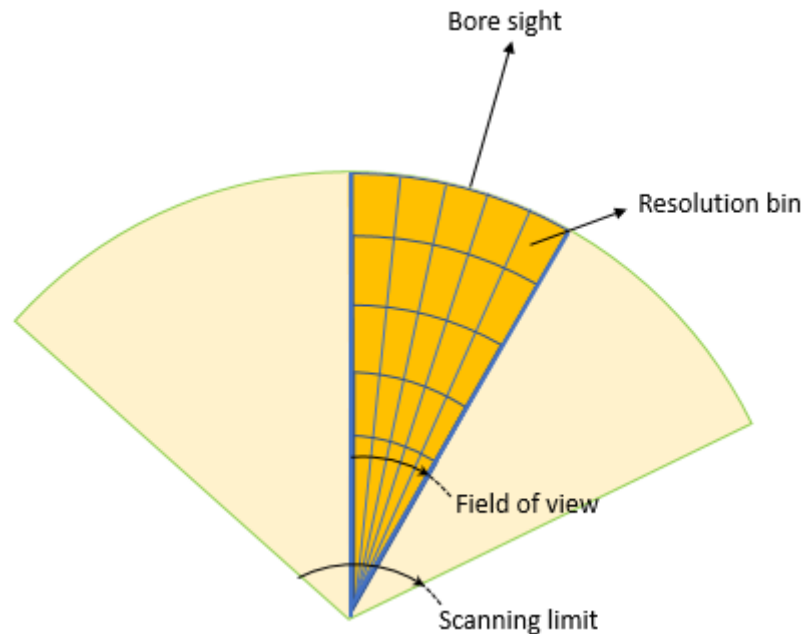
Mounting Radar on Platform

To interpret the detection generated by the radar, you need to understand how the radar is mounted on the platform. The radar mounting frame (M) origin can be displaced from the platform frame origin (P). This displacement is usually specified by the `MountingLocation` property of a sensor object, such as `fusionRadarSensor`. The radar mounting frame can also angularly displaced from the platform frame. You can specify this angle displacement represented by three rotation angles in the z - y - x sequence using the `MountingAngles` property. Initially, the radar scanning frame (S) is aligned with its mounting frame (M). However, when the radar starts scanning, the radar can scan around the z - and y -axes of the mounting frame. The x -direction of the radar scanning frame is aligned with the current boresight direction of the radar.

Detection Ability and Quality

Sensor Coverage

In most cases, a radar is working in a scanning mode in which the sensor beams sweep back and forth, with width equal to its field of view (FOV), through a space region defined by the radar scanning limit. The FOV is typically 3 decibels (dB) of the radar beam width. The speed of the sweeping is specified by the `UpdateRate` property of the sensor object. You can obtain the scanning speed of the sensor using its field of view and update rate. For example, if the update rate is 20 Hz and the field of view is 2 degrees, then the radar scanning speed is 40 degrees per second. For more details on radar sensor coverage, see the “Scanning Radar Mode Configuration” on page 6-122 example.



Resolution and Accuracy

Sensor resolution defines the ability of a sensor to distinguish between two targets. In a 3-D space, the resolution bin of a radar is formed by the azimuth boundary, elevation boundary, and range boundary. If two targets fall within the same resolution bin, then the radar cannot distinguish between them and reports them as one target in the detection.

Sensor accuracy can be described by the standard deviation of the measurement error. The accuracy is mainly affected by two factors: the signal to noise ratio (SNR) of the sensor and the detection bias of the sensor. 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. A larger SNR results in a smaller measurement error and higher accuracy. For radar, SNR is usually a function of the radar cross section (RCS) of the target. The bias of a sensor is mainly due to imperfect alignment or calibration, and is often assumed to be a constant value. In each radar object, you can specify its bias as a fraction of the sensor resolution bin size using properties such as `AzimuthBiasFraction`. The larger the bias is, the more errors the detection incorporates.

Detection Statistics

Radar can also make an incorrect assessment of the surveillance region. The probability of false alarm (p_{FA}) represents the probability that the radar reports a detection on a resolution bin even though the resolution bin is not occupied by a target. The probability of detection (P_D) represents the probability that the radar 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 the radar. P_D is mainly a function of the SNR of the target and the p_{FA} of the radar.

When a radar operates in an environment where other undesirable radio frequency (RF) emissions interfere with the waveforms emitted by the radar, the radar can experience a degradation in detection performance in the direction of the interfering signals.

Reported Target Range and Range-Rate

In many cases, a radar has the maximum unambiguous range and range-rate limits. If the distance between a target and the sensor is greater than the maximum unambiguous range, then the sensor wraps the detected range in the range of $[0, R_{\max}]$, where R_{\max} is the maximum unambiguous range. For example, assume the target's range R_t is larger than R_{\max} , then the reported range of the target is $\text{mod}(R_t, R_{\max})$, where mod is the remainder after division function in MATLAB. In a radar object, you can disable this limitation by setting the `HasMaxUnambiguousRange` property to `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 identification and characteristics of the target, such as shape and reflectivity. The kinematic measurements for radar are described here.

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

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

In the Sensor Fusion and Tracking Toolbox, sensor objects output detections in the form of `objectDetection` objects. 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 about 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 pose (position, velocity, and orientation) and coordinate frame information for measurements at the time of detection. For more details, see “Measurement Parameters” and the “Convert Detections to objectDetection Format” on page 6-435 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 of 9.81m/s^2 . 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 most of the inertial sensor filters (marked as tunable in the table below).

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

Tip Other than the filters listed in this table, you can use the `insEKF` object to build a flexible inertial sensor fusion framework, in which you can use built-in or custom motion models and sensor models. For more details, see “Fuse Inertial Sensor Data Using insEKF-Based Flexible Fusion Framework” on page 3-7.

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used	Tunable
<code>ecompass</code>	<ul style="list-style-type: none"> • Accelerometer • Magnetometer 	Orientation	The filter assumes no sustained linear and angular acceleration other than gravitational acceleration.	TRIAD method	No
<code>ahrsfilter</code>	<ul style="list-style-type: none"> • Accelerometer • Gyroscope • Magnetometer 	Orientation and angular velocity	The filter assumes no sustained linear and angular acceleration other than gravitational acceleration.	Error-state Kalman filter	Yes

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used	Tunable
ahrs10filter	<ul style="list-style-type: none"> 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	Yes
imufilter	<ul style="list-style-type: none"> Accelerometer Gyroscope 	Orientation and angular velocity	The filter assumes no sustained linear and angular acceleration other than gravitational acceleration.	Error-state Kalman filter	Yes
complementaryFilter	<ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> Use high and low pass filters to reduce noise in various sensor readings. Fuse the filtered sensor readings based on their assigned weights. 	No

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used	Tunable
insfilterMARG	<ul style="list-style-type: none"> Accelerometer Gyroscope Magnetometer GPS 	Orientation, position, velocity, delta angle bias, delta velocity bias, geomagnetic field vector, magnetometer bias	<p>The prediction step takes the accelerometer and gyroscope inputs. Therefore, the filter assumes:</p> <ul style="list-style-type: none"> Piecewise constant linear acceleration. Piecewise constant angular velocity. Accelerometer and gyroscope run at the same rate with no sample dropping. 	Discrete extended Kalman filter	Yes
insfilterAsync	<ul style="list-style-type: none"> Accelerometer Gyroscope Magnetometer GPS 	Orientation, angular velocity, position, velocity, acceleration, accelerometer bias, gyroscope bias, geomagnetic field vector, magnetometer bias	<p>The filter assumes:</p> <ul style="list-style-type: none"> Constant angular velocity Constant acceleration <p>The filter does not require the sensors to be synchronous and each sensor can have sample dropping.</p>	Continuous discrete extended Kalman Filter	Yes

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used	Tunable
insfilterNonholonomic	<ul style="list-style-type: none"> Accelerometer Gyroscope GPS 	Orientation, position, velocity, gyroscope bias, accelerometer bias	<p>The prediction step takes the accelerometer and gyroscope inputs. Therefore, the filter assumes:</p> <ul style="list-style-type: none"> Piece-wise constant linear acceleration. Piece-wise constant angular velocity. Accelerometer and gyroscope run at the same rate with no sample dropping. <p>Also, the filter assumes the platform moves forward without side slip.</p>	Discrete extended Kalman Filter	Yes

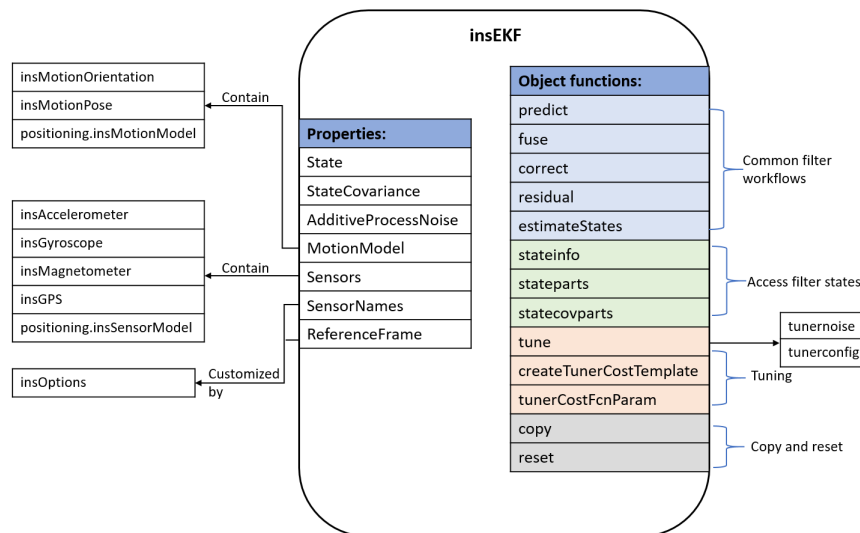
Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used	Tunable
insfilterErrorState	<ul style="list-style-type: none"> • Accelerometer • Gyroscope • Magnetometer • GPS • Visual odometry scale 	Orientation, position, velocity, gyroscope bias, accelerometer bias, and visual odometry scale	<p>The prediction step takes the accelerometer and gyroscope inputs. Therefore, the filter assumes:</p> <ul style="list-style-type: none"> • 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	Yes

See Also
tunerconfig

Fuse Inertial Sensor Data Using insEKF-Based Flexible Fusion Framework

The `insEKF` filter object provides a flexible framework that you can use to fuse inertial sensor data. You can fuse measurement data from various inertial sensors by selecting or customizing the sensor models used in the filter, and estimate different platform states by selecting or customizing the motion model used in the filter. The `insEKF` object is based on a continuous-discrete extended Kalman filter, in which the state prediction step is continuous, and the measurement correction or fusion step is discrete.

Overall, the object contains 7 properties and 13 object functions. The object also interacts with other objects and functions to support regular filter workflows, as well as filter tuning.



Object Properties

The `insEKF` object maintains major filter variables and information of the motion model and sensor models using these properties:

- **State** — Platform state saved in the filter, specified as a vector. This property maintains the whole state vector, including the platform states defined by the motion model, such as orientation, position, and velocity, as well as the sensor states defined by sensor models, such as sensor biases.

You can interact with this property in various ways:

- Display the state information saved in the vector by using the `stateinfo` object function.
- Directly specify this property. This may not be convenient since the dimension of the state vector can be large.
- Get or set a part or component of the state vector by using the `stateparts` object function.
- **StateCovariance** — State estimate error covariance, specified as a matrix. This property maintains the whole covariance matrix corresponding to the state vector. You can interact with this property in various ways:

- Directly specify this property. This may not be convenient since the dimension of the covariance matrix can be large.
- Get or set a part of the covariance matrix by using the `statecovparts` object function.
- `AdditiveProcessNoise` — Additive process noise covariance, specified as a matrix. This property maintains the whole additive process noise covariance matrix corresponding to the state vector.
- `MotionModel` — Motion model to predict the filter state, specified as an object. You can use one of the these options to specify this property when constructing the filter:
 - `insMotionOrientation` — Model orientation-only platform motion assuming a *constant angular velocity*. Using this object adds the quaternion and angular velocity state to the `State` property.
 - `insMotionPose` — Model 3-D motion assuming *constant angular velocity* and *constant linear acceleration*. Using this model adds the quaternion, angular velocity, position, linear velocity, and acceleration state to the `State` property.
 - `positioning.INSMotionModel` — An interface class to implement a motion model object used with the filter. To customize a motion model object, you must inherit from this interface class and implement at least two methods: `modelstates` and `stateTransition`.
- `Sensors` — Sensor models that model the corresponding sensor measurement based on the platform state, specified as a cell array of INS sensor objects. You can specify each INS sensor object using one of the these options:
 - `insAccelerometer` — Model accelerometer readings, including the gravitational acceleration and sensor bias by default. The model also includes the sensor acceleration if the `State` property of the filter includes the `Acceleration` state.
 - `insMagnetometer` — Model magnetometer readings, including the geomagnetic vector and the sensor bias.
 - `insGyroscope` — Model gyroscope readings, including the angular velocity vector and the sensor bias.
 - `insGPS` — Model GPS readings, including the latitude, longitude, and altitude coordinates. If you fuse velocity data from the GPS sensor, the object additionally models velocity measurements.
 - `positioning.INSSensorModel` — An interface class to implement a sensor model object used with the filter. To customize a sensor model object, you must inherit from this interface class and implement at least the `measurement` method.
- `SensorNames` — Names of the sensors added to the filter, specified as a cell array of character vector. The filter object assigns default names to the added sensors. These sensor names are useful when you use various object functions of the filter.
 - To customize the sensor names, you must use the `insOptions` object.
- `ReferenceFrame` — Reference frame of the extended Kalman filter object, specified as "NED" for the north-east-down frame by default. To customize it as "ENU" for the east-north-up frame, you must use the `insOptions` object.

Note You can also specify the data type used in the filter as `double` (default) or `single` by specifying the `DataType` property of the `insOptions` object.

Object Functions

The `insEKF` object provides various object functions for implementing common filter workflows, accessing filter states and covariances, and filter tuning.

These object functions support common filter workflows:

- `predict` — Predict the filter state forward in time based on the motion model used in the filter.
- `fuse` — Fuse or correct the filter state using a measurement based on a sensor model previously added to the filter. You can use this object function multiple times if you need to fuse multiple measurements.
- `correct` — Correct the filter using direct state measurement of the filter state. A direct measurement contains a subset of the filter state vector elements. You can use this object function multiple times for multiple direct measurements.
- `residual` — Return the residual and residual covariance of a measurement based on the current state of the filter.
- `estimateStates` — Obtain the state estimates based on a timetable of sensor data. The object function processes the measurements one-by-one and returns the corresponding state estimates.

These object functions enable you to access various states and variables maintained by the filter:

- `stateinfo` — Return or display the state components saved in the `State` property of the filter. Using this function, you can observe what components the state consists of and the indices for all the state components.
- `stateparts` — Get or set parts of the state vector. Since the state vector contain many components, this object function enables you to get or set only a component of the whole state vector.
- `statecovparts` — Get or set parts of the state estimate error covariance matrix. Similar to the `stateparts` function, this object function enables you to get or set only a part of the state estimate error covariance matrix.

To obtain more accurate state estimates, you often need to tune the filter parameters to reduce estimate errors. The `insEKF` object provides these object functions to facilitate filter tuning:

- `tune` — Adjust the `AdditiveProcessNoise` property of the filter object and measurement noise of the sensors to reduce the state estimation error between the filter estimates and the ground truth.
 - You can use the `tunernoise` function to obtain an example for the measure noise structure, required by the `tune` function.
 - You can optionally use the `tunerconfig` object to specify tuning parameters, such as function tolerance and the cost function, as well as which elements of the process noise matrix to tuned.
- `createTunerCostTemplate` — Creates a template for a tuner cost function that you can further modify to customize your own cost function.
- `tunerCostFcnParam` — Creates a required structure for tuning an `insEKF` filter with a custom cost function. The structure is useful when generating C-code for a cost function using MATLAB® Coder™.

You can use the `copy` and `reset` object functions to conveniently create a new `insEKF` object based on an existing `insEKF` object.

- `copy` — Create a copy of the `insEKF` object.
- `reset` — Resets the `State` and `StateCovariance` properties of the `insEKF` object to their default values.

Example: Fuse Inertial Sensor Data Using `insEKF`

This example introduces the basic workflows for fusing inertial sensor data using the `insEKF` object, which supports a flexible fusion framework based on a continuous discrete Kalman filter.

Create `insEKF` Object

Create an `insEKF` object directly.

```
filter1 = insEKF

filter1 =
  insEKF with properties:
        State: [13×1 double]
   StateCovariance: [13×13 double]
 AdditiveProcessNoise: [13×13 double]
      MotionModel: [1×1 insMotionOrientation]
         Sensors: {[1×1 insAccelerometer] [1×1 insGyroscope]}
   SensorNames: {'Accelerometer' 'Gyroscope'}
  ReferenceFrame: 'NED'
```

From the `Sensors` property, note that the object contains two sensor models, `insAccelerometer` and `insGyroscope` by default. These enables you to fuse accelerometer and gyroscope data, respectively. From the `MotionModel` property, note that the object defaults to an `insMotionOrientation` model, which models rotation-only motion and not translational motion. Due to the specified motion model and sensor models, the state of the filter is a 13-by-1 vector. Get the components and corresponding indices from the state vector using the `stateinfo` object function.

```
stateinfo(filter1)

ans = struct with fields:
   Orientation: [1 2 3 4]
 AngularVelocity: [5 6 7]
 Accelerometer_Bias: [8 9 10]
 Gyroscope_Bias: [11 12 13]
```

Note that, in addition to the orientation and angular velocity states, the filter also includes the accelerometer bias and gyroscope bias.

You can explicitly specify the motion model and sensor models when constructing the filter. For example, create an `insEKF` object and specify the motion model as an `insMotionPose` object, which models both rotational motion and translational motion, and specify the sensors as an `insAccelerometer`, an `insGPS`, and another `insGPS` object. This enables the fusion of one set of accelerometer data and two sets of GPS data.

```
filter2 = insEKF(insAccelerometer,insGPS,insGPS,insMotionPose)
```

```

filter2 =
    insEKF with properties:
        State: [19x1 double]
        StateCovariance: [19x19 double]
        AdditiveProcessNoise: [19x19 double]
        MotionModel: [1x1 insMotionPose]
        Sensors: {[1x1 insAccelerometer] [1x1 insGPS] [1x1 insGPS]}
        SensorNames: {'Accelerometer' 'GPS' 'GPS_1'}
        ReferenceFrame: 'NED'
    
```

The `State` property is a 19-by-1 vector that contains these components:

```

stateinfo(filter2)

ans = struct with fields:
    Orientation: [1 2 3 4]
    AngularVelocity: [5 6 7]
    Position: [8 9 10]
    Velocity: [11 12 13]
    Acceleration: [14 15 16]
    Accelerometer_Bias: [17 18 19]
    
```

The `SensorNames` property enables you to indicate specific sensors when using various object functions of the filter. The filter generates the names for added sensors in a default format. To provide custom names for the sensors, you must use the `insOptions` object. The `insOptions` object can also specify the data type of variables used in the filter and the `ReferenceFrame` of the filter.

```

options = insOptions(SensorNamesSource="Property", ...
    SensorNames={'Sensor1','Sensor2','Sensor3'}, ...
    Datatype="single", ...
    ReferenceFrame="ENU");
filter3 = insEKF(insAccelerometer,insGPS,insGPS,insMotionPose,options)

filter3 =
    insEKF with properties:
        State: [19x1 single]
        StateCovariance: [19x19 single]
        AdditiveProcessNoise: [19x19 single]
        MotionModel: [1x1 insMotionPose]
        Sensors: {[1x1 insAccelerometer] [1x1 insGPS] [1x1 insGPS]}
        SensorNames: {'Sensor1' 'Sensor2' 'Sensor3'}
        ReferenceFrame: 'ENU'
    
```

You can also directly obtain the indices of a state component based on the sensor names. For example,

```

stateinfo(filter3,'Sensor1_Bias')

ans = 1x3
    17    18    19
    
```

Configure Filter Properties

Create a new `insEKF` object with an accelerometer and a gyroscope. Explicitly define these two sensors for later usage.

```
accSensor = insAccelerometer;
gyroSensor = insGyroscope;
filter = insEKF(accSensor,gyroSensor)

filter =
  insEKF with properties:
        State: [13×1 double]
    StateCovariance: [13×13 double]
AdditiveProcessNoise: [13×13 double]
    MotionModel: [1×1 insMotionOrientation]
        Sensors: {[1×1 insAccelerometer] [1×1 insGyroscope]}
    SensorNames: {'Accelerometer' 'Gyroscope'}
    ReferenceFrame: 'NED'
```

Load prerecorded data for an accelerometer and a gyroscope. The sample rate of the recorded data is 100 Hz. It contains the sensor data, ground truth, and the initial orientation represented by a quaternion.

```
ld = load("accelGyroINSEKFData.mat")

ld = struct with fields:
    sampleRate: 100
    sensorData: [300×2 timetable]
    groundTruth: [300×1 timetable]
    initOrient: [1×1 quaternion]
```

Before fusing the sensor data, you need to set up the initial orientation for the filter state. First, observe the state information.

```
stateinfo(filter)

ans = struct with fields:
    Orientation: [1 2 3 4]
    AngularVelocity: [5 6 7]
    Accelerometer_Bias: [8 9 10]
    Gyroscope_Bias: [11 12 13]
```

Query the index of a state component directly by using the corresponding sensor.

```
stateinfo(filter,accSensor,"Bias")

ans = 1×3
     8     9    10
```

Set only the `Orientation` component of the state vector using the `stateparts` object function.

```

quatElements = compact(ld.initOrient); % Convert the quaternion object to a vector of four elements
stateparts(filter,"Orientation",quatElements); % Specify the Orientation state component
stateparts(filter,"Orientation") % Show the specified Orientation state component

```

```
ans = 1x4
```

```
    1.0000    -0.0003     0.0001     0.0002
```

Specify the covariance matrix corresponding to the orientation as a diagonal matrix.

```
statecovparts(filter,"Orientation",1e-2);
```

Fuse Sensor Data

You can fuse sensor data by recursively calling the `predict` and `fuse` object functions.

Preallocate variables for saving estimated results.

```

N = size(ld.sensorData,1);
estOrient = quaternion.zeros(N,1);
dt = seconds(diff(ld.sensorData.Properties.RowTimes));

```

Predict the filter state, fuse the sensor data, and obtain the estimates.

```

for ii = 1:N
    if ii ~= 1
        % Predict forward in time.
        predict(filter,dt(ii-1));
    end
    % Fuse accelerometer data.
    fuse(filter,accSensor,ld.sensorData.Accelerometer(ii,:),1);
    % Fuse gyroscope data.
    fuse(filter,gyroSensor,ld.sensorData.Gyroscope(ii,:),1);
    % Extract the orientation state estimate using the stateparts object function.
    estOrient(ii) = quaternion(stateparts(filter,"Orientation"));
end

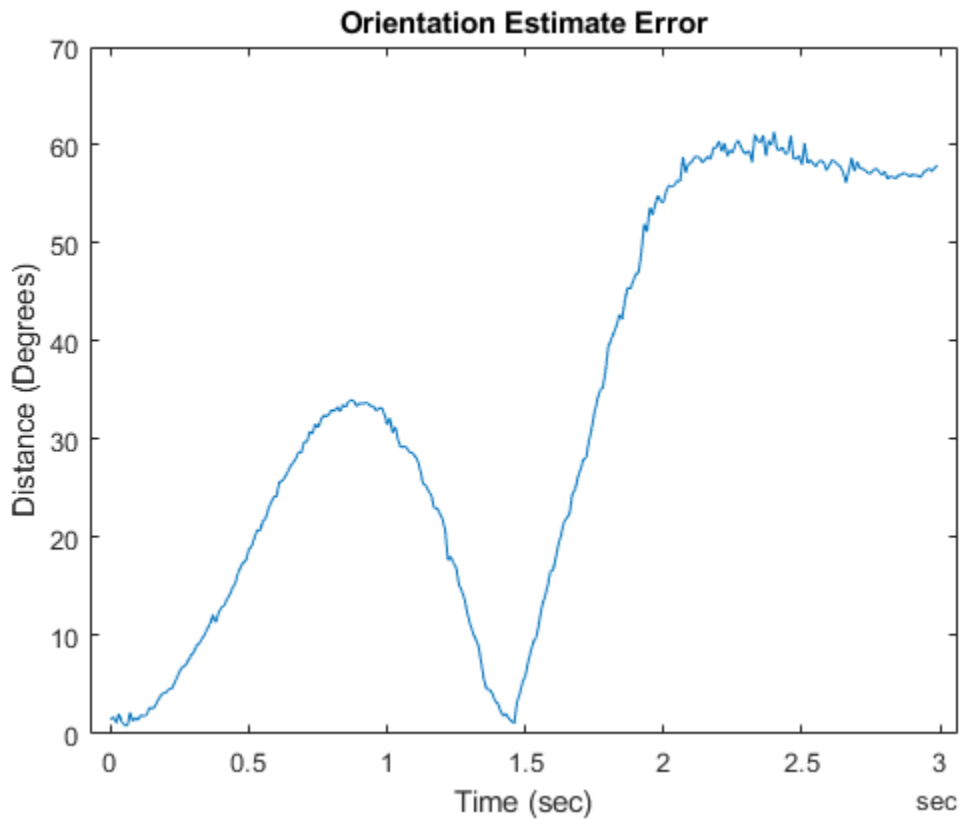
```

Visualize the estimate error, in quaternion distance, using the `dist` object function of the quaternion object.

```

figure
times = ld.groundTruth.Properties.RowTimes;
distance = rad2deg(dist(estOrient,ld.groundTruth.Orientation));
plot(times,distance)
xlabel("Time (sec)")
ylabel("Distance (Degrees)")
title("Orientation Estimate Error")

```



The results indicate that the estimate errors are large. You can also use the `estimateStates` object function to process the sensor data and obtain the same results.

Tune Filter

Given that the estimation results are not ideal, you can try to tune the filter parameters to reduce estimate errors.

Create a measurement noise structure and a `tunerconfig` object used to configure the tuning parameters.

```
mnoise = tunernoise(filter);
cfg = tunerconfig(filter,MaxIterations=10,ObjectiveLimit=1e-4);
```

Reinitialize the filter. Use the `tune` object function to tune the filter and obtain the tuned noise.

```
stateparts(filter,"Orientation",quatElements);
statecovparts(filter,"Orientation",1e-2);

tunedmn = tune(filter,mnoise,ld.sensorData,ld.groundTruth,cfg);
```

Iteration	Parameter	Metric
1	AdditiveProcessNoise(1)	0.3787
1	AdditiveProcessNoise(15)	0.3761
1	AdditiveProcessNoise(29)	0.3695
1	AdditiveProcessNoise(43)	0.3655
1	AdditiveProcessNoise(57)	0.3533

1	AdditiveProcessNoise(71)	0.3446
1	AdditiveProcessNoise(85)	0.3431
1	AdditiveProcessNoise(99)	0.3428
1	AdditiveProcessNoise(113)	0.3427
1	AdditiveProcessNoise(127)	0.3426
1	AdditiveProcessNoise(141)	0.3298
1	AdditiveProcessNoise(155)	0.3206
1	AdditiveProcessNoise(169)	0.3200
1	AccelerometerNoise	0.3199
1	GyroscopeNoise	0.3198
2	AdditiveProcessNoise(1)	0.3126
2	AdditiveProcessNoise(15)	0.3098
2	AdditiveProcessNoise(29)	0.3018
2	AdditiveProcessNoise(43)	0.2988
2	AdditiveProcessNoise(57)	0.2851
2	AdditiveProcessNoise(71)	0.2784
2	AdditiveProcessNoise(85)	0.2760
2	AdditiveProcessNoise(99)	0.2744
2	AdditiveProcessNoise(113)	0.2744
2	AdditiveProcessNoise(127)	0.2743
2	AdditiveProcessNoise(141)	0.2602
2	AdditiveProcessNoise(155)	0.2537
2	AdditiveProcessNoise(169)	0.2527
2	AccelerometerNoise	0.2524
2	GyroscopeNoise	0.2524
3	AdditiveProcessNoise(1)	0.2476
3	AdditiveProcessNoise(15)	0.2432
3	AdditiveProcessNoise(29)	0.2397
3	AdditiveProcessNoise(43)	0.2381
3	AdditiveProcessNoise(57)	0.2255
3	AdditiveProcessNoise(71)	0.2226
3	AdditiveProcessNoise(85)	0.2221
3	AdditiveProcessNoise(99)	0.2202
3	AdditiveProcessNoise(113)	0.2201
3	AdditiveProcessNoise(127)	0.2201
3	AdditiveProcessNoise(141)	0.2090
3	AdditiveProcessNoise(155)	0.2070
3	AdditiveProcessNoise(169)	0.2058
3	AccelerometerNoise	0.2052
3	GyroscopeNoise	0.2052
4	AdditiveProcessNoise(1)	0.2051
4	AdditiveProcessNoise(15)	0.2027
4	AdditiveProcessNoise(29)	0.2019
4	AdditiveProcessNoise(43)	0.2000
4	AdditiveProcessNoise(57)	0.1909
4	AdditiveProcessNoise(71)	0.1897
4	AdditiveProcessNoise(85)	0.1882
4	AdditiveProcessNoise(99)	0.1871
4	AdditiveProcessNoise(113)	0.1870
4	AdditiveProcessNoise(127)	0.1870
4	AdditiveProcessNoise(141)	0.1791
4	AdditiveProcessNoise(155)	0.1783
4	AdditiveProcessNoise(169)	0.1751
4	AccelerometerNoise	0.1748
4	GyroscopeNoise	0.1747
5	AdditiveProcessNoise(1)	0.1742
5	AdditiveProcessNoise(15)	0.1732
5	AdditiveProcessNoise(29)	0.1712

5	AdditiveProcessNoise(43)	0.1712
5	AdditiveProcessNoise(57)	0.1626
5	AdditiveProcessNoise(71)	0.1615
5	AdditiveProcessNoise(85)	0.1598
5	AdditiveProcessNoise(99)	0.1590
5	AdditiveProcessNoise(113)	0.1589
5	AdditiveProcessNoise(127)	0.1589
5	AdditiveProcessNoise(141)	0.1517
5	AdditiveProcessNoise(155)	0.1508
5	AdditiveProcessNoise(169)	0.1476
5	AccelerometerNoise	0.1473
5	GyroscopeNoise	0.1470
6	AdditiveProcessNoise(1)	0.1470
6	AdditiveProcessNoise(15)	0.1470
6	AdditiveProcessNoise(29)	0.1463
6	AdditiveProcessNoise(43)	0.1462
6	AdditiveProcessNoise(57)	0.1367
6	AdditiveProcessNoise(71)	0.1360
6	AdditiveProcessNoise(85)	0.1360
6	AdditiveProcessNoise(99)	0.1350
6	AdditiveProcessNoise(113)	0.1350
6	AdditiveProcessNoise(127)	0.1350
6	AdditiveProcessNoise(141)	0.1289
6	AdditiveProcessNoise(155)	0.1288
6	AdditiveProcessNoise(169)	0.1262
6	AccelerometerNoise	0.1253
6	GyroscopeNoise	0.1246
7	AdditiveProcessNoise(1)	0.1246
7	AdditiveProcessNoise(15)	0.1244
7	AdditiveProcessNoise(29)	0.1205
7	AdditiveProcessNoise(43)	0.1203
7	AdditiveProcessNoise(57)	0.1125
7	AdditiveProcessNoise(71)	0.1122
7	AdditiveProcessNoise(85)	0.1117
7	AdditiveProcessNoise(99)	0.1106
7	AdditiveProcessNoise(113)	0.1104
7	AdditiveProcessNoise(127)	0.1104
7	AdditiveProcessNoise(141)	0.1058
7	AdditiveProcessNoise(155)	0.1052
7	AdditiveProcessNoise(169)	0.1035
7	AccelerometerNoise	0.1024
7	GyroscopeNoise	0.1014
8	AdditiveProcessNoise(1)	0.1014
8	AdditiveProcessNoise(15)	0.1012
8	AdditiveProcessNoise(29)	0.1012
8	AdditiveProcessNoise(43)	0.1005
8	AdditiveProcessNoise(57)	0.0948
8	AdditiveProcessNoise(71)	0.0948
8	AdditiveProcessNoise(85)	0.0938
8	AdditiveProcessNoise(99)	0.0934
8	AdditiveProcessNoise(113)	0.0931
8	AdditiveProcessNoise(127)	0.0931
8	AdditiveProcessNoise(141)	0.0896
8	AdditiveProcessNoise(155)	0.0889
8	AdditiveProcessNoise(169)	0.0867
8	AccelerometerNoise	0.0859
8	GyroscopeNoise	0.0851
9	AdditiveProcessNoise(1)	0.0851

```

9      AdditiveProcessNoise(15)    0.0850
9      AdditiveProcessNoise(29)    0.0824
9      AdditiveProcessNoise(43)    0.0819
9      AdditiveProcessNoise(57)    0.0771
9      AdditiveProcessNoise(71)    0.0771
9      AdditiveProcessNoise(85)    0.0762
9      AdditiveProcessNoise(99)    0.0759
9      AdditiveProcessNoise(113)   0.0754
9      AdditiveProcessNoise(127)   0.0754
9      AdditiveProcessNoise(141)   0.0734
9      AdditiveProcessNoise(155)   0.0724
9      AdditiveProcessNoise(169)   0.0702
9      AccelerometerNoise          0.0697
9      GyroscopeNoise              0.0689
10     AdditiveProcessNoise(1)     0.0689
10     AdditiveProcessNoise(15)    0.0686
10     AdditiveProcessNoise(29)    0.0658
10     AdditiveProcessNoise(43)    0.0655
10     AdditiveProcessNoise(57)    0.0622
10     AdditiveProcessNoise(71)    0.0620
10     AdditiveProcessNoise(85)    0.0616
10     AdditiveProcessNoise(99)    0.0615
10     AdditiveProcessNoise(113)   0.0607
10     AdditiveProcessNoise(127)   0.0606
10     AdditiveProcessNoise(141)   0.0590
10     AdditiveProcessNoise(155)   0.0578
10     AdditiveProcessNoise(169)   0.0565
10     AccelerometerNoise          0.0562
10     GyroscopeNoise              0.0557
    
```

```
filter.AdditiveProcessNoise
```

```
ans = 13×13
```

```

0.5849    0    0    0    0    0    0    0    0
0    0.6484    0    0    0    0    0    0    0
0    0    0.5634    0    0    0    0    0    0
0    0    0    1.4271    0    0    0    0    0
0    0    0    0    4.3574    0    0    0    0
0    0    0    0    0    2.9527    0    0    0
0    0    0    0    0    0    1.3071    0    0
0    0    0    0    0    0    0    4.3574    0
0    0    0    0    0    0    0    0    2.2415
0    0    0    0    0    0    0    0    0
:
    
```

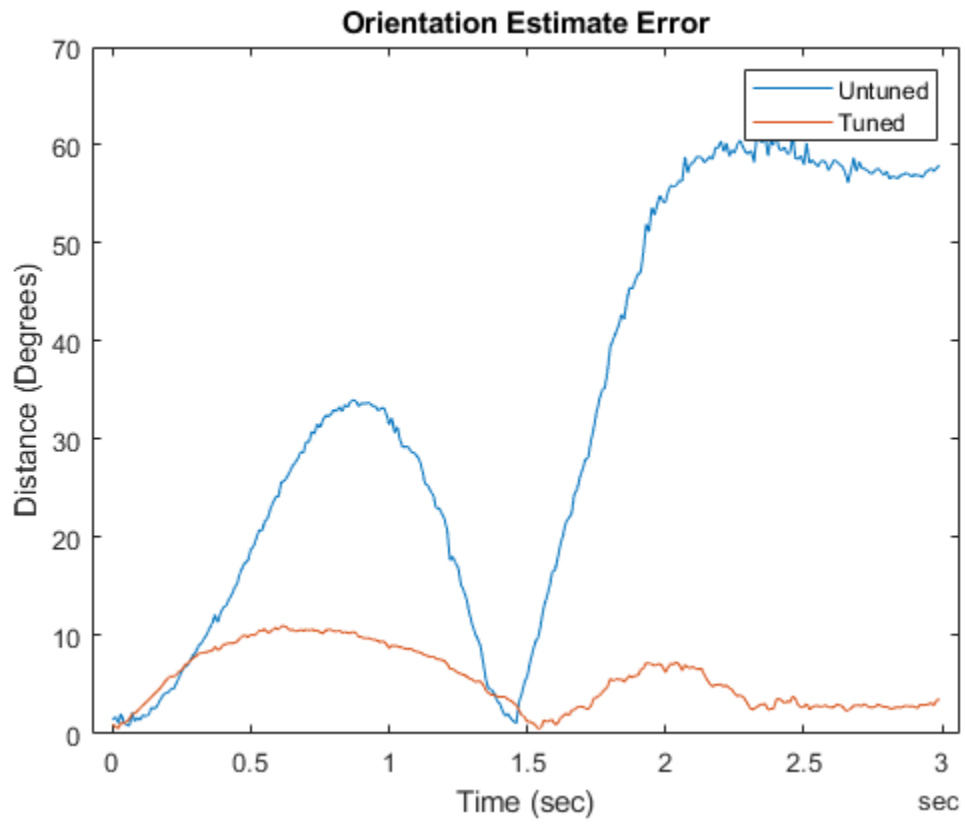
Batch-process the sensor data using the `estimateStates` object function. You can also recursively call the `predict` and `fuse` object functions to obtain the same results.

```
tunedEst = estimateStates(filter,ld.sensorData,tunedmn);
```

Compare the tuned and untuned estimates against the truth data. The results indicate that the tuning process greatly reduces the estimate errors.

```

distanceTuned = rad2deg(dist(tunedEst.Orientation,ld.groundTruth.Orientation));
hold on
plot(times,distanceTuned)
legend("Untuned","Tuned")
    
```



Multi-Object Tracking

- “Tracking and Tracking Filters” on page 4-2
- “Introduction to Estimation Filters” on page 4-9
- “Introduction to Out-of-Sequence Measurement Handling” on page 4-18
- “Motion Model, State, and Process Noise” on page 4-22
- “Linear Kalman Filters” on page 4-28
- “Extended Kalman Filters” on page 4-36
- “Introduction to Multiple Target Tracking” on page 4-44
- “Introduction to Assignment Methods in Tracking Systems” on page 4-49
- “Introduction to Track-To-Track Fusion” on page 4-56
- “Multiple Extended Object Tracking” on page 4-59
- “Configure Time Scope MATLAB Object” on page 4-61

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 6-168 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 6-269 and “Tracking Maneuvering Targets” on page 6-193 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 6-78 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, <code>initcaekf</code> 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 <code>help trackHistoryLogic</code> or <code>help trackScoreLogic</code> at the command line.

ConfirmationThreshold	<p>Specify the threshold for track confirmation. The threshold depends on the setting for TrackLogic</p> <ul style="list-style-type: none"> • '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	<p>Specify the threshold for track deletion. The threshold depends on the setting of TrackLogic</p> <ul style="list-style-type: none"> • '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	<p>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'.</p>
FalseAlarmRate	<p>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'.</p>
Beta	<p>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'.</p>

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 <code>TrackLogic</code> 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 <code>SensorIndex</code> value used in the <code>objectDetection</code> input to the <code>step</code> method. This property determines how many sets of <code>ObjectAttributes</code> each track can have.
HasDetectableTrackIDsInput	Set this property to <code>true</code> if you want to provide a list of detectable track IDs as input to the <code>step</code> 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 <code>true</code> if you want to provide an assignment cost matrix as input to the <code>step</code> 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 2-12). The input arguments are listed here.

trackerGNN Input

<code>tracker</code>	A <code>trackerGNN</code> object.
<code>detections</code>	Cell array of <code>objectDetection</code> objects (see “Detections” on page 2-12).
<code>time</code>	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.
<code>costmatrix</code>	Cost matrix for assigning detections to tracks. A real T -by- D matrix, where T is the number of tracks listed in the <code>allTracks</code> argument returned from the previous call to <code>step</code> . D is the number of detections that are input in the current call. A larger cost matrix entry means a lower likelihood of assignment.
<code>detectableTrackIDs</code>	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 <code>TrackID</code> 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	<p>The current state of the track logic.</p> <ul style="list-style-type: none"> • 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:

$$\begin{aligned}x_{k+1} &= f(x_k) \\ y_k &= h(x_k)\end{aligned}$$

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:

$$\begin{aligned}x_{k+1} &= f(x_k, w_k) \\ y_k &= h(x_k, v_k)\end{aligned}$$

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:

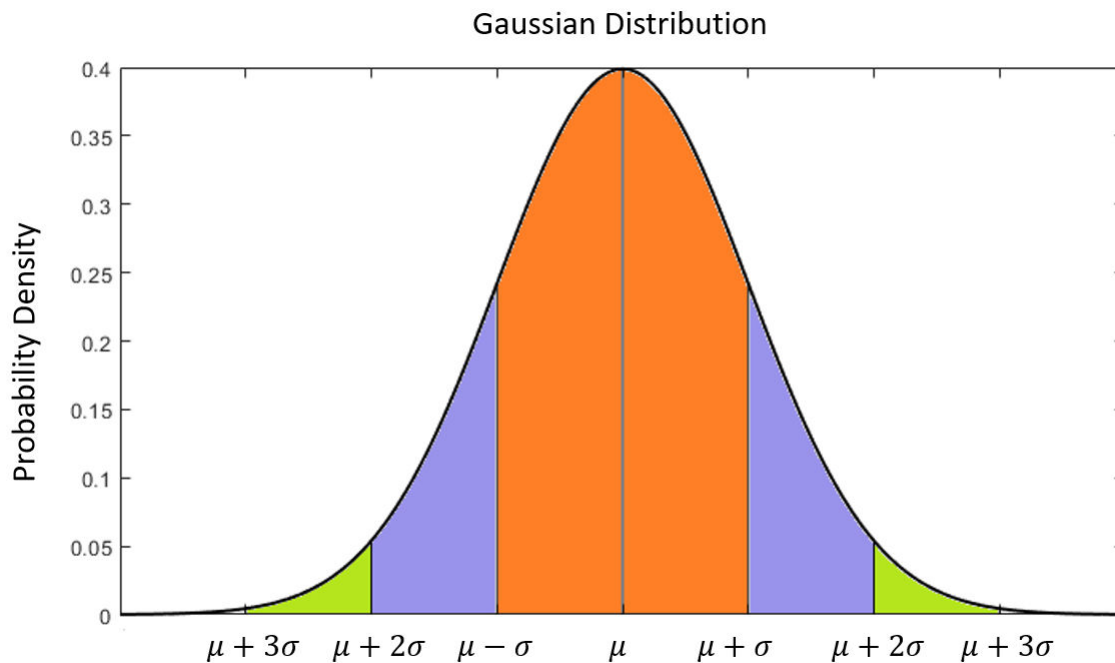
$$\mu = E[x]$$

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:

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

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.



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 ($k = 1, 2, 3, \dots, N$) 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.

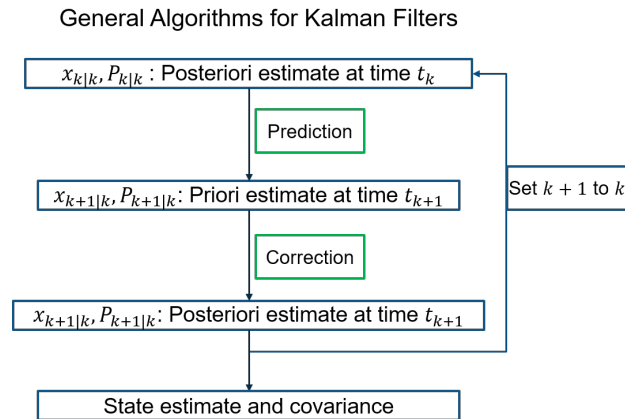
Correction

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.



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:

$$\begin{aligned}x_{k+1} &= A_k x_k + w_k \\ y_k &= H_k x_k + v_k\end{aligned}$$

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

$$\begin{aligned}w_k &\sim N(0, Q_k) \\ v_k &\sim N(0, R_k)\end{aligned}$$

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:

$$\begin{aligned}x_{k+1|k+1} &= x_{k+1|k} + K_k (y_k - H_k x_{k+1|k}) \\ P_{k+1|k+1} &= (I - K_k H_k) P_{k+1|k}\end{aligned}$$

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

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

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 4-28.

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 $h(x_k)$:

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

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

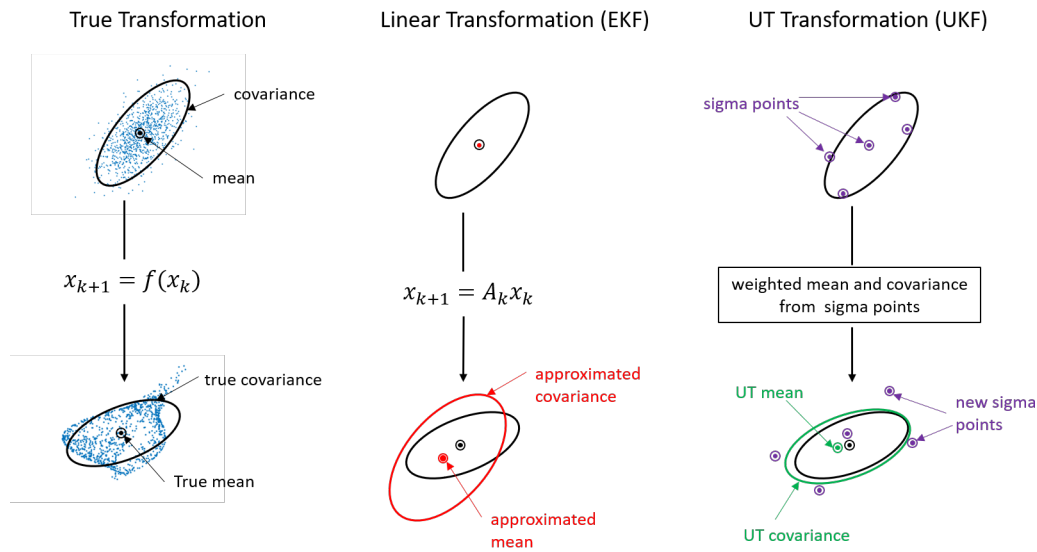
If the true dynamics of the estimation system are close to the linearized dynamics, then using this linear approximation does not yield significant errors for a short period of time. For this reason, an EKF can produce relatively accurate state estimates for a mildly nonlinear estimation system with 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:

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

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 6-269 for an example.

Interactive Multiple Model Filter

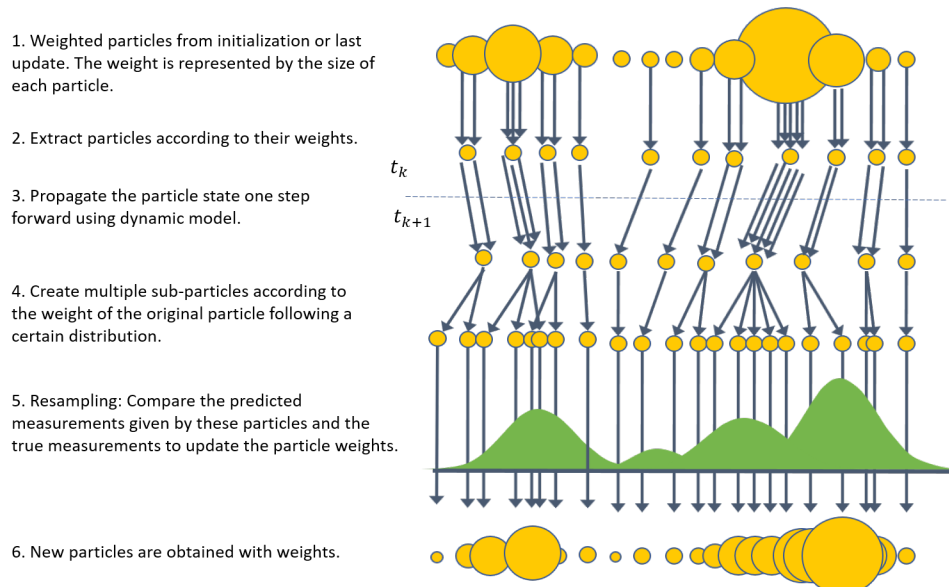
The interactive multiple model filter (`trackingIMM`) uses multiple Gaussian filters to track the position of a target. In highly maneuverable systems, the system dynamics can switch between multiple models (constant velocity, constant acceleration, and constant turn for example). Modelling the motion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

$$\begin{aligned}x_{k+1}^i &= f_i(x_k^i, w_k^i) \\ y_k^i &= h_i(x_k^i, v_k^i)\end{aligned}$$

where $i = 1, 2, \dots, M$, 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 6-193 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.



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>.

Introduction to Out-of-Sequence Measurement Handling

In this section...

“Introduction” on page 4-18

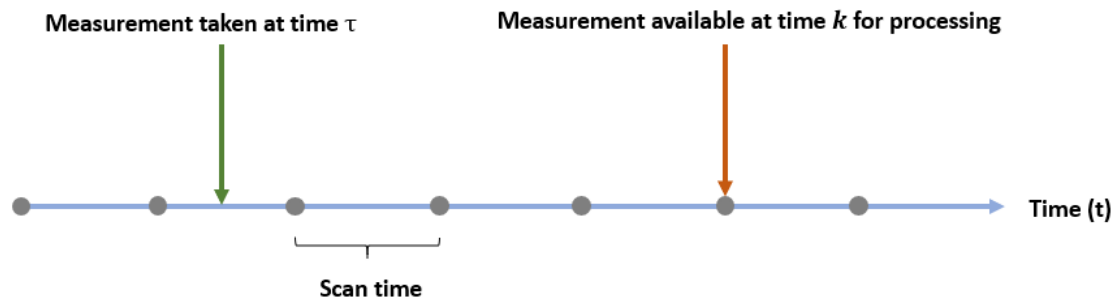
“Report an Error for OOSM” on page 4-18

“Neglect OOSM” on page 4-19

“Process OOSM Using Retrodiction” on page 4-19

Introduction

In a multi-sensor tracking system, time lag can result in the time at which a sensor generates a measurement differing from the time at which the tracker or filter receives the measurement for processing. If, before receiving a measurement, the filter or tracker is updated by another measurement with a later timestamp, the delayed measurement is known as a out-of-sequence measurement (OOSM). For example, assume a measurement is taken at time τ , but is not available until time step k , where $\tau < k$, then the measurement taken at time τ is an OOSM at the processing time k .



Regular sequential filters and object trackers do not have the ability to process OOSMs correctly, because they only maintain the current state estimates and covariances. At the filter level, neglecting OOSMs can increase the state estimate error covariances and thus increase the uncertainty of the state estimate. On the other hand, directly applying an OOSM without considering the measurement time can destabilize the state estimates. At the tracker level, incorrect or insufficient use of OOSMs can additionally result in false associations and false tracks, while valuable information, such as object classification, can be missed entirely. To simulate out-of-sequence detections, use `objectDetectionDelay`.

Sensor Fusion and Tracking Toolbox provides these options to handle OOSMs.

Report an Error for OOSM

In Sensor Fusion and Tracking Toolbox, as the default behavior, these multi-object trackers and tracker blocks report an error when encountering an OOSM:

- `trackerGNN`
- `trackerJPDA`

- `trackerTOMHT`
- Global Nearest Neighbor Multi Object Tracker
- Joint Probabilistic Data Association Multi Object Tracker
- Track-Oriented Multi-Hypothesis Tracker

Reporting an error for an OOSM stops the execution of a tracker. Use this option when you do not expect OOSMs, and consider them to be errors in your tracking system.

Neglect OOSM

You can specify for the tracker to neglect OOSMs by specifying the `OOSMHandling` property of these multi-object trackers to 'Neglect':

- `trackerGNN`
- `trackerJPDA`
- `trackerTOMHT`

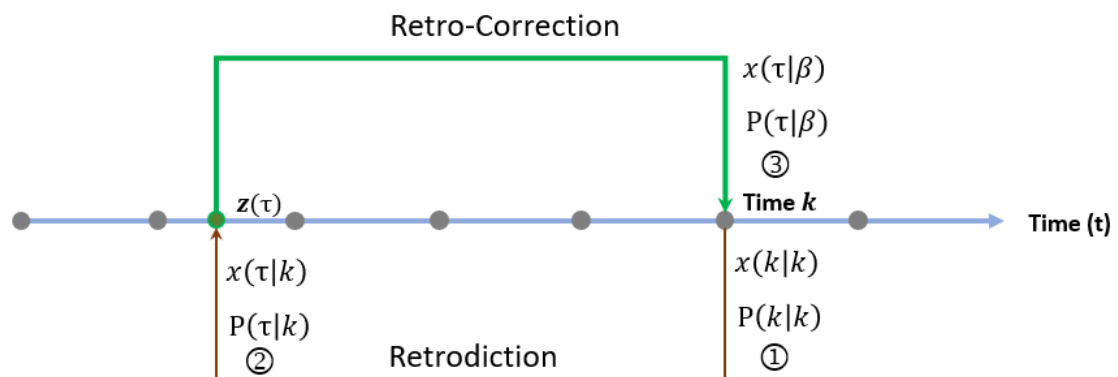
With this setting, the tracker neglects any OOSM it encounters and continues to run. Similarly, you can set the **Out-of-sequence measurement handling parameter** in these tracker blocks to `Neglect`:

- Global Nearest Neighbor Multi Object Tracker
- Joint Probabilistic Data Association Multi Object Tracker
- Track-Oriented Multi-Hypothesis Tracker

In most cases, neglecting an OOSM is a good choice when the tracking system is updated with high frequency, such that the loss of a measurement is not very costly. Also, this option does not require extra processing operations and storage. For tracking systems with low update rates, however, this option can affect the state estimate quality, and may result in incorrect detection-track association.

Process OOSM Using Retrodiction

Retrodiction is an effective and efficient approach for handling OOSMs. For details on the algorithm, see "Retrodiction and Retro-Correction", [1], and [2]. This diagram illustrates the retrodiction and retro-correction steps.



Assume that, at time k , the filter obtains the posteriori state estimate $x(k|k)$ and state estimate error covariance $P(k|k)$, based on all the measurements that have arrived at the filter up to that time. Also, assume an OOSM taken at time τ , where $\tau < k$, is now available at time k . To process the OOSM using retrodiction, the filter first retrodicts (predicts backward in time) the state and state covariance to obtain the retrodicted state $x(\tau|k)$ and state covariance $P(\tau|k)$. Then, using the OOSM $z(\tau)$ and its covariance $R(\tau)$, the filter corrects the state estimate and obtains the retro-corrected state $x(k|\tau)$ and state covariance $P(k|\tau)$ for the current time step k , including the OOSM. Note that, to successfully apply the algorithm, the filter needs to store the state estimate covariance matrices from timestamps prior to the OOSM time τ to time k .

Compared with a linear or extended Kalman filter, an interactive multiple model (IMM) filter (`trackingIMM`) needs to maintain the probability of each member-filter in the retrodiction steps. See [2] for more details.

Retrodiction in Tracking Filters

Sensor Fusion and Tracking Toolbox currently provides retrodiction capability for the `trackingKF`, `trackingEKF`, and `trackingIMM` filter objects. To enable retrodiction in a filter, specify its `MaxNumOOSMSteps` property as a positive integer, which represents the number of previous state covariances preserved in the filter. Then, use the `retrodict` object function to retrodict the state to the OOSM time, and use the `retroCorrect` or `retroCorrectJPDA` object function to update the current state using the OOSM.

Retrodiction in Multi-Object Trackers

Sensor Fusion and Tracking Toolbox currently provides retrodiction capability for these tracker objects and blocks:

- `trackerGNN` System object™
- `trackerJPDA` System object
- Global Nearest Neighbor Multi Object Tracker block
- Joint Probabilistic Data Association Multi Object Tracker block

To enable retrodiction in `trackerGNN` or `trackerJPDA`, specify its `OOSMHandling` property as 'Retrodiction', and specify the `MaxNumOOSMSteps` property as a positive integer. Similarly, you can enable retrodiction in the Global Nearest Neighbor Multi Object Tracker block or the Joint Probabilistic Data Association Multi Object Tracker block via the **Out-of-sequence measurements handling** and the **Maximum number of OOSM steps** parameters. In the tracker object or the tracker block, you must specify a filter initialization function that returns a `trackingKF`, `trackingEKF`, or `trackingIMM` object.

The tracker follows these steps to handle the OOSM:

- If the OOSM timestamp is beyond the track history maintained by the tracker, the tracker neglects the OOSM.
- If the OOSM timestamp is within the track history maintained by the tracker, the tracker first retrodicts all the existing tracks to the time of the OOSM. Then, the tracker tries to associate the OOSM to any of the retrodicted tracks.
 - If the tracker successfully associates the OOSM to a retrodicted track, the tracker updates the retrodicted tracks using the OOSM by applying the retro-correction algorithm to obtain current, corrected tracks.

- If the tracker cannot associate the OOSM to any retrodicted track, the tracker creates a new track based on the OOSM and predicts the track to the current time.

See Also

objectDetectionDelay | trackingKF | trackingEKF | retrodict | retroCorrect | trackerGNN | trackerJPDA | trackerTOMHT | Global Nearest Neighbor Multi Object Tracker | Joint Probabilistic Data Association Multi Object Tracker | Track-Oriented Multi-Hypothesis Tracker

References

- [1] Bar-Shalom, Y., Huimin Chen, and M. Mallick. "One-Step Solution for the Multistep out-of-Sequence-Measurement Problem in Tracking." *IEEE Transactions on Aerospace and Electronic Systems* 40, no. 1 (January 2004): 27-37.
- [2] Bar-shalom, Y. and Huimin Chen. "IMM Estimator with Out-of-Sequence Measurements." *IEEE Transactions on Aerospace and Electronic Systems*, vol. 41, no. 1, Jan. 2005, pp. 90-98.

Motion Model, State, and Process Noise

In this section...

“Introduction” on page 4-22

“Constant Velocity Model” on page 4-22

“Constant Acceleration Model” on page 4-23

“Constant Turn Rate Model” on page 4-24

“Singer Model” on page 4-25

“Summary” on page 4-26

Introduction

A motion model describes how a target or object moves with respect to time and is usually expressed as an equation of motion governing the transition of target states, such as position and velocity. You can use a motion model to simulate the ideal motion of a target. In estimation filters, you use a motion model to predict the estimated state from one time step to the next.

In real applications, modeling the exact motion of targets is often impossible, because the target motion can be perturbed by unknown external effects. Process noise is commonly used to account for these motion uncertainties in an estimation system.

There are many conventions for defining motion models, states, and process noise in the literature. However, this topic focuses on the conventions used in the Sensor Fusion and Tracking Toolbox, which assume the process noise is constant between discrete time steps.

Constant Velocity Model

The `constvel` function models constant velocity motion, which assumes that the velocity vector is constant. In many cases, even though the target velocity vector is not exactly constant, you can use this model if the velocity vector does not change consistently. In fact, because of the simplicity of the constant velocity model, you can often try to use the constant velocity model before using other motion models.

The `constvel` function defines the state as $[x, v_x, y, v_y, z, v_z]$, with the variables specified in this order.

State Component	Definition
x	x -coordinate of the target, specified as a scalar.
v_x	x -direction velocity of the target, specified as a scalar.
y	y -coordinate of the target, specified as a scalar.
v_y	y -direction velocity of the target, specified as a scalar.
z	z -coordinate of the target, specified as a scalar.
v_z	z -direction velocity of the target, specified as a scalar.

Since the motion model assumes decoupled x -, y -, and z -motion, you can use the `constvel` function to model 1-D, 2-D, or 3-D constant velocity motion.

With process noise, the `constvel` model for the 1-D x -motion is:

$$\begin{bmatrix} x(k+1) \\ v_x(k+1) \end{bmatrix} = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x(k) \\ v_x(k) \end{bmatrix} + \begin{bmatrix} \frac{1}{2}T^2 \\ T \end{bmatrix} w_x(k)$$

where T is the time step size of the discrete model, k is the time step index, and $w_x(k)$ is the process noise in the x -direction at the k -th time step. From the equation, the function treats the process noise as an acceleration disturbance. If you do not specify the process noise, the function assumes the process noise to be zero.

To model 3-D motion, the function repeats the equation for the y - and z -directions as:

$$\begin{bmatrix} x(k+1) \\ v_x(k+1) \\ y(k+1) \\ v_y(k+1) \\ z(k+1) \\ v_z(k+1) \end{bmatrix} = \begin{bmatrix} 1 & T & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & T & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x(k) \\ v_x(k) \\ y(k) \\ v_y(k) \\ z(k) \\ v_z(k) \end{bmatrix} + \begin{bmatrix} \frac{1}{2}w_x(k)T^2 \\ w_x(k)T \\ \frac{1}{2}w_y(k)T^2 \\ w_y(k)T \\ \frac{1}{2}w_z(k)T^2 \\ w_z(k)T \end{bmatrix}$$

where w_y and w_z represent the process noise for the y - and z -directions, respectively.

Constant Acceleration Model

The `constacc` function models constant velocity motion, which assumes that the acceleration vector is constant. In many cases, even though the target acceleration vector is not exactly constant, you can use the constant acceleration model if the target acceleration vector does not change consistently.

The `constvel` function defines the state as $[x, v_x, a_x, y, v_y, a_y, z, v_z, a_z]$, with the variables specified in this order. .

State Component	Definition
x	x -coordinate of the target, specified as a scalar.
v_x	x -direction velocity of the target, specified as a scalar.
a_x	x -direction acceleration of the target, specified as a scalar.
y	y -coordinate of the target, specified as a scalar.
v_y	y -direction velocity of the target, specified as a scalar.
a_y	y -direction acceleration of the target, specified as a scalar.

State Component	Definition
z	z -coordinate of the target, specified as a scalar.
v_z	z -direction velocity of the target, specified as a scalar.
a_z	z -direction acceleration of the target, specified as a scalar.

Since the motion model assumes decoupled x -, y -, and z -motion, you can use the `constacc` function to model 1-D, 2-D, or 3-D constant acceleration motion.

With process noise, the constant acceleration model for the 1-D x -motion is:

$$\begin{bmatrix} x(k+1) \\ v_x(k+1) \\ a_x(k+1) \end{bmatrix} = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x(k) \\ v_x(k) \\ a_x(k) \end{bmatrix} + \begin{bmatrix} \frac{1}{2}T^2 \\ T \\ 1 \end{bmatrix} w_x(k)$$

where T is the time step size of the discrete model, k is the time step index, and w_x is the process noise in the x -direction. From the equation, the `constacc` function models the process noise as an acceleration disturbance. If you do not specify the process noise, the function assumes the process noise is zero.

To model 2-D or 3-D motion, simply repeat the equations for the y - and z -directions.

Constant Turn Rate Model

The `constturn` function models the horizontal constant turn motion, in which the turn rate and turn radius of the target trajectory are constant. In many cases, even though the target turn rate and turn radius are not exactly constant, you can use the constant turn rate model if the turn rate and turn radius do not change consistently.

The `constturn` function defines the state as $[x, v_x, y, v_y, \omega, z, v_z]$, with the variables specified in this order.

State Component	Definition
x	x -coordinate of the target, specified as a scalar.
v_x	x -direction velocity of the target, specified as a scalar.
y	y -coordinate of the target, specified as a scalar.
v_y	y -direction velocity of the target, specified as a scalar.
ω	Turn rate in the x - y plane, specified as a scalar.
z	z -coordinate of the target, specified as a scalar.
v_z	z -direction velocity of the target, specified as a scalar.

In the model, the x - and y -motion depends on the turnrate, ω . The equation for the 2-D motion is:

$$\begin{bmatrix} x(k+1) \\ v_x(k+1) \\ y(k+1) \\ v_y(k+1) \\ \omega(k+1) \end{bmatrix} = \begin{bmatrix} 1 & TS & 0 & -TC & 0 \\ 0 & \cos\omega T & 0 & -\sin\omega T & 0 \\ 0 & TC & 1 & TS & 0 \\ 0 & \sin\omega T & 0 & \cos\omega T & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x(k) \\ v_x(k) \\ y(k) \\ v_y(k) \\ \omega(k) \end{bmatrix} + \begin{bmatrix} \frac{1}{2}w_x(k)T^2 \\ w_x(k)T \\ \frac{1}{2}w_y(k)T^2 \\ w_y(k)T \\ w_\omega(k)T \end{bmatrix}$$

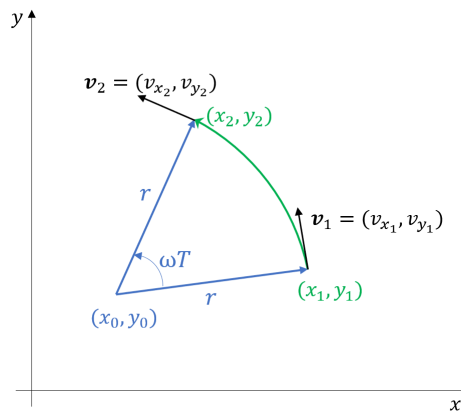
where T is the time step size of the discrete model, k is the time step index, and w_x , w_y , and w_ω represent the process noise for the x -motion, the y -motion, and the turn rate, respectively. The two variables TS and TC are:

$$TS = \frac{\sin(\omega(k)T)}{\omega(k)}$$

$$TC = \frac{[1 - \cos(\omega(k)T)]}{\omega(k)}$$

This figure shows the target motion from one time step to the next step using the 2-D constant turn rate model without process noise. During one time step, the target moves from (x_1, y_1) to (x_2, y_2) following an arc of radius r and angle ωT . The velocities \mathbf{v}_1 and \mathbf{v}_2 are perpendicular to their respective radial directions. The magnitude of the velocity is constant along the entire arc as:

$$v = \omega r = \sqrt{v_{x_i}^2 + v_{y_i}^2}, \quad i = 1, 2.$$



From the equation of motion, the `constturn` function treats the process noise for the x - and y -motion as acceleration disturbances and treats the process noise for the turn rate as the disturbance of the turn rate.

To model 3-D motion, simply add the z -motion equation of the constant velocity model since the z -motion is decoupled from the x - y motion in the constant turn model.

Singer Model

The `singer` function models the target motion with a state definition similar to that of the constant acceleration motion. However, the Singer model assumes that the target acceleration decays over time. Thus, the Singer model allows you to model targets that change their motion between different constant velocity sections by using limited-time linear acceleration or turn.

Like the `constacc` function, the `constvel` function defines the state as $[x, v_x, a_x, y, v_y, a_y, z, v_z, a_z]$, with the variables specified in this order.

State Component	Definition
x	x -coordinate of the target, specified as a scalar.
v_x	x -direction velocity of the target, specified as a scalar.
a_x	x -direction acceleration of the target, specified as a scalar.
y	y -coordinate of the target, specified as a scalar.
v_y	y -direction velocity of the target, specified as a scalar.
a_y	y -direction acceleration of the target, specified as a scalar.
z	z -coordinate of the target, specified as a scalar.
v_z	z -direction velocity of the target, specified as a scalar.
a_z	z -direction acceleration of the target, specified as a scalar.

Since the motion model assumes decoupled x -, y -, and z -motion, you can use the `singer` function to model 1-D, 2-D, or 3-D Singer motion.

Without the process noise, the Singer model for the 1-D x -motion can be expressed as:

$$\begin{bmatrix} x(k+1) \\ v_x(k+1) \\ a_x(k+1) \end{bmatrix} = \begin{bmatrix} 1 & T & (\alpha T - 1 - e^{-\alpha T})/\alpha^2 \\ 0 & 1 & (1 - e^{-\alpha T})/\alpha \\ 0 & 0 & e^{-\alpha T} \end{bmatrix} \begin{bmatrix} x(k) \\ v_x(k) \\ a_x(k) \end{bmatrix}$$

where T is the time step size of the discrete model, k is the time step index, and $\alpha = 1/\tau$ is the reciprocal of the target maneuver time constant τ . If $\tau \rightarrow +\infty$, the Singer model reduces to the constant acceleration model.

The process noise in the singer model follows a Markov process and is not white noise. You can use the `singerProcessNoise` function to generate the process noise and add to the state. See [1] or [2] for more details.

Summary

This table summarizes the motion models, model states, and their application assumptions. You can always start by using a simpler model, such as the constant velocity model, and review the estimation performance before applying more sophisticated models.

Function	Motion Model	1-D State	2-D State	3-D State	Application Assumption
constvel	Constant velocity	$[x, v_x]$	$[x, v_x, y, v_y]$	$[x, v_x, y, v_y, z, v_z]$	The velocity vector of the target is roughly constant.
constacc	Constant acceleration	$[x, v_x, a_x]$	$[x, v_x, a_x, y, v_y, a_y]$	$[x, v_x, a_x, y, v_y, a_y, z, v_z, a_z]$	The acceleration vector of the target is roughly constant.
constturn	Constant turn rate	N/A	$[x, v_x, y, v_y, \omega]$	$[x, v_x, y, v_y, \omega, z, v_z]$	The turn rate and turn radius of the target are roughly constant.
singer	Singer	$[x, v_x, a_x]$	$[x, v_x, a_x, y, v_y, a_y]$	$[x, v_x, a_x, y, v_y, a_y, z, v_z, a_z]$	The target can often perform maneuvers.

References

- [1] Singer, Robert. "Estimating optimal tracking filter performance for manned maneuvering targets." *IEEE Transactions on Aerospace and Electronic Systems* 4 (1970): 473-483.
- [2] Blackman, Samuel S., and Robert Popoli. *Design and Analysis of Modern Tracking Systems*. Artech House Radar Library, Boston, 1999.
- [3] Li, X. Rong, and Vesselin P. Jilkov. "Survey of maneuvering target tracking: dynamic models." *Signal and Data Processing of Small Targets* 2000, vol. 4048, pp. 212-235. International Society for Optics and Photonics, 2000.

Linear Kalman Filters

In this section...

“Motion Model” on page 4-28

“Measurement Models” on page 4-29

“Filter Loop” on page 4-29

“Built-In Motion Models in trackingKF” on page 4-31

“Example: Estimate 2-D Target States Using trackingKF” on page 4-32

Kalman filters track an object using a sequence of detections or measurements to estimate the state of the object based on the motion model of the object. In a motion model, state is a collection of quantities that represent the status of an object, such as its position, velocity, and acceleration. An object motion model is defined by the evolution of the object state.

The linear Kalman filter (`trackingKF`) is an optimal, recursive algorithm for estimating the state of an object if the estimation system is linear and Gaussian. An estimation system is linear if both the motion model and measurement model are linear. The filter works by recursively predicting the object state using the motion model and correcting the state using measurements.

Motion Model

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

Consider an object moving in the x -direction at a constant acceleration. You can write the equation of motion, using Newtonian equations, as:

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

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

Furthermore, if you define the state as:

$$x_1 = x$$

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

you can write the equation of motion in state-space form as:

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

In most cases, a motion model does not fully model the motion of an object, and you need to include the process noise to compensate the uncertainty in the motion model. For the constant velocity model, you can add process noise as an acceleration term.

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

Here, v_k is the unknown noise perturbation of the acceleration. For the filter to be optimal, you must assume the process noise is zero-mean, white Gaussian 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 in this equation is the state transition matrix. For independent x - and y -motions, this matrix is block diagonal.

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

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

where 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. For details on how to obtain the discretized process noise from a continuous system, See [1].

You can generalize the state equation to:

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

where A_k is the state transition matrix and B_k is the control matrix. The control matrix accounts for any known forces acting on the object. v_k represents discretized process noise, following a Gaussian distribution of mean 0 and covariance Q_k . G_k is the process noise gain matrix.

Measurement Models

Measurements are what you observe or measure in a system. Measurements depend on the state vector, but are usually not 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. A linear Kalman filter assumes the measurements are a linear function of the state vector. To apply nonlinear measurement models, you can choose to use an extended Kalman filter (trackingEKF) or an unscented Kalman filter (trackingUKF).

You can represent a linear measurement as:

$$z_k = H_k x_k + w_k$$

Here, H_k is the measurement matrix and w_k represents measurement noise at the current time step. For an optimal filter, the measurement noise must be zero-mean, Gaussian white noise. Assume the covariance matrix of the measurement noise is R_k .

Filter Loop

The filter starts with best estimates of the state $x_{0|0}$ and the state covariance $P_{0|0}$. The filter performs these steps in a recursive loop.

- 1 Propagate the state to the next step using the motion equation:

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

Propagate the covariance matrix as well:

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

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

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

- 2 Use the difference between the actual measurement and the predicted measurement to correct the state at the $k+1$ step. To correct the state, the filter must compute the Kalman gain. First, the filter computes the measurement prediction covariance (innovation) as:

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

Then, the filter computes the Kalman gain as:

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

- 3 The filter corrects the predicted estimate by using the measurement. The estimate, after correction using the measurement z_{k+1} , is

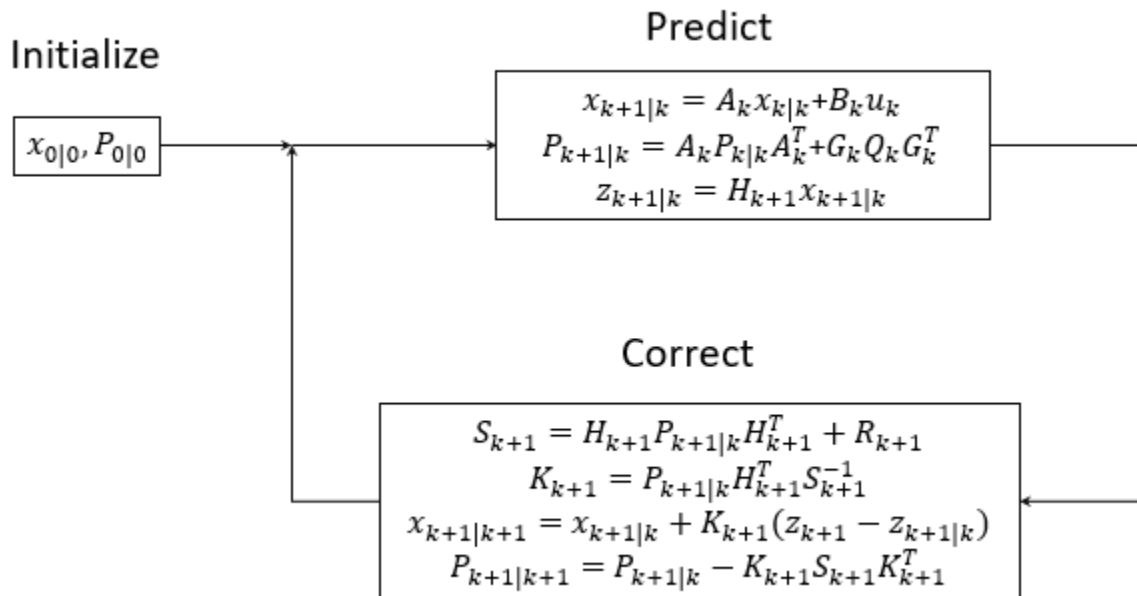
$$x_{k+1|k+1} = x_{k+1|k} + K_{k+1}(z_{k+1} - z_{k+1|k})$$

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 including the measurement.

The filter corrects the state covariance matrix as:

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

This figure summarizes the Kalman loop operations. Once initialized, a Kalman filter loops between prediction and correction until reaching the end of the simulation.



Built-In Motion Models in trackingKF

When you only need to use the standard 1-D, 2-D, or 3-D constant velocity or constant acceleration motion models, you can specify the `MotionModel` property of `trackingKF` as one of these:

- "1D Constant Velocity"
- "1D Constant Acceleration"
- "2D Constant Velocity"
- "2D Constant Acceleration"
- "3D Constant Velocity"
- "3D Constant Acceleration"

To customize your own motion model, specify the `MotionModel` property as "Custom", and then specify the state transition matrix in the `StateTransitionModel` property of the filter.

For the 3-D constant velocity model, the state equation is:

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

For the 3-D constant acceleration model, the state equation is:

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

Example: Estimate 2-D Target States Using trackingKF

Initialize Estimation Model

Specify an initial position and velocity for a target that you assume moving in 2-D. The simulation lasts 20 seconds with a sample time of 0.2 seconds.

```
rng(2021); % For repeatable results
dt = 0.2; % seconds
simTime = 20; % seconds
tspan = 0:dt:simTime;
trueInitialState = [30; 2; 40; 2]; % [x;vx;y;vy]
processNoise = diag([0; 1; 0; 1]); % Process noise matrix
```

Create a measurement noise covariance matrix, assuming that the target measurements consist of its position states.

```
measureNoise = diag([4 4]); % Measurement noise matrix
```

The matrix specifies a standard deviation of 2 meters in both the x- and y-directions.

Preallocate variables in which to save estimation results.

```
numSteps = length(tspan);
trueStates = NaN(4,numSteps);
trueStates(:,1) = trueInitialState;
estimateStates = NaN(size(trueStates));
```

Obtain True States and Measurements

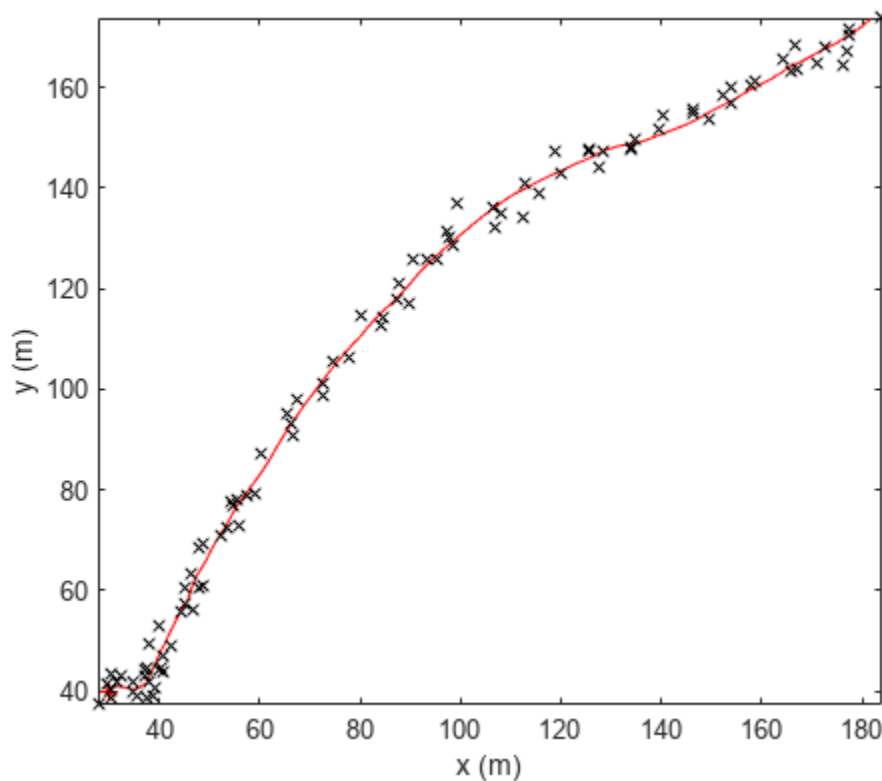
Propagate the constant velocity model, and generate the measurements with noise.

```
F = [1 dt 0 0;
     0 1 0 0;
     0 0 1 dt;
     0 0 0 1];
H = [1 0 0 0;
     0 0 1 0];
for i = 2:length(tspan)
    trueStates(:,i) = F*trueStates(:,i-1) + sqrt(processNoise)*randn(4,1);
```

```
end
measurements = H*trueStates + sqrt(measureNoise)*randn(2,numSteps);
```

Plot the true trajectory and the measurements.

```
figure
plot(trueStates(1,1),trueStates(3,1),"r*",DisplayName="True Initial")
hold on
plot(trueStates(1,:),trueStates(3,:),"r",DisplayName="Truth")
plot(measurements(1,:),measurements(2,:),"kx",DisplayName="Measurements")
xlabel("x (m)")
ylabel("y (m)")
axis image
```



Initialize Linear Kalman Filter

Initialize the filter with a state of $[40; 0; 160; 0]$, which is far from the true initial state. Normally, you can use the initial measurement to construct an initial state as $[\text{measurements}(1,1); 0; \text{measurements}(2,1); 0]$. Here, you use an erroneous initial state, which enables you to test if the filter can quickly converge on the truth.

```
filter = trackingKF(MotionModel="2D Constant Velocity",State=[40; 0; 160;0], ...
    MeasurementModel=H,MeasurementNoise=measureNoise)
```

```
filter =
    trackingKF with properties:
        State: [4x1 double]
```

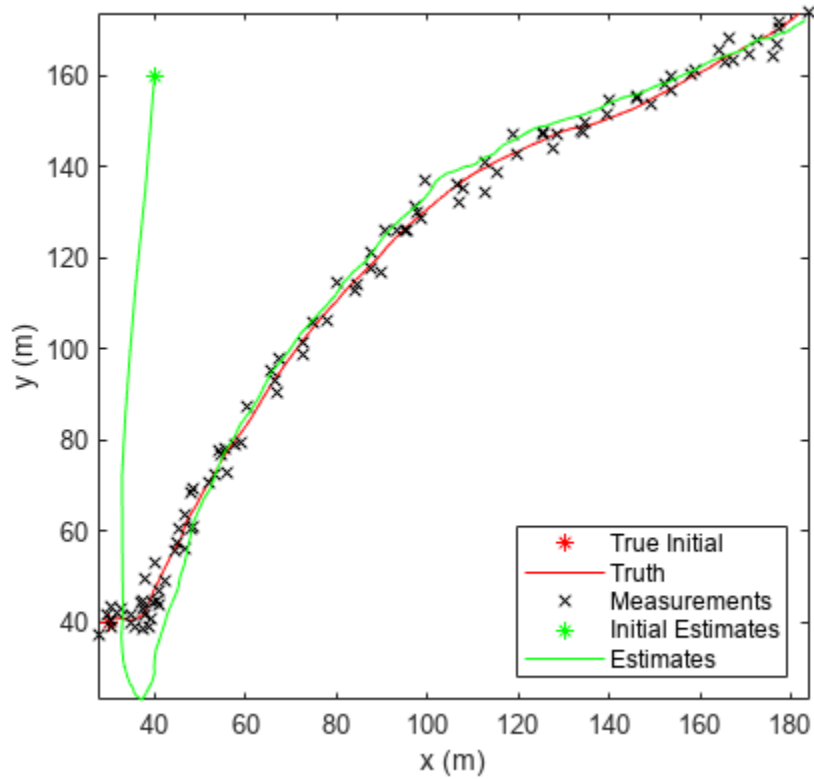
```
StateCovariance: [4x4 double]
    MotionModel: '2D Constant Velocity'
    ProcessNoise: [2x2 double]
MeasurementModel: [2x4 double]
MeasurementNoise: [2x2 double]
    MaxNumOOSMSteps: 0
    EnableSmoothing: 0
```

```
estimateStates(:,1) = filter.State;
```

Run Linear Kalman Filter and Show Results

Run the filter by recursively calling the `predict` and `correct` object functions. From the results, the estimates converge on the truth quickly. In fact, the linear Kalman filter has an exponential convergence speed.

```
for i=2:length(tspan)
    predict(filter,dt)
    estimateStates(:,i) = correct(filter,measurements(:,i));
end
plot(estimateStates(1,1),estimateStates(3,1),"g*",DisplayName="Initial Estimates")
plot(estimateStates(1,:),estimateStates(3,:),"g",DisplayName="Estimates")
legend(Location="southeast")
```



See Also

trackingKF | trackingEKF | trackingUKF | "Extended Kalman Filters" on page 4-36 | "Introduction to Estimation Filters" on page 4-9

References

- [1] Li, X. Rong, and Vesselin P. Jilkov. "Survey of Maneuvering Target Tracking: Dynamic Models". Edited by Oliver E. Drummond, 2000, pp. 212-35. DOI.org (Crossref), <https://doi.org/10.1117/12.391979>.

Extended Kalman Filters

In this section...

“State Update Model” on page 4-36

“Measurement Model” on page 4-37

“Extended Kalman Filter Loop” on page 4-37

“Predefined Extended Kalman Filter Functions” on page 4-38

“Example: Estimate 2-D Target States with Angle and Range Measurements Using `trackingEKF`” on page 4-39

When you use a filter to track objects, you use a sequence of detections or measurements to estimate the state of an object based on the motion model of the object. In a motion model, state is a collection of quantities that represent the status of an object, such as its position, velocity, and acceleration. Use an extended Kalman filter (`trackingEKF`) when object motion follows a nonlinear state equation or when the measurements are nonlinear functions of the state. For example, consider using an extended Kalman filter when the measurements of the object are expressed in spherical coordinates, such as azimuth, elevation, and range, but the states of the target are expressed in Cartesian coordinates.

The formulation of an extended Kalman is based on the linearization of the state equation and measurement equation. Linearization enables you to propagate the state and state covariance in an approximately linear format, and requires Jacobians of the state equation and measurement equation.

Note If your estimate system is linear, you can use the linear Kalman filter (`trackingKF`) or the extended Kalman filter (`trackingEKF`) to estimate the target state. If your system is nonlinear, you should use a nonlinear filter, such as the extended Kalman filter or the unscented Kalman filter (`trackingUKF`).

State Update Model

Assume a closed-form expression for the predicted state as a function of the previous state x_k , controls u_k , noise w_k , and time t .

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

The Jacobian of the predicted state with respect to the previous state is obtained by partial derivatives as:

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

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

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

These functions take simpler forms when the noise is additive in the state update equation:

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

In this case, $F^{(w)}$ is an identity matrix.

You can specify the state Jacobian matrix using the `StateTransitionJacobianFcn` property of the `trackingEKF` object. If you do not specify this property, the object computes Jacobians using numeric differencing, which is slightly less accurate and can increase the computation time.

Measurement Model

In an extended Kalman filter, the measurement can also 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 is additive in the measurement equation:

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

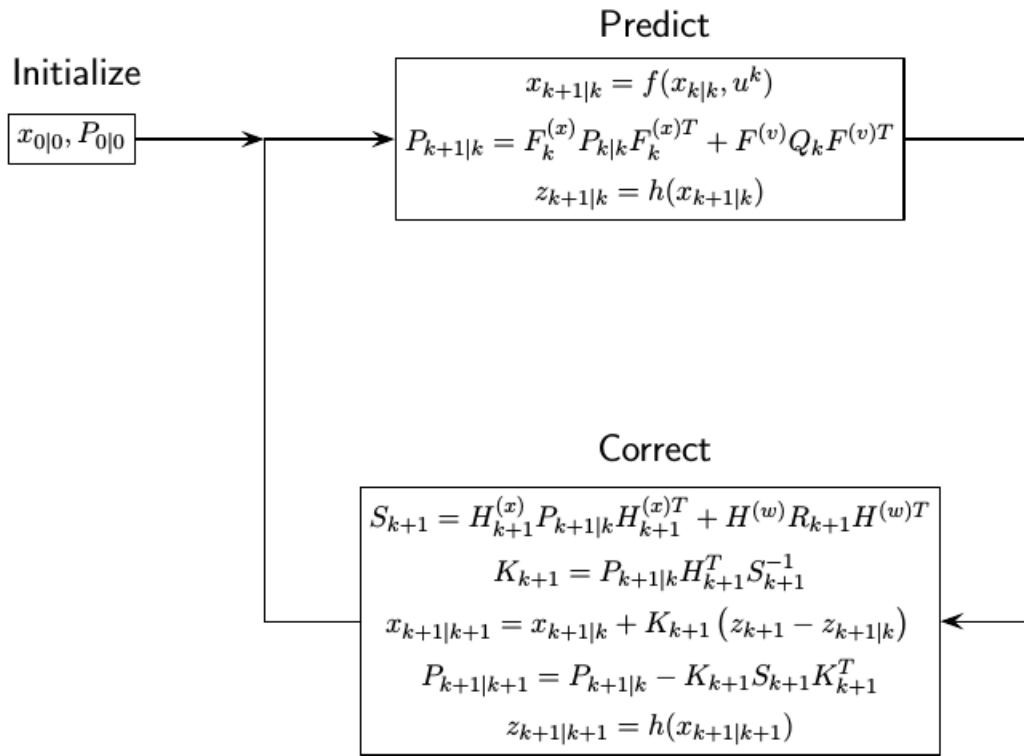
In this case, $H^{(v)}$ is an identity matrix.

In `trackingEKF`, you can specify the measurement Jacobian matrix using the `MeasurementJacobianFcn` property. If you do not specify this property, the object computes the Jacobians using numeric differencing, which is slightly less accurate and can increase the computation time.

Extended Kalman Filter Loop

The extended Kalman filter loop is almost identical to the loop of “Linear Kalman Filters” on page 4-28 except that:

- The filter uses the exact nonlinear state update and measurement functions whenever possible.
- The state Jacobian replaces the state transition matrix.
- The measurement jacobian replaces the measurement matrix.



Predefined Extended Kalman Filter Functions

The toolbox provides predefined state update and measurement functions to use in `trackingEKF`.

Motion Model	Function Name	Function Purpose	State Representation
Constant velocity	<code>constvel</code>	Constant-velocity state update model	<ul style="list-style-type: none"> • 1-D — $[x; vx]$ • 2-D — $[x; vx; y; vy]$
	<code>constveljac</code>	Constant-velocity state update Jacobian	<ul style="list-style-type: none"> • 3-D — $[x; vx; y; vy; z; vz]$
	<code>cvmeas</code>	Constant-velocity measurement model	where
	<code>cvmeasjac</code>	Constant-velocity measurement Jacobian	<ul style="list-style-type: none"> • $x, y,$ and z represents the position in the x-, y-, and z-directions, respectively. • $vx, vy,$ and vz represents the velocity in the x-, y-, and z-directions, respectively.

Motion Model	Function Name	Function Purpose	State Representation
Constant acceleration	constacc	Constant-acceleration state update model	<ul style="list-style-type: none"> • 1-D — $[x; vx; ax]$ • 2-D — $[x; vx; ax; y; vy; ay]$ • 3-D — $[x; vx; ax; y; vy; ay; z; v_z; az]$ where <ul style="list-style-type: none"> • ax, ay, and az represents the acceleration in the x-, y-, and z-directions, respectively.
	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	<ul style="list-style-type: none"> • 2-D — $[x; vx; y; vy; \omega]$ • 3-D — $[x; vx; y; vy; \omega; z; v_z]$ where ω represents the turn-rate.
	constturnjac	Constant turn-rate state update Jacobian	
	ctmeas	Constant turn-rate measurement model	
	ctmeasjac	Constant turn-rate measurement Jacobian	

Example: Estimate 2-D Target States with Angle and Range Measurements Using trackingEKF

Initialize Estimation Model

Assume a target moves in 2D with the following initial position and velocity. The simulation lasts 20 seconds with a sample time of 0.2 seconds.

```
rng(2022);    % For repeatable results
dt = 0.2;    % seconds
simTime = 20; % seconds
tspan = 0:dt:simTime;
trueInitialState = [30; 1; 40; 1]; % [x;vx;y;vy]
initialCovariance = diag([100,1e3,100,1e3]);
processNoise = diag([0; .01; 0; .01]); % Process noise matrix
```

Assume the measurements are the azimuth angle relative to the positive-x direction and the range to from the origin to the target. The measurement noise covariance matrix is:

```
measureNoise = diag([2e-6;1]); % Measurement noise matrix. Units are m^2 and rad^2.
```

Preallocate variables in which to save results.

```

numSteps = length(tspan);
trueStates = NaN(4,numSteps);
trueStates(:,1) = trueInitialState;
estimateStates = NaN(size(trueStates));
measurements = NaN(2,numSteps);

```

Obtain True States and Measurements

Propagate the constant velocity model and generate the measurements with noise.

```

for i = 2:length(tspan)
    if i ~= 1
        trueStates(:,i) = stateModel(trueStates(:,i-1),dt) + sqrt(processNoise)*randn(4,1);
    end
    measurements(:,i) = measureModel(trueStates(:,i)) + sqrt(measureNoise)*randn(2,1);
end

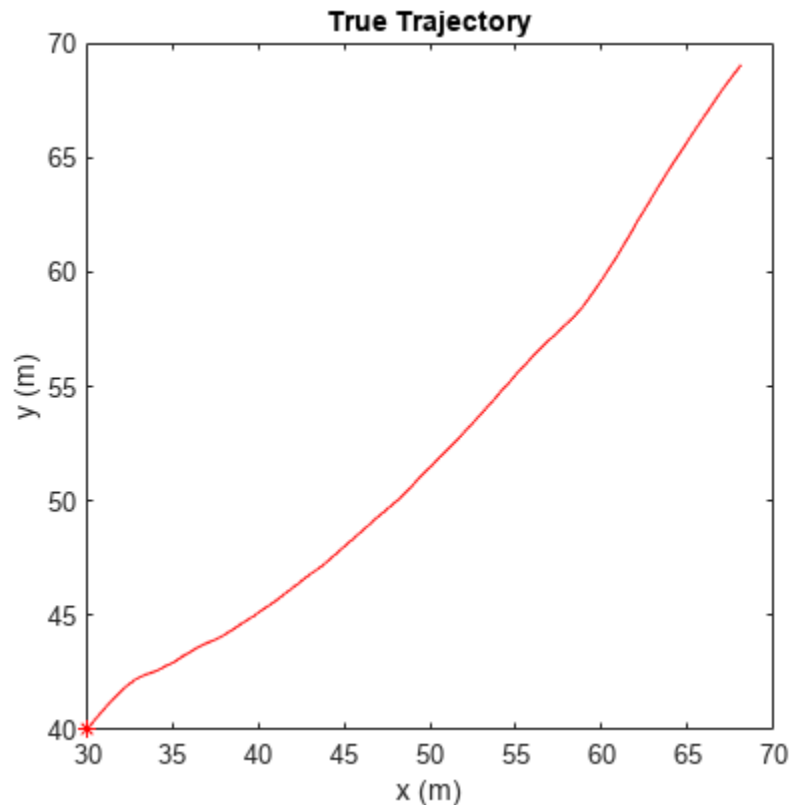
```

Plot the true trajectory and the measurements.

```

figure(1)
plot(trueStates(1,1),trueStates(3,1),"r*","DisplayName="Initial Truth")
hold on
plot(trueStates(1,:),trueStates(3,:),"r","DisplayName="True Trajectory")
xlabel("x (m)")
ylabel("y (m)")
title("True Trajectory")
axis square

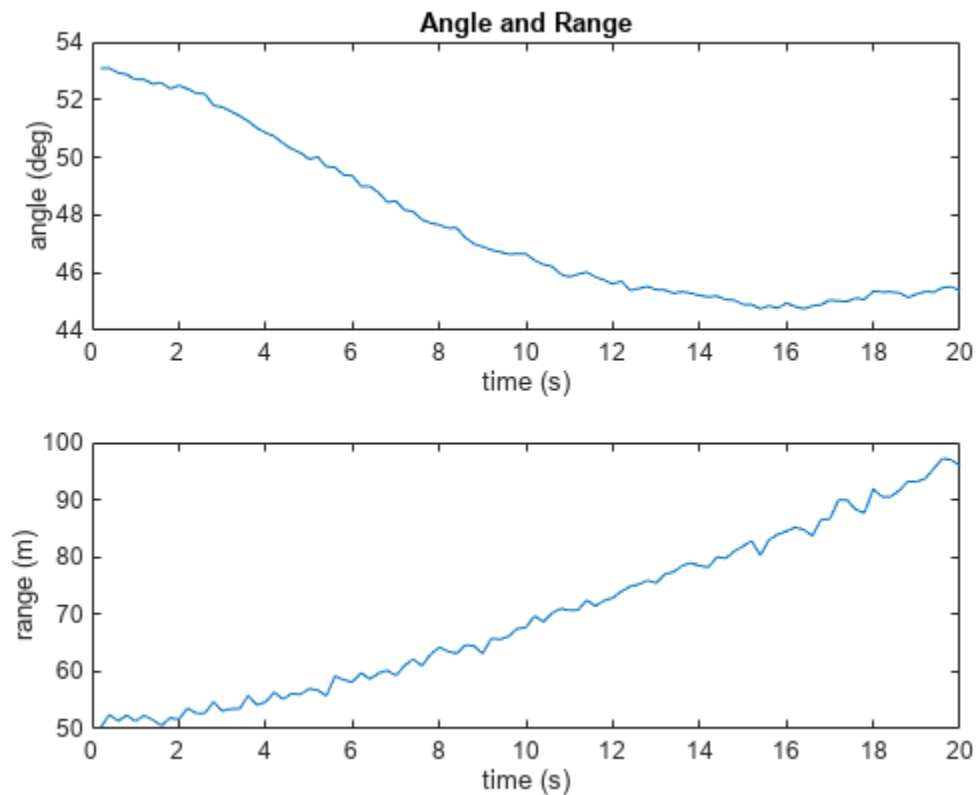
```



```

figure(2)
subplot(2,1,1)
plot(tspan,measurements(1,:)*180/pi)
xlabel("time (s)")
ylabel("angle (deg)")
title("Angle and Range")
subplot(2,1,2)
plot(tspan,measurements(2,:))
xlabel("time (s)")
ylabel("range (m)")

```



Initialize Extended Kalman Filter

Initialize the filter with an initial state estimate at [35; 0; 45; 0].

```

filter = trackingEKF(State=[35; 0; 45; 0],StateCovariance=initialCovariance, ...
    StateTransitionFcn=@stateModel,ProcessNoise=processNoise, ...
    MeasurementFcn=@measureModel,MeasurementNoise=measureNoise);
estimateStates(:,1) = filter.State;

```

Run Extended Kalman Filter And Show Results

Run the filter by recursively calling the predict and correct object functions.

```

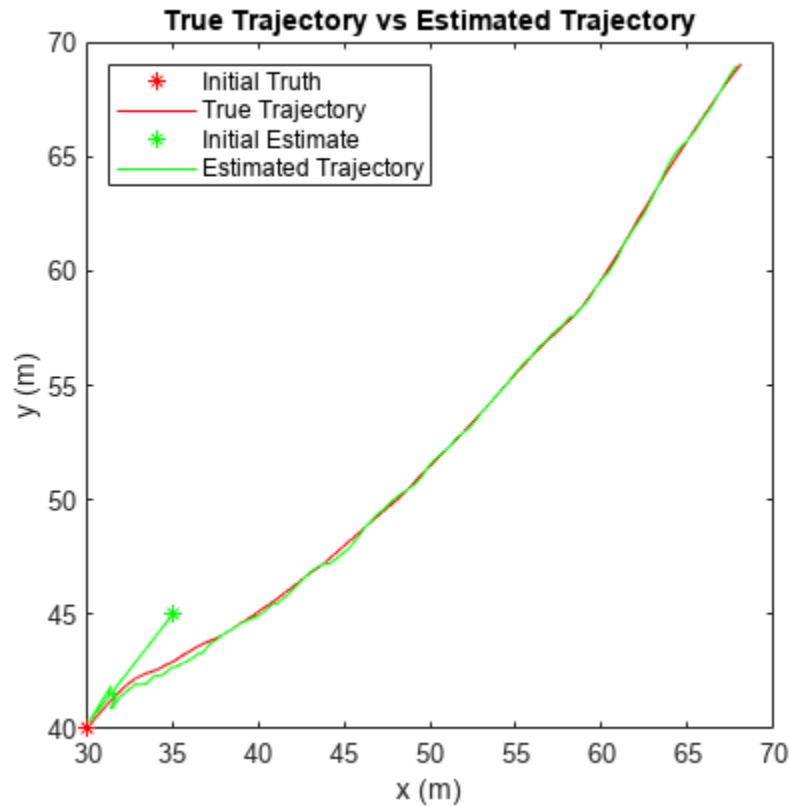
for i=2:length(tspan)
    predict(filter,dt);
    estimateStates(:,i) = correct(filter,measurements(:,i));
end

```

```

figure(1)
plot(estimateStates(1,1),estimateStates(3,1),"g*",DisplayName="Initial Estimate")
plot(estimateStates(1,:),estimateStates(3,:),"g",DisplayName="Estimated Trajectory")
legend(Location="northwest")
title("True Trajectory vs Estimated Trajectory")

```



Helper Functions

stateModel modeled constant velocity motion without process noise.

```

function stateNext = stateModel(state,dt)
    F = [1 dt 0 0;
         0 1 0 0;
         0 0 1 dt;
         0 0 0 1];
    stateNext = F*state;
end

```

measureModel models range and azimuth angle measurements without noise.

```

function z = measureModel(state)
    angle = atan(state(3)/state(1));
    range = norm([state(1) state(3)]);

```

```
        z = [angle;range];  
end
```

See Also

[trackingKF](#) | [trackingEKF](#) | [trackingUKF](#) | “Linear Kalman Filters” on page 4-28 | “Introduction to Estimation Filters” on page 4-9

Introduction to Multiple Target Tracking

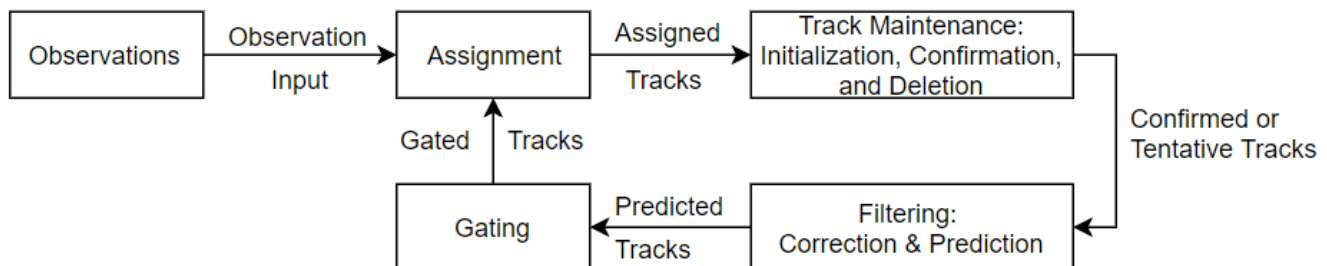
Background

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 6-148 for more details.

Gating and Assignment

For details about gating and assignment, see “Introduction to Assignment Methods in Tracking Systems” on page 4-49, 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 6-168 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 6-78 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

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

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, \dots) 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:

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

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

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

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:

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

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 ($i = 1, \dots, n$) 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.

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.

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:

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

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.

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, $i = 1, 2, \dots, M_1$. 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:

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

subject to three constraints:

$$\begin{aligned} \sum_{i=0}^{M_1} \sum_{j=0}^{M_2} z_{ijk} &= 1, \forall k = 1, 2, \dots, M_3 \\ \sum_{i=0}^{M_1} \sum_{k=0}^{M_3} z_{ijk} &= 1, \forall j = 1, 2, \dots, M_2 \\ \sum_{j=0}^{M_2} \sum_{k=0}^{M_3} z_{ijk} &= 1, \forall i = 1, 2, \dots, M_1 \end{aligned}$$

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 $L(\lambda)$:

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

where $\lambda_k, k = 1, 2, \dots, M_3$ 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 6-227 for the application of S-D assignment in static detection fusion.

See Also

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

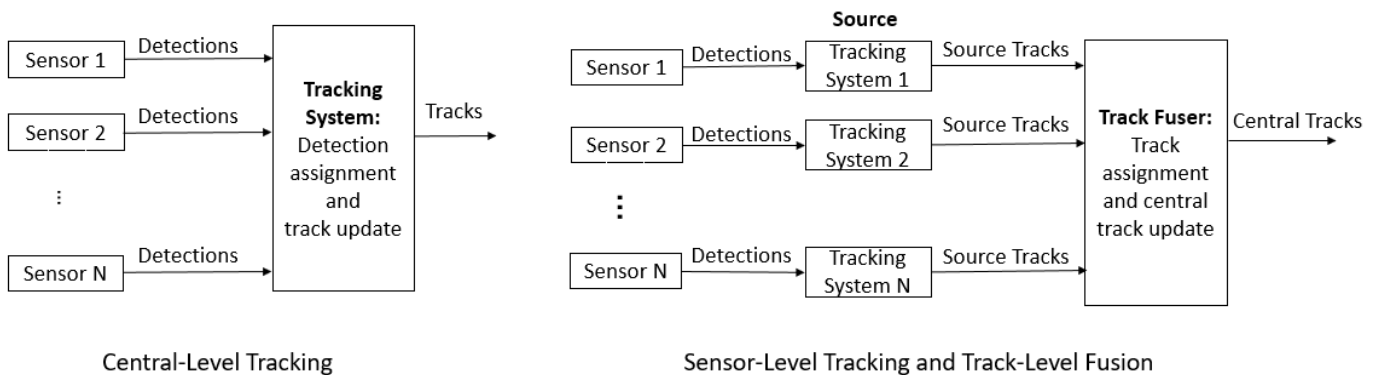
References

- [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.



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

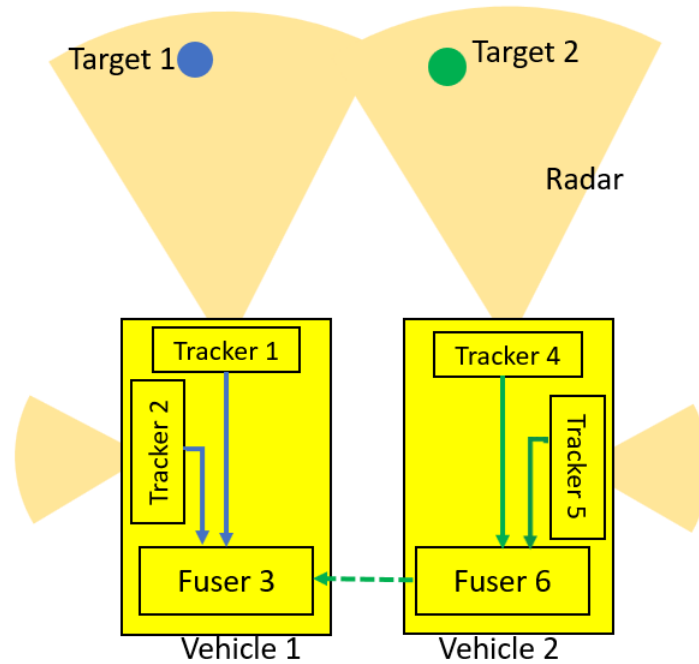
You can use the `trackFuser` in Sensor Fusion and Tracking Toolbox for the purpose of track-to-track fusion. The `trackFuser` System object provides two algorithms to combine source tracks considering the correction effects between different tracks. You can choose the algorithm by specifying the `StateFusion`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

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

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 6-148 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 6-352 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 6-148 example, the GGIW-PHD Extended Object Tracker approach represents vehicle shapes as ellipses, and the GM-PHD Tracker approach represents vehicle shapes as rectangles.

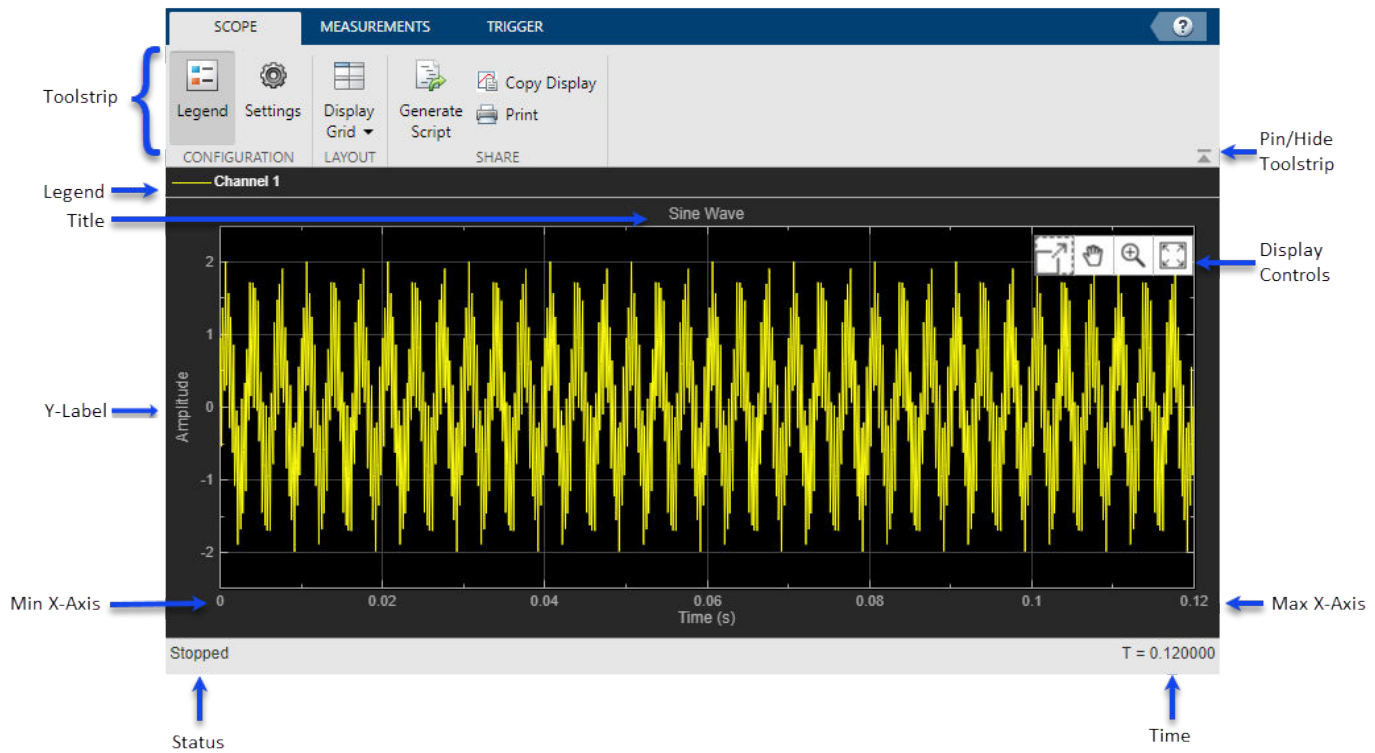
In the “Extended Object Tracking With Radar For Marine Surveillance” on page 6-370 example, the GGIW-PHD tracker models the ship shapes as ellipses.


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 $10/\text{SampleRate}$.

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**

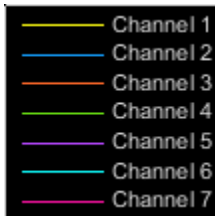
- **Scope** tab — Customize and share the time scope. For example, showing and hiding the legend
- **Measurements** tab — Turn on and control different measurement tools.
- **Trigger** tab — Turn on and modify triggers.

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

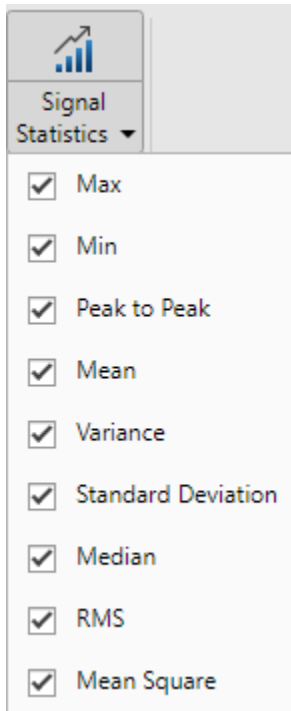
Use the **Data Cursors** button to display screen cursors. Each cursor tracks a vertical line along the signal. The difference between x - and y -values of the signal at the two cursors is displayed in the box between the cursors.

Signal Statistics

Use the **Signal Statistics** button to display 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  at 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.
- **Variance** -- Variance of the values within the displayed portion of the input signal.
- **Standard Deviation** -- Standard deviation of the values within the displayed portion of the input signal.
- **Median** — Median value within the displayed portion of the input signal.
- **RMS** — Root mean square of the input signal.
- **Mean Square** -- Mean square of the values within the displayed portion of the input signal.

To customize which statistics to show and compute, use the **Signal Statistics** list.



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

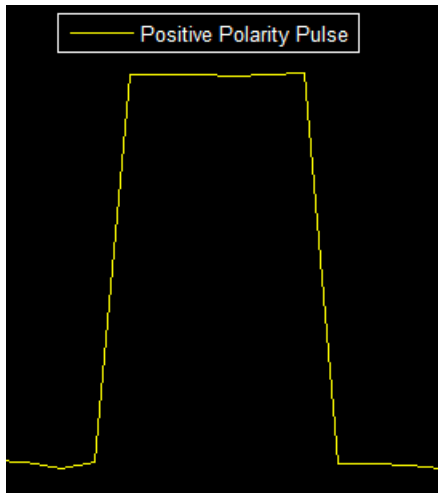
With bilevel measurements, you can measure transitions, aberrations, and cycles.

Bilevel Settings

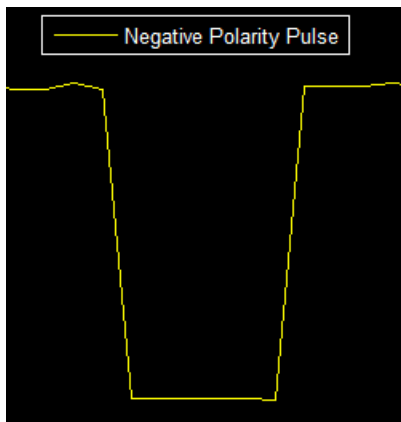
When using bilevel measurements, you can set these properties:

- **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.

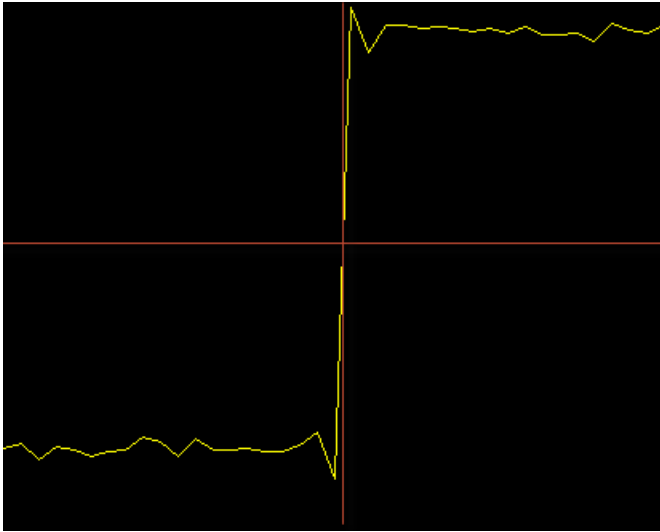
- **High** — Manually specify the value that denotes a positive polarity or high-state level.



- **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.

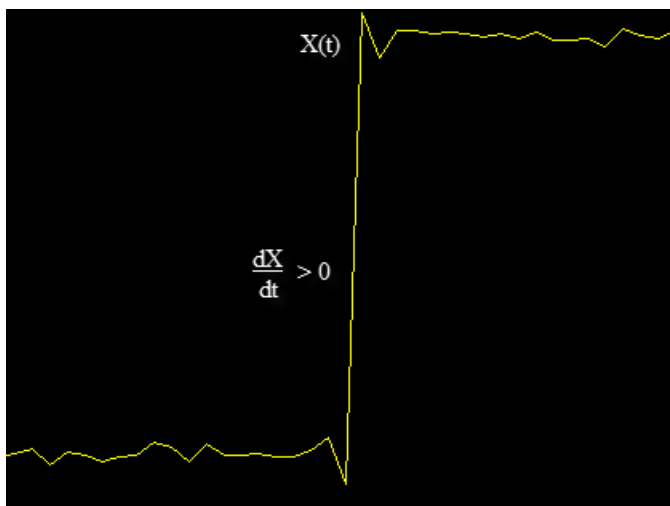


- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs is used for computing a valid settling time. Settling time is displayed in the **Aberrations** pane.

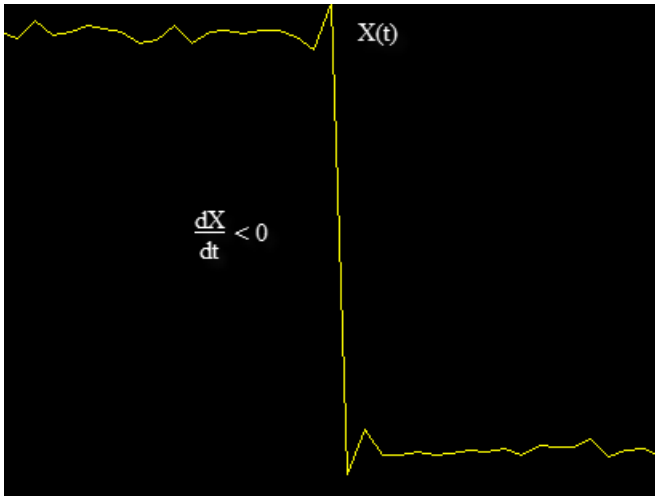
Transitions

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.

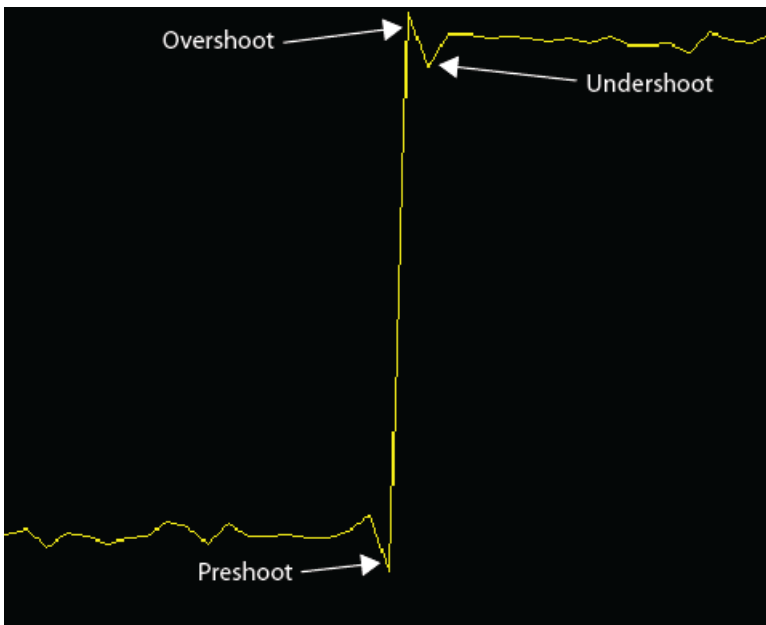


The Transition measurements assume that the amplitude of the input signal is in units of volts. For the transition measurements to be valid, you must convert all input signals to volts.

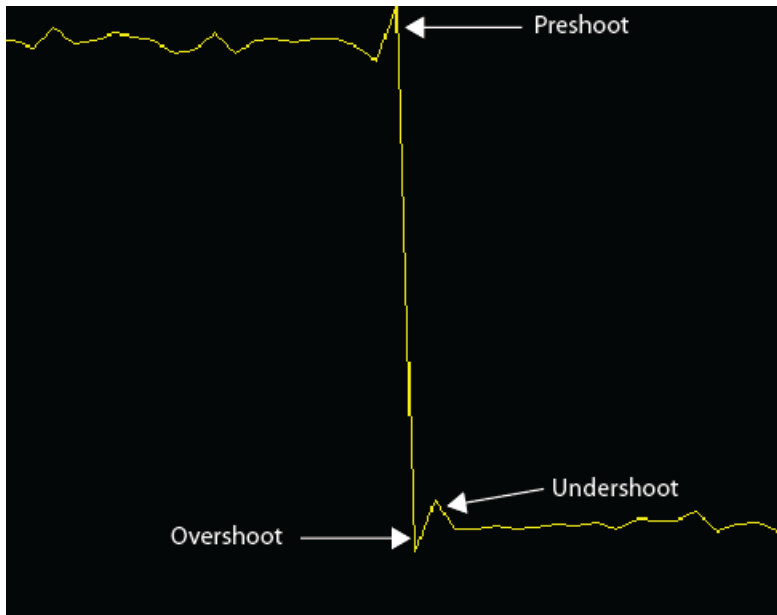
Aberrations

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.



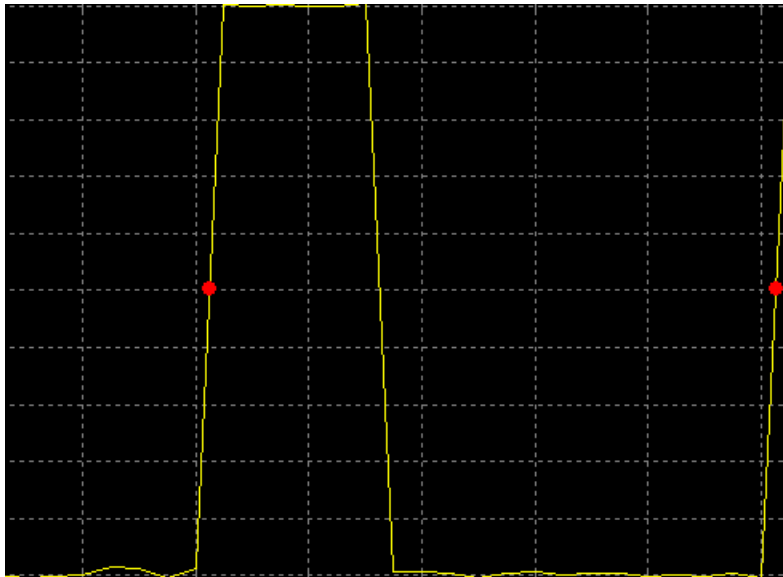
The next figure shows preshoot, overshoot, and undershoot for a falling-edge transition.



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

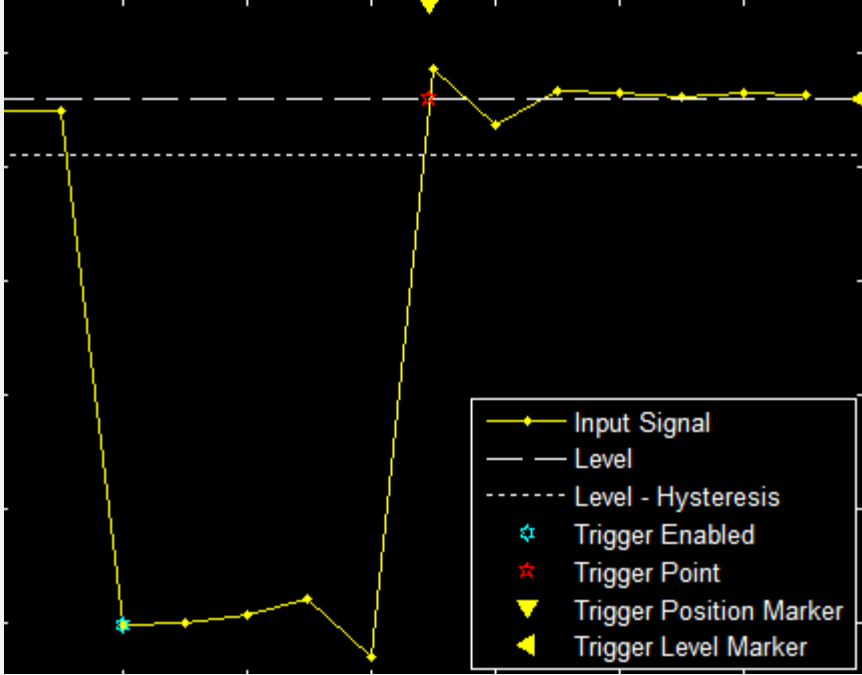
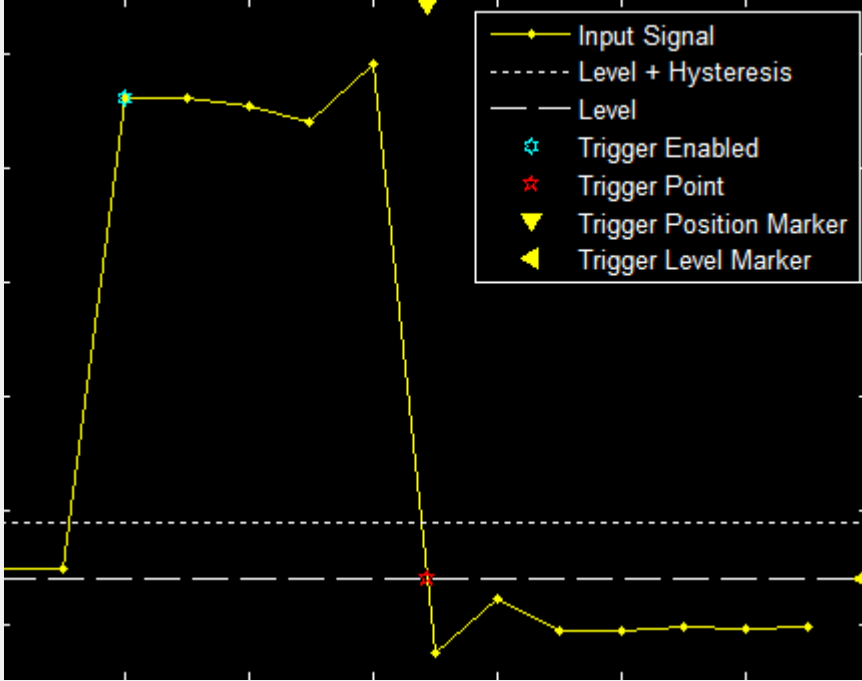
Define a trigger event to identify the simulation time of specified input signal characteristics. You can use trigger events to stabilize periodic signals such as a sine wave or capture non-periodic signals such as a pulse that occurs intermittently.

To define a trigger:

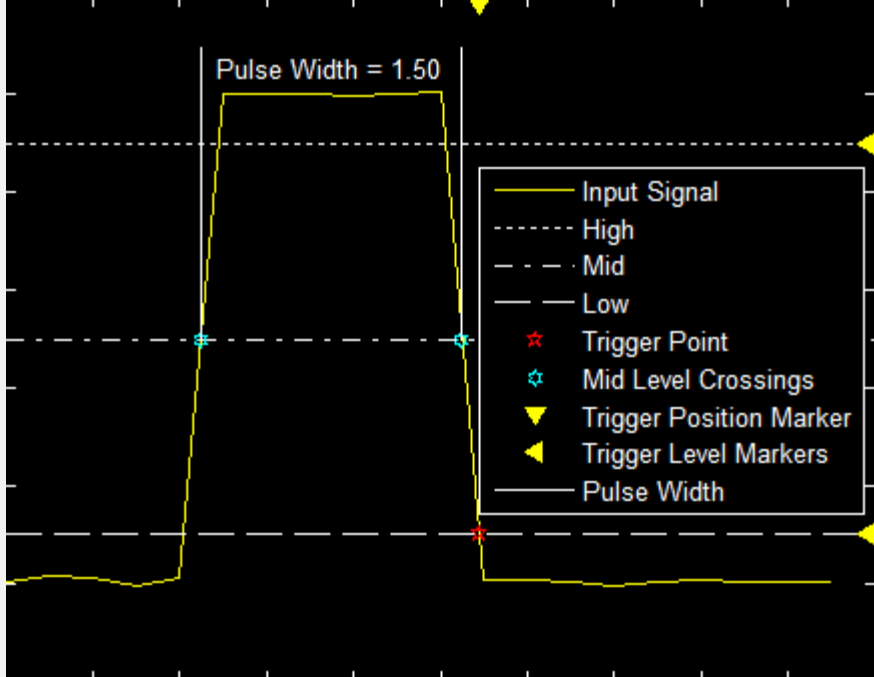
- 1 On the Trigger tab of the scope window, select the channel you want to trigger.
- 2 Specify when the display updates by selecting a triggering **Mode**.
 - **Auto** — Display data from the last trigger event. If no event occurs after one time span, display the last available data.
Normal — Display 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**.

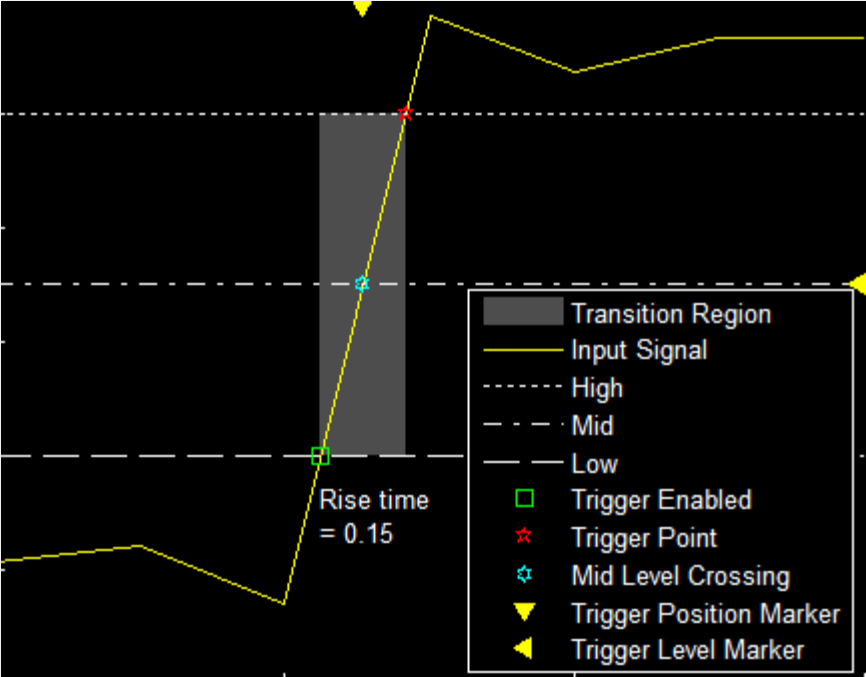
You can set the trigger position to specify the position of the time pointer along the y-axis. You can also drag the time pointer to the left or right to adjust its position.

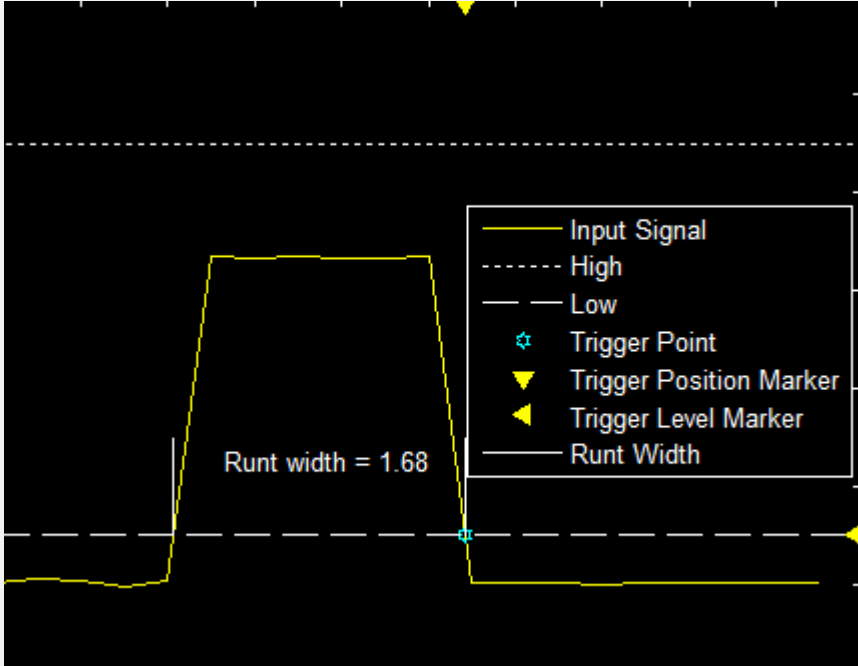
Trigger Properties


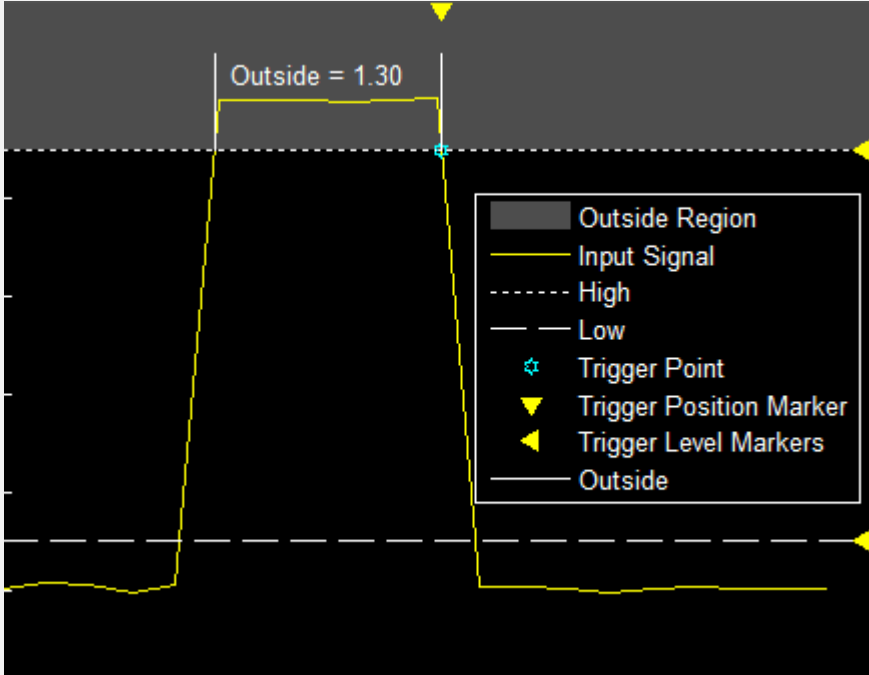
Trigger Type	Trigger Parameters
Edge — Trigger when the signal crosses a threshold.	<p>Polarity — Select the polarity for an edge-triggered signal.</p> <ul style="list-style-type: none"> Rising — Trigger when the signal is increasing.  <ul style="list-style-type: none"> Falling — Trigger when the signal value is decreasing. 

Trigger Type	Trigger Parameters
	<ul style="list-style-type: none">• Either — Trigger when the signal is increasing or decreasing. <p>Level — Enter a threshold value for an edge-triggered signal. Auto level is 50%</p> <p>Hysteresis — Enter a value for an edge-triggered signal. See “Hysteresis of Trigger Signals” on page 4-78</p> <p>Delay — Offset the trigger by a fixed delay in seconds.</p> <p>Holdoff — Set the minimum possible time between triggers.</p> <p>Position — Set horizontal position of the trigger on the screen.</p>

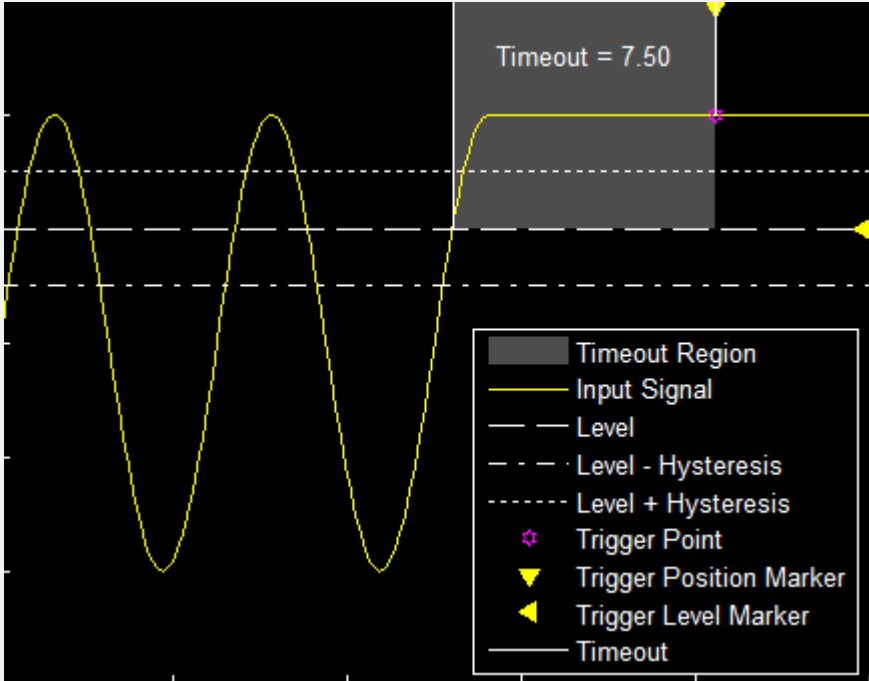
Trigger Type	Trigger Parameters
<p>Pulse Width — Trigger when the signal crosses a low threshold and a high threshold twice within a specified time.</p>	<p>Polarity — Select the polarity for a pulse width-triggered signal.</p> <ul style="list-style-type: none"> • Positive — Trigger on a positive-polarity pulse when the pulse crosses the low threshold for a second time.  <ul style="list-style-type: none"> • 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.
	<p>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.</p>
	<p>High — Enter a high value for a pulse-width-triggered signal. Auto level is 90%.</p>
	<p>Low — Enter a low value for a pulse-width-triggered signal. Auto level is 10%.</p>
	<p>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.</p>
	<p>Max Width — Enter the maximum pulse width for a pulse-width-triggered signal.</p>
	<p>Delay — Offset the trigger by a fixed delay in seconds.</p>

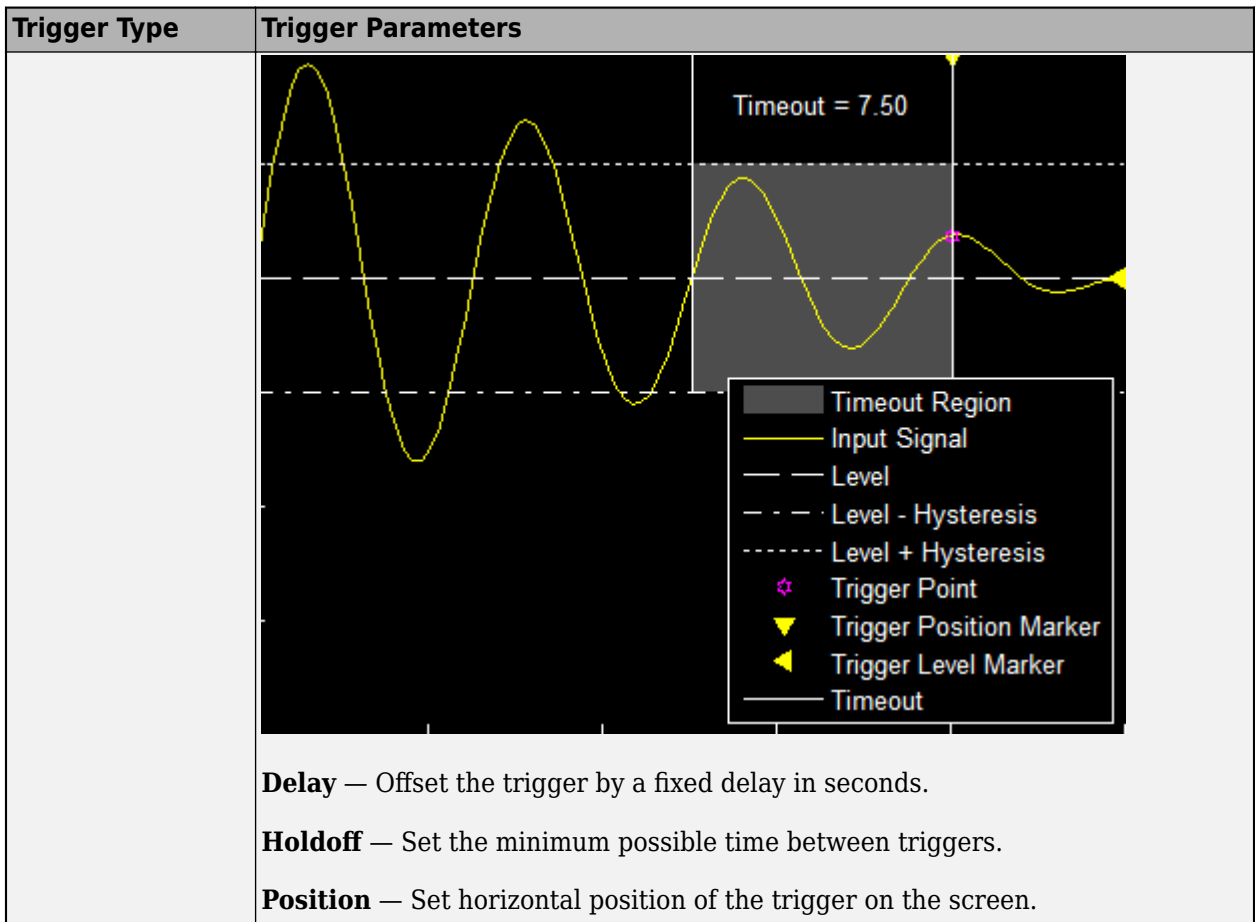
Trigger Type	Trigger Parameters
	<p>Holdoff — Set the minimum possible time between triggers.</p> <p>Position — Set horizontal position of the trigger on the screen.</p>
<p>Transition — Trigger on the rising or falling edge of a signal that crosses the high and low levels within a specified time range.</p>	<p>Polarity — Select the polarity for a transition-triggered signal.</p> <ul style="list-style-type: none"> • Rise Time — Trigger on an increasing signal when the signal crosses the high threshold.  <ul style="list-style-type: none"> • Fall Time — Trigger on a decreasing signal when the signal crosses the low threshold. • Either — Trigger on an increasing or decreasing signal. <p>High — Enter a high value for a transition-triggered signal. Auto level is 90%.</p> <p>Low — Enter a low value for a transition-triggered signal. Auto level is 10%.</p> <p>Min Time — Enter a minimum time duration for a transition-triggered signal.</p> <p>Max Time — Enter a maximum time duration for a transition-triggered signal.</p> <p>Delay — Offset the trigger by a fixed delay in seconds.</p> <p>Holdoff — Set the minimum possible time between triggers.</p> <p>Position — Set horizontal position of the trigger on the screen.</p>

Trigger Type	Trigger Parameters
<p>Runt— Trigger when a signal crosses a low threshold or a high threshold twice within a specified time.</p>	<p>Polarity — Select the polarity for a runt-triggered signal.</p> <ul style="list-style-type: none"> • Positive — Trigger on a positive-polarity pulse when the signal crosses the low threshold a second time without crossing the high threshold.  <ul style="list-style-type: none"> • Negative — Trigger on a negative-polarity pulse. • Either — Trigger on both positive-polarity and negative-polarity pulses. <p>High — Enter a high value for a runt-triggered signal. Auto level is 90%.</p> <p>Low — Enter a low value for a runt-triggered signal. Auto level is 10%.</p> <p>Min Width — Enter a minimum width for a runt-triggered signal. Pulse width is measured between the first and second crossing of a threshold.</p> <p>Max Width — Enter a maximum pulse width for a runt-triggered signal.</p> <p>Delay — Offset the trigger by a fixed delay in seconds.</p> <p>Holdoff — Set the minimum possible time between triggers.</p> <p>Position — Set horizontal position of the trigger on the screen.</p>

Trigger Type	Trigger Parameters
<p>Window — Trigger when a signal stays within or outside a region defined by the high and low thresholds for a specified time.</p>	<p>Polarity — Select the region for a window-triggered signal.</p> <ul style="list-style-type: none"> • Inside — Trigger when a signal leaves a region between the low and high levels.  <ul style="list-style-type: none"> • Outside — Trigger when a signal enters a region between the low and high levels. 

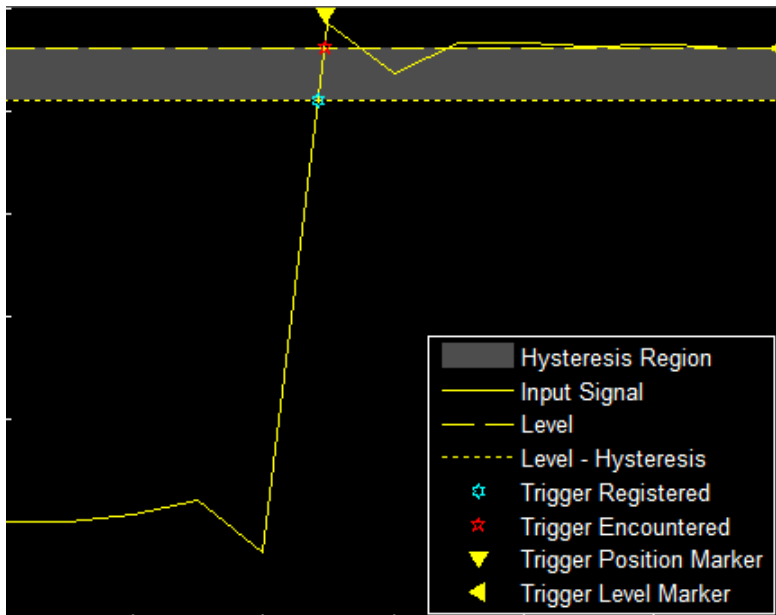
Trigger Type	Trigger Parameters
	<ul style="list-style-type: none"> • Either — Trigger when a signal leaves or enters a region between the low and high levels. High — Enter a high value for a window-triggered signal. Auto level is 90%. Low — Enter a low value for a window-trigger signal. Auto level is 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
<p>Timeout — Trigger when a signal stays above or below a threshold longer than a specified time</p>	<p>Polarity — Select the polarity for a timeout-triggered signal.</p> <ul style="list-style-type: none"> • 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.  <ul style="list-style-type: none"> • 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 <p>Level — Enter a threshold value for a timeout-triggered signal.</p> <p>Hysteresis — Enter a value for a timeout-triggered signal. See “Hysteresis of Trigger Signals” on page 4-78.</p> <p>Timeout — Enter a time duration for a timeout-triggered signal.</p> <p>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.</p>

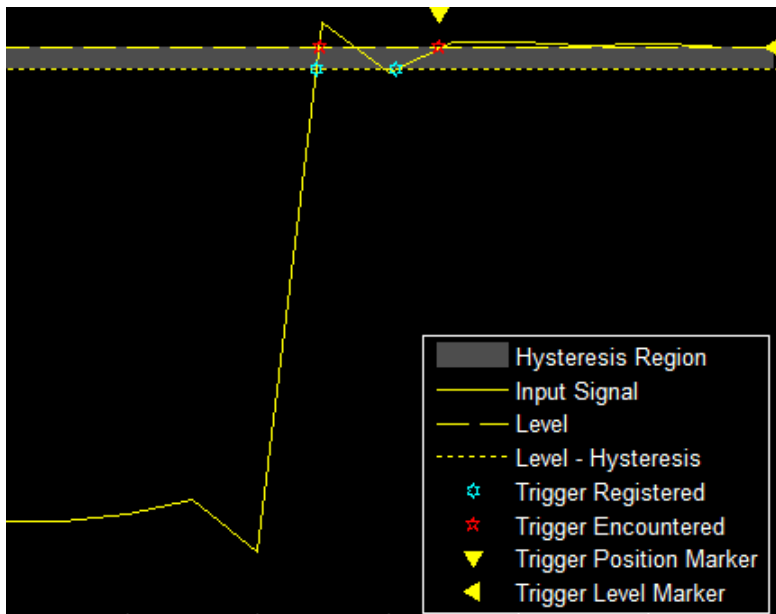


Hysteresis of Trigger Signals

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, hide all labels and inset the axes values.
-  — Zoom in on the plot.
-  — Pan the plot.
-  — Autoscale the axes to fit the shown data.

See Also

Objects
timescope

Code Generation

Generate Code with Strict Single-Precision and Non-Dynamic Memory Allocation

Introduce functions, objects, and blocks that support strict single-precision and non-dynamic memory allocation code generation in Sensor Fusion and Tracking Toolbox.

Introduction to Strict Single-Precision and Non-Dynamic Memory Allocation

In Sensor Fusion and Tracking Toolbox (SFTT), many functions, objects, and Simulink blocks support C/C++ code generation. You can see if a function, object, or block supports code generation, as well as any limitations in the Extended Capabilities section of its reference page. For details on how to generate code at the command line, or by using the MATLAB Coder App, see “Generate C Code at the Command Line” (MATLAB Coder) and “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder), respectively. For generating code for SFTT applications, see these examples:

- “How to Generate C Code for a Tracker” on page 6-296
- “Generate Code for a Track Fuser with Heterogeneous Source Tracks” on page 6-588
- “Processor-in-the-Loop Verification of JPDA Tracker for Automotive Applications” on page 6-886

Sensor Fusion and Tracking Toolbox widely supports double-precision code generation, and the generated code can use dynamic memory allocation if necessary. Though double-precision variables can provide more accurate calculations, they have increased memory requirements over single-precision variables. Similarly, though dynamic memory allocation allows for flexible variable allocation, this process is typically slower than non-dynamic memory allocation. For these and other reasons, Sensor Fusion and Tracking Toolbox provides strict single-precision and non-dynamic memory allocation support for some algorithms.

In *strict single-precision* code generation, the generated C/C++ code, including the input, code body, and output, does not use double-precision variables. In other words, it can only use single-precision variables and integer-type variables up to 32 bits. For algorithms that support strict single-precision code generation in SFTT, you can enable it by passing single-precision input arguments. To enable strict single-precision code generation:

- For a function that supports strict single-precision code generation, specify single-precision inputs.
- For an object that supports strict single-precision code generation, specify the inputs as non-double-precision variables, and specify any custom setups as non-double precision. For example, to generate single-precision code from the `trackerGNN` System object:
 - You must specify the inputs, such as the detections, as non-double-precision values.
 - You must specify the `FilterInitializationFcn` property of the tracker to return a single-precision filter.

In *non-dynamic memory allocation* code generation, the memory allocation of each variable is determined during compilation time, before running the code. Non-dynamic memory allocation is usually faster than dynamic memory allocation. For information on how to disable dynamic memory allocation, see “Generate Code for Variable-Size Data” (MATLAB Coder) and “Control Memory Allocation for Variable-Size Arrays” (MATLAB Coder).

The functions, objects, and blocks listed in the table of the following sections are verified to support strict single-precision and non-dynamic memory allocation code generation. Other unverified

functions, object, and blocks in SFTT can possibly support strict single-precision and non-dynamic memory allocation.

Supported Trackers and Tracking Filters

These trackers and tracking filters support strict single-precision and non-dynamic memory allocation code generation with the specified limitations.

Objects or Blocks	Strict single-Precision Code Generation Limitations	Non-Dynamic Memory Allocation Code Generation Limitations
trackerJPDA or Joint Probabilistic Data Association Multi Object Tracker	<ul style="list-style-type: none"> • Must set the tracker to the K-best JPDA mode. • Filter must be one of these, configured in single-precision: <ul style="list-style-type: none"> • trackingKF • trackingEKF • trackingUKF • trackingCKF • trackingIMM 	<ul style="list-style-type: none"> • Must set the tracker to the K-best JPDA mode. • Filter must be one of these: <ul style="list-style-type: none"> • trackingKF • trackingEKF • trackingUKF • trackingCKF • trackingIMM • For trackerJPDA, the MaxNumDetections property must be finite.
trackerGNN or Global Nearest Neighbor Multi Object Tracker	<ul style="list-style-type: none"> • Filter must be one of these, configured in single-precision: <ul style="list-style-type: none"> • trackingKF • trackingEKF • trackingUKF • trackingCKF • trackingIMM • Assignment algorithm must be Jonker-Volgenant. 	<ul style="list-style-type: none"> • Filter must be one of these: <ul style="list-style-type: none"> • trackingKF • trackingEKF • trackingUKF • trackingCKF • trackingIMM • Assignment algorithm must be Jonker-Volgenant or MatchPairs. • For trackerGNN, the MaxNumDetections property must be finite.

Objects or Blocks	Strict single-Precision Code Generation Limitations	Non-Dynamic Memory Allocation Code Generation Limitations
trackerPHD or Probability Hypothesis Density (PHD) Tracker	<ul style="list-style-type: none"> You must specify the filter initialization function as single-precision in each <code>trackingSensorConfiguration</code> object. The filter specified in each <code>trackingSensorConfiguration</code> object must use state transition and measurement functions that support single-precision. 	<ul style="list-style-type: none"> You must specify the <code>MaxNumDetections</code> and <code>MaxNumDetsPerObject</code> properties in each <code>trackingSensorConfiguration</code> object as finite integers.
trackingKF	No limitations	No limitations
trackingEKF	The state transition function and measurement function must support single-precision.	No limitations
trackingUKF	The state transition function and measurement function must support single-precision.	No limitations
trackingCKF	The state transition function and measurement function must support single-precision.	No limitations
trackingIMM	The <code>trackingIMM</code> filter must be configured with either <code>trackingKF</code> , <code>trackingEKF</code> , <code>trackingUKF</code> , or <code>trackingCKF</code> objects set in single-precision.	The <code>trackingIMM</code> filter must be configured with either <code>trackingKF</code> , <code>trackingEKF</code> , <code>trackingUKF</code> , or <code>trackingCKF</code> objects.
gmphd	The motion model and measurement model used in the filter must support single-precision.	No limitations
ggiwphd	The motion model and measurement model used in the filter must support single-precision.	No limitations

Supported Assignment and Partition Functions

These assignment functions support strict single-precision and non-dynamic memory allocation code generation with the specified limitations.

Functions	Strict Single-Precision Code Generation Limitations	Non-Dynamic Memory Allocation Code Generation Limitations
assignkbest	Must use the Jonker-Volgenant algorithm.	Must use the Jonker-Volgenant algorithm.
assignjv	No limitations	No limitations
jpdaEvents	Must use K-best joint event.	Must use K-best joint event.
partitionDetections	No limitations	No limitations
mergeDetections	No limitations	No limitations

Supported Motion Model Functions

These motion model functions support strict single-precision and non-dynamic memory allocation code generation without limitations.

- constvel
- constveljac
- cvmeas
- cvmeasjac
- constacc
- constaccjac
- cameas
- cameasjac
- constturn
- constturnjac
- ctmeas
- ctmeasjac
- switchimm
- ctrect
- ctrectjac
- ctrectmeas
- ctrectmeasjac

Supported Filter Initialization Functions

These filter initialization functions support single-precision and non-dynamic memory allocation code generation without limitations.

- initcvkf
- initcakf
- initcvekf
- initcaekf

- `initctekf`
- `initcvukf`
- `initcaukf`
- `initctukf`
- `initcvckf`
- `initctckf`
- `initcackf`
- `initekfimm`
- `initcvggiwphd`
- `initcaggiwphd`
- `initctggiwphd`
- `initcvgmphd`
- `initcagmphd`
- `initctgmphd`
- `initctrectgmphd`

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 `fusionRadarSensor`.

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

radar = fusionRadarSensor(1,'Rotator', ...
    'UpdateRate', updaterate, ... % Hz
    'FieldOfView', fov, ... % [az;el] deg
    'MaxAzimuthScanRate', 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.MechanicalElevationLimits = [-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 `fusionRadarSensor` 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 `trackingGlobeViewer` 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.

```
origin = [42.366978, -71.022362, 50];  
mapViewer = trackingGlobeViewer('ReferenceLocation',origin,...  
    'Basemap','streets-dark');  
campos(mapViewer, origin + [0 0 1e5]);  
drawnow;  
plotScenario(mapViewer,scenario);  
snapshot(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 = objectTrack.empty;

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

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

```
while advance(scenario)

    % Update airliner positions
    plotPlatform(mapViewer, scenario.Platforms([2 3 4]), 'TrajectoryMode','Full');

    % 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);

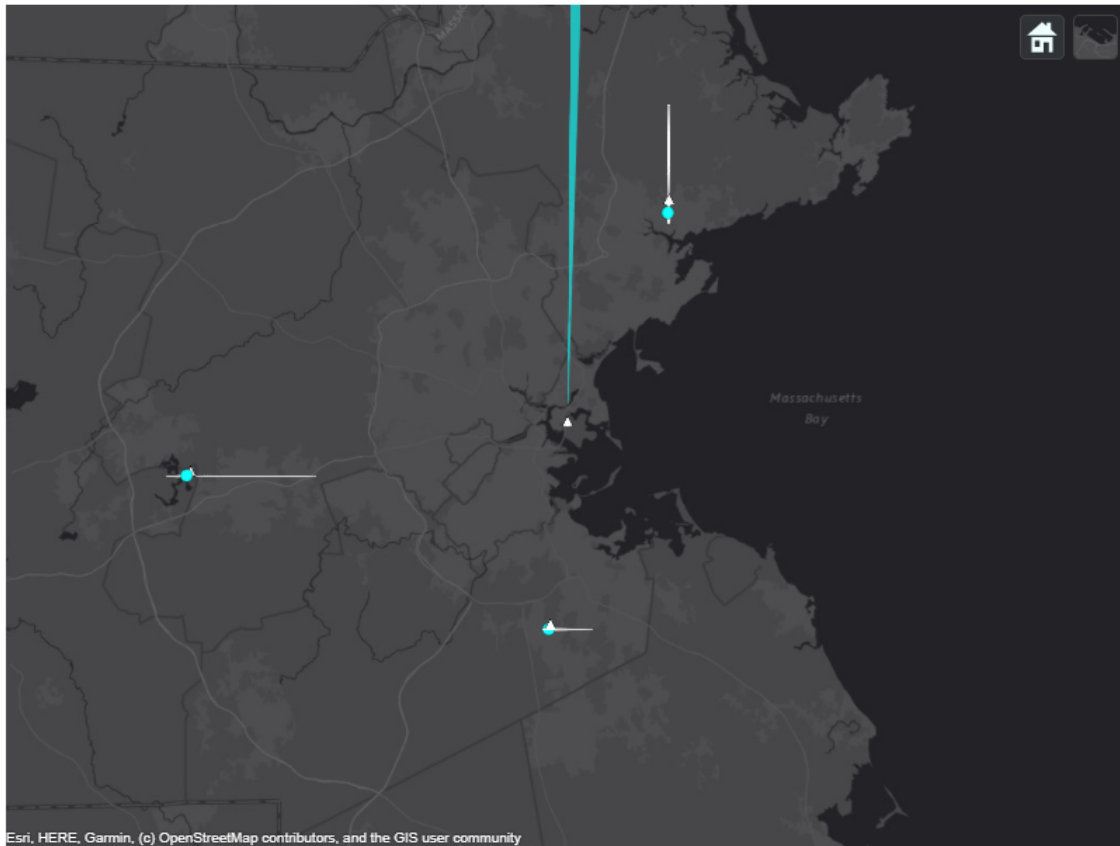
    % Update tracks when a 360 degree scan is complete
    simTime = scenario.SimulationTime;
    isScanDone = config.IsScanDone;
    if isScanDone
        scanCount = scanCount+1;
        % Update tracker
        [tracks,~,~,info] = tracker(scanBuffer,simTime);
        % Clear scan buffer for next scan
        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, {snapshot(mapViewer)}];
```

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

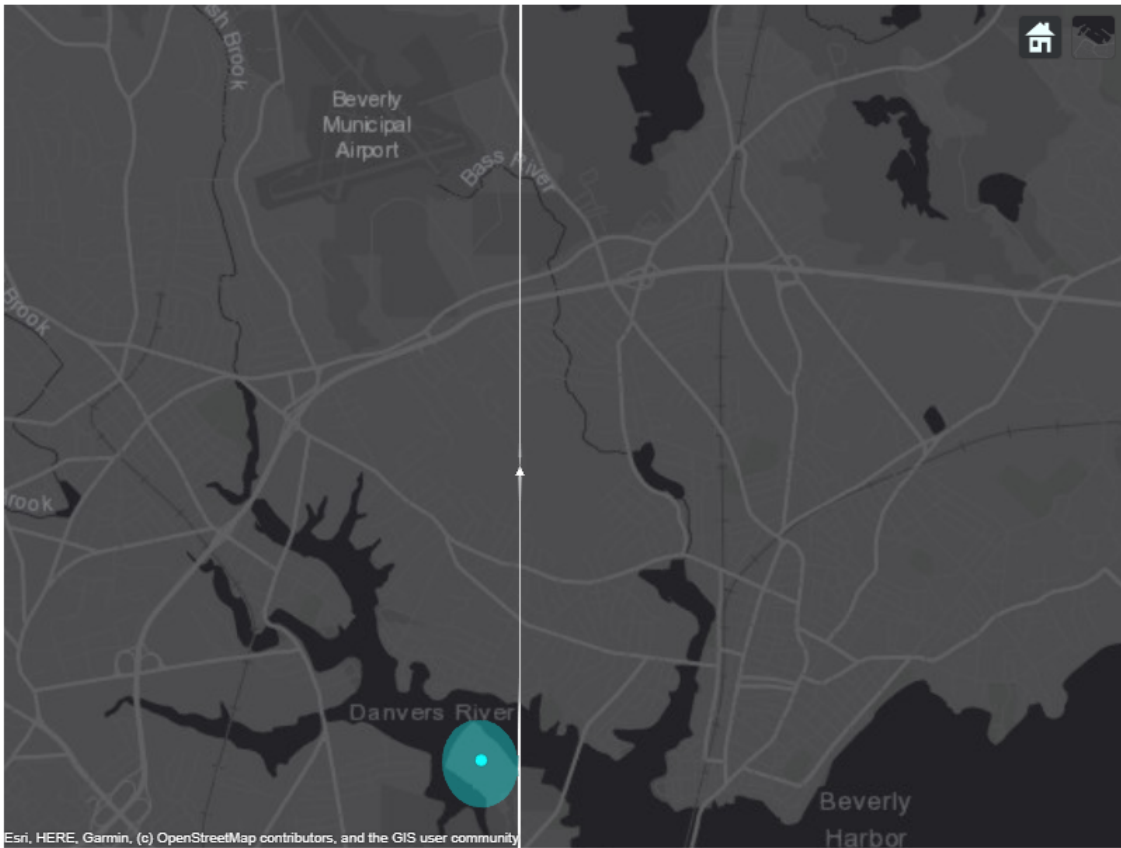
```
figure
imshow(allsnaps{1});
```

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});
```

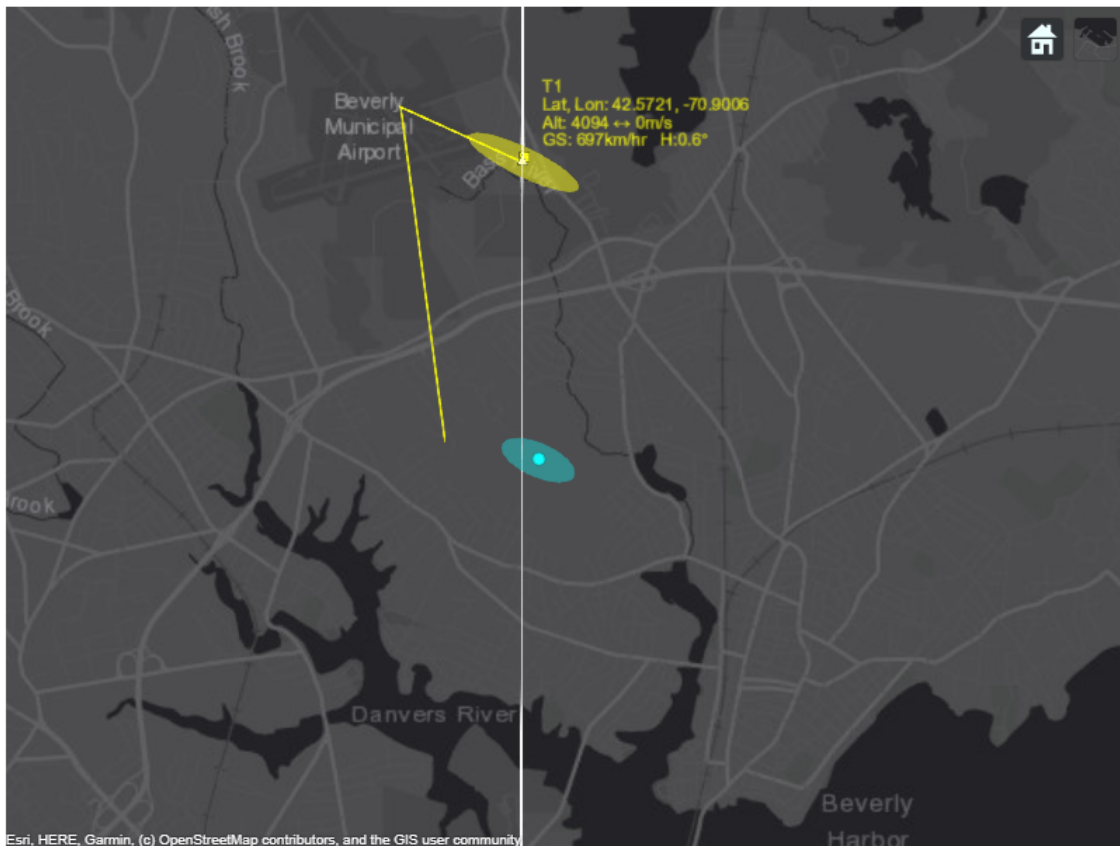


The previous figures show the track picture before and immediately after the tracker updates after the radar's second scan. The detection in the figure before the tracker update is used to update and confirm the initialized track from the previous scan's detection for this airliner. The next figure shows the confirmed track's position and velocity. The uncertainty of the track's position estimate is shown as the yellow 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 north at 700 km/hr.

```
figure  
imshow(allsnaps{4});
```

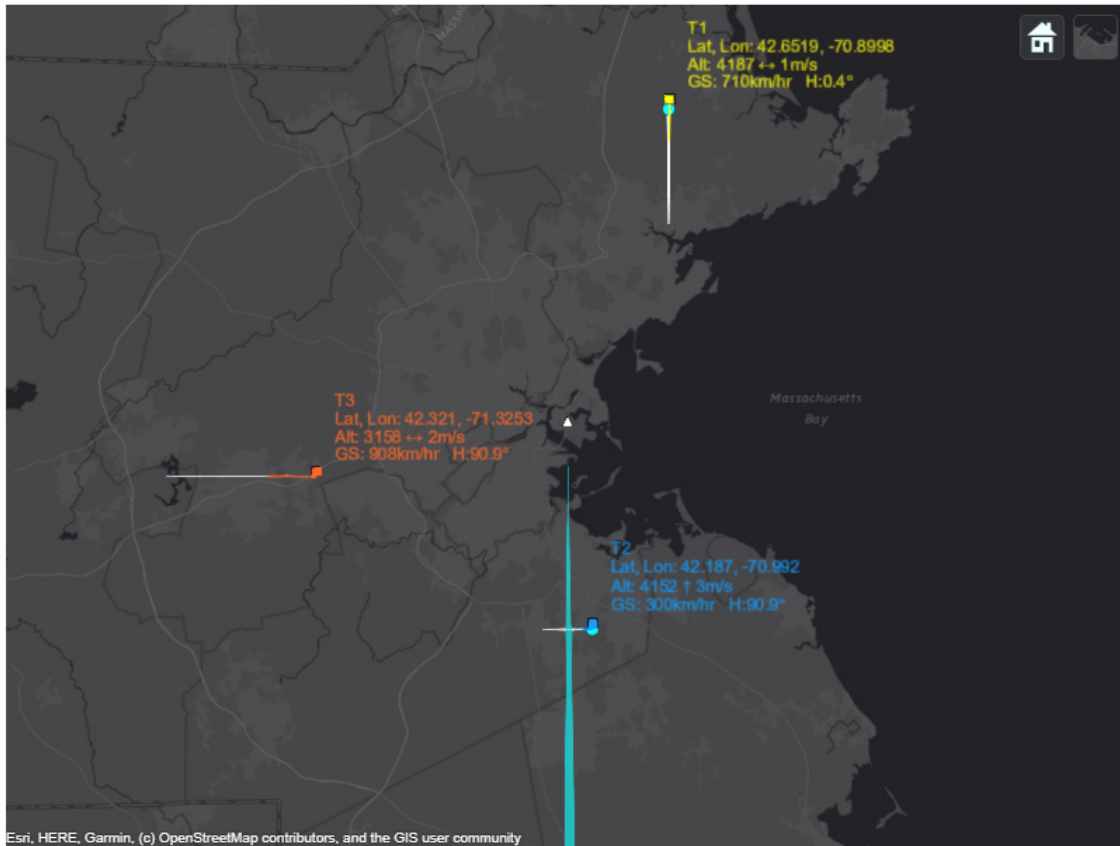


```
figure  
imshow(allsnaps{5});
```



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=3x4 table
  TrackID      Altitude      Heading      Speed
            True  Estimated  True  Estimated  True  Estimated
-----
  "T1"        4000      4051      90    90        700    710
  "T2"        4000      4070       0    359       300    300
  "T3"        3000      3057       0    359       900    908
```

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 = [235, -17, 0]; % Looking south west, 17 degrees below the horizon
campos(mapViewer, camPosition);
```

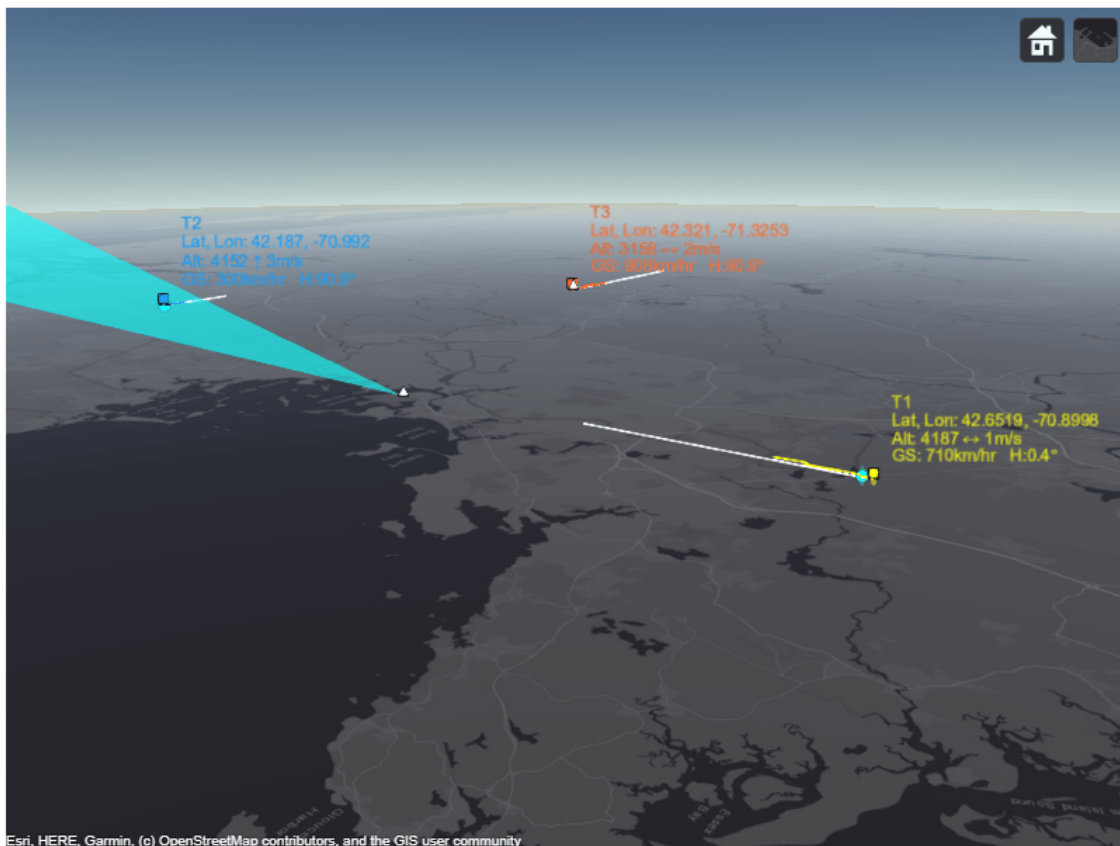
```
camorient(mapViewer, camOrientation);
drawnow
```

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.

```
snapshot(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', 'True','Estimated');
truthTrackTable(6).Properties.VariableNames = {'True','Estimated'};
truthTrackTable = mergevars(truthTrackTable, (4:5), 'NewVariableName', 'Heading', 'MergeAsTable', 'True','Estimated');
truthTrackTable(4).Properties.VariableNames = {'True','Estimated'};
truthTrackTable = mergevars(truthTrackTable, (2:3), 'NewVariableName', 'Altitude', 'MergeAsTable', 'True','Estimated');
truthTrackTable(2).Properties.VariableNames = {'True','Estimated'};
truthTrackTable.TrackID = "T" + string(truthTrackTable.TrackID);
end

```

snapPlotTrack

This function handles moving the 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])
    newsnap = snapshot(mapViewer);
    allsnaps = [allsnaps, {newsnap}];
    %move camera
    if scanCount == 2
        % Show the outbound airliner
        campos(mapViewer, [42.5650 -70.8990 7e3]);
        drawnow
        newsnap = snapshot(mapViewer);
        allsnaps = [allsnaps, {newsnap}];
    end
end

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

if isScanDone && any(scanCount == [2 3])
    % Take a snapshot of confirmed track
    drawnow
    newsnap = snapshot(mapViewer);
    allsnaps = [allsnaps, {newsnap}];
    % Reset Camera view to full scene
end

```

```
    if scanCount == 3
      origin = [42.366978, -71.022362, 50];
      campos(mapViewer, origin + [0 0 1e5]);
      drawnow
    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 = 0.0862; % Magnetometer measurement noise
Rvel = 0.0051; % GPS Velocity measurement noise
Rpos = 5.169; % GPS Position measurement noise

% Process noises
fusionfilt.AccelerometerBiasNoise = 0.010716;
fusionfilt.AccelerometerNoise = 9.7785;
fusionfilt.GyroscopeBiasNoise = 1.3436e-14;
fusionfilt.GyroscopeNoise = 0.00016528;
fusionfilt.MagnetometerBiasNoise = 2.189e-11;
```

```
fusionfilt.GeoMagneticVectorNoise = 7.67e-13;
```

```
% Initial error covariance
fusionfilt.StateCovariance = 1e-9*eye(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', ...
        [-3, 3
        -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);
```

```

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;

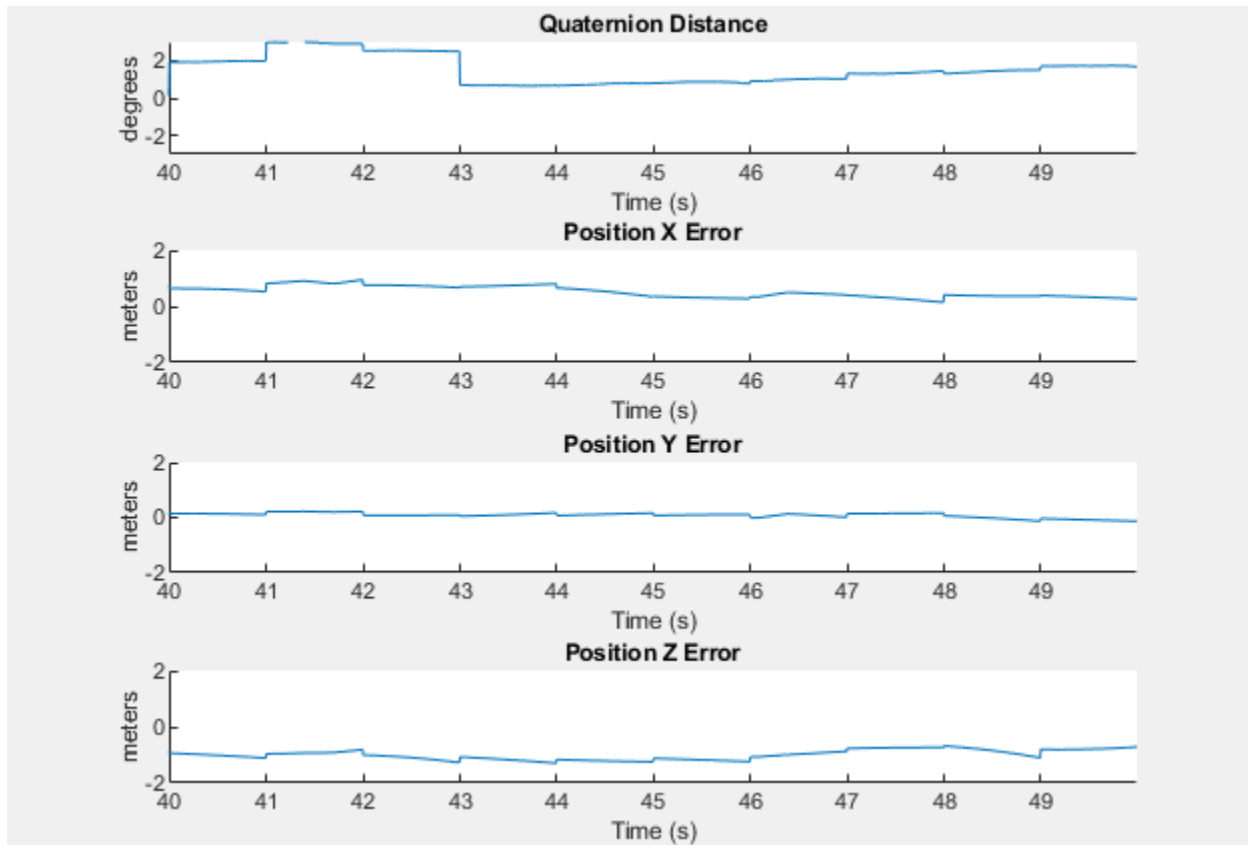
        % Compute the errors and plot.
        if useErrScope
            orientErr = rad2deg(dist(fusedOrient, ...
                trajOrient(fcnt) ));
            posErr = fusedPos - trajPos(fcnt,:);
            errscape(orientErr, posErr(1), posErr(2), posErr(3));
        end

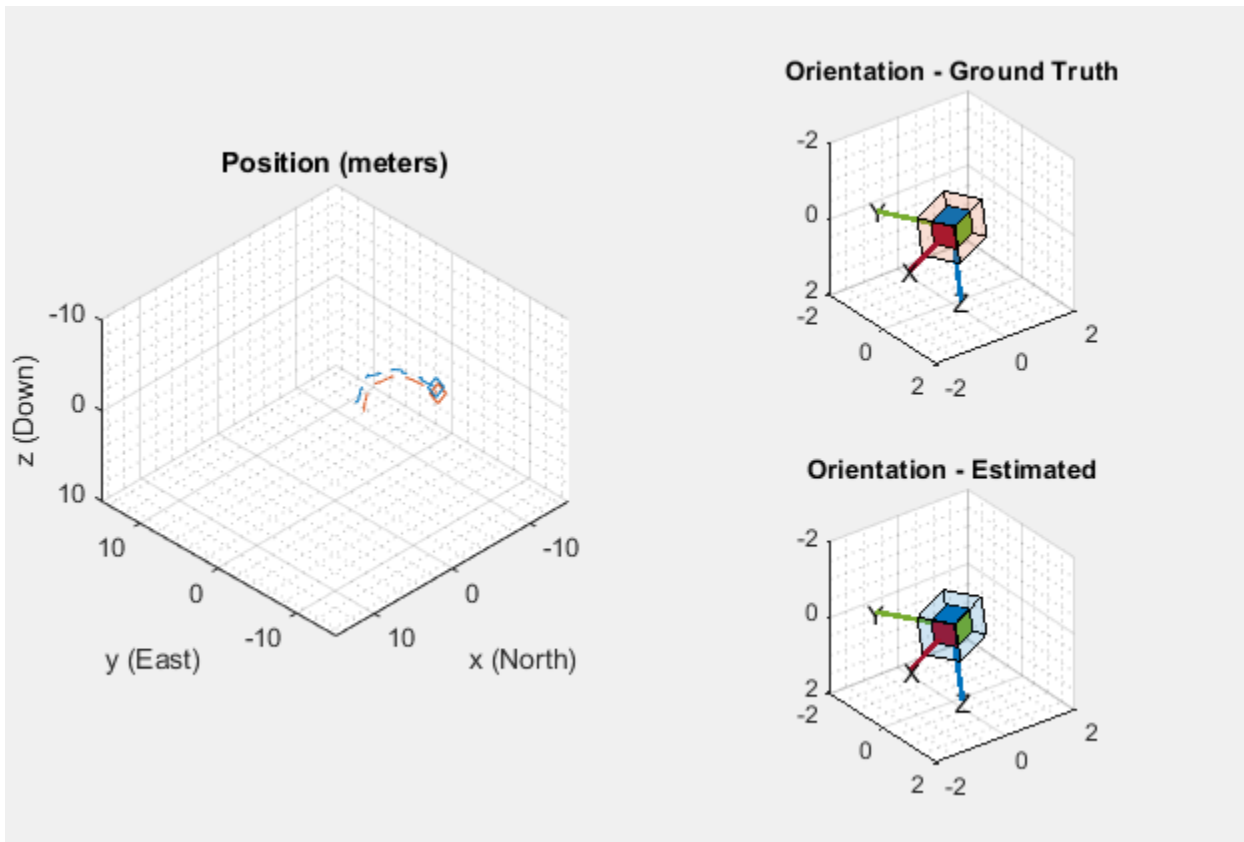
        % Update the pose viewer.
        if usePoseView
            posescope(pqpos(fcnt,:), pqorient(fcnt), trajPos(fcnt,:), ...
                trajOrient(fcnt,:) );
        end
        fcnt = fcnt + 1;
    end

    % 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 = pqpPos(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.
```

```
quadd = 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: 0.50 , Y: 0.79, Z: 0.65 (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)));
    1.45 (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 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*eye(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
    errscape = 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.

```
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);

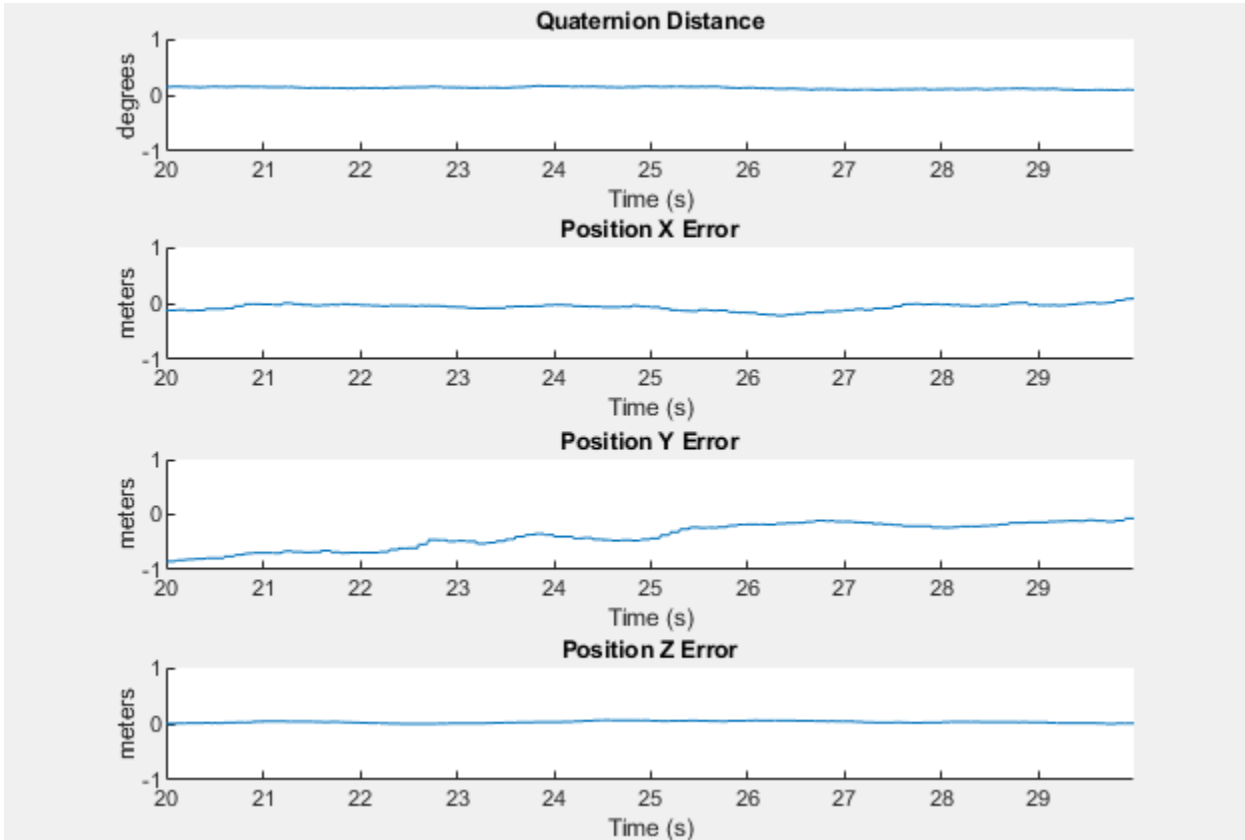
            % Compute the errors and plot.
            if useErrScope
                orientErr = rad2deg( ...
                    dist(estOrientation(idx,:), trueOrientation(idx,:)));
                posErr = estPosition(idx,:) - truePosition(idx,:);
                errsScope(orientErr, posErr(1), posErr(2), posErr(3));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

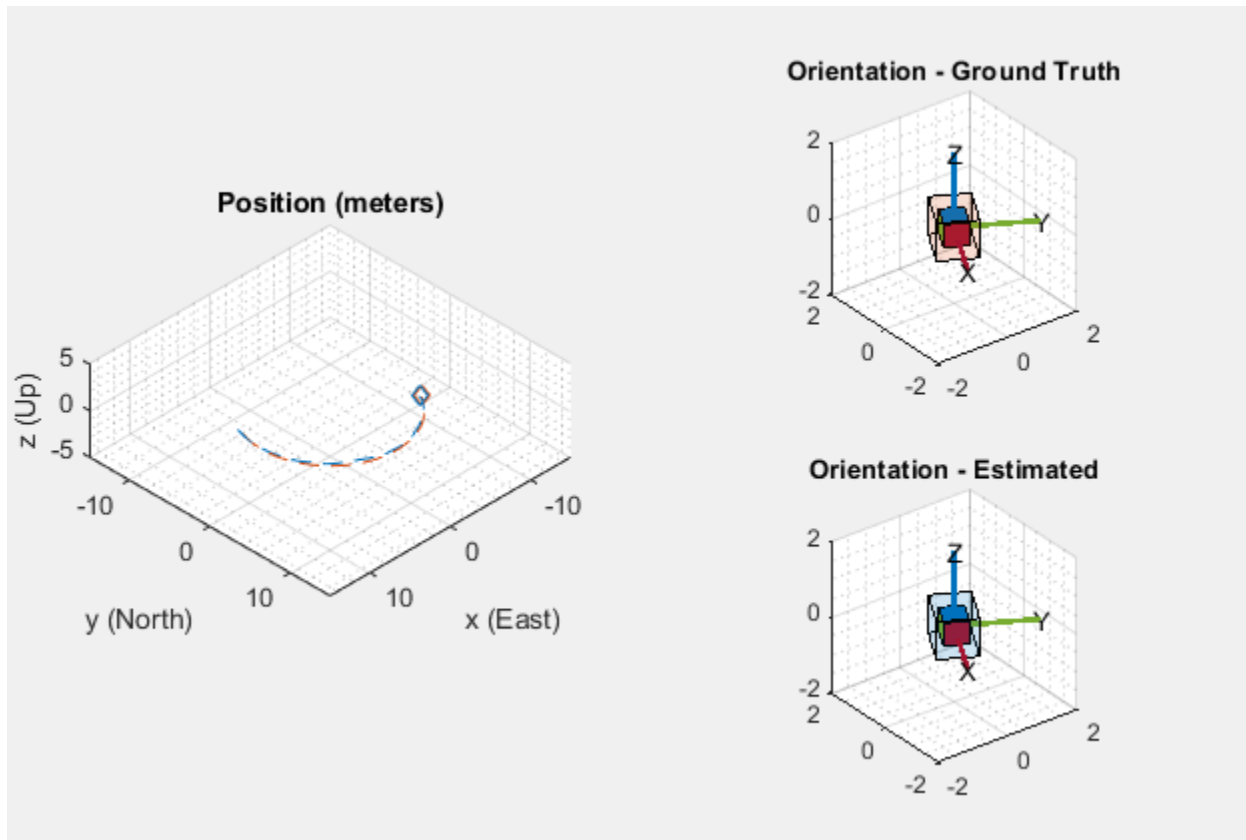
            % Update the pose viewer.
            if usePoseView
                viewer(estPosition(idx,:), estOrientation(idx,:), ...
                    truePosition(idx,:), estOrientation(idx,:));
            end
        end
    end
end

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
```





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.
```

```
quadd = rad2deg(dist(estOrientation, trueOrientation));
```

```
% 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: 1.16 , Y: 0.98, Z: 0.03 (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)));
```

```
0.11 (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.

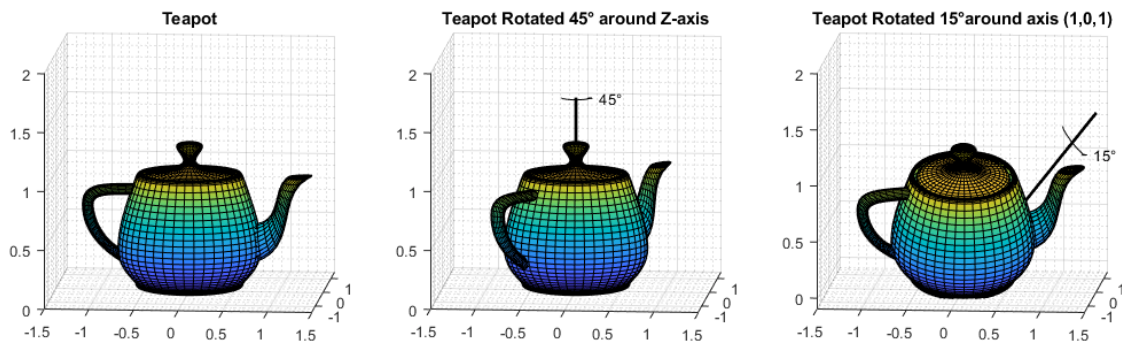
The `HelperDrawRotation` class is used to illustrate several portions of this example.

```
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.

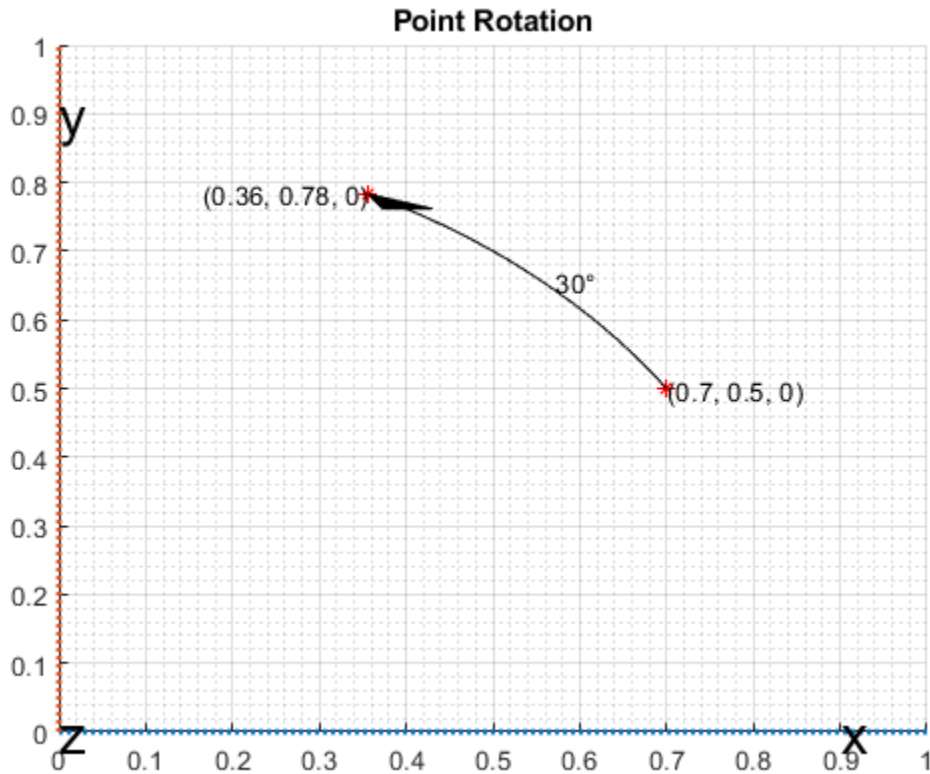
```
dr.drawTeapotRotations;
```



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.

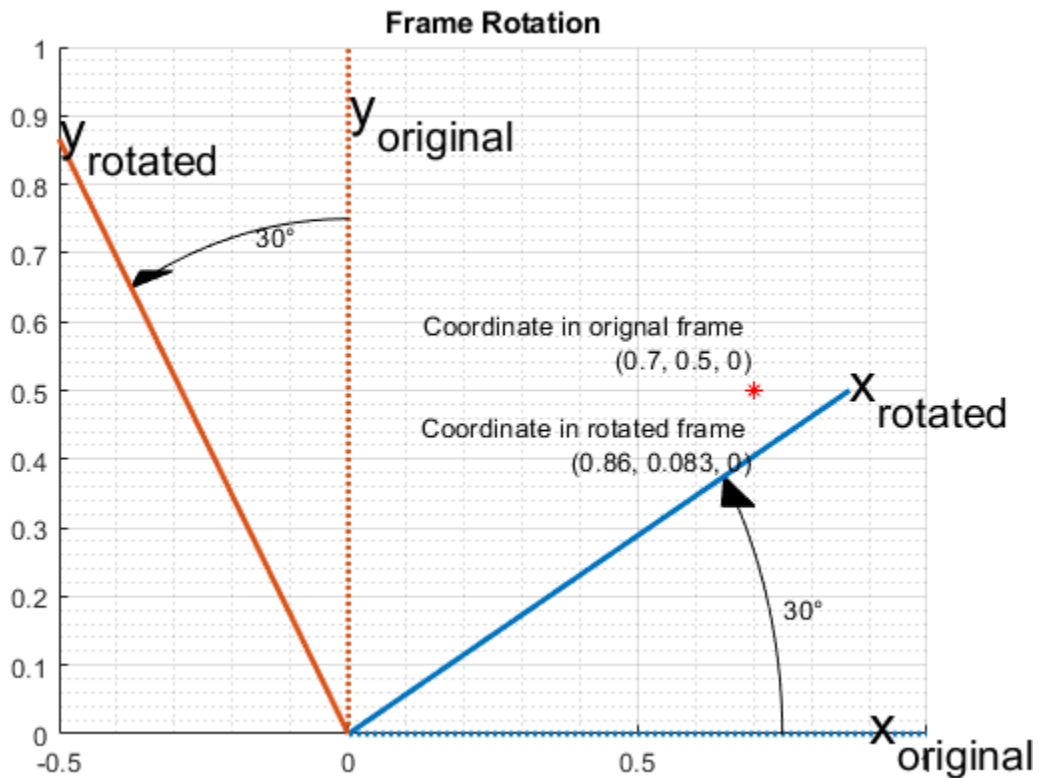
```
figure;
dr.draw2DPointRotation(gca);
```



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);
```

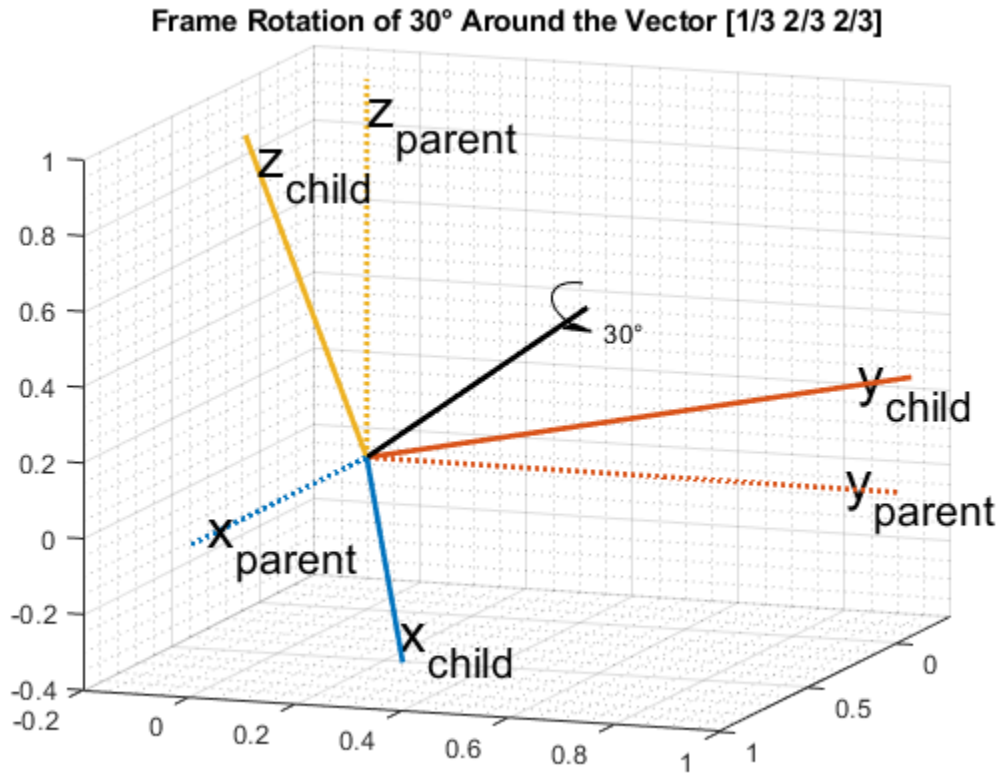


Orientation

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.draw3D0rientation(gca, [1/3 2/3 2/3], 30);
```



Quaternions

Quaternions are numbers of the form

$$a + bi + cj + dk$$

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

$$\cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)(xi + yj + zk)$$

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

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

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 $p^{-1}q$.

$p \setminus q$

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:

$$qv_{quat}q^*$$

where v_{quat} is

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

and q^* 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

$$q^* v_{quat} q$$

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)
```

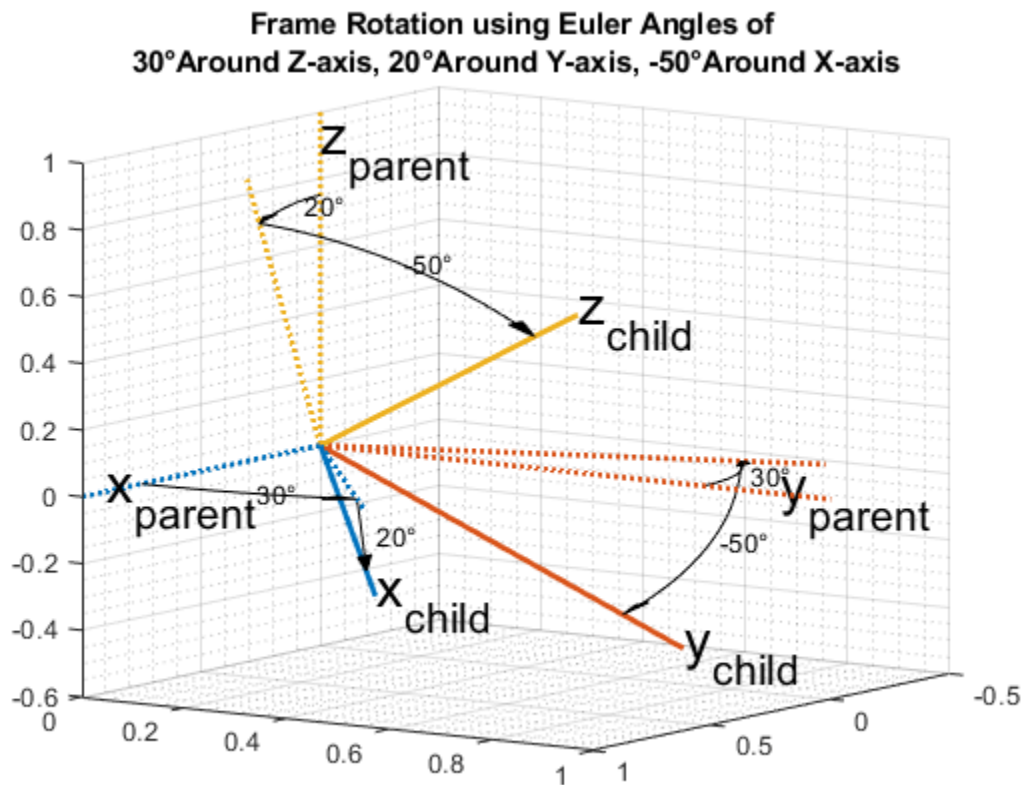
```
ans =
    0.7000    0.5000    0
```

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);
```



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 π .

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:

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

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

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

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

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 q_0 to q_1 , 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 q_0 and q_1 , 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 q_0 and q_1 , 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.

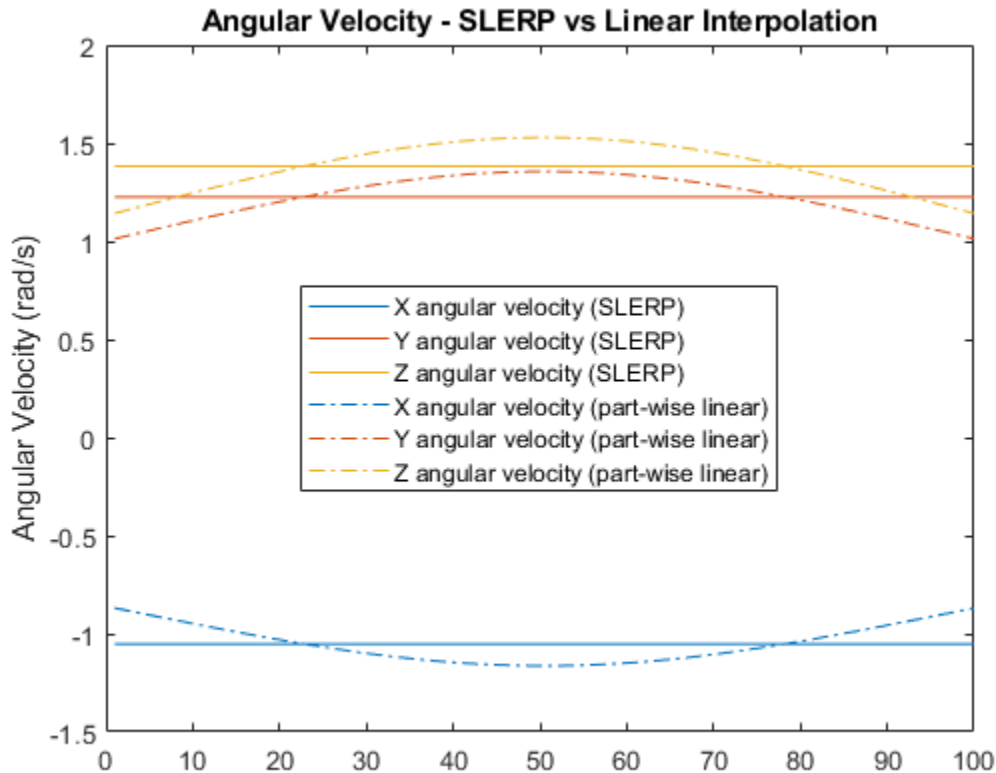
```
trajLerped = normalize( quaternion(partsLinInterp) );
```

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.

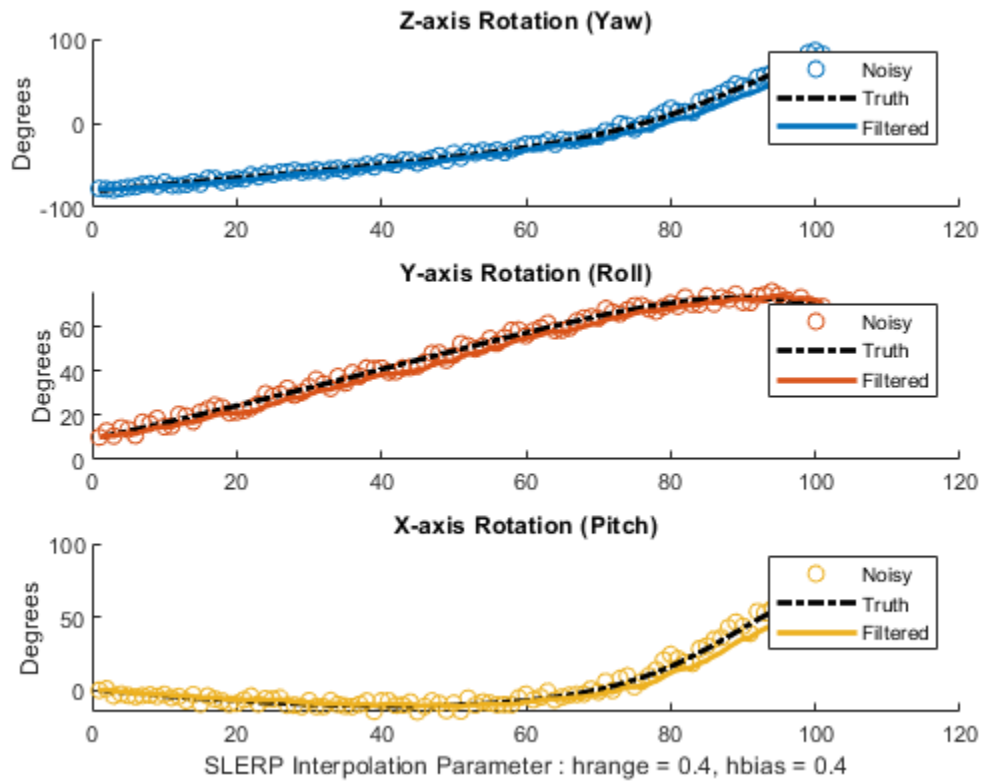
```
y = trajNoisy(1); % initial filter state
qout = zeros(size(y), 'like', y); % preallocate filter output
qout(1) = y;
```

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 SIGGRAPH 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);

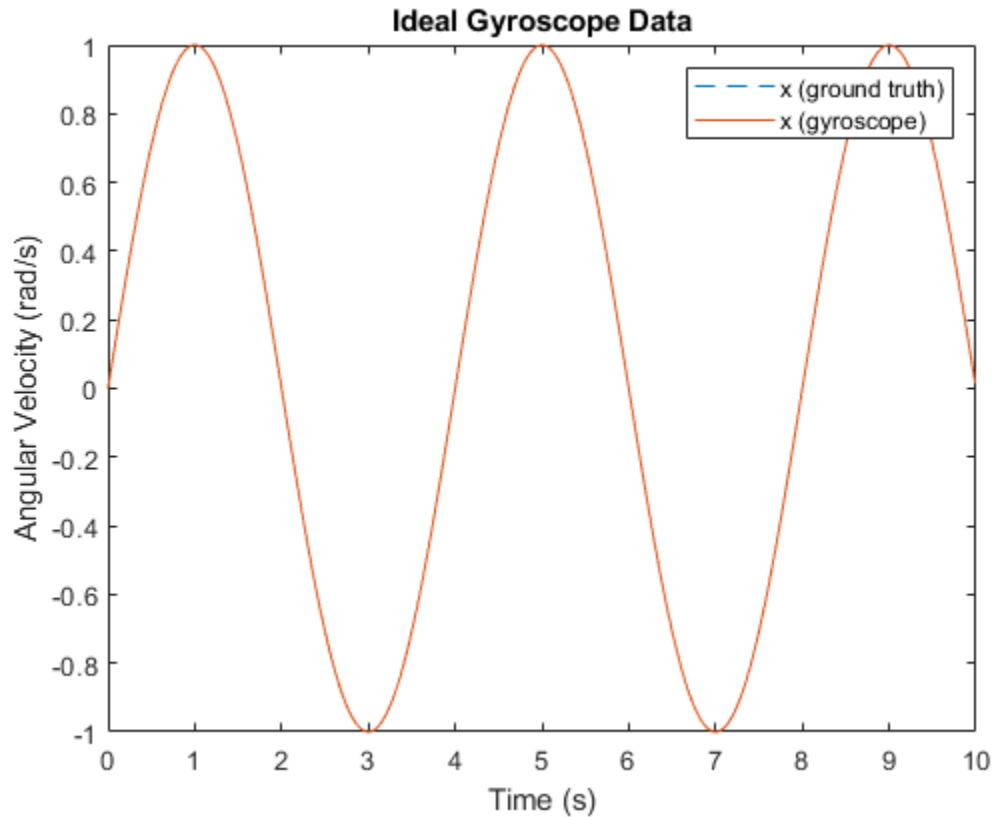
figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Ideal Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')

params =
    gyroparams with properties:

    MeasurementRange: Inf          rad/s
    Resolution: 0                (rad/s)/LSB
    ConstantBias: [0 0 0]        rad/s
    AxesMisalignment: [3x3 double] %

    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

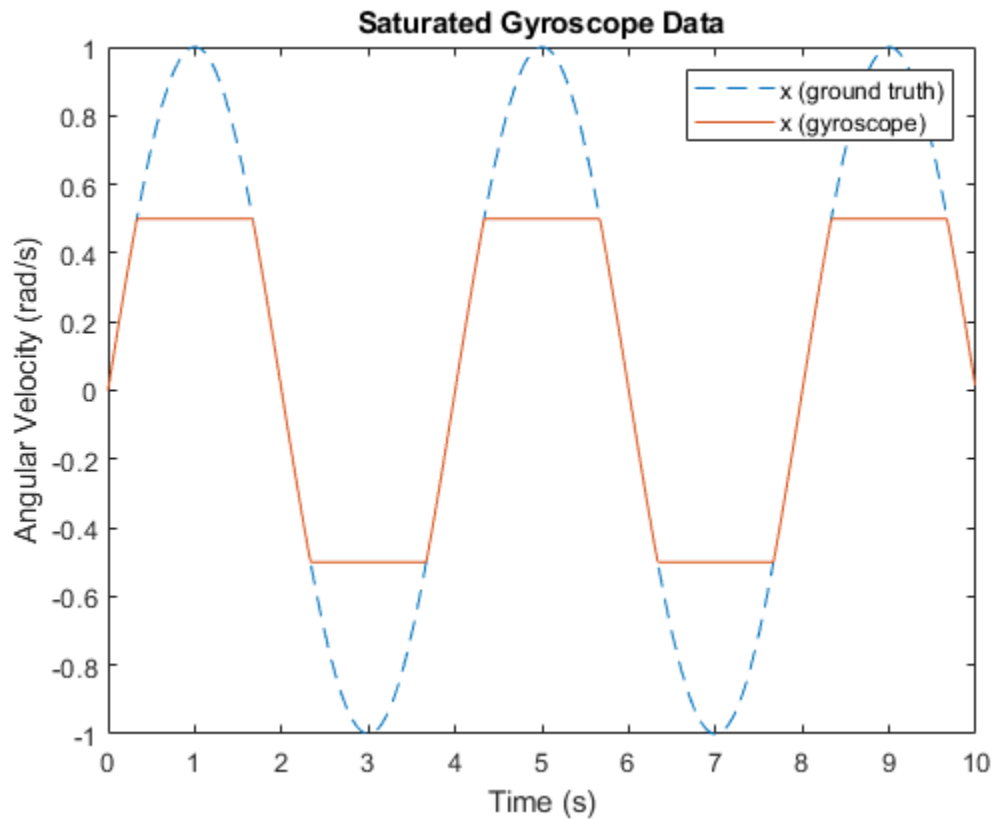
The following parameters model hardware limitations or defects. Some can be corrected through calibration.

`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)')
```

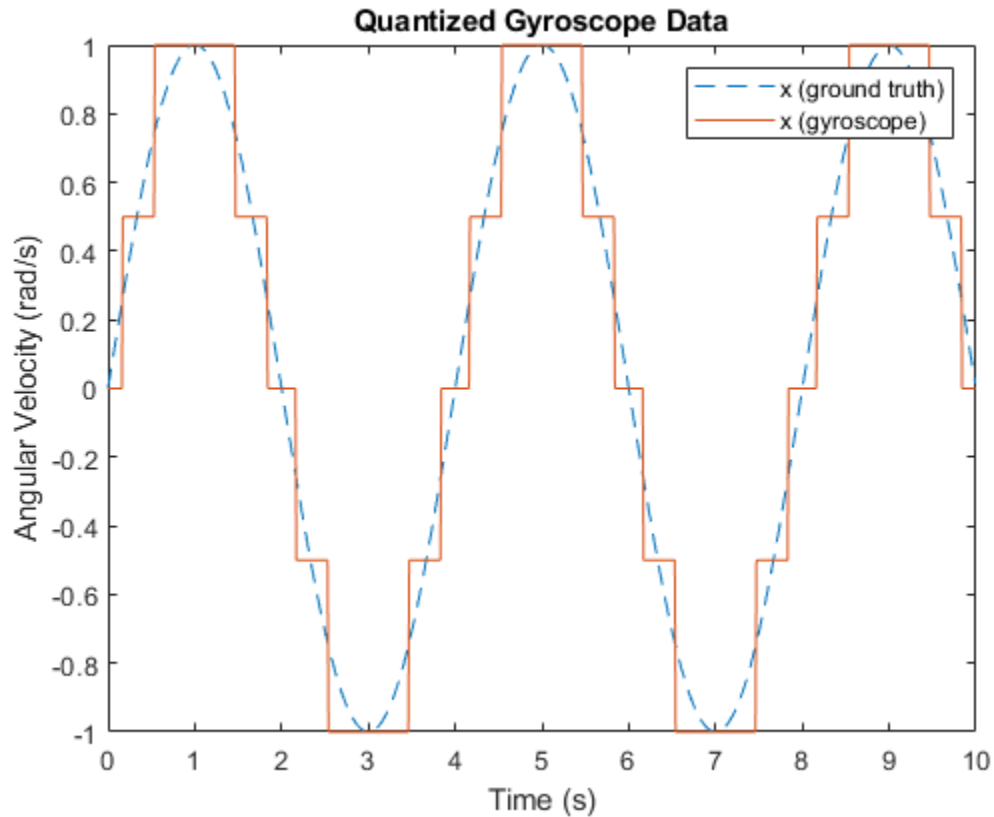


Resolution affects the step size of the digital measurements. Use this parameter to model the quantization effects from the analog-to-digital converter (ADC). The effect is shown by increasing the parameter to a much larger value than is typical.

```
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)')
```

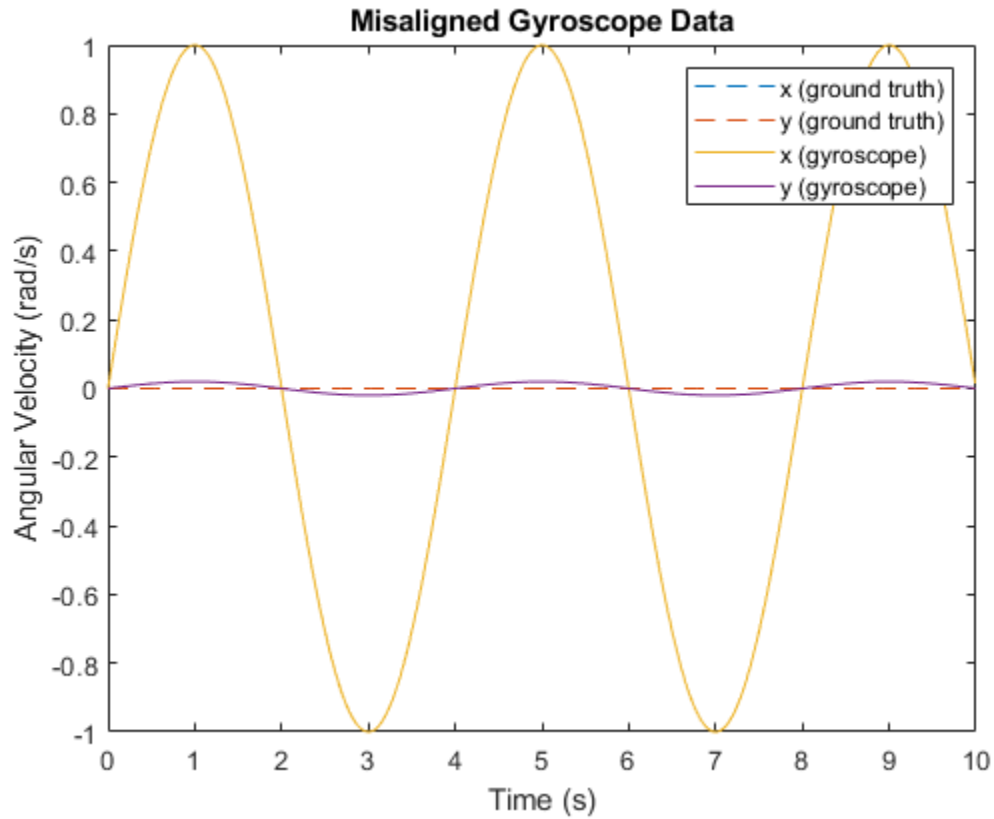



`AxesMisalignment` is the amount of skew in the sensor axes. This skew normally occurs when the sensor is mounted to the PCB and can be corrected through calibration. The effect is shown by skewing the x-axis slightly and plotting both the x-axis and y-axis.

```
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)')
```

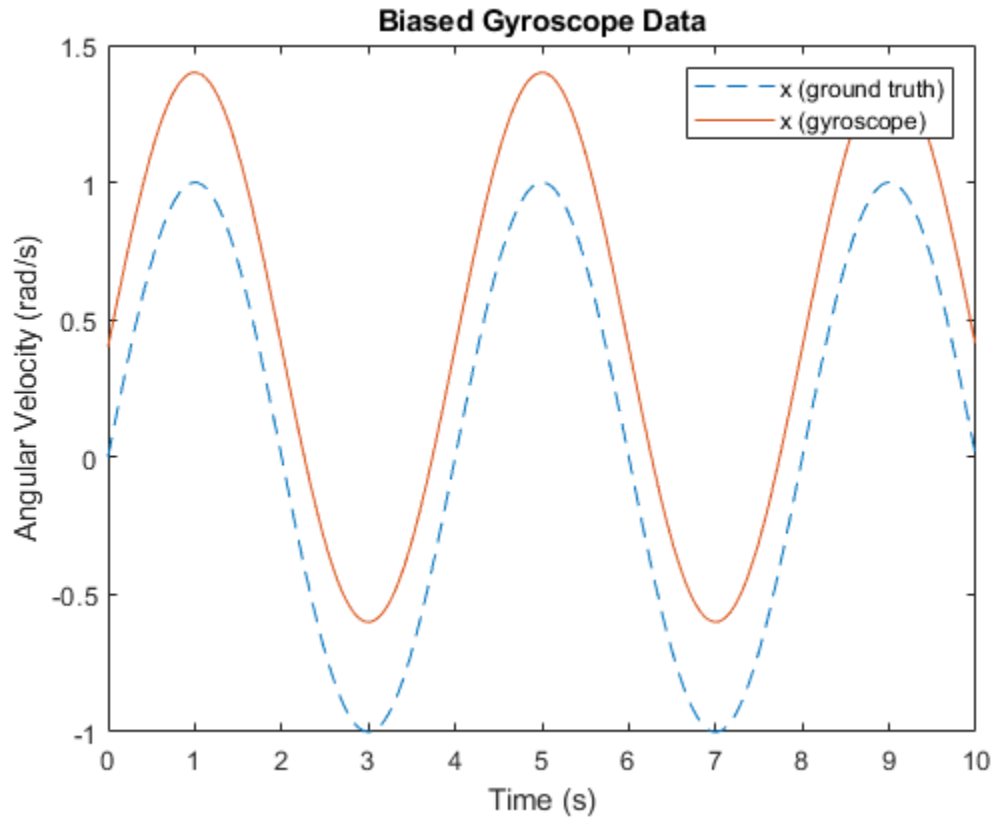


ConstantBias occurs in sensor measurements due to hardware defects. Since this bias is not caused by environmental factors, such as temperature, it can be corrected through calibration.

```
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)')
```



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 6-97 example.

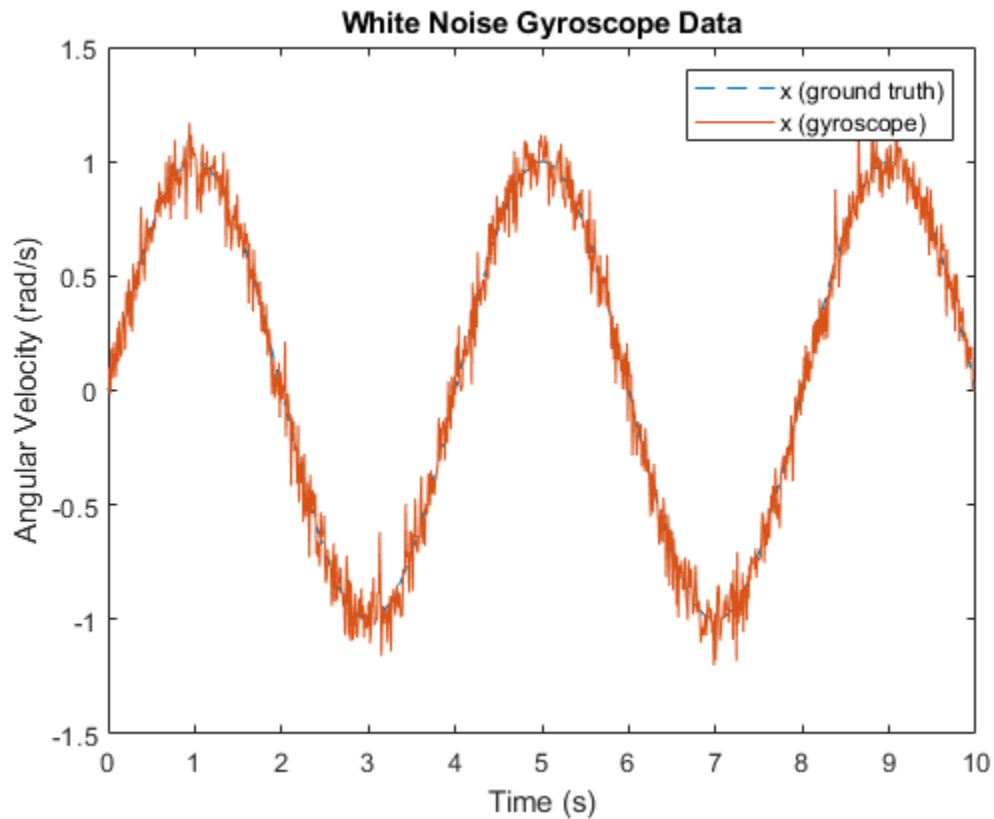
NoiseDensity is the amount of white noise in the sensor measurement. It is sometimes called angle random walk for gyroscopes or velocity random walk for accelerometers.

```
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)')
```

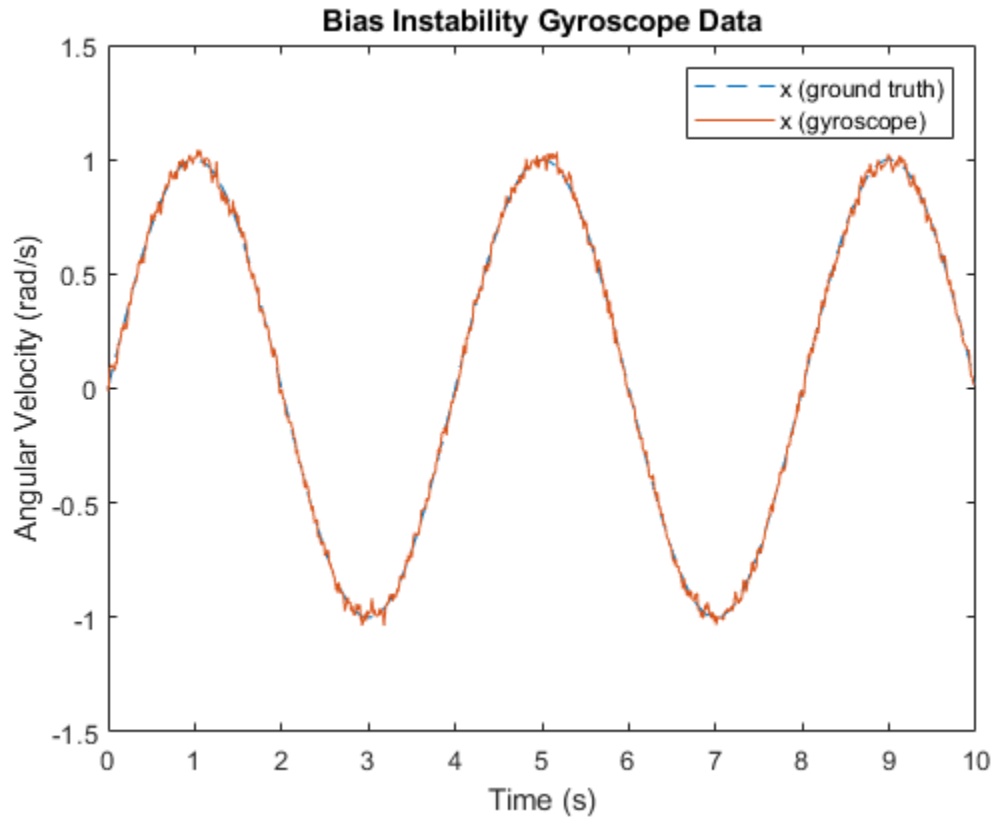


BiasInstability is the amount of pink or flicker noise in the sensor measurement.

```
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)')
```

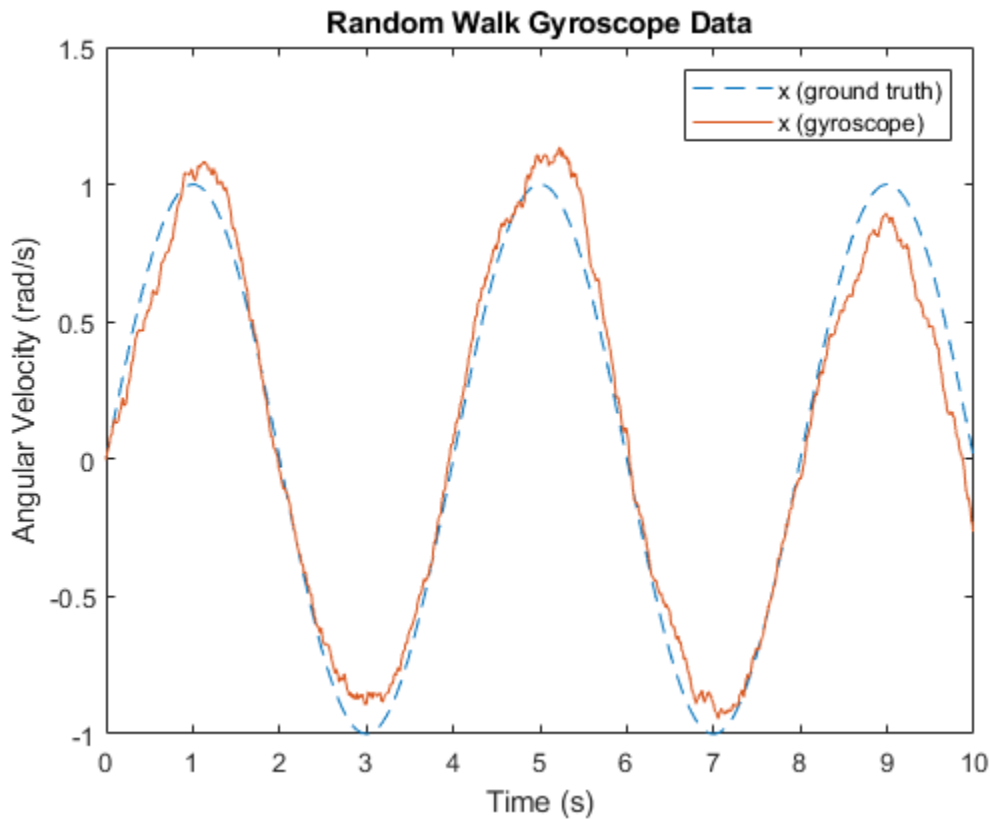


RandomWalk is the amount of Brownian noise in the sensor measurement. It is sometimes called rate random walk for gyroscopes or acceleration random walk for accelerometers.

```
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

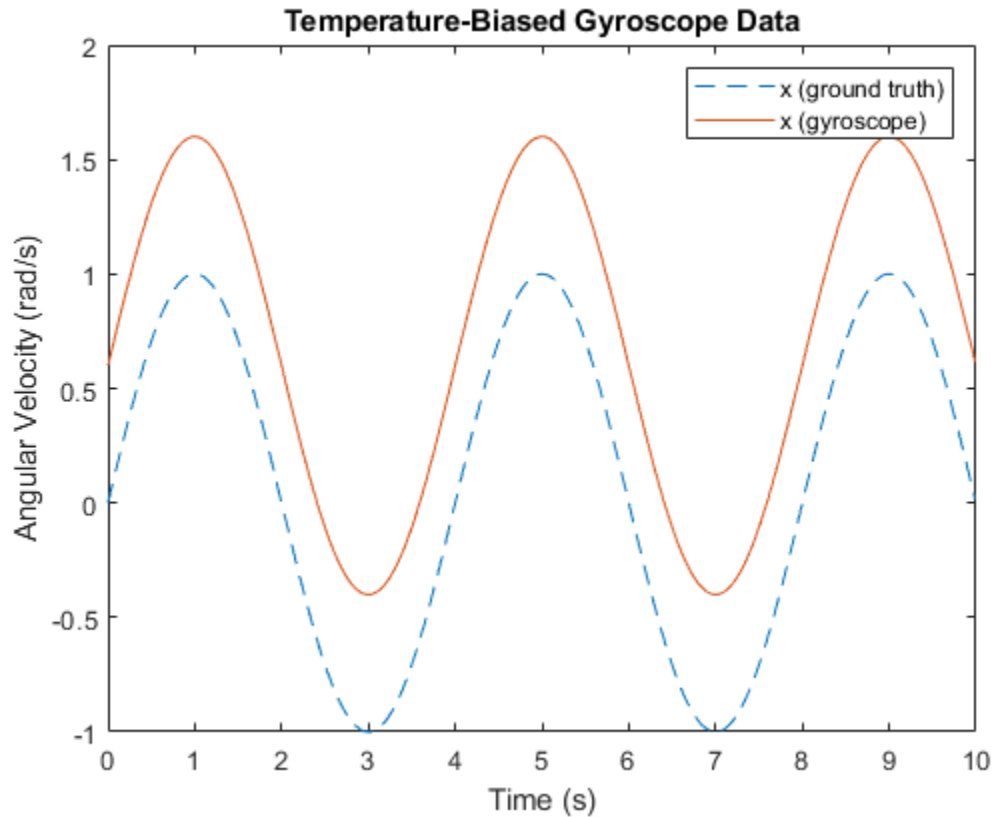
The following parameters model noise that arises from changes to the environment of the sensor.

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.

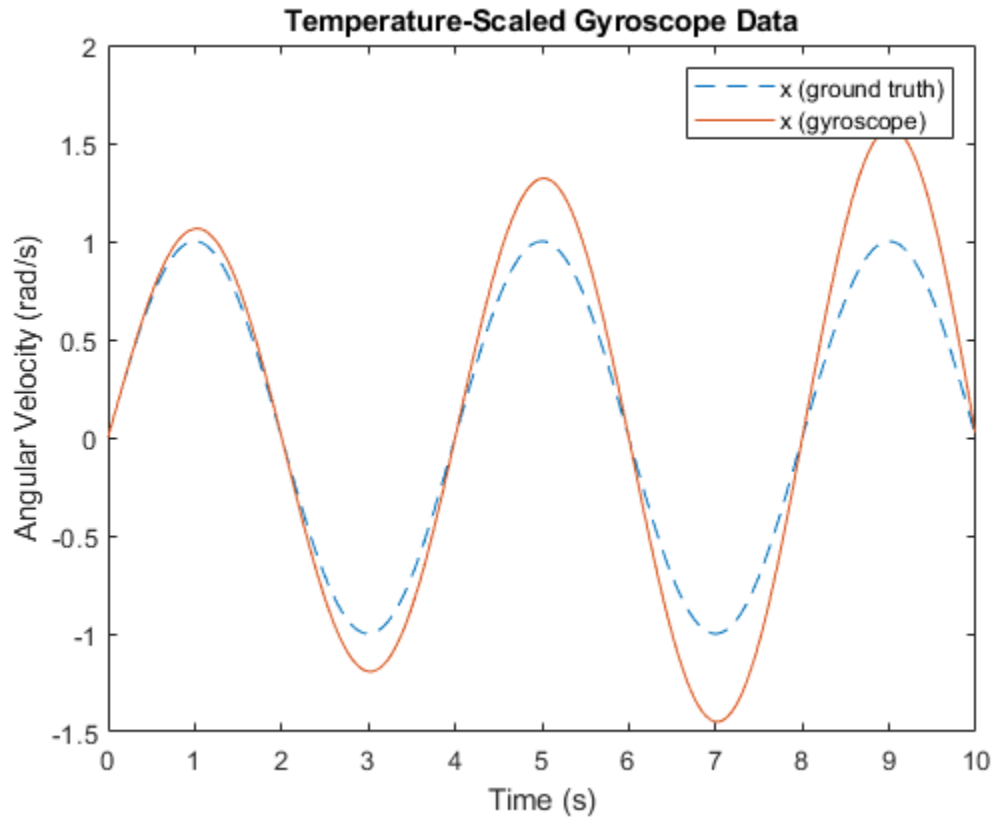
```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.TemperatureScaleFactor = 3.2; % %/(degrees C)

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)')
```



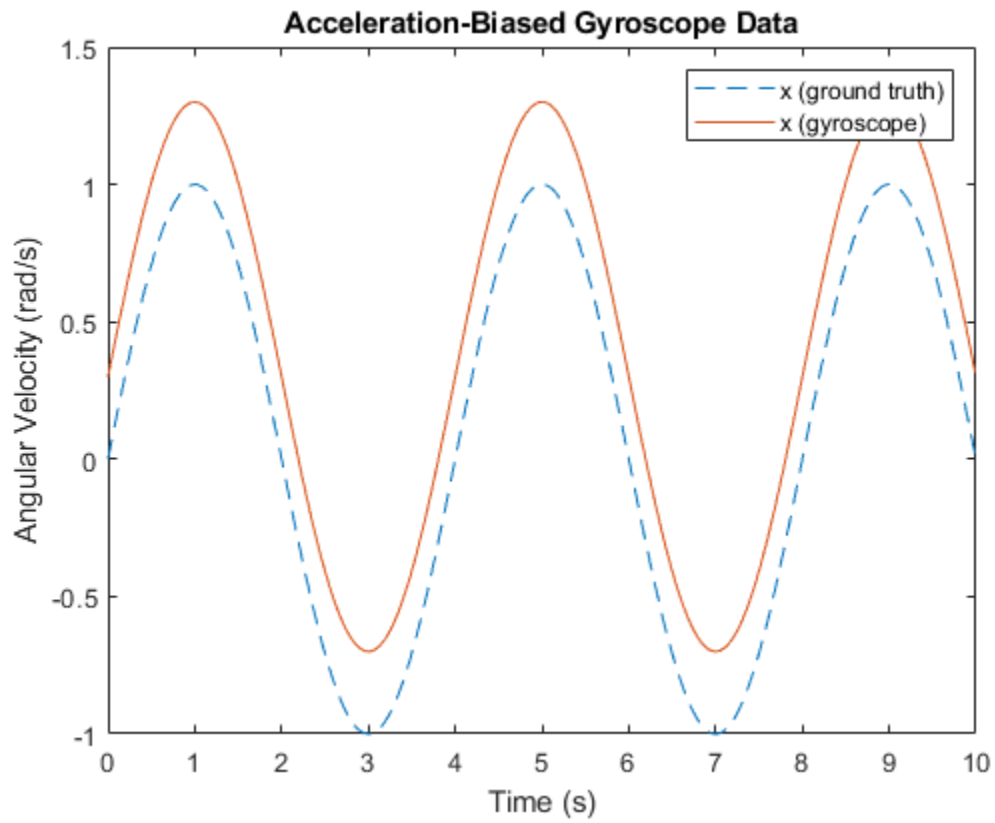
AccelerationBias is the bias added to the gyroscope measurement due to linear accelerations. This parameter is specific to the gyroscope. This bias is shown by setting the parameter to a non-zero value and using a non-zero input acceleration.

```
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)')
```


**See Also**

[imuSensor](#) | [gyroparams](#) | [accelparams](#) | [magparams](#)

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'
      MaxNumTracks: 100
      MaxNumDetections: Inf
      MaxNumSensors: 20
      Assignment: 'Auction'
  AssignmentThreshold: [30 Inf]
  AssignmentClustering: 'off'
      OOSMHandling: 'Terminate'
      TrackLogic: 'History'
  ConfirmationThreshold: [2 3]
  DeletionThreshold: [5 5]
      HasCostMatrixInput: false
  HasDetectableTrackIDsInput: false
      StateParameters: [1x1 struct]
      NumTracks: 0
  NumConfirmedTracks: 0
```

```

ClassFusionMethod: 'None'

EnableMemoryManagement: false

```

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
ObjectClassParameters: []
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
ObjectClassProbabilities: 1
          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
ObjectClassProbabilities: 1
        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
ObjectClassProbabilities: 1
        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'
    MaxNumTracks: 100
    MaxNumDetections: Inf
    MaxNumSensors: 20

    Assignment: 'Auction'
    AssignmentThreshold: [30 Inf]
    AssignmentClustering: 'off'

    OOSMHandling: 'Terminate'

    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

    ClassFusionMethod: 'None'

    EnableMemoryManagement: false
```

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.

```
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  
      ObjectClassProbabilities: 1  
      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  
      ObjectClassProbabilities: 1  
      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
    ObjectClassProbabilities
    TrackLogic
    TrackLogicState
    IsConfirmed
    IsCoasted
    IsSelfReported
    ObjectAttributes

```

```
allTracks =
```

```
2x1 objectTrack array with properties:
```

```

    TrackID
    BranchID
    SourceIndex
    UpdateTime
    Age
    State
    StateCovariance
    StateParameters
    ObjectClassID
    ObjectClassProbabilities
    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
ObjectClassProbabilities: 1
        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 `initcvkf` (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 = 'initekfirm'

```

```

tracker =

```

```

    trackerGNN with properties:

```

```

        TrackerIndex: 0
    FilterInitializationFcn: 'initekfirm'
        MaxNumTracks: 100
        MaxNumDetections: Inf
        MaxNumSensors: 20

        Assignment: 'Auction'
    AssignmentThreshold: [30 Inf]
    AssignmentClustering: 'off'

        OOSMHandling: 'Terminate'

```



```

        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

    ClassFusionMethod: 'None'

    EnableMemoryManagement: false

```

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
    HasMeasurementWrapping: 1
        MeasurementNoise: [3x3 double]
    HasAdditiveMeasurementNoise: 1

    MaxNumOOSMSteps: 0

    EnableSmoothing: 0

```

trackingEKF with properties:

```

                State: [9x1 double]
                StateCovariance: [9x9 double]

                StateTransitionFcn: @constacc
                StateTransitionJacobianFcn: @constaccjac
                ProcessNoise: [3x3 double]
                HasAdditiveProcessNoise: 0

                MeasurementFcn: @cameas
                MeasurementJacobianFcn: @cameasjac
                HasMeasurementWrapping: 1
                MeasurementNoise: [3x3 double]
                HasAdditiveMeasurementNoise: 1

                MaxNum00SMSteps: 0

                EnableSmoothing: 0

trackingEKF with properties:

                State: [7x1 double]
                StateCovariance: [7x7 double]

                StateTransitionFcn: @constturn
                StateTransitionJacobianFcn: @constturnjac
                ProcessNoise: [4x4 double]
                HasAdditiveProcessNoise: 0

                MeasurementFcn: @ctmeas
                MeasurementJacobianFcn: @ctmeasjac
                HasMeasurementWrapping: 1
                MeasurementNoise: [3x3 double]
                HasAdditiveMeasurementNoise: 1

                MaxNum00SMSteps: 0

                EnableSmoothing: 0

```

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 = 'initcvkf';
tracker.HasDetectableTrackIDsInput = true % Allows the tracker to get input about the track detection

```

```

% 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: 'initcvckf'
        MaxNumTracks: 100
        MaxNumDetections: Inf
        MaxNumSensors: 20

        Assignment: 'Auction'
    AssignmentThreshold: [30 Inf]
    AssignmentClustering: 'off'

        OOSMHandling: 'Terminate'

        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

        ClassFusionMethod: 'None'

    EnableMemoryManagement: false

```

```

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
ObjectClassProbabilities: 1
        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
ObjectClassProbabilities: 1
        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
ObjectClassProbabilities: 1
        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 second
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
```

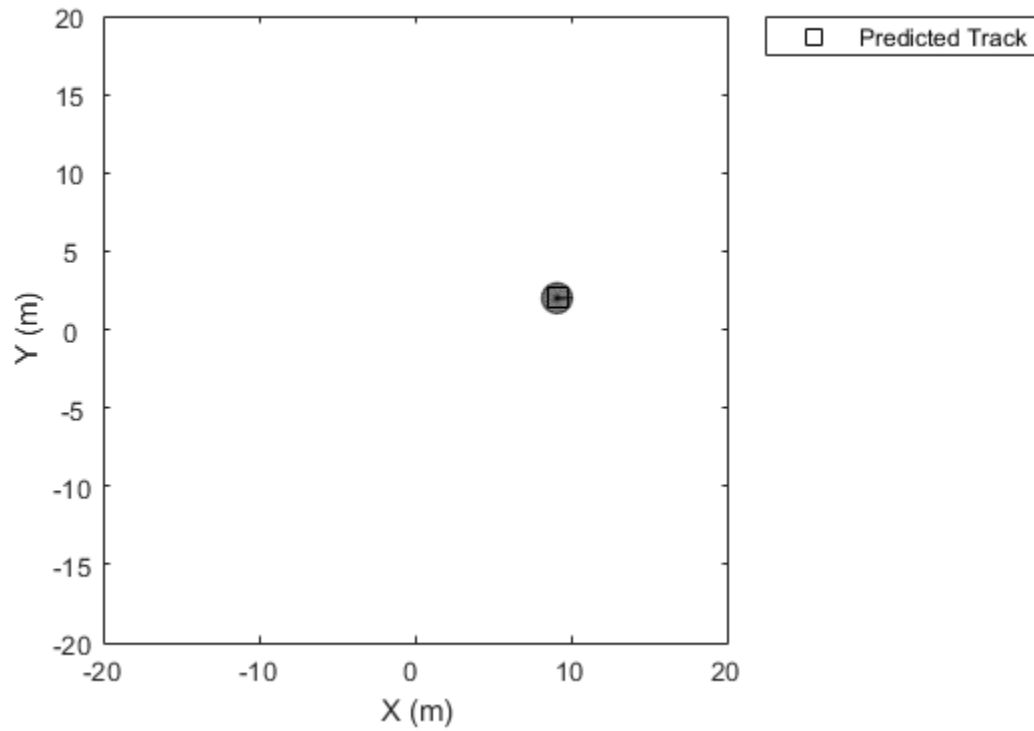
```
ans =
```

```
10
0
0
0
0
0
```

You can use the `predictTracksToTime` method to visualize the predicted state of the tracks.

```
% 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 $N_{\max} = \max(N_{\text{conf}}, N_{\text{del}})$. In this example, N_{\max} 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, β , 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(confFlag)])
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
```

```
rng(r); % Return the random seed to its previous setting
```

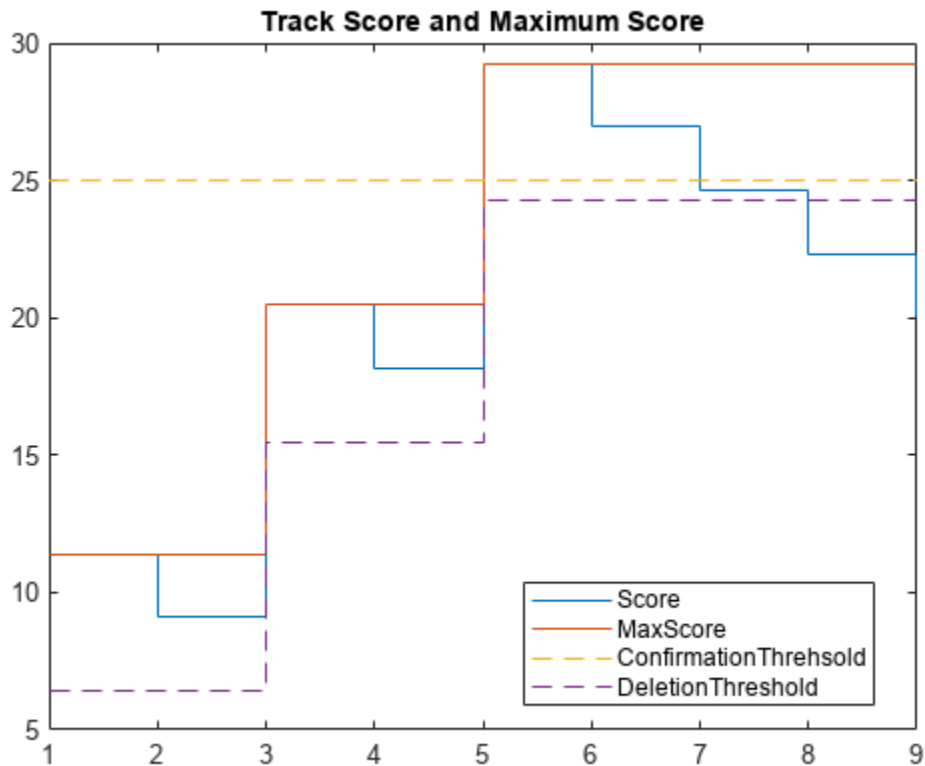
Notice how the track score increases with every successful update and decreases with every missed detection. Once the track score is higher than the ConfirmationThreshold, the checkConfirmation function returns true, meaning that the track is confirmed now.

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.

```
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

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'
      MaxNumTracks: 100
      MaxNumDetections: Inf
      MaxNumSensors: 20

      Assignment: 'Auction'
  AssignmentThreshold: [30 Inf]
  AssignmentClustering: 'off'

      OOSMHandling: 'Terminate'

      TrackLogic: 'History'
  ConfirmationThreshold: [3 5]
      DeletionThreshold: [5 6]
```

```

        HasCostMatrixInput: false
        HasDetectableTrackIDsInput: false
        StateParameters: [1x1 struct]

        NumTracks: 0
        NumConfirmedTracks: 0

        ClassFusionMethod: 'None'

        EnableMemoryManagement: false

```

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
    ObjectClassProbabilities: 1
        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
ObjectClassProbabilities: 1
        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
ObjectClassProbabilities: 1
        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
ObjectClassProbabilities: 1
        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
ObjectClassProbabilities: 1
        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).

```

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
ObjectClassProbabilities: 1
        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

```

```
ObjectClassProbabilities: 1
    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
    ObjectClassProbabilities: 1
        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
    ObjectClassProbabilities
    TrackLogic
    TrackLogicState
    IsConfirmed
    IsCoasted
    IsSelfReported
    ObjectAttributes
```

```
confirmedTrack =
    0x1 objectTrack array with properties:
```

```
    TrackID
    BranchID
```


SourceIndex
UpdateTime
Age
State
StateCovariance
StateParameters
ObjectClassID
ObjectClassProbabilities
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
    SimulationStatus: NotStarted
           Platforms: {}
        SurfaceManager: [1x1 fusion.scenario.SurfaceManager]
```

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: [1x1 struct]
Mesh: [1x1 extendedObjectMesh]
Trajectory: [1x1 kinematicTrajectory]
PoseEstimator: [1x1 insSensor]
Emitters: {}
Sensors: {}
Signatures: {[1x1 rcsSignature] [1x1 irSignature] [1x1 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.

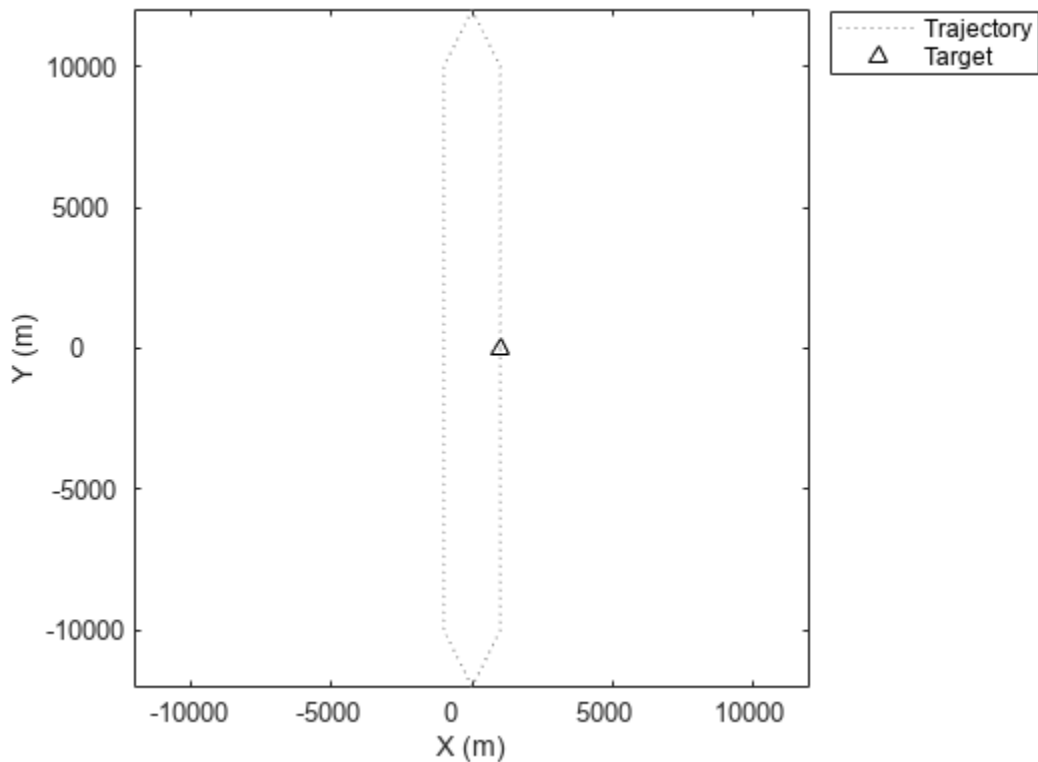
```

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);
```

Use a `fusionRadarSensor` in monostatic mode to implement 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 `fusionRadarSensor`. The available output frames are 'Body' (default), 'Scenario', 'Sensor rectangular', and 'Sensor spherical'.

```
radar1 = fusionRadarSensor(1, 'UpdateRate', 2.5, ...
    'MountingLocation', [0 0 -15], 'FieldOfView', [4;45], ...
    'MechanicalAzimuthLimits', [-60 60], 'MechanicalElevationLimits', [0 0], ...
    'HasElevation', true, 'FalseAlarmRate', 1e-7)
```

```
radar1 =
    fusionRadarSensor with properties:

        SensorIndex: 1
        UpdateRate: 2.5000
        DetectionMode: 'Monostatic'
        ScanMode: 'Mechanical'
        InterferenceInputPort: 0
        EmissionsInputPort: 0

        MountingLocation: [0 0 -15]
        MountingAngles: [0 0 0]

        FieldOfView: [4 45]
        LookAngle: [0 0]
        RangeLimits: [0 100000]

        DetectionProbability: 0.9000
        FalseAlarmRate: 1.0000e-07
        ReferenceRange: 100000

        TargetReportFormat: 'Clustered detections'

    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;
```

```

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);

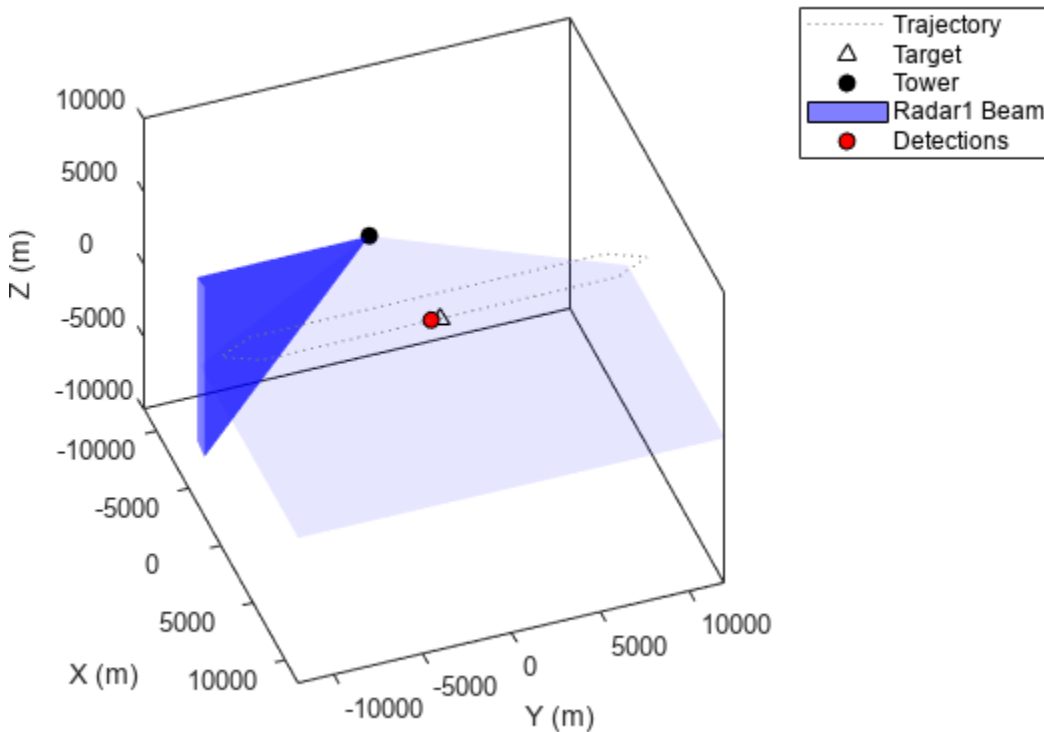
% Extract detection positions and transform them to the scenario frame.
detPos = zeros(numDets, 3);
for i=1:numDets
    detPos(i,:) = tower.Trajectory.Position + detections{i}.Measurement';
end

% 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 = fusionRadarSensor(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);

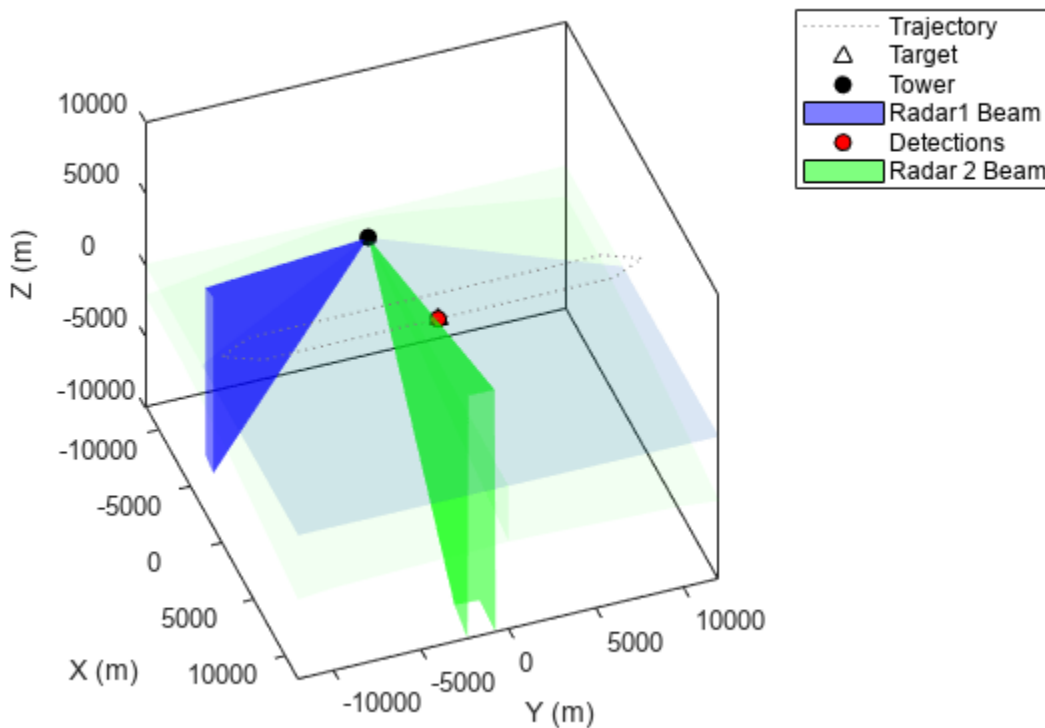
    % Plot the coverage of sensors.
    configs = coverageConfig(scene);
    plotCoverage(radar1Plotter, configs(1));
    plotCoverage(radar2Plotter, configs(2));

    % 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 = fusionRadarSensor(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);

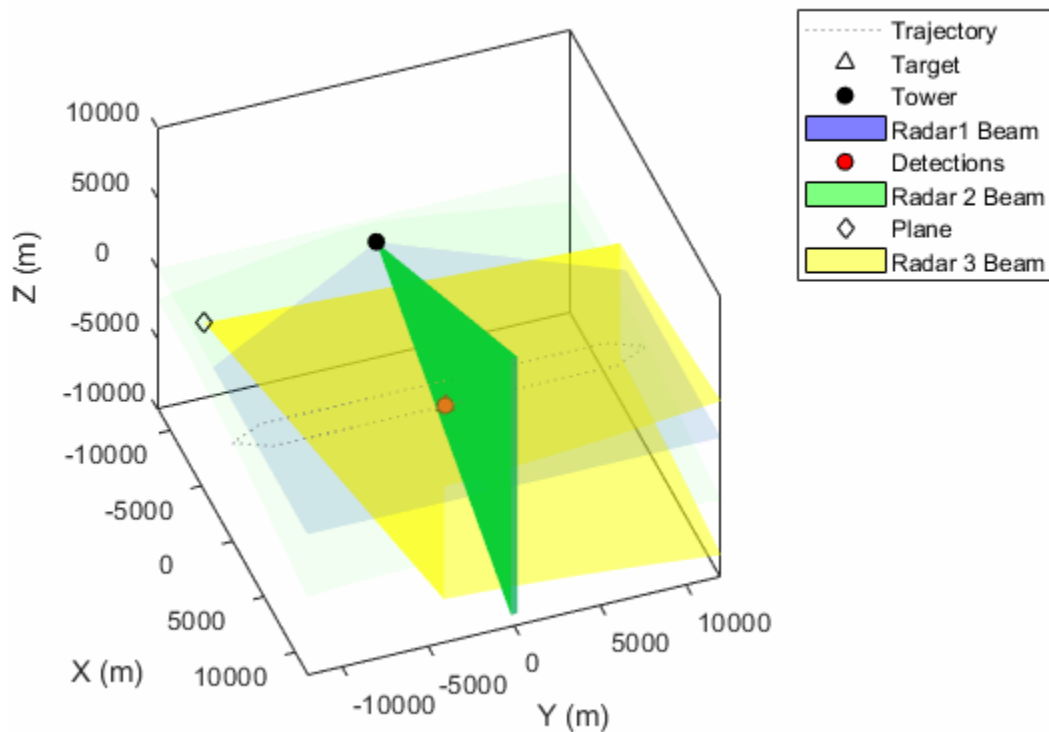
    % Plot the coverage of sensors.
    configs = coverageConfig(scene);
    plotCoverage(radar1Plotter, configs(1));
    plotCoverage(radar2Plotter, configs(2));
    plotCoverage(radar3Plotter, configs(3));

    % 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
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 $\tau_0, 2\tau_0, \dots, m\tau_0, (m < (L - 1)/2)$ 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:

$$\sigma^2(\tau) = \frac{1}{2\tau^2} \langle (\theta_{k+2m} - 2\theta_{k+m} + \theta_k)^2 \rangle$$

where $\tau = m\tau_0$ and $\langle \rangle$ is the ensemble average.

The ensemble average can be expanded to:

$$\sigma^2(\tau) = \frac{1}{2\tau^2(L - 2m)} \sum_{k=1}^{L-2m} (\theta_{k+2m} - 2\theta_{k+m} + \theta_k)^2$$

```
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.

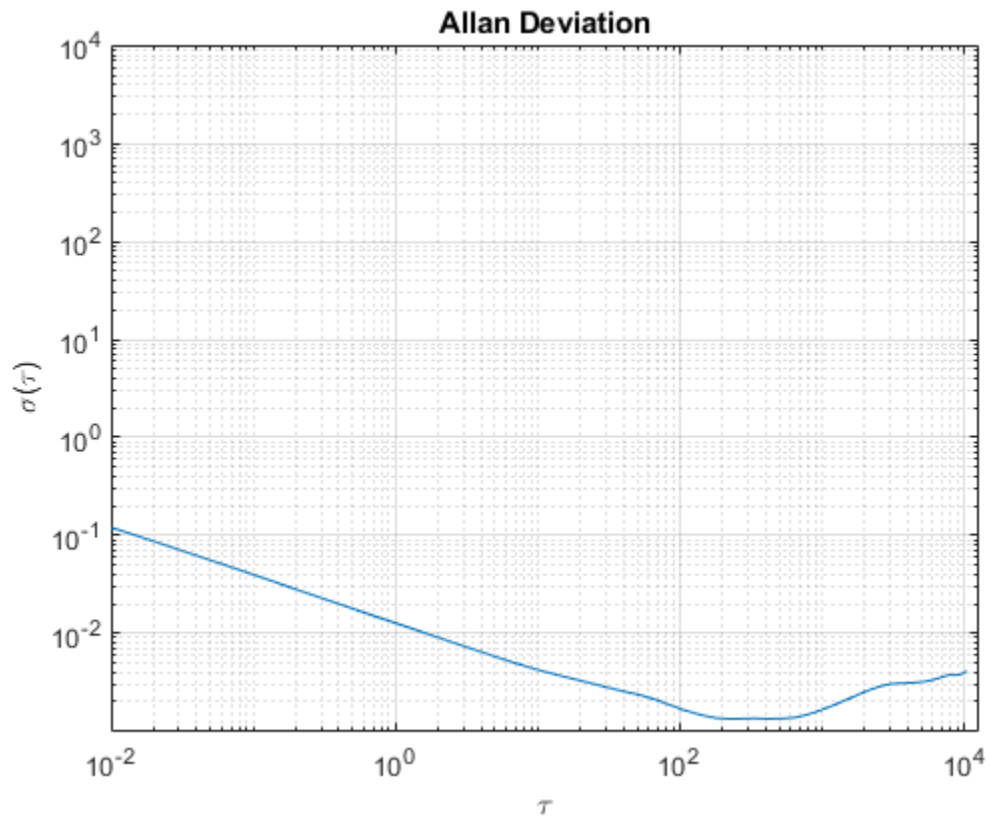
tau = m*t0;

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);
```

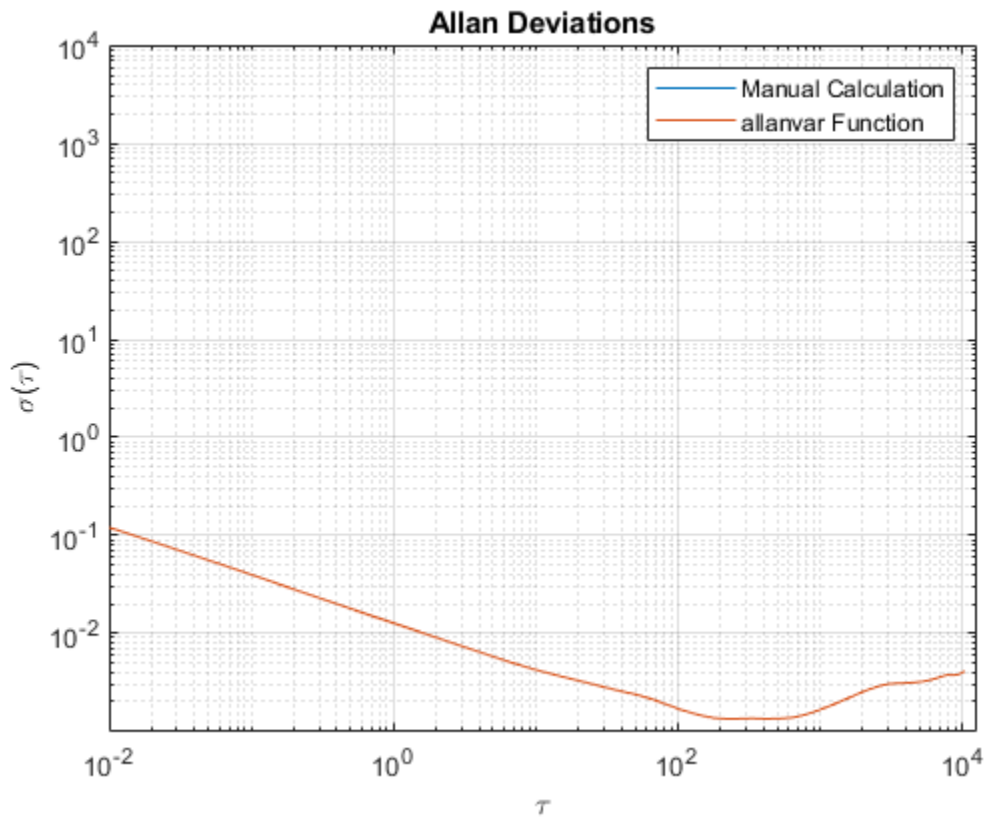
```
figure
loglog(tau, adev)
title('Allan Deviation')
xlabel('\tau');
ylabel('\sigma(\tau)')
grid on
axis equal
```



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
```



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 $\tau = 1$. The units of N are $(\text{rad/s})/\sqrt{\text{Hz}}$.

```
% 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.
```

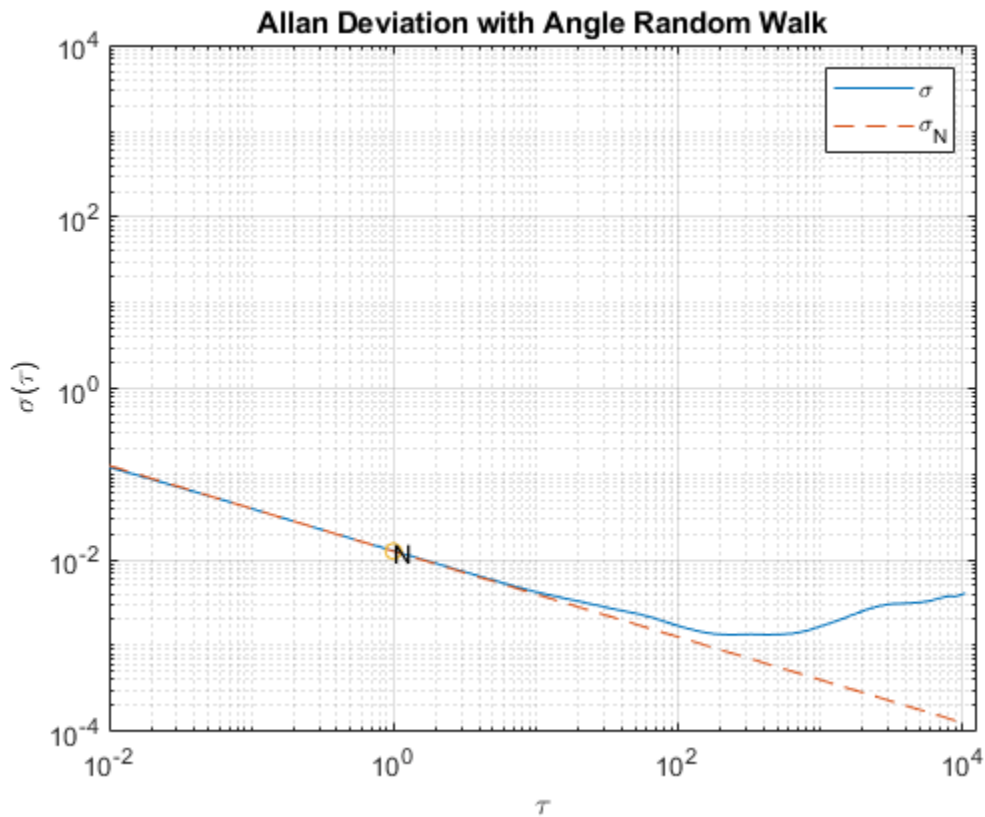
```
logN = slope*log(1) + b;
N = 10^logN
```

```
% 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:

$$S_{\Omega}(f) = \left(\frac{K}{2\pi}\right)^2 \frac{1}{f^2}$$

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 $\tau = 3$. The units of K are $(rad/s)\sqrt{Hz}$.

`% 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);

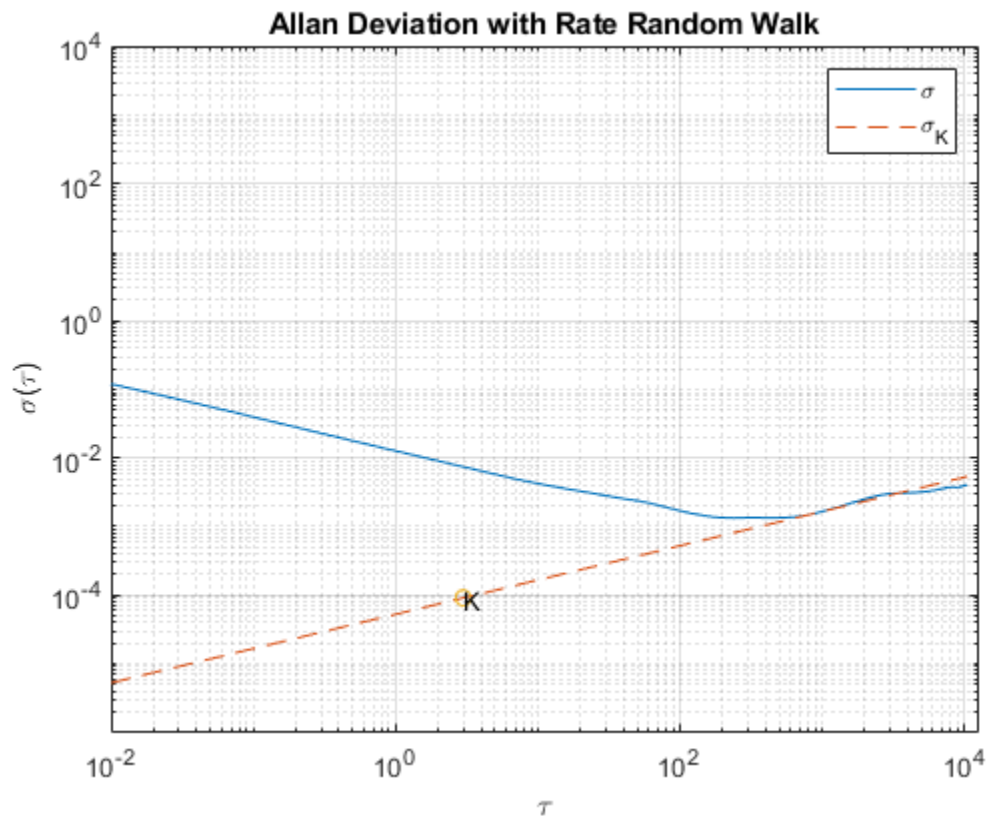
% Determine the rate random walk coefficient from the line.
logK = slope*log10(3) + b;
K = 10^logK

% 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



Bias Instability

The bias instability is characterized by the pink noise (flicker noise) spectrum of the gyroscope output. The PSD is represented by:

$$S_{\Omega}(f) = \begin{cases} (\frac{B^2}{2\pi})\frac{1}{f} & : f \leq f_0 \\ 0 & : f > f_0 \end{cases}$$

where

B = bias instability coefficient

f_0 = cut-off frequency

Substituting into the original PSD equation and performing integration yields:

$$\sigma^2(\tau) = \frac{2B^2}{\pi} [\ln 2 + -\frac{\sin^3 x}{2x^2} (\sin x + 4x \cos x) + Ci(2x) - Ci(4x)]$$

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 $\sqrt{\frac{2 \ln 2}{\pi}} \approx 0.664$. The units of B are *rad/s*.

% 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));
```

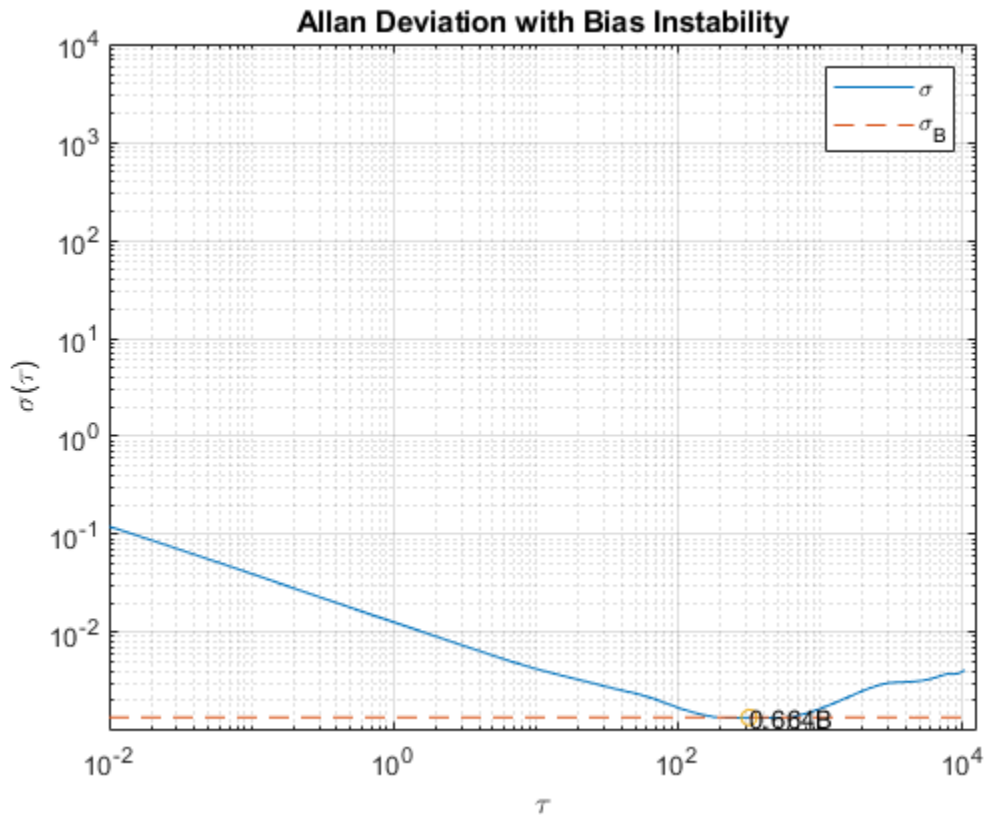
```

figure
loglog(tau, adev, tau, lineB, '--', tauB, scfB*B, 'o')
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



Now that all the noise parameters have been calculated, plot the Allan deviation with all of the lines used for quantifying the parameters.

```

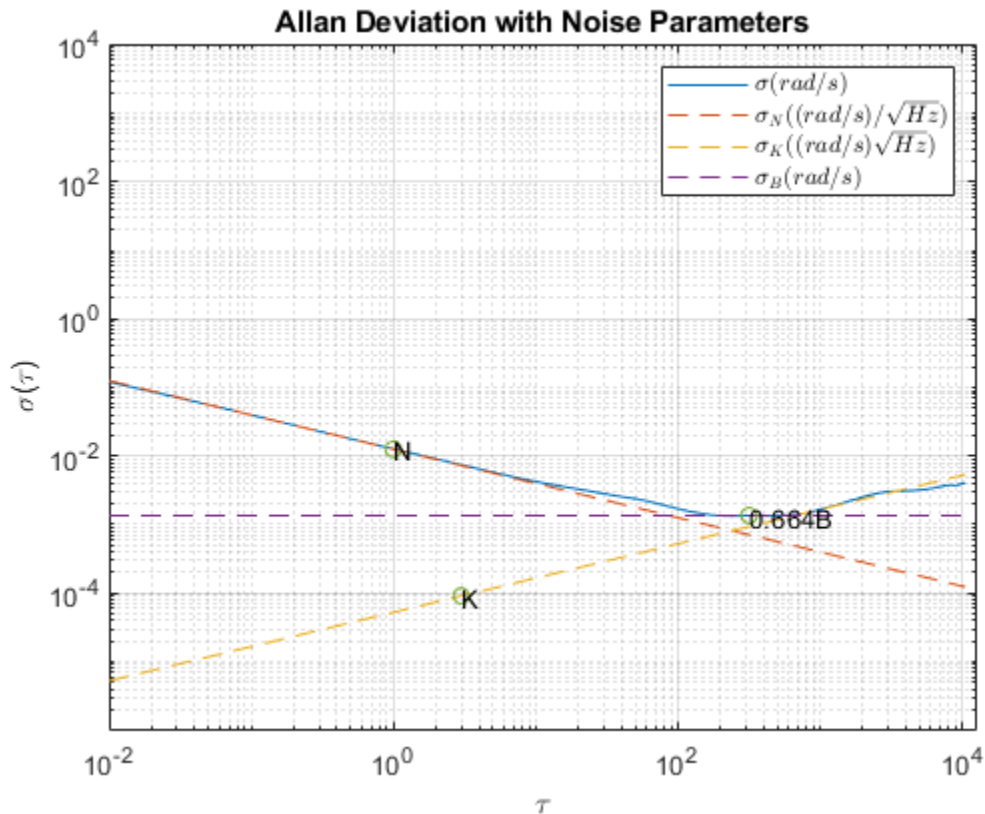
tauParams = [tauN, tauK, tauB];
params = [N, K, scfB*B];
figure
loglog(tau, adev, tau, [lineN, lineK, lineB], '--', ...
      tauParams, params, 'o')
title('Allan Deviation with Noise Parameters')
xlabel('\tau')
ylabel('\sigma(\tau)')
legend('\sigma (rad/s)', '\sigma_N ((rad/s)/\sqrt{Hz})$', ...

```

```

    '$\sigma_K ((rad/s)\sqrt{Hz})$', '$\sigma_B (rad/s)$', 'Interpreter', 'latex')
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;

```

```

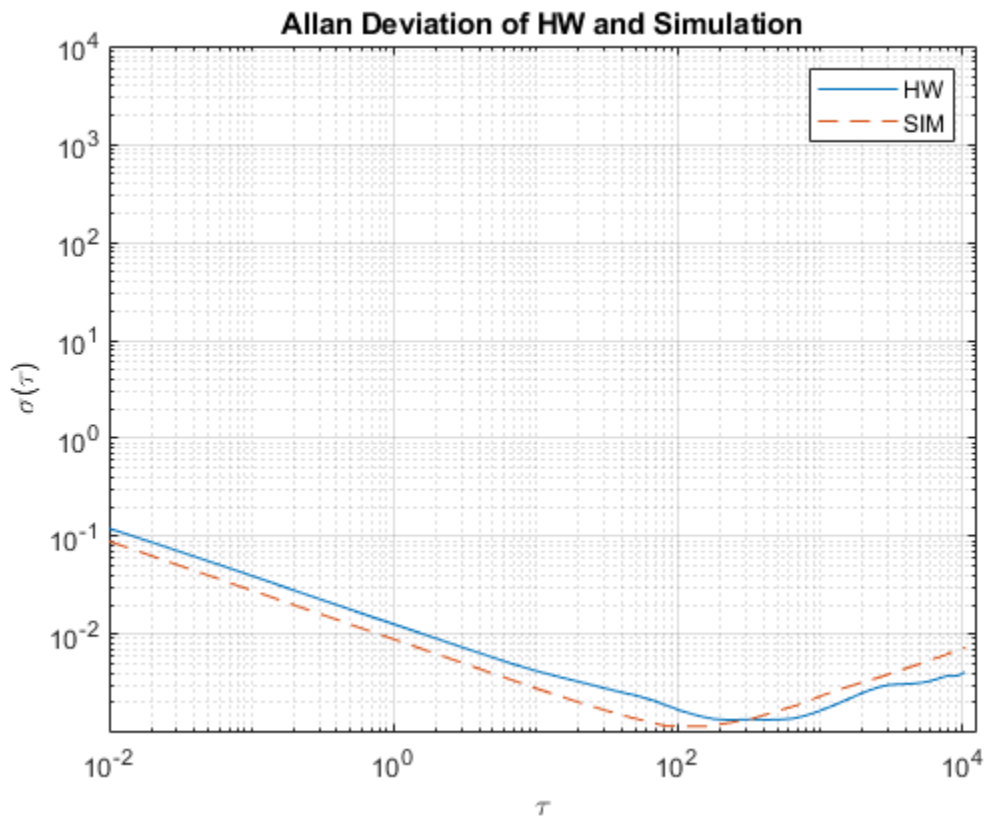
if generateSimulatedData
    % Set the gyroscope parameters to the noise parameters determined
    % above.
    gyro = gyroparams('NoiseDensity', N, 'RandomWalk', K, ...
        'BiasInstability', B);
    omegaSim = helperAllanVarianceExample(L, Fs, gyro);
else
    load('SimulatedSingleAxisGyroscope', '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.

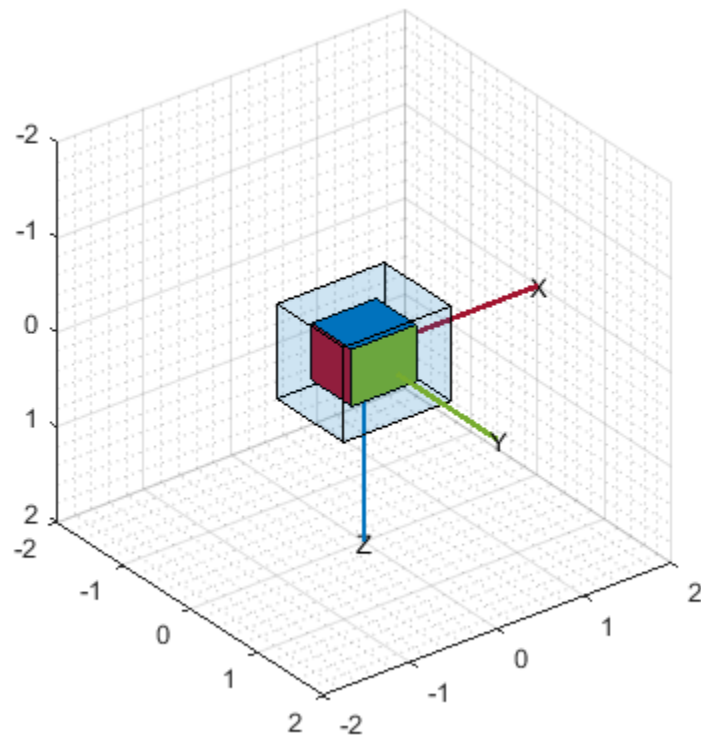
Types of Sensors

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.

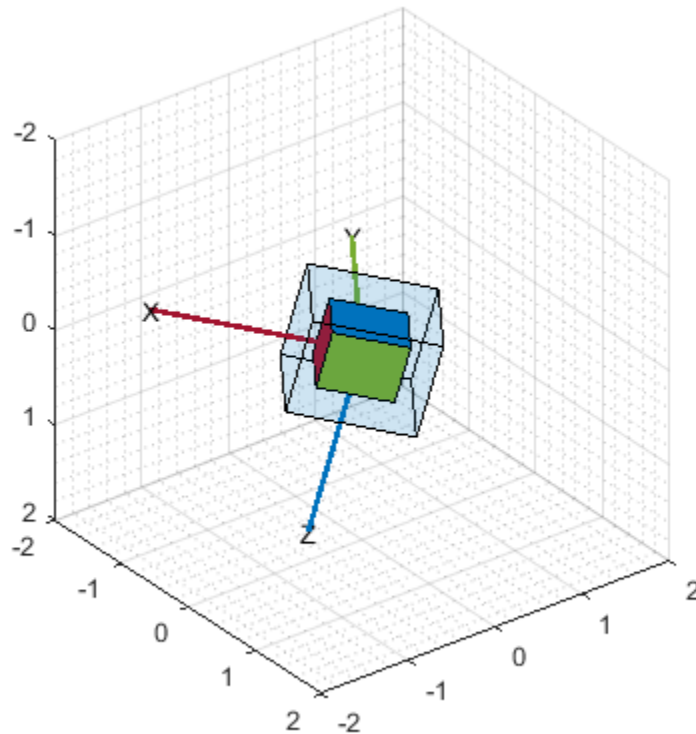
```
ld = load('rpy_9axis.mat');  
  
acc = ld.sensorData.Acceleration;  
gyro = ld.sensorData.AngularVelocity;  
mag = ld.sensorData.MagneticField;  
  
pp = poseplot;
```



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)  
    set(pp, "Orientation", qe(ii))  
    drawnow limitrate  
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 6-48 could be used to smooth the motion.

Accelerometer-Gyroscope Fusion

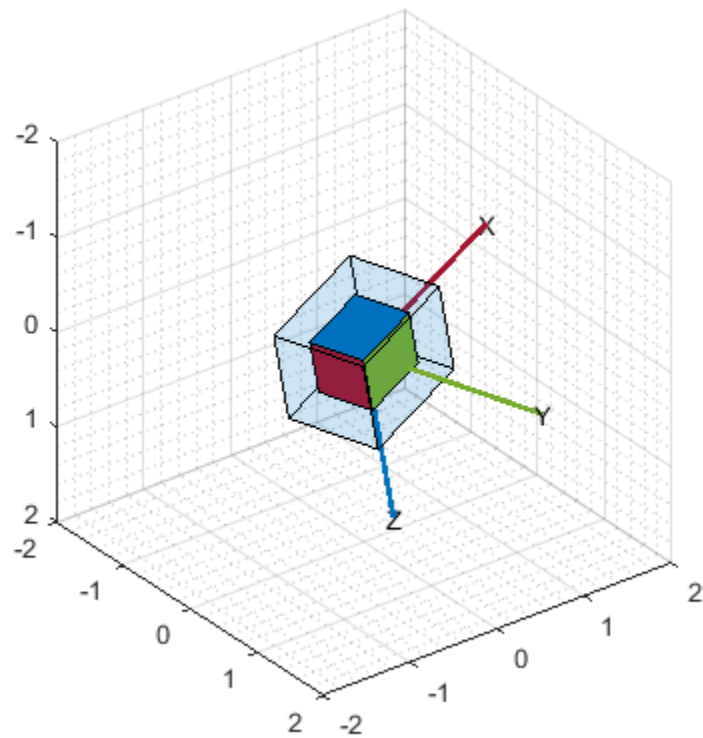
The following objects estimate orientation using either an error-state Kalman filter or a complementary filter. The error-state Kalman filter is the standard estimation filter and allows for many different aspects of the system to be tuned using the corresponding noise parameters. The complementary filter 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.

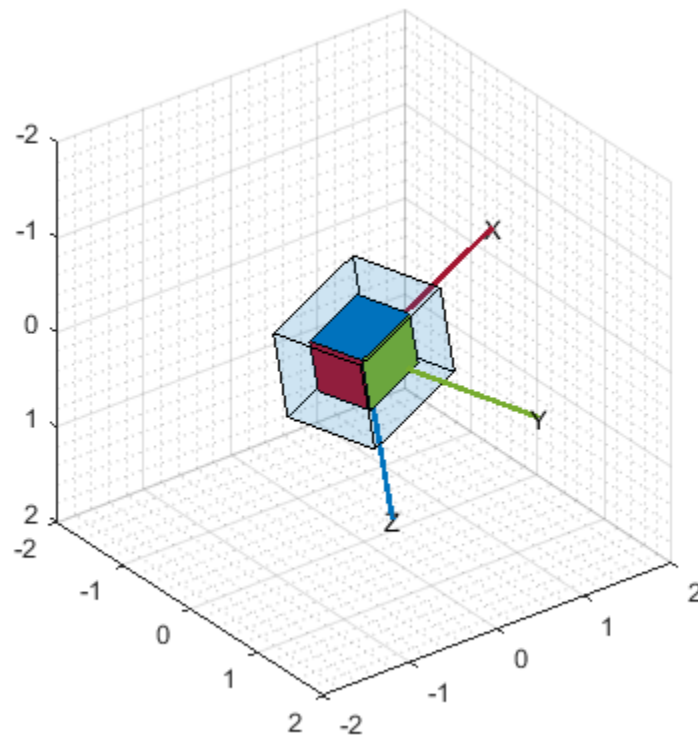
```

ifilt = imufilter('SampleRate', ld.Fs);
for ii=1:size(acc,1)
    qimu = ifilt(acc(ii,:), gyro(ii,:));
    set(pp, "Orientation", qimu)
    drawnow limitrate
end

```

```
% Disable magnetometer input.  
cfilt = complementaryFilter('SampleRate', ld.Fs, 'HasMagnetometer', false);  
for ii=1:size(acc,1)  
    qimu = cfilt(acc(ii,:), gyro(ii,:));  
    set(pp, "Orientation", qimu)  
    drawnow limitrate  
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`.

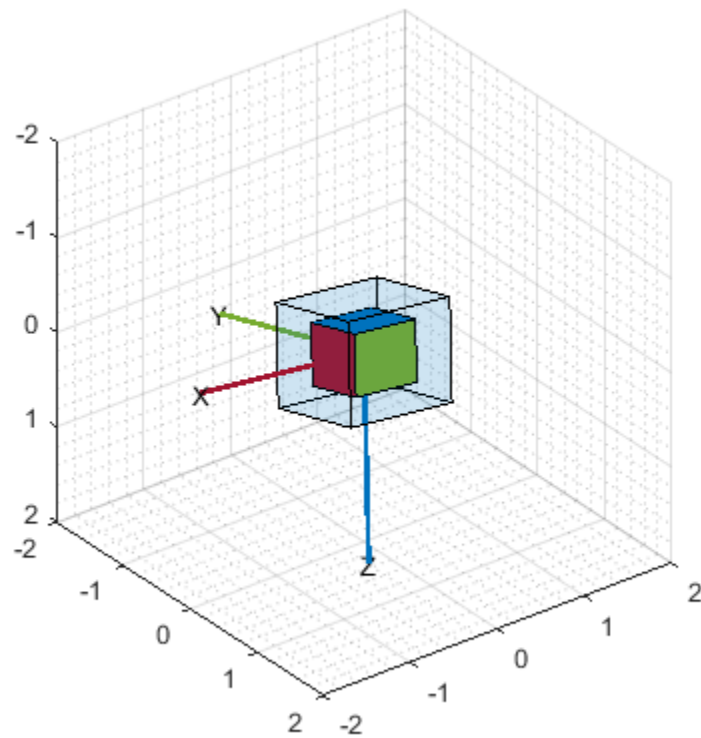
Accelerometer-Gyroscope-Magnetometer Fusion

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.

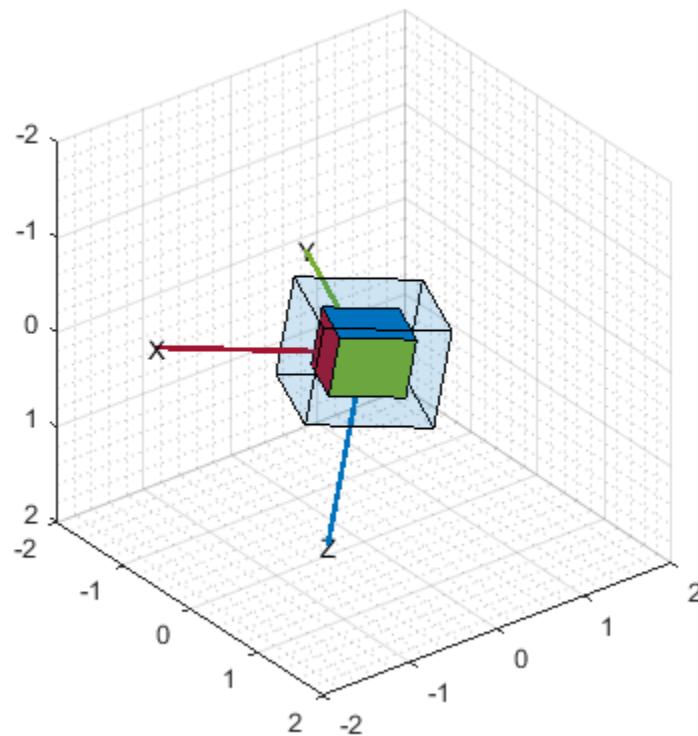
```

ifilt = ahrsfilter('SampleRate', ld.Fs);
for ii=1:size(acc,1)
    qahrs = ifilt(acc(ii,:), gyro(ii,:), mag(ii,:));
    set(pp, "Orientation", qahrs)
    drawnow limitrate
end

```



```
cfilt = complementaryFilter('SampleRate', ld.Fs);  
for ii=1:size(acc,1)  
    qahrs = cfilt(acc(ii,:), gyro(ii,:), mag(ii,:));  
    set(pp, "Orientation", qahrs)  
    drawnow limitrate  
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;

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.

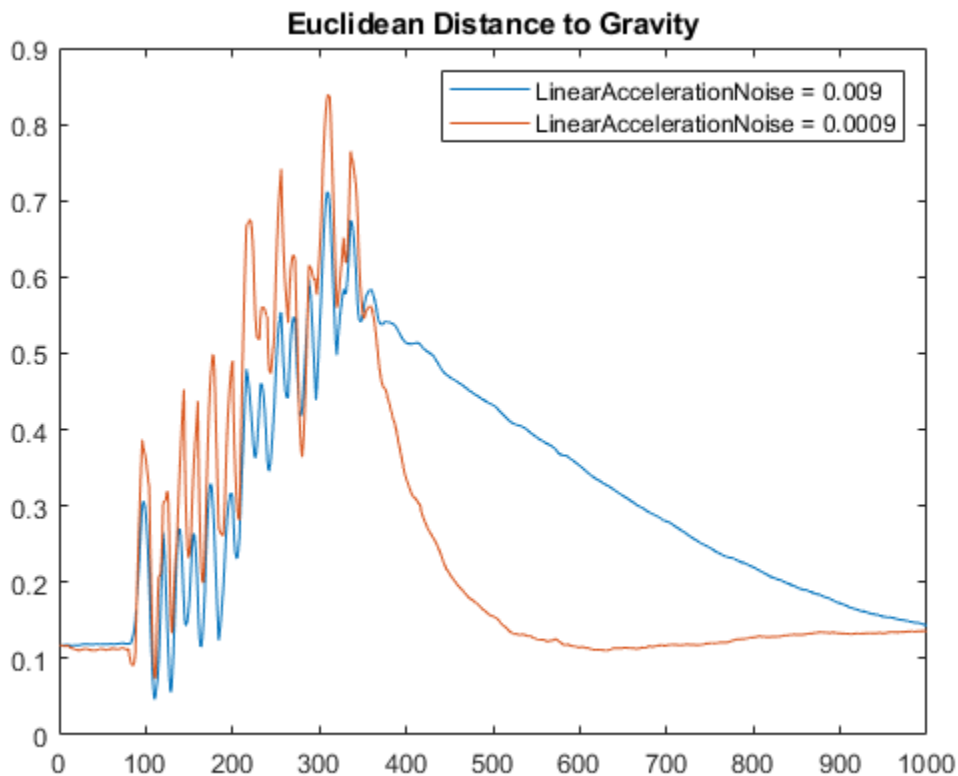
```

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.

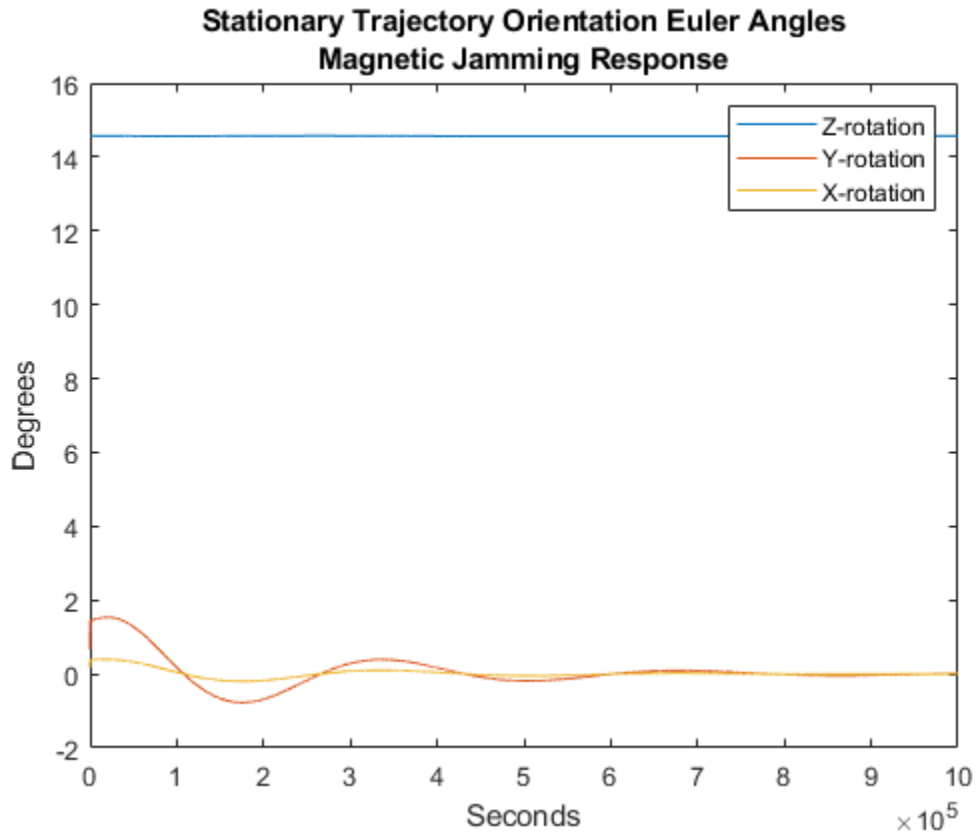
The `MagneticDisturbanceNoise` property enables modeling magnetic disturbances (non-geomagnetic noise sources) in much the same way `LinearAccelerationNoise` models linear acceleration.

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.

```
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.

```
% 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.

```
% 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

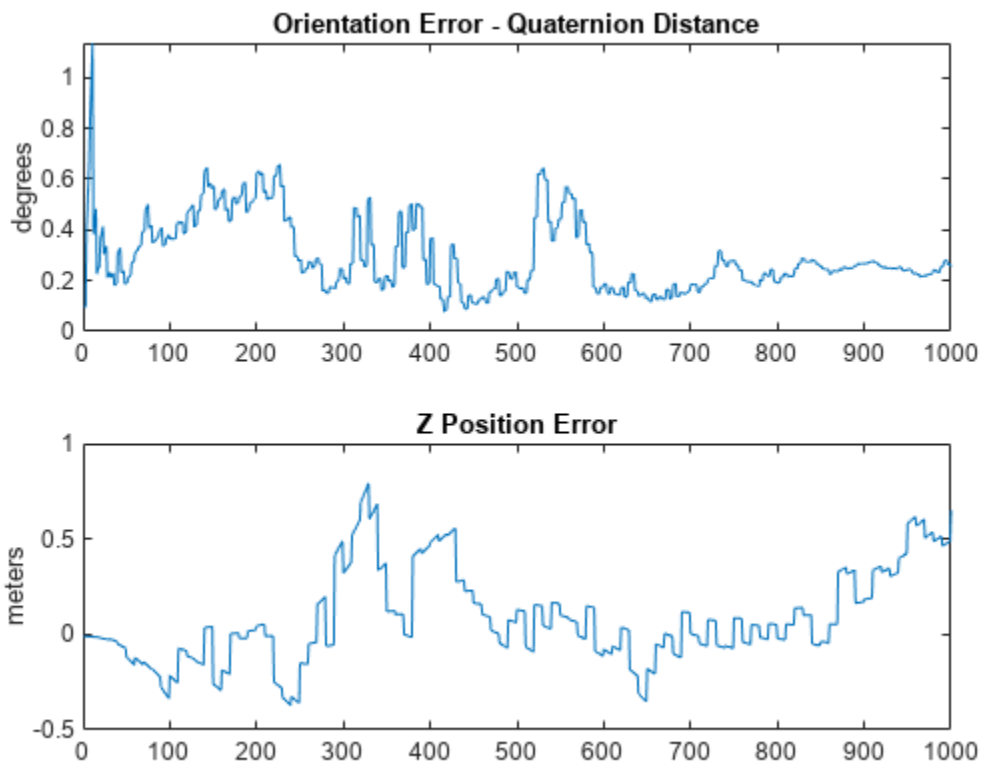
```

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 use 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 `fusionRadarSensor`. This example shows how to configure the `fusionRadarSensor` 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 6-88 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 = fusionRadarSensor(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 =
```

```

fusionRadarSensor with properties:

    SensorIndex: 1
    UpdateRate: 1
    DetectionMode: 'Monostatic'
    ScanMode: 'Mechanical'
InterferenceInputPort: 0
EmissionsInputPort: 0

    MountingLocation: [0 0 -15]
    MountingAngles: [-135 0 0]

    FieldOfView: [5 10.0000]
    LookAngle: [0 0]
    RangeLimits: [0 100000]

    DetectionProbability: 0.9000
    FalseAlarmRate: 1.0000e-06
    ReferenceRange: 100000

    TargetReportFormat: 'Clustered detections'

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;

```

```

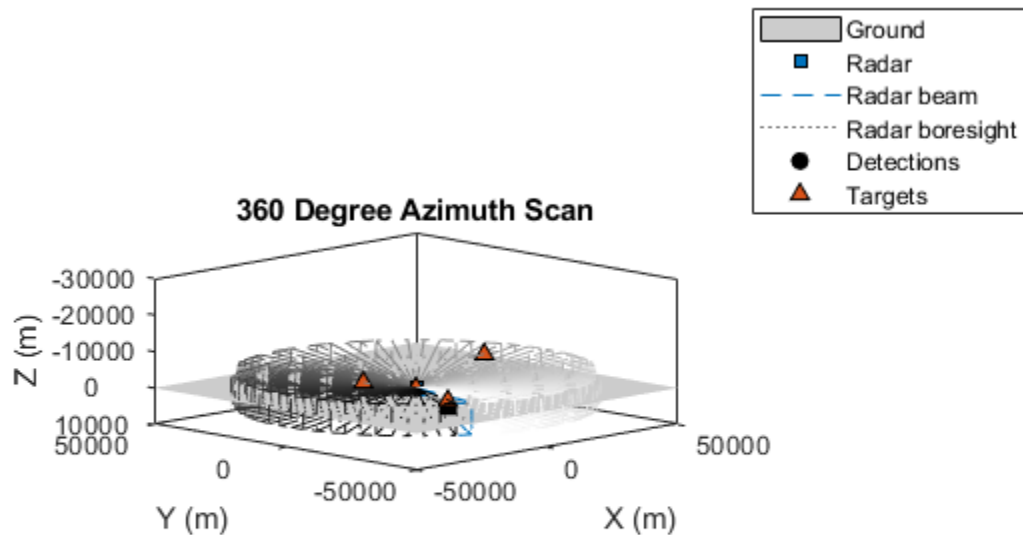
% Current target positions
targets = targetPoses(platform);

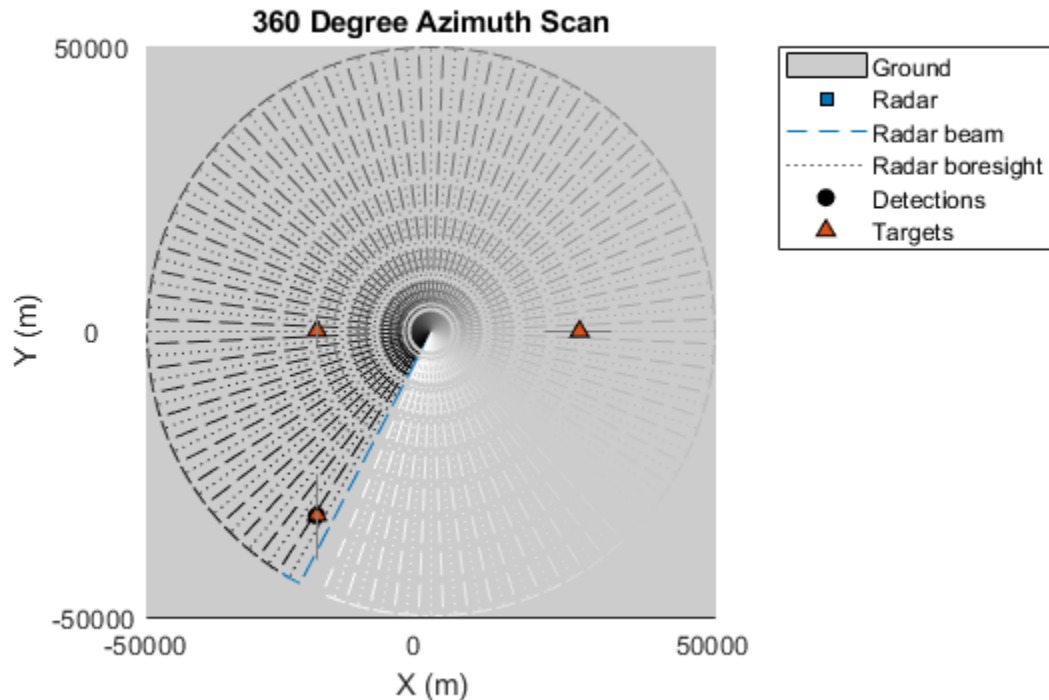
% Generate target detections at current scan position
dets = radar(targets,simTime);

% 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
end

```





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 elevation scan limits, 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.MechanicalAzimuthLimits  
radar.MechanicalElevationLimits
```

```
ans =  
  
    0    360
```

```
ans =  
  
   -10     0
```

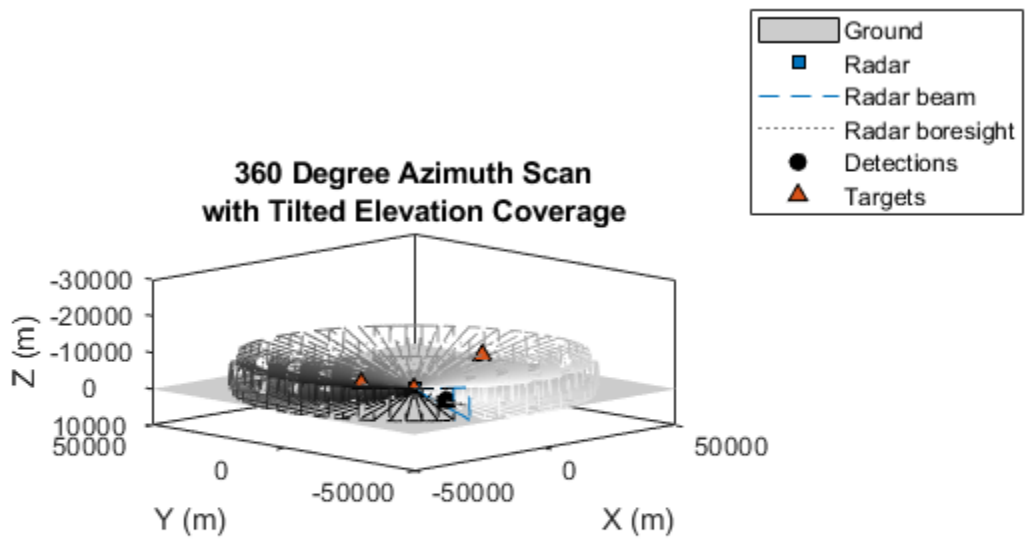
```
% Confirm elevation field of view is slightly larger than the elevation  
% spanned by the scan limits so that raster scan is not performed  
elSpan = diff(radar.MechanicalElevationLimits)  
isLarger = radar.FieldOfView(2)>elSpan
```

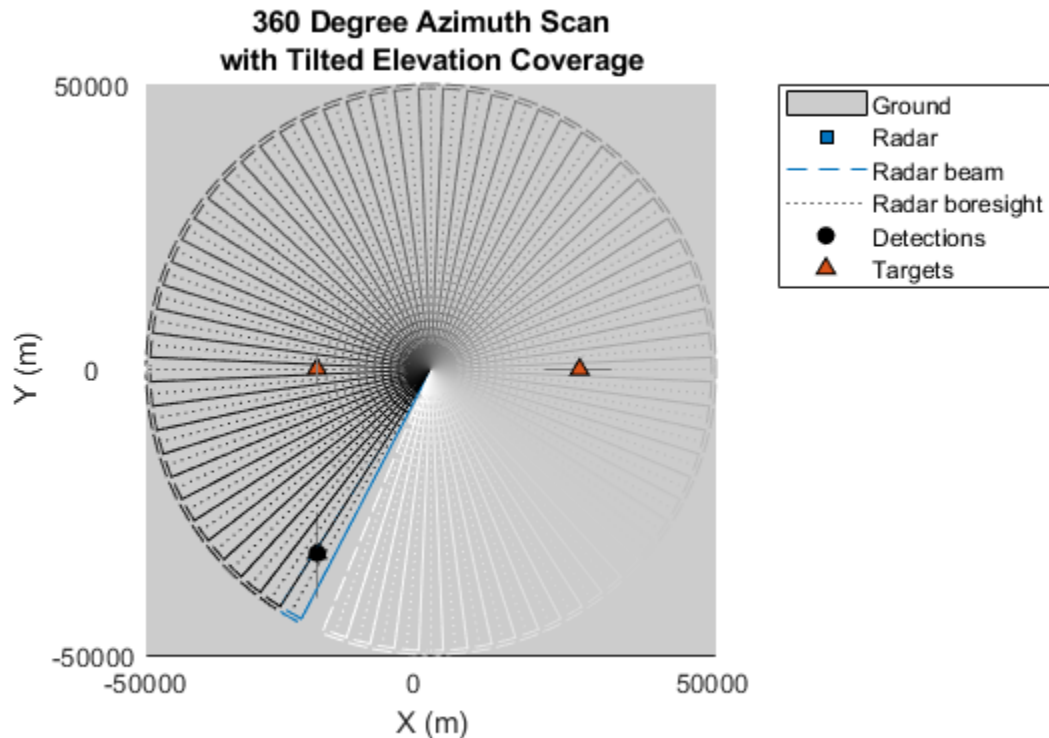
```
elSpan =  
  
    10
```

```
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);
```



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.

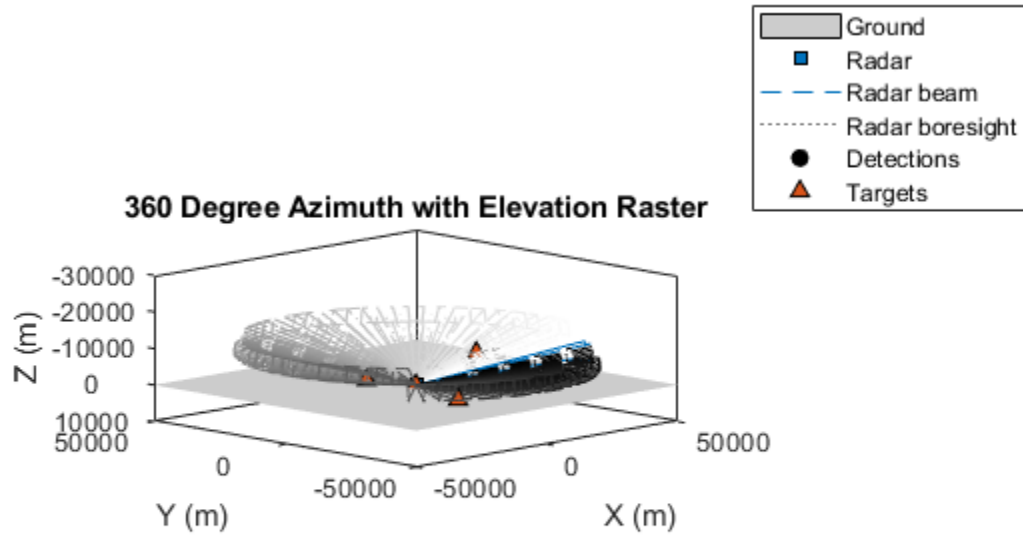
Setup the radar to scan a 10 degree region above the ground using a beam which spans 5 degrees in elevation. This produces 3 elevation raster bars at 0, -5, and -10 degrees elevation.

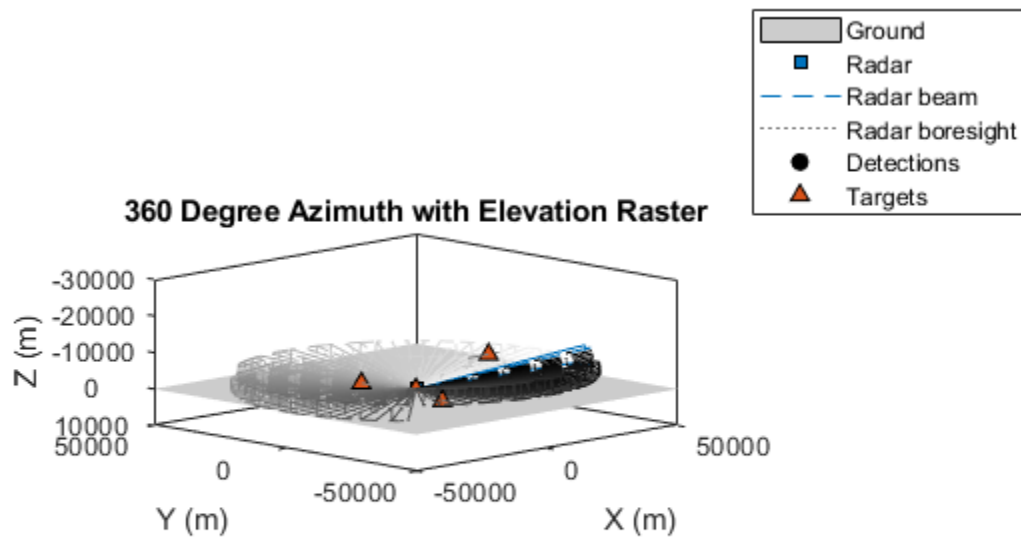
```
release(radar);
elFov = 5; % deg
radar.FieldOfView(2) = elFov;
radar.MechanicalElevationLimits = [-10 0]-elFov/2;
```

```
% 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;
```

```
% Run the simulation  
helperScanRadarModesExample('Run simulation',ts,platform,radar,3);
```





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.

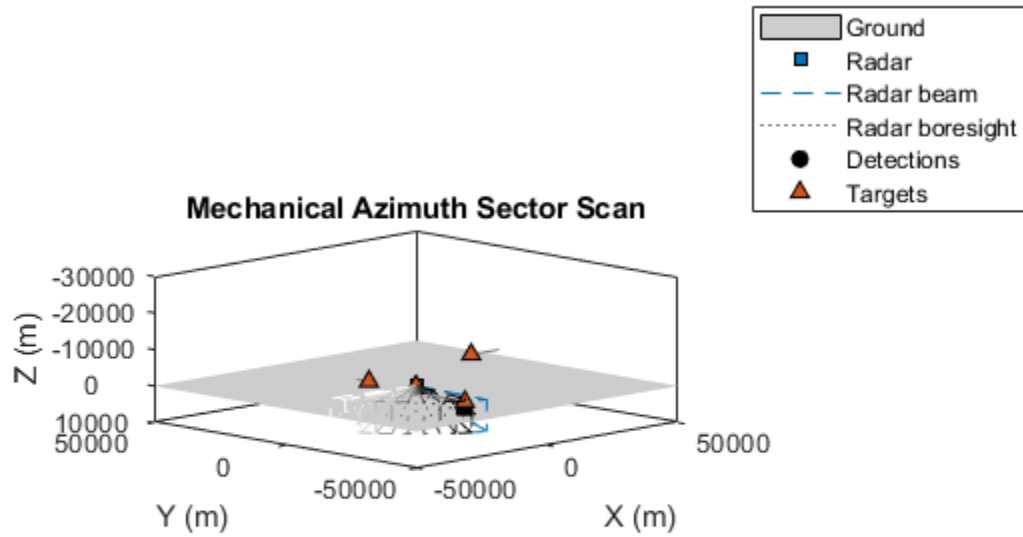
Setup the radar to scan a 90 degree azimuth sector by setting the mechanical scan limits to span 45 degrees on either side of the radar's mounted orientation. Disable elevation to constrain the radar's beam to the horizontal plane.

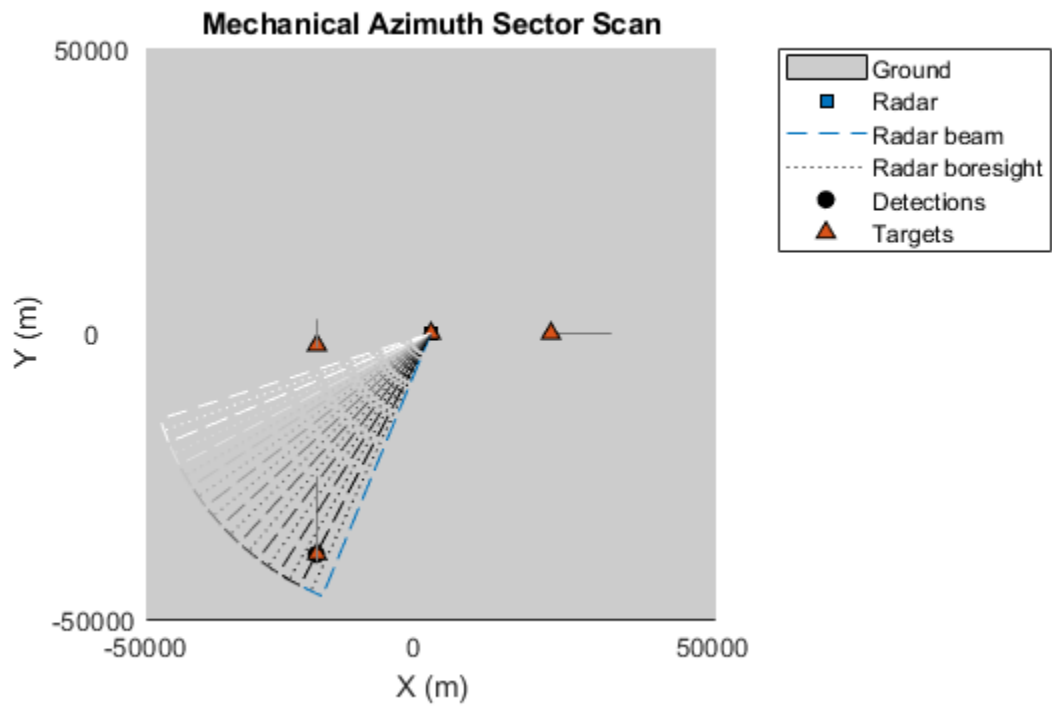
Setup a `fusionRadarSensor` with this configuration by specifying the 'Sector' configuration for the radar.

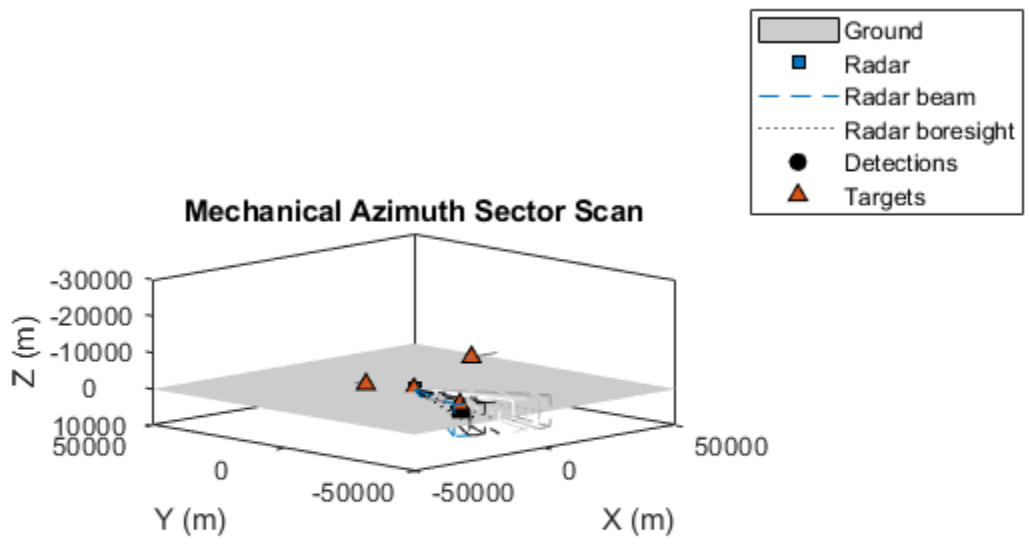
```
radar = fusionRadarSensor(1, 'Sector', 'MountingAngles', [-135 0 0], 'MountingLocation', [0 0 -15]);
% Set the radar's azimuth field of view to 5 degrees to display larger beams
radar.FieldOfView(1) = 5;
```

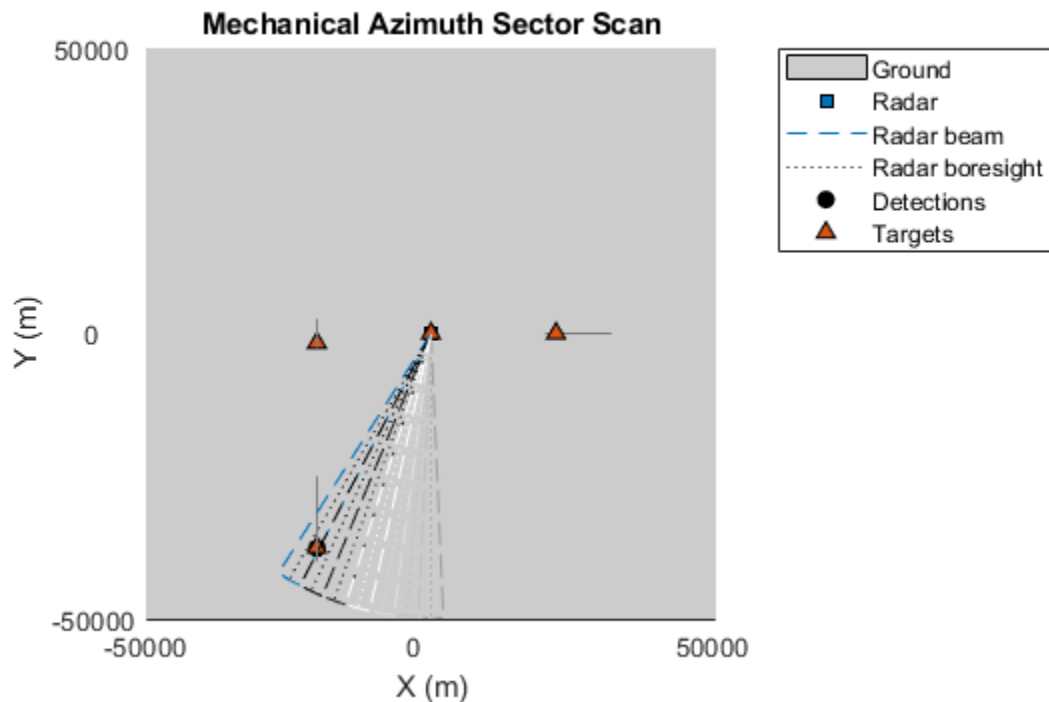
```
% Set the update rate to show multiple cycles of the sector scan
rpm = 2;
fov = radar.FieldOfView;
scanrate = rpm*360/60;           % deg/s
updaterate = scanrate/fov(1);    % Hz
radar.UpdateRate = updaterate;

% Run the simulation
helperScanRadarModesExample('Run simulation',ts,platform,radar,4);
```









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
% azimuth scan limits to zero to disable electronic azimuth scan. Set the
% mechanical elevation scan limits to zero to disable mechanical elevation
% scan
radar.ElectronicAzimuthLimits = [0 0];
radar.MechanicalElevationLimits = [0 0];

% Confirm that the elevation field of view is greater than the elevation
% spanned by the scan limits
elSpan = diff(radar.ElectronicElevationLimits)
isLarger = radar.FieldOfView(2)>elSpan

elSpan =

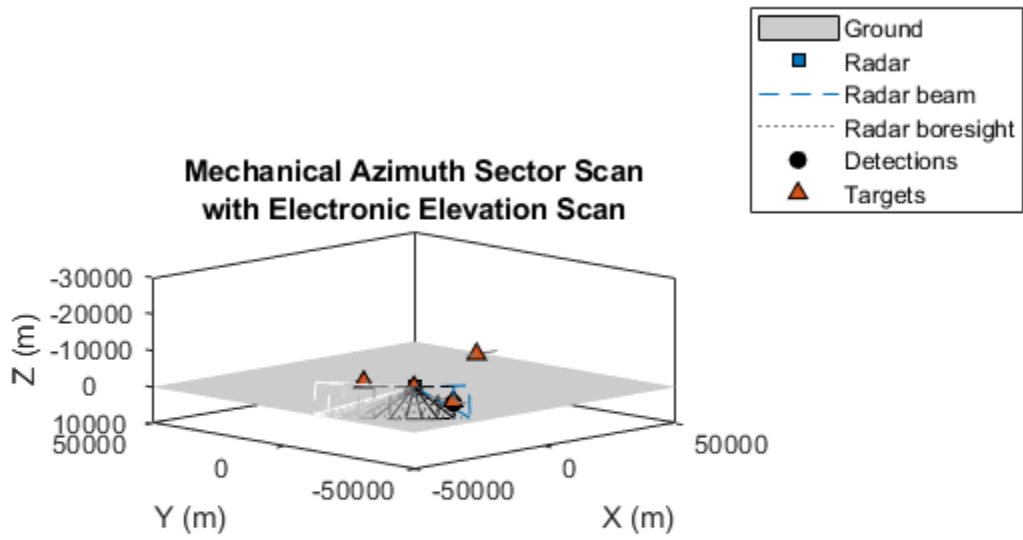
    10

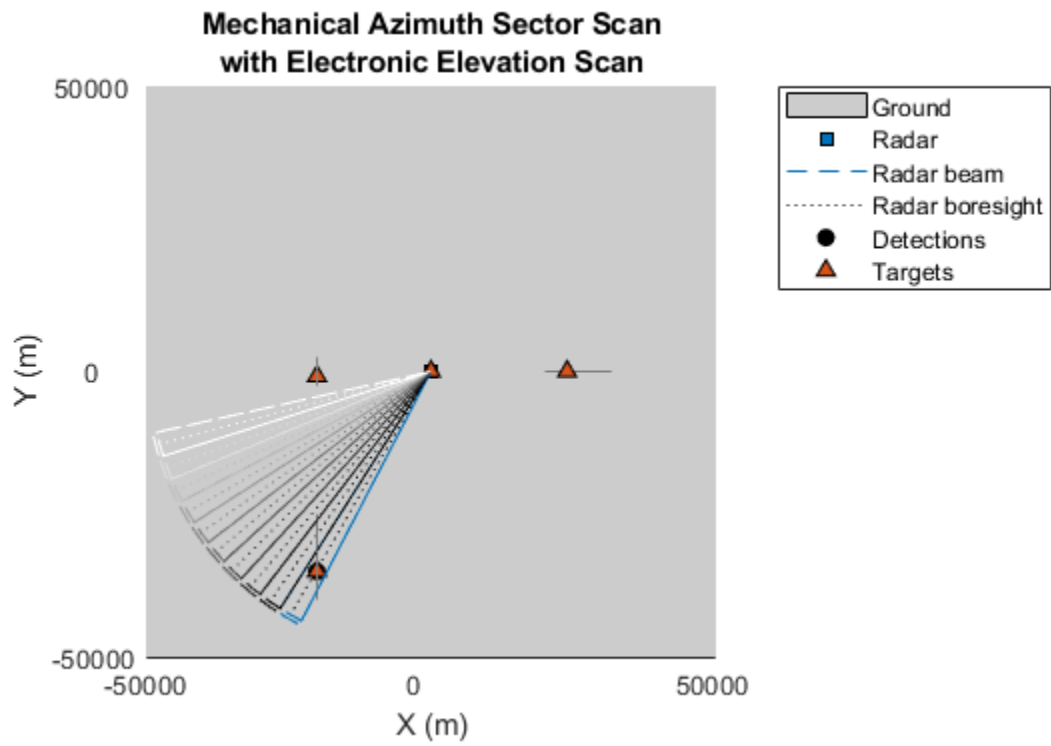
isLarger =

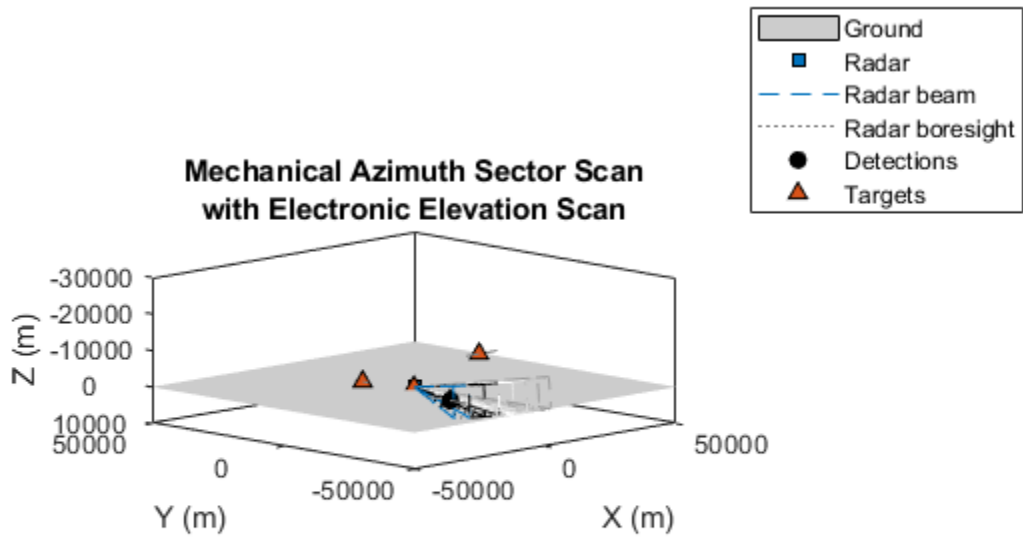
    logical

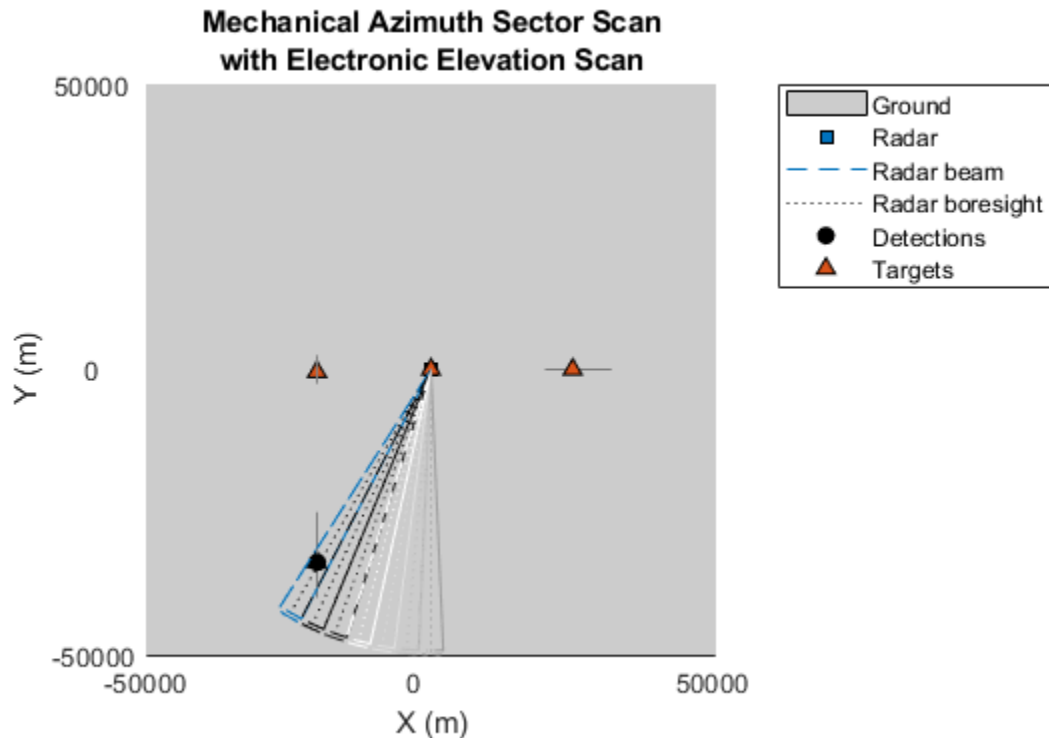
     1

% Run the simulation
helperScanRadarModesExample('Run simulation',ts,platform,radar,5);
```









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

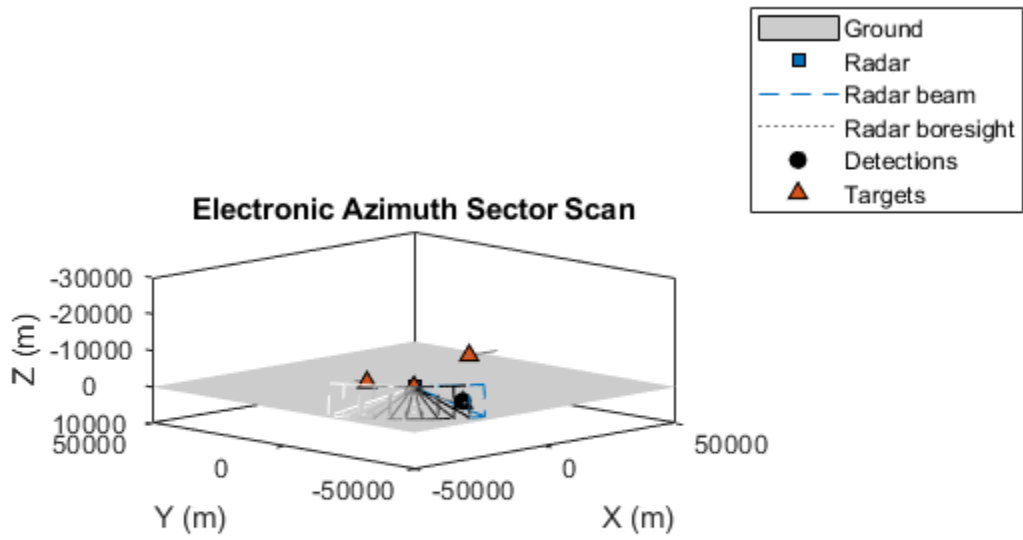
The azimuth sector can be surveyed using electronic scanning as well. Electronically scan the same azimuth sector by creating a `fusionRadarSensor` using the 'Sector' scan configuration with its scan mode set to 'Electronic' instead of 'Mechanical'. Enable elevation so that the region above the ground is scanned by stacking beams in elevation to span the entire elevation scan limits.

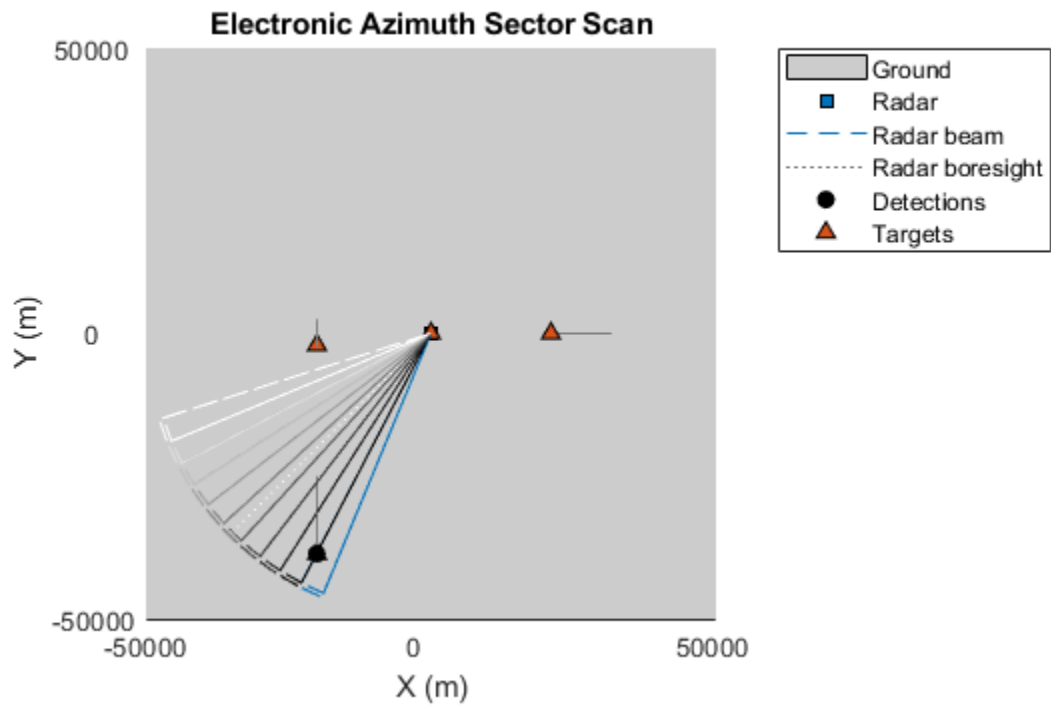
```
radar = fusionRadarSensor(1,'Sector','ScanMode','Electronic','HasElevation',true, ...
    'MountingAngles',[-135 0 0],'MountingLocation',[0 0 -15]);
```

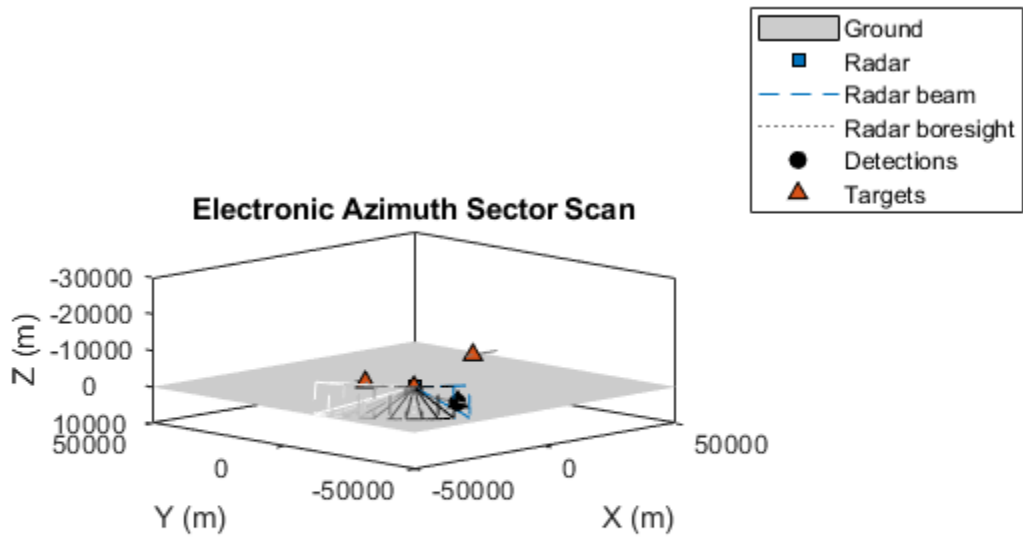
```
% Set update rate to show multiple cycles of the raster scan pattern
radar.UpdateRate = updatarate;
```

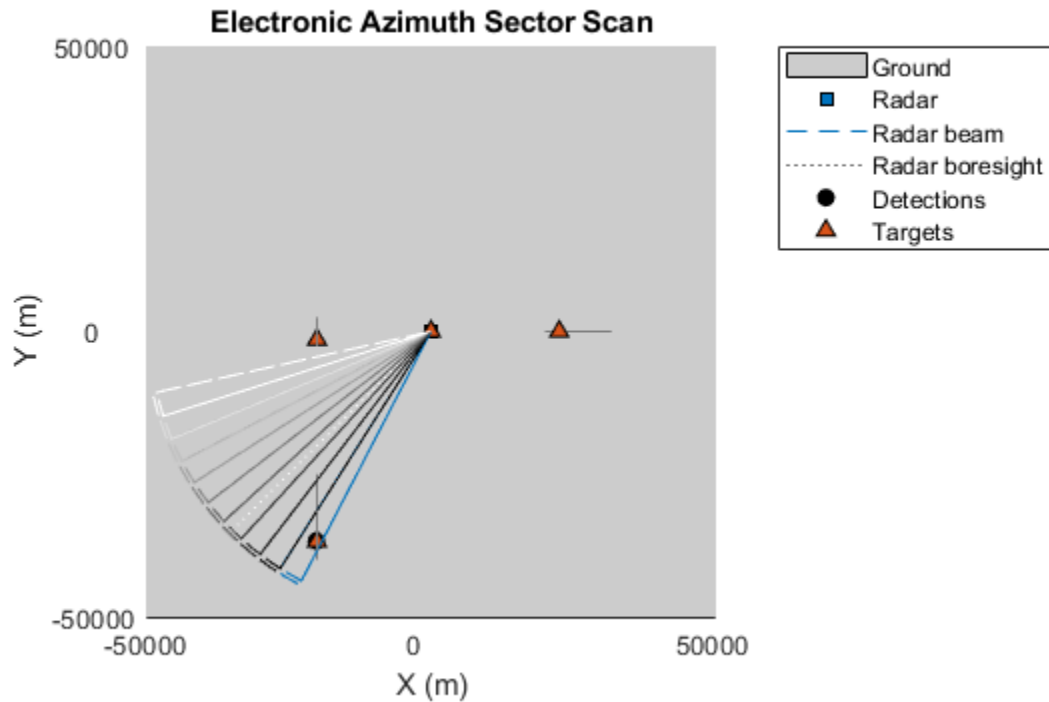
```
% Set the radar's azimuth field of view to 5 degrees to display larger beams
radar.FieldOfView(1) = 5;
```

```
% Run the simulation
helperScanRadarModesExample('Run simulation',ts,platform,radar,6);
```









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

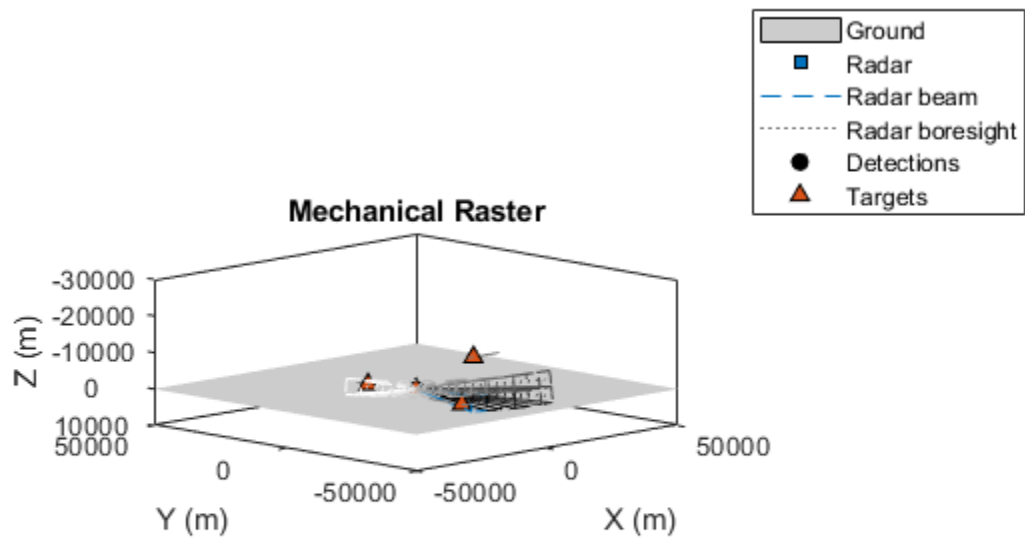
The radar can be easily configured to execute a mechanical raster scan pattern as follows.

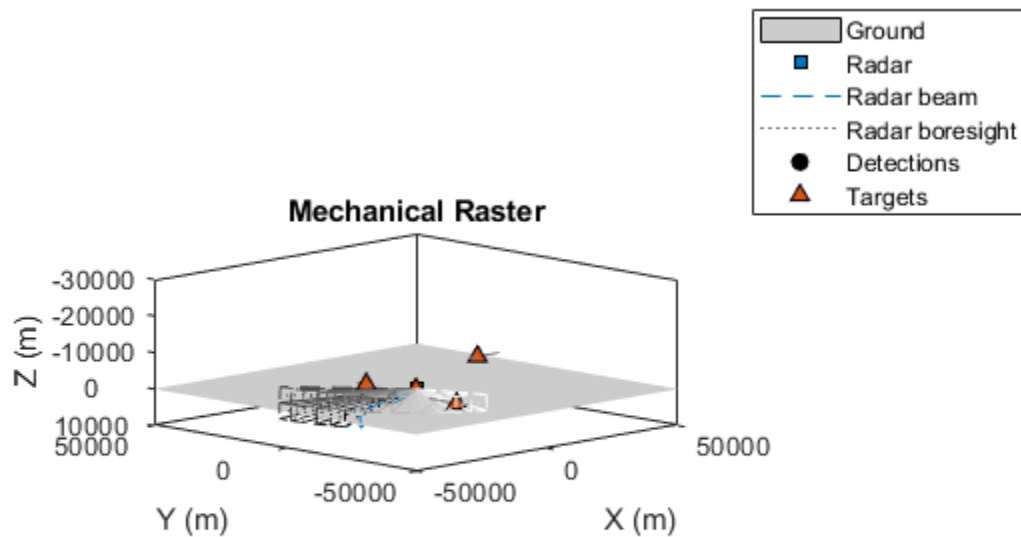
```
radar = fusionRadarSensor(1,'Raster','MountingAngles', [-135 0 0], 'MountingLocation', [0 0 -15]);

% Set update rate to show multiple cycles of the raster scan pattern
radar.UpdateRate = updatarate;

% 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);
```





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.

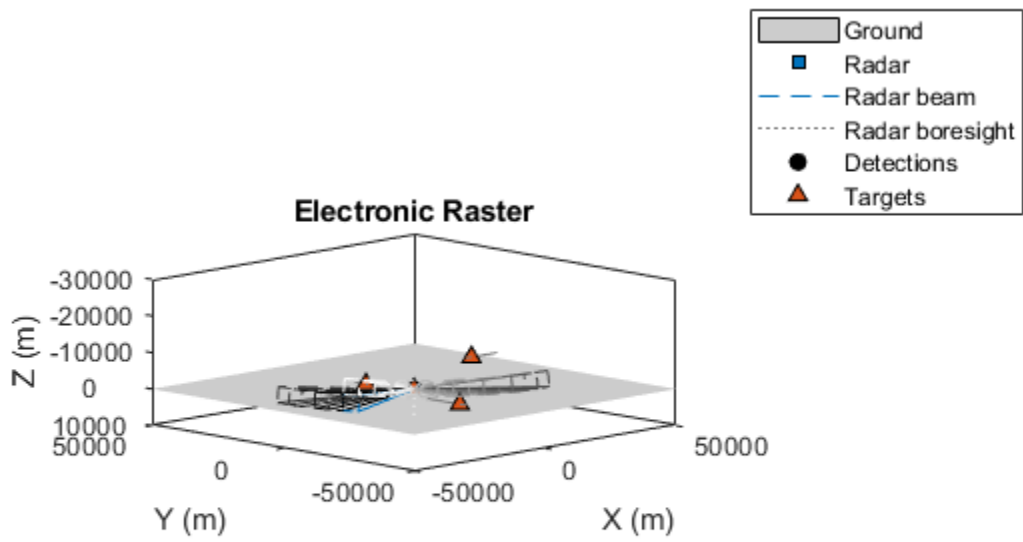
Configure the radar to perform an electronic raster scan by setting its scan mode to 'Electronic'.

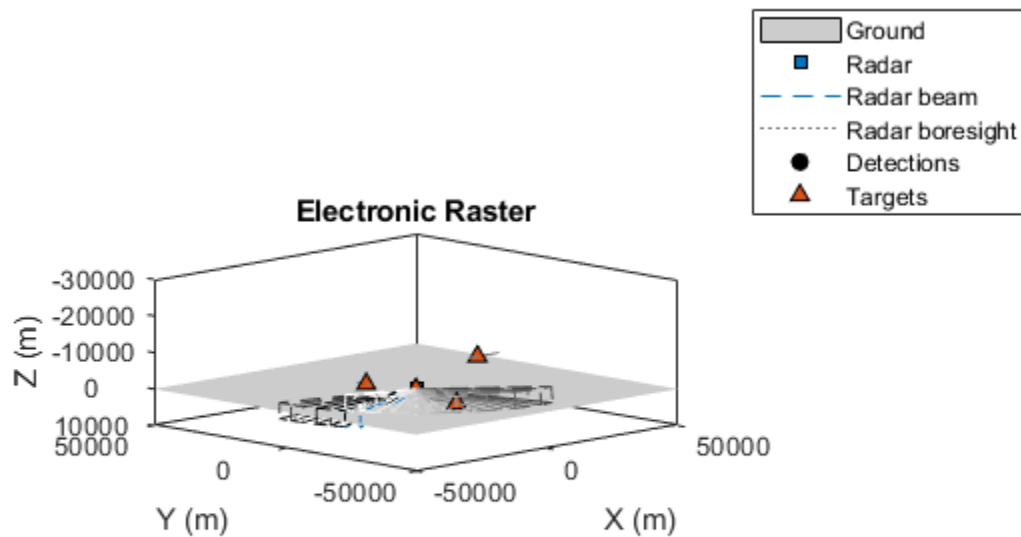
```
radar = fusionRadarSensor(1,'Raster','ScanMode','Electronic','MountingAngles',[-135 0 0],'MountingAngles',[-135 0 0]);

% Set update rate to show multiple cycles of the raster scan pattern
radar.UpdateRate = updatarate;

% 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);
```





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

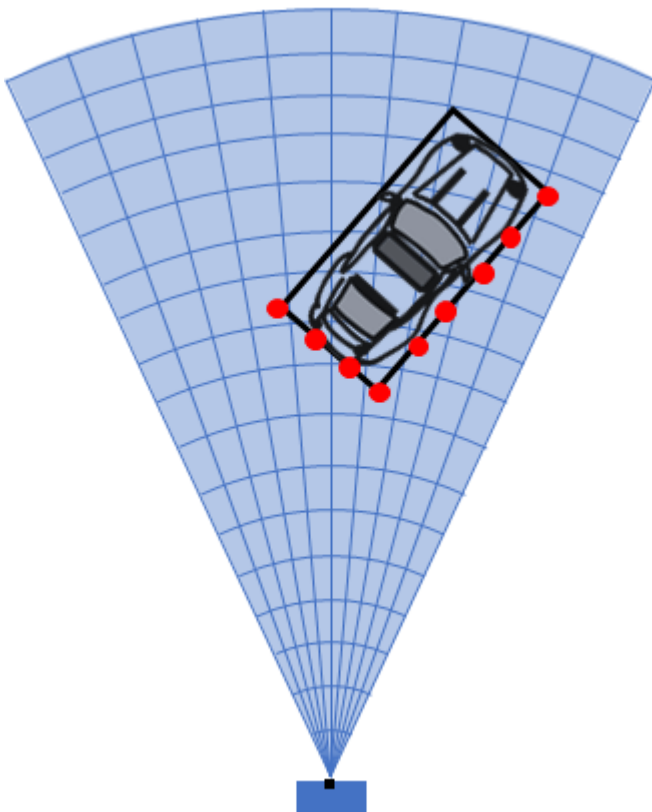
This example shows you how to model many common scan patterns using the `fusionRadarSensor`. This model provides presets that enable you to quickly configure the radar and make additional minor adjustments to the scan parameters to match the radar you are modeling.

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

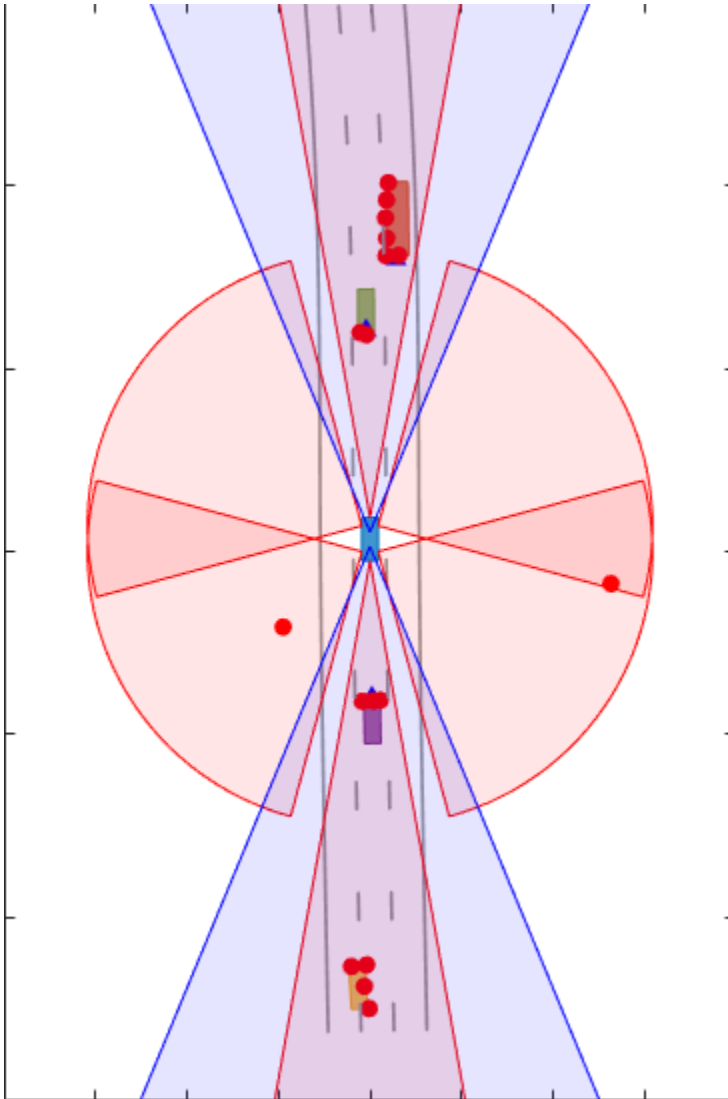
In this example, there is an ego vehicle and four other vehicles: a vehicle ahead of the ego vehicle in the center lane, a vehicle behind the ego vehicle in the center lane, a truck ahead of the ego vehicle in the right lane and an overtaking vehicle in the left lane.

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.

```
% Function to return the errors given track and truth.
errorFcn = @(track,truth)helperExtendedTargetError(track,truth);

% Function to return the distance between track and truth.
distFcn = @(track,truth)helperExtendedTargetDistance(track,truth);

% 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 = 'Body';
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

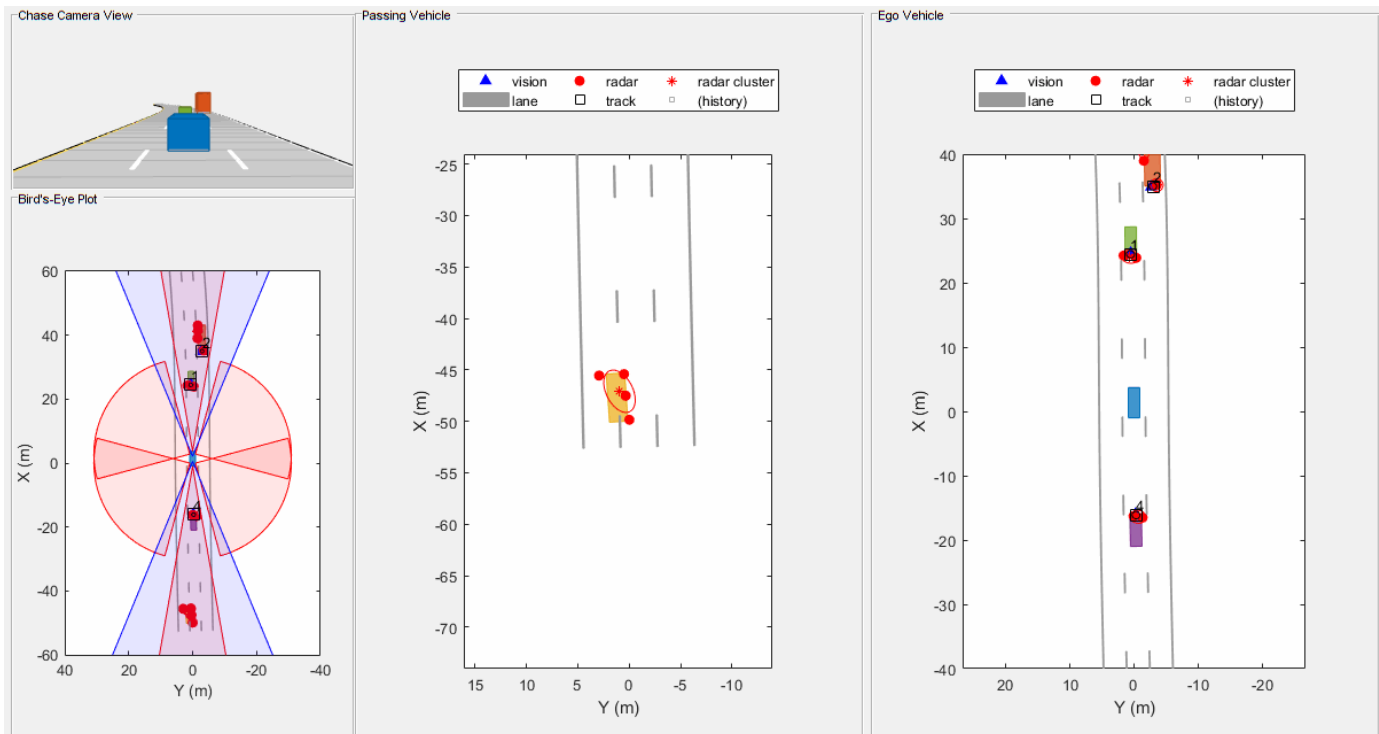
% Capture the cumulative track metrics. The error metrics show the averaged
% value of the error over the simulation.
assignmentMetricsMOT = tam.trackMetricsTable;
errorMetricsMOT = tem.cumulativeTruthMetrics;

% 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.

```
% Release and restart all objects.
restart(scenario);
release(tem);
release(tam);
% No penalty 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)

% Set up sensor configurations

% Set Ego pose
egoPose.Position = egoVehicle.Position;
egoPose.Velocity = egoVehicle.Velocity;
egoPose.Orientation = rotmat(quaternion([egoVehicle.Yaw egoVehicle.Pitch egoVehicle.Roll], 'euler'));

sensorConfigurations = cell(numel(sensors),1);
for i = 1:numel(sensors)
    sensorConfigurations{i} = trackingSensorConfiguration(sensors{i},egoPose, ...
        FilterInitializationFcn = @helperInitGGIWFilter, ...
        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 evaluates 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. The helper function wraps around `partitionDetections` and doesn't use range-rate measurements for partitioning detections from side radars.

```
partFcn = @(x)helperPartitioningFcn(x,3,5);
```

```
tracker = trackerPHD( SensorConfigurations = sensorConfigurations,...
    PartitioningFcn = partFcn,...
    AssignmentThreshold = 450,...% Minimum negative log-likelihood of a detection cell (multiple
    ExtractionThreshold = 0.75,...% Weight threshold of a filter component to be declared a track
    ConfirmationThreshold = 0.85,...% 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 s
);
```

Run the simulation. 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);

    % 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.Operators(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;

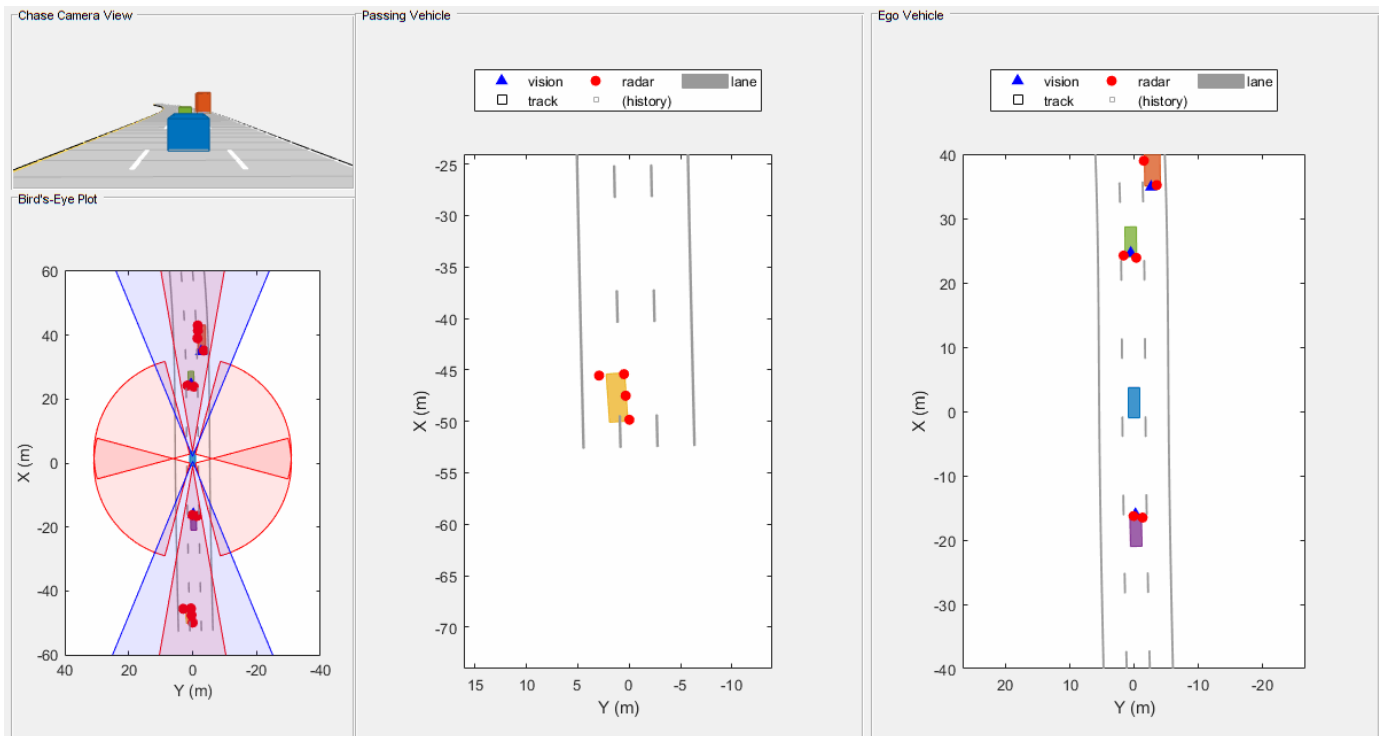
% Capture frames for GIF
gifWriter();
end

% 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 detection
end

% Define tracker using new sensor configurations
tracker = trackerPHD( SensorConfigurations = sensorConfigurations,...
    PartitioningFcn = partFcn,...
    AssignmentThreshold = 600,... % Minimum negative log-likelihood of a detection cell to add to filter
    ExtractionThreshold = 0.85,... % Weight threshold of a filter component to be declared a track
    ConfirmationThreshold = 0.95,... % Weight threshold of a filter component to be declared a confirmed track
    MergingThreshold = 50,... % Threshold to merge components
    HasSensorConfigurationsInput = true... % Tracking is performed in scenario frame and hence sensors are in scenario frame
);

% 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);

    % 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.Operators(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
```



```
end
```

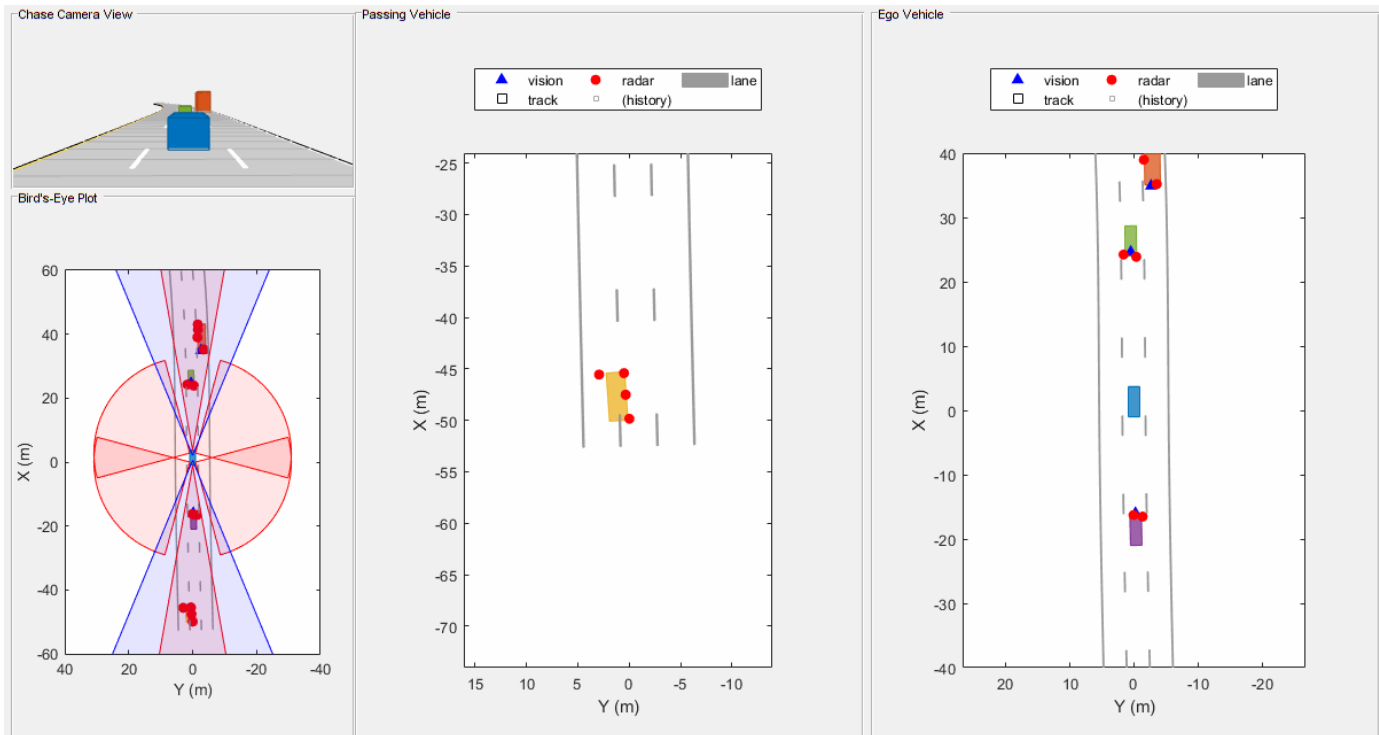
```
% Capture the truth and track metrics tables
assignmentMetricsGMPHD = tam.trackMetricsTable;
errorMetricsGMPHD = tem.cumulativeTruthMetrics;
```

```
% 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 a similar approach as the 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.

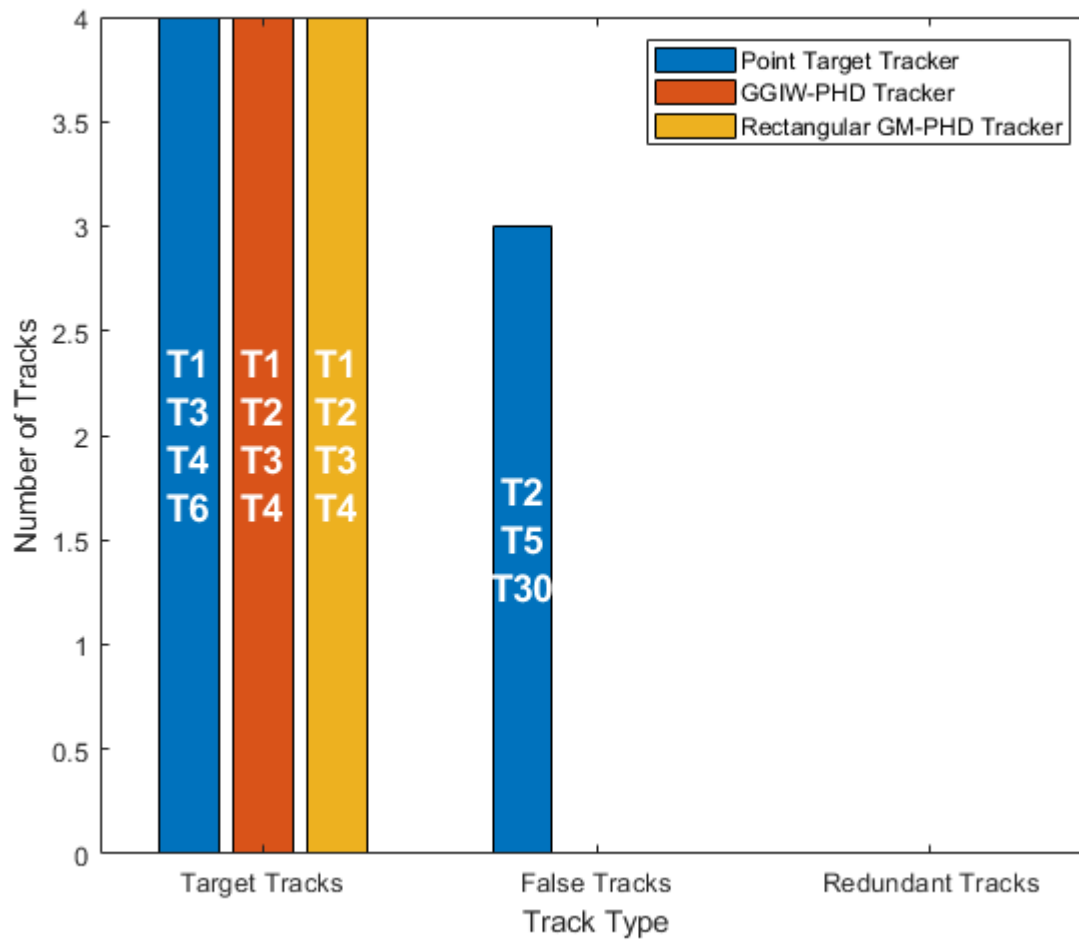


Evaluate Tracking Performance

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

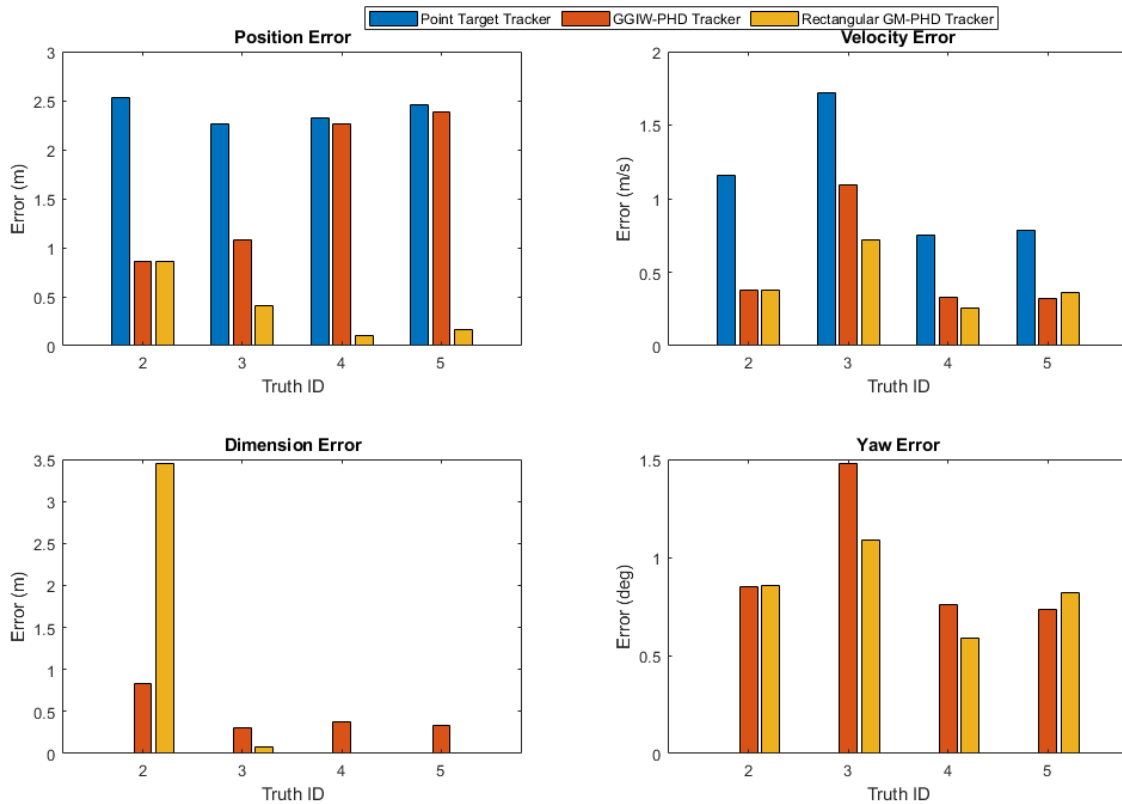
```
helperPlotAssignmentMetrics(assignmentMetricsMOT, assignmentMetricsGGIWPHD, assignmentMetricsGMPHD)
```



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.

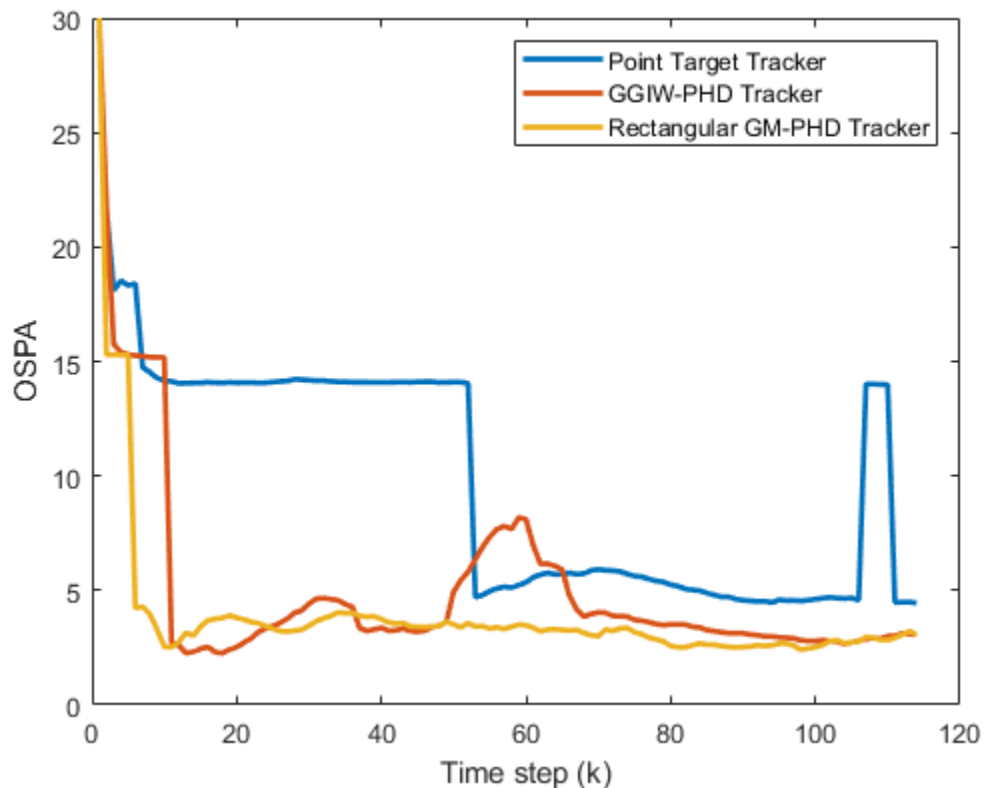
As described earlier, the GGIW-PHD tracker assumes that measurements are distributed around the object's extent, which results in center of the tracks on observable parts of the vehicle. This can also be seen in the position error metrics for TruthID 2 and 4. The tracker is able to estimate the dimensions 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 the 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.

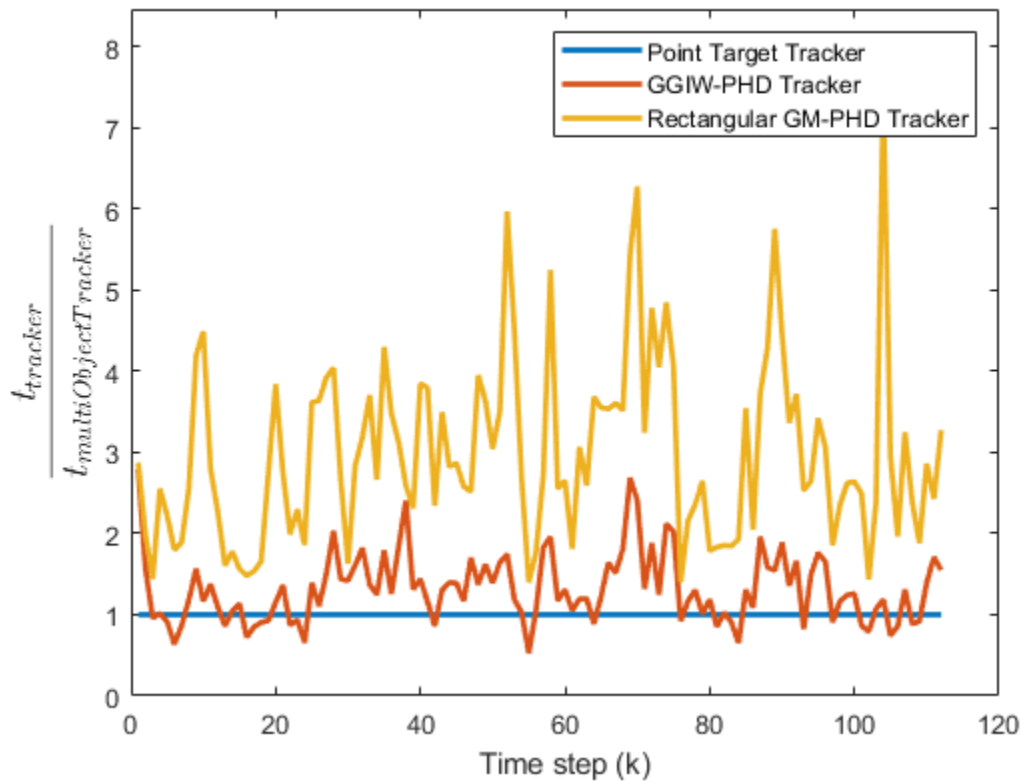
```
ospaFig = figure;
plot(ospaMetric, 'LineWidth', 2);
legend('Point Target Tracker', 'GGIW-PHD Tracker', 'Rectangular GM-PHD Tracker');
xlabel('Time step (k)');
ylabel('OSPA');
```



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)
% Errors as a function of target track and associated truth.
```

```

% 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(:);
trueDims = [truth.Length;truth.Width];

% Get estimated value from track.
% GGIW-PHD tracker outputs a struct field 'Extent' and 'SourceIndex'
% GM-PHD tracker outputs struct with but not 'Extent'
% multiObjectTracker outputs objectTrack

if isa(track,'objectTrack')
    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.

```

function dist = helperExtendedTargetDistance(track,truth)
% This function computes the distance between track and a truth.

% Copyright 2019-2020 The MathWorks, Inc.

% Errors in each aspect
[posError,velError,dimError,yawError] = helperExtendedTargetError(track,truth);

```

```

% 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.

```

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};

    % 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(2);0;detections{i}.Measurement(3);0;detections{i}.Measurement(4);0;detections{i}.Measurement(5);0;detections{i}.Measurement(6);0;detections{i}.Measurement(7);0;detections{i}.Measurement(8);0;detections{i}.Measurement(9);0;detections{i}.Measurement(10);0];
        detections{i}.MeasurementNoise = blkdiag(detections{i}.MeasurementNoise(1,1),0.4,detections{i}.MeasurementNoise(2,2),0.4,detections{i}.MeasurementNoise(3,3),0.4,detections{i}.MeasurementNoise(4,4),0.4,detections{i}.MeasurementNoise(5,5),0.4,detections{i}.MeasurementNoise(6,6),0.4,detections{i}.MeasurementNoise(7,7),0.4,detections{i}.MeasurementNoise(8,8),0.4,detections{i}.MeasurementNoise(9,9),0.4,detections{i}.MeasurementNoise(10,10),0.4);
        detections{i}.MeasurementParameters(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.

```

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

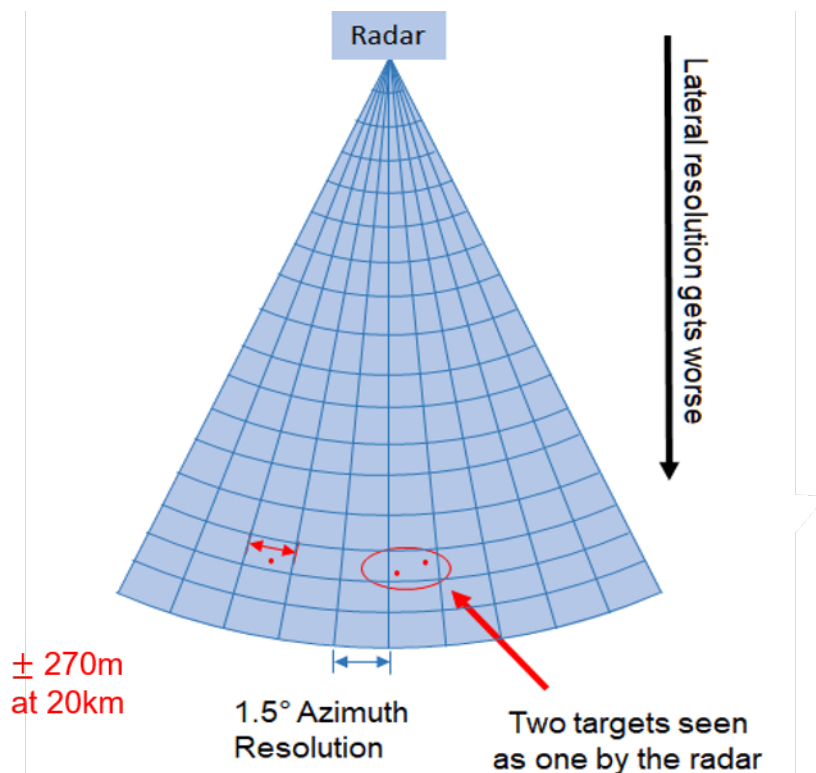
This example shows how to track objects when the association of sensor detections to tracks is ambiguous. In this example, you use a single-hypothesis tracker, a multiple-hypothesis tracker, and a probabilistic data association tracker to compare how the trackers handle this ambiguity. To track the maneuvering 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.

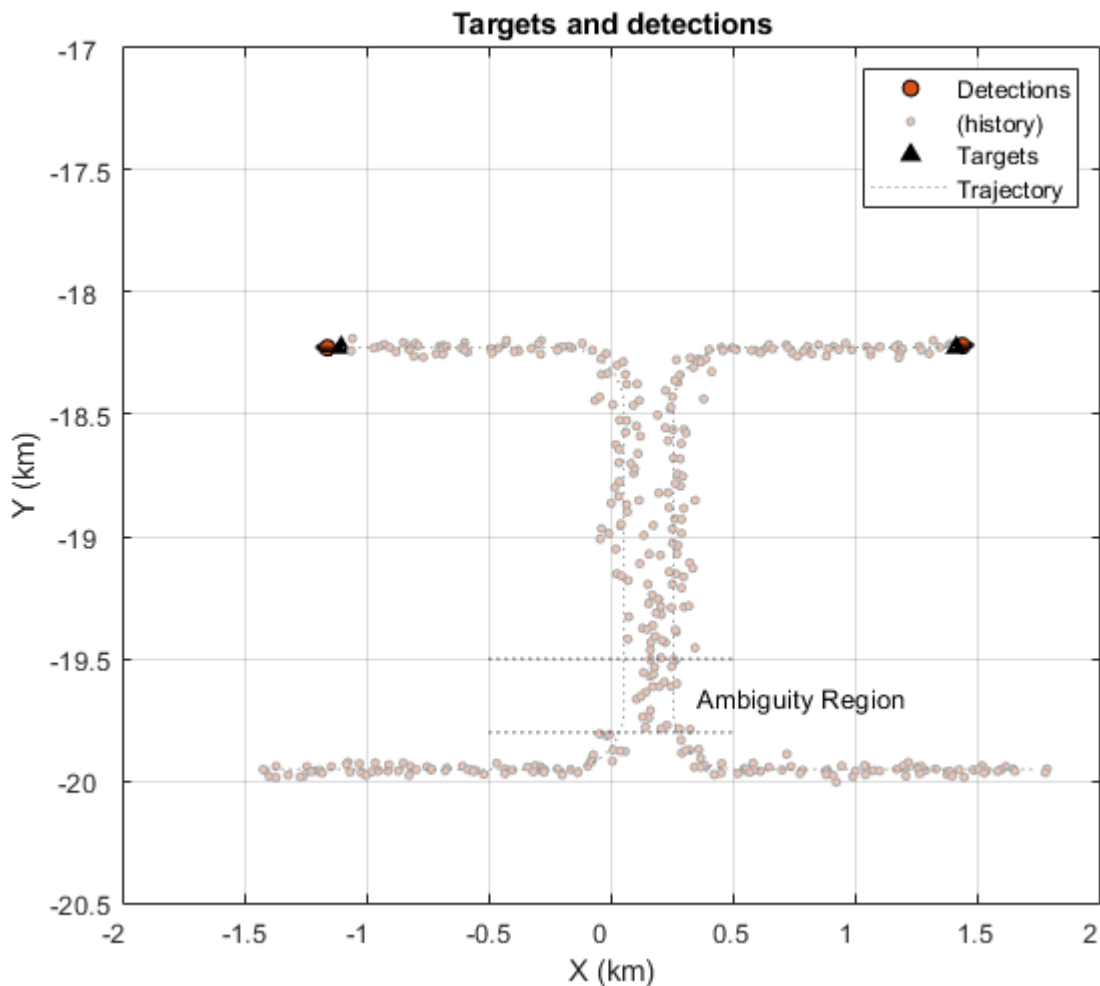


Scenario

To demonstrate a case where sensor reports are ambiguously assigned to tracks, you create a simple scenario. In this scenario, a single radar 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 $Y = -19.5$ km 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.

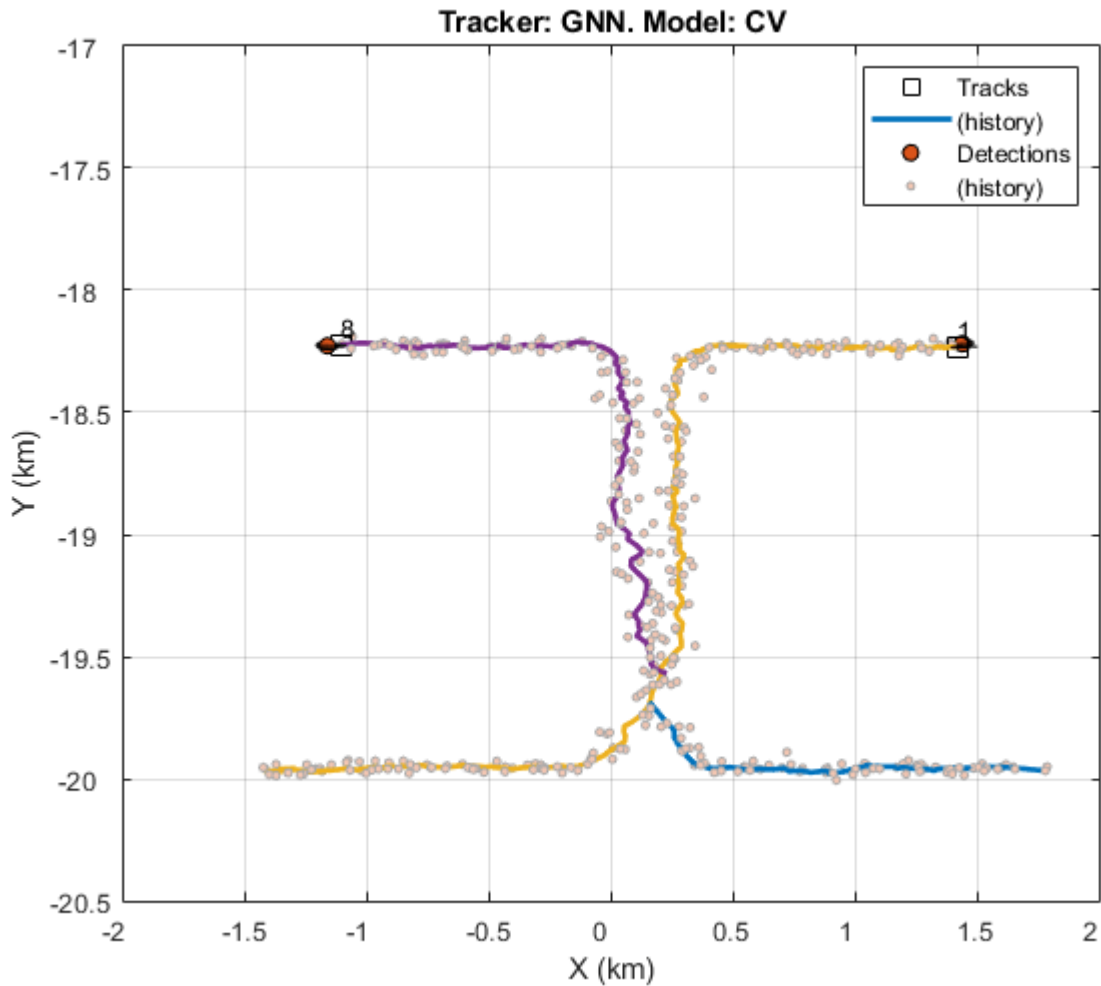
```
numTracks = 20; % Maximum number of tracks
gate = 45;     % Association gate
vol = 1e9;    % Sensor bin volume
beta = 1e-14; % Rate of new targets in a unit volume
pd = 0.8;     % Probability of detection
far = 1e-6;   % False alarm rate
```

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.

```
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.

```
[trackSummary, truthSummary, trackMetrics, truthMetrics,timeGNNCV] = helperRunTracker(dataLog,tra
```



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)
```

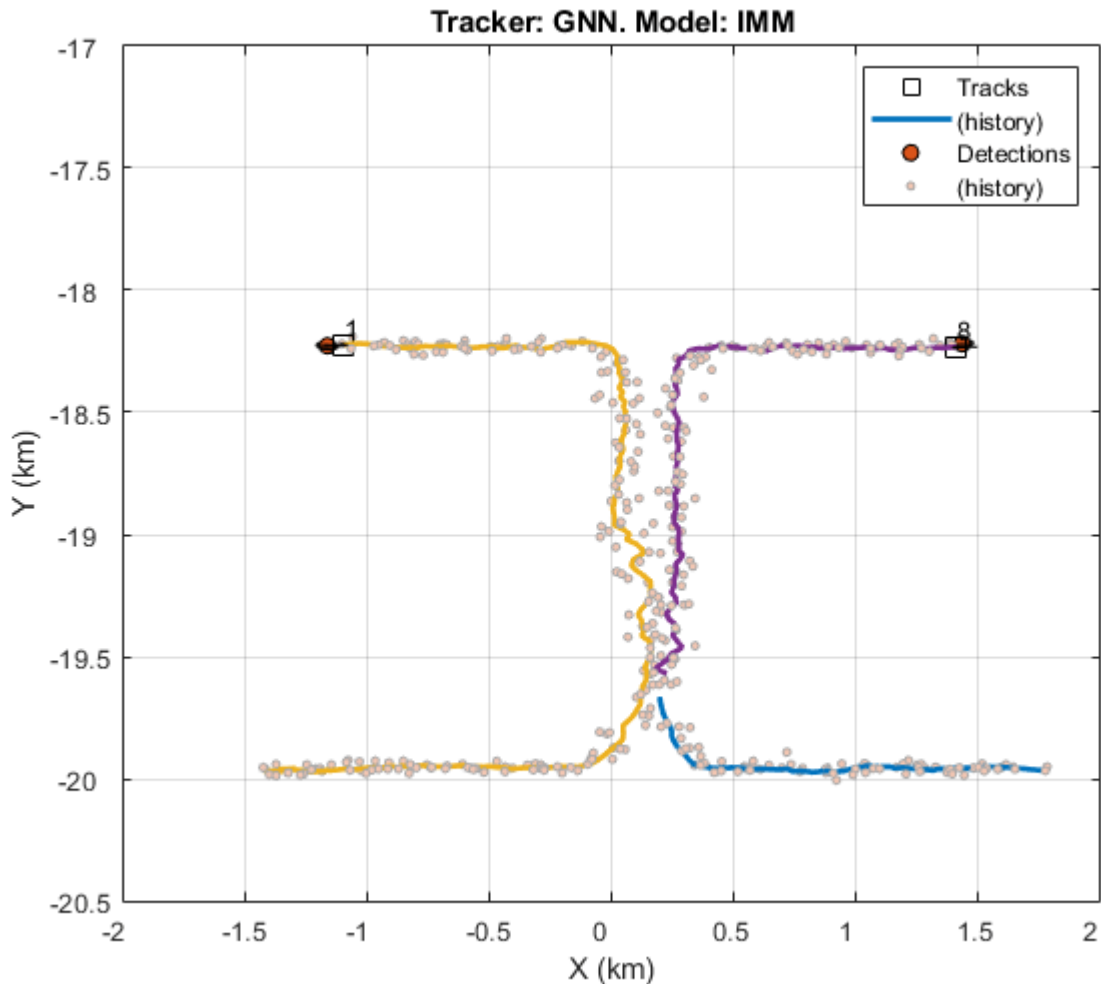
TrackID	posRMS	velRMS	posANEES	velANEES
1	59.461	26.119	7.9331	3.495
2	51.803	39.433	5.7494	3.0362
8	62.914	28.877	7.2004	3.0465
TruthID	posRMS	velRMS	posANEES	velANEES
2	62.147	27.546	8.3597	3.9608
3	56.218	32.109	6.3234	2.6642

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:

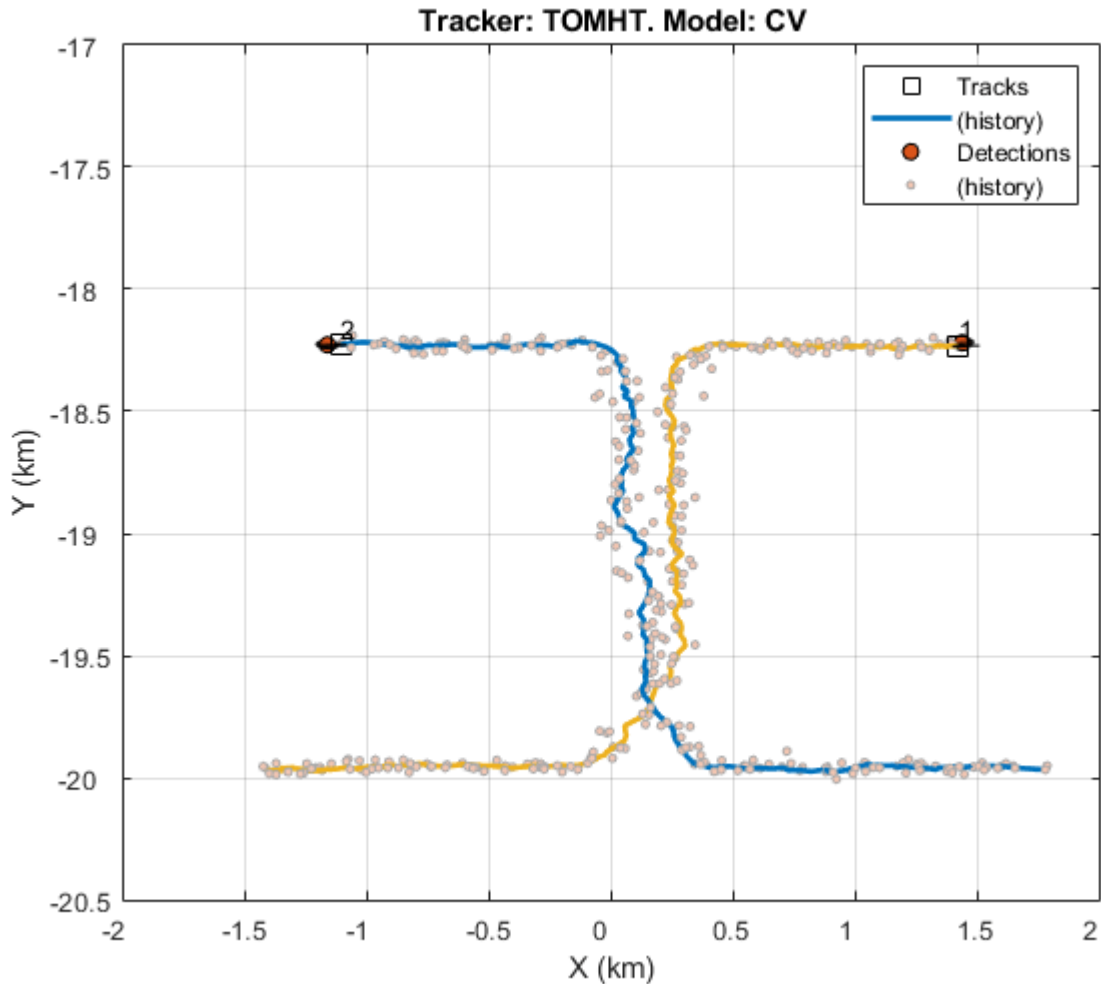
- `AssignmentThreshold` allows a track to be considered unassigned even when there is an assigned detection. This setting allows multiple branches per track. However, the second element of the gate disallows new tracks from assigned detections, to reduce the total number of tracks and improve performance.
- `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
```



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)
```

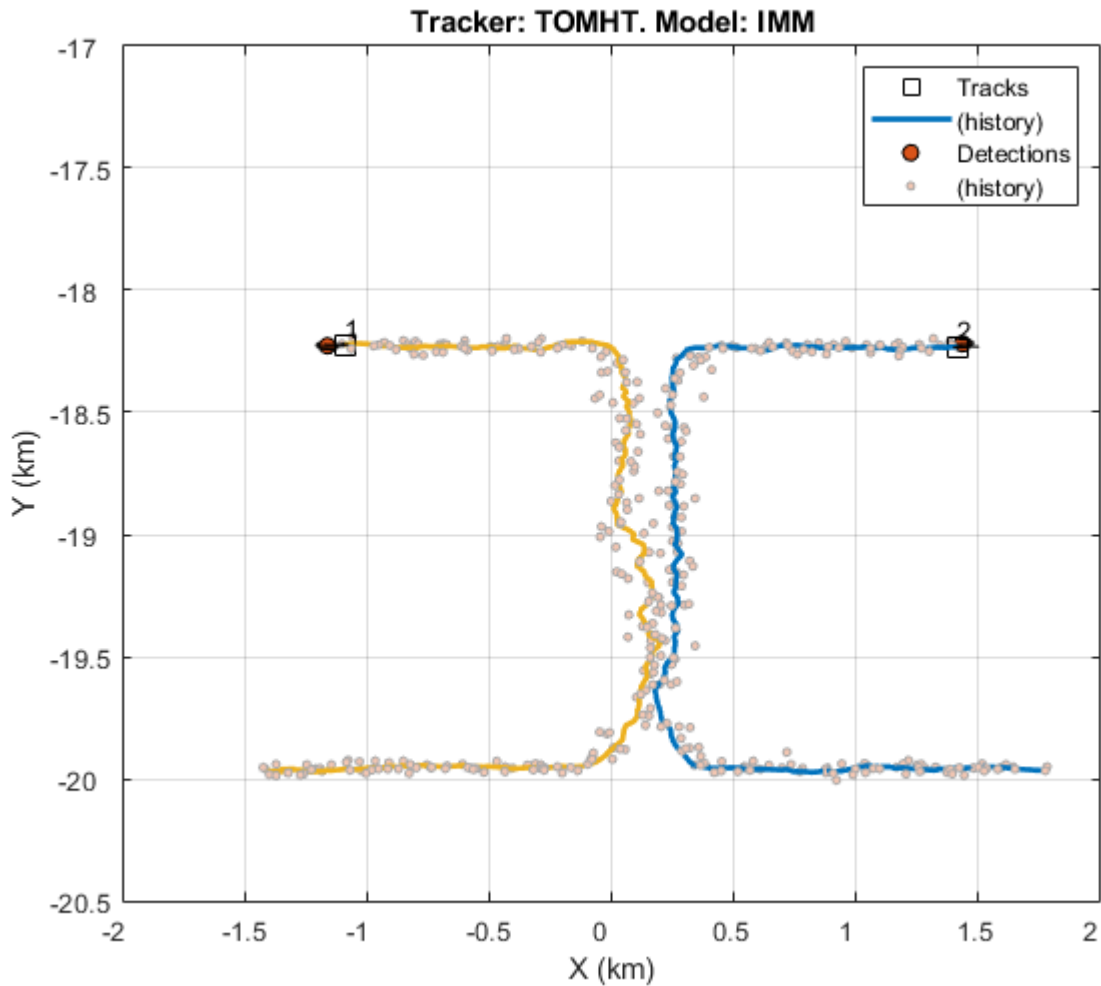
TrackID	posRMS	velRMS	posANEES	velANEES
1	45.553	28.805	4.5831	3.2817
2	61.951	29.849	6.7165	2.733

TruthID	posRMS	velRMS	posANEES	velANEES
2	48.913	28.578	5.0099	3.2762
3	58.178	29.891	6.1356	2.8033

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(dataLo
```



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	55.917	24.434	6.9456	5.5389
2	37.78	25.963	5.249	4.0935

TruthID	posRMS	velRMS	posANEES	velANEES
2	55.917	24.434	6.9456	5.5389
3	37.78	25.963	5.249	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.

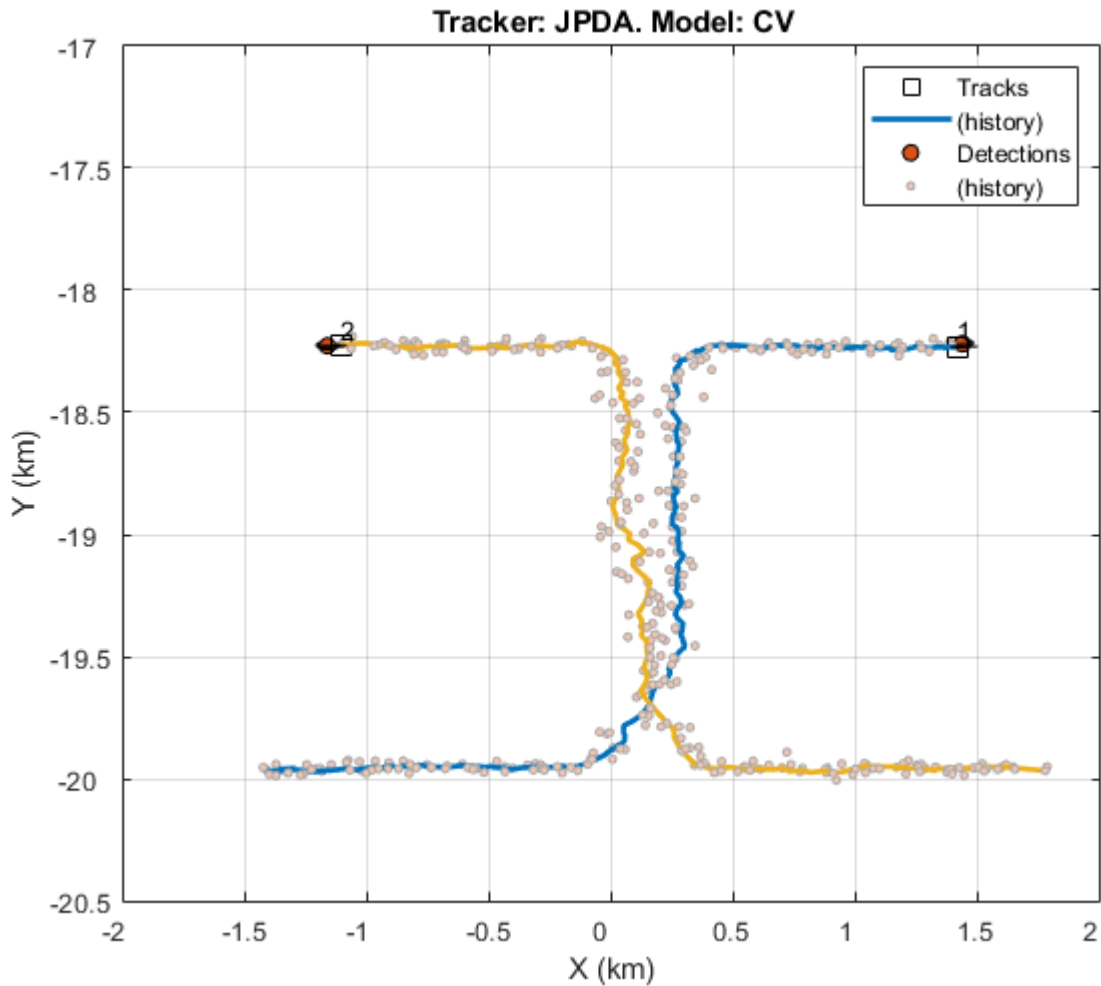
`trackerJPDA` uses the same parameters as `trackerGNN` except:

- `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, ...
```



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)
```

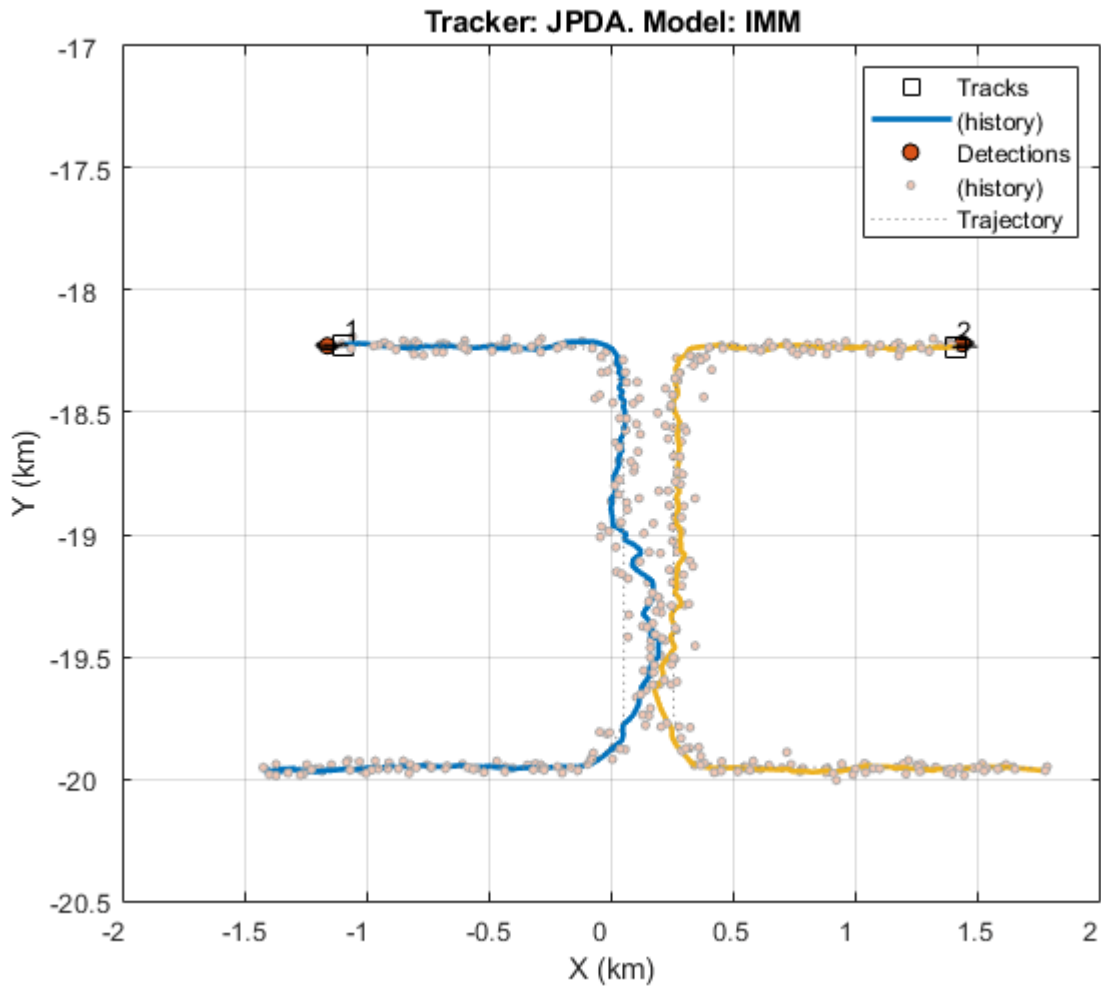
TrackID	posRMS	velRMS	posANEES	velANEES
1	44.918	29.911	5.1894	3.5599
2	58.904	31.484	7.0023	3.0944
TruthID	posRMS	velRMS	posANEES	velANEES
2	47.304	29.606	5.4677	3.5093
3	56.106	31.564	6.5976	3.1815

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)
```

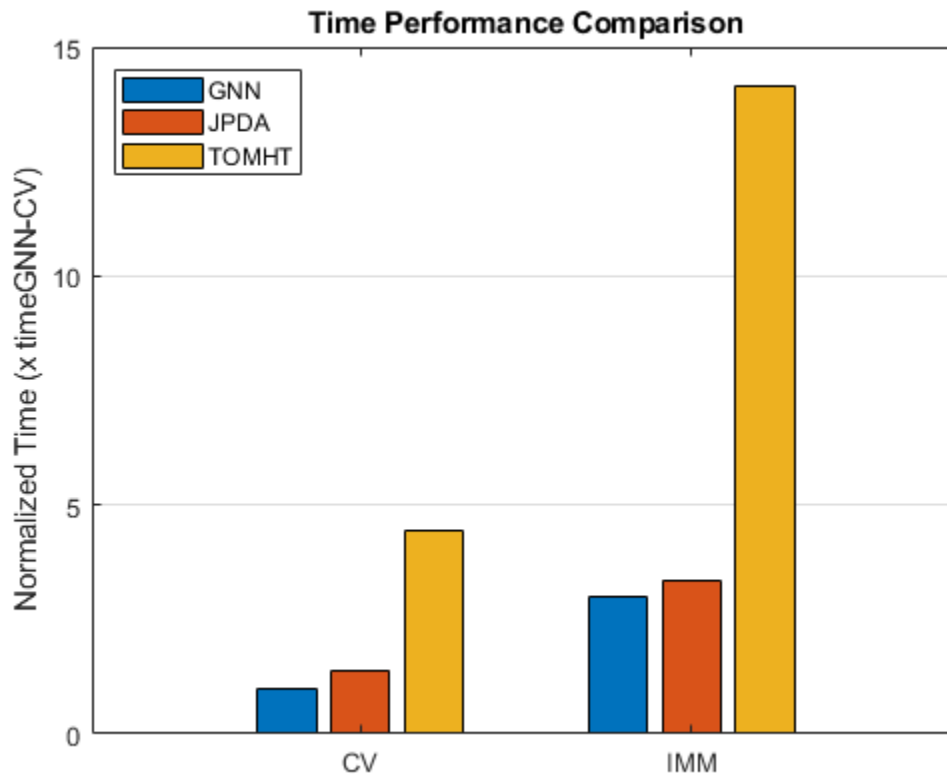
TrackID	posRMS	velRMS	posANEES	velANEES
1	57.172	27.523	7.6831	6.2685
2	39.21	29.184	6.1317	4.769

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 timeJPDA CV 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;
```

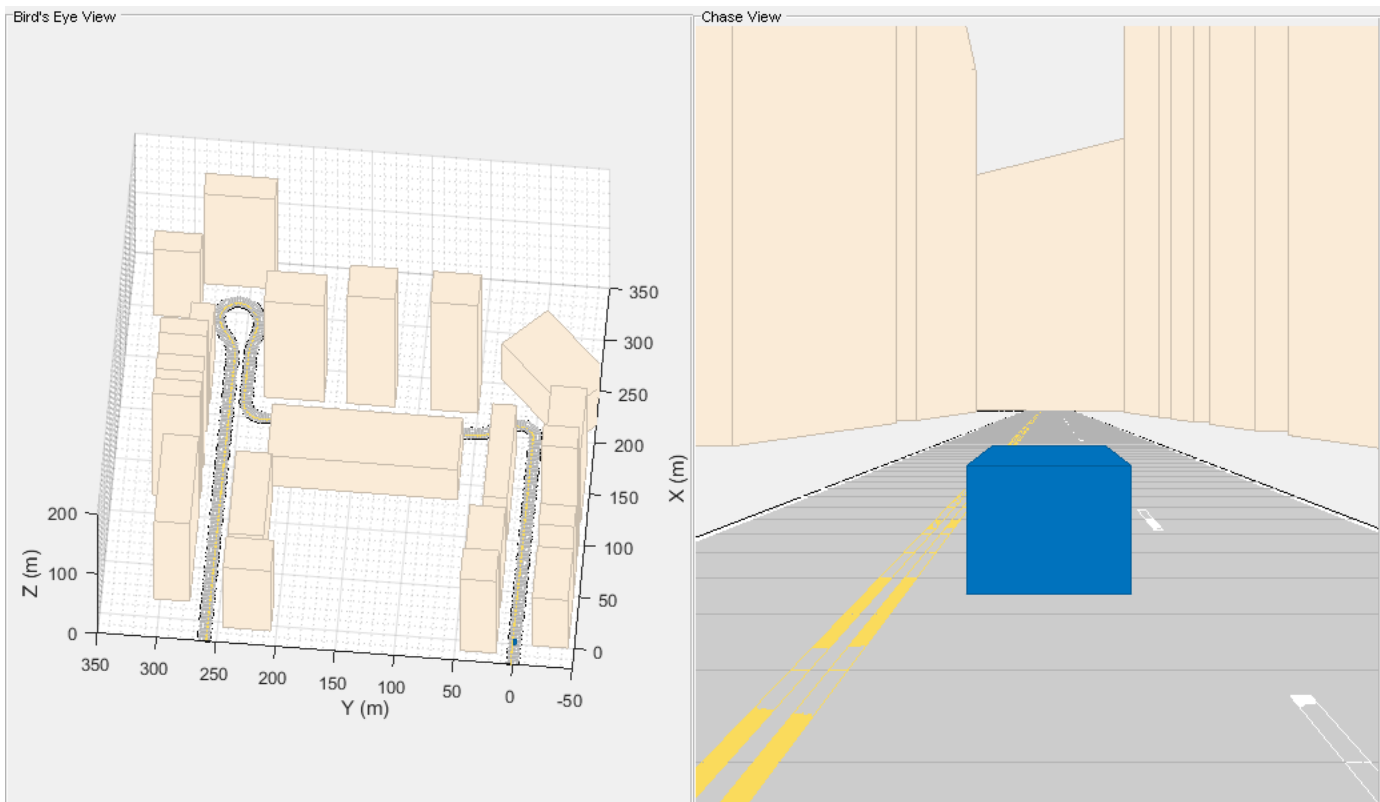
```

        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, ...
 posV0, orientV0, ...
 posEst, orientEst, velEst] ...
= helperPreallocateData(numIMUSamples, numCameraSamples);

% Set measurement noise parameters for the visual odometry fusion.
RposV0 = 0.1;
RorientV0 = 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.
```

```
[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);

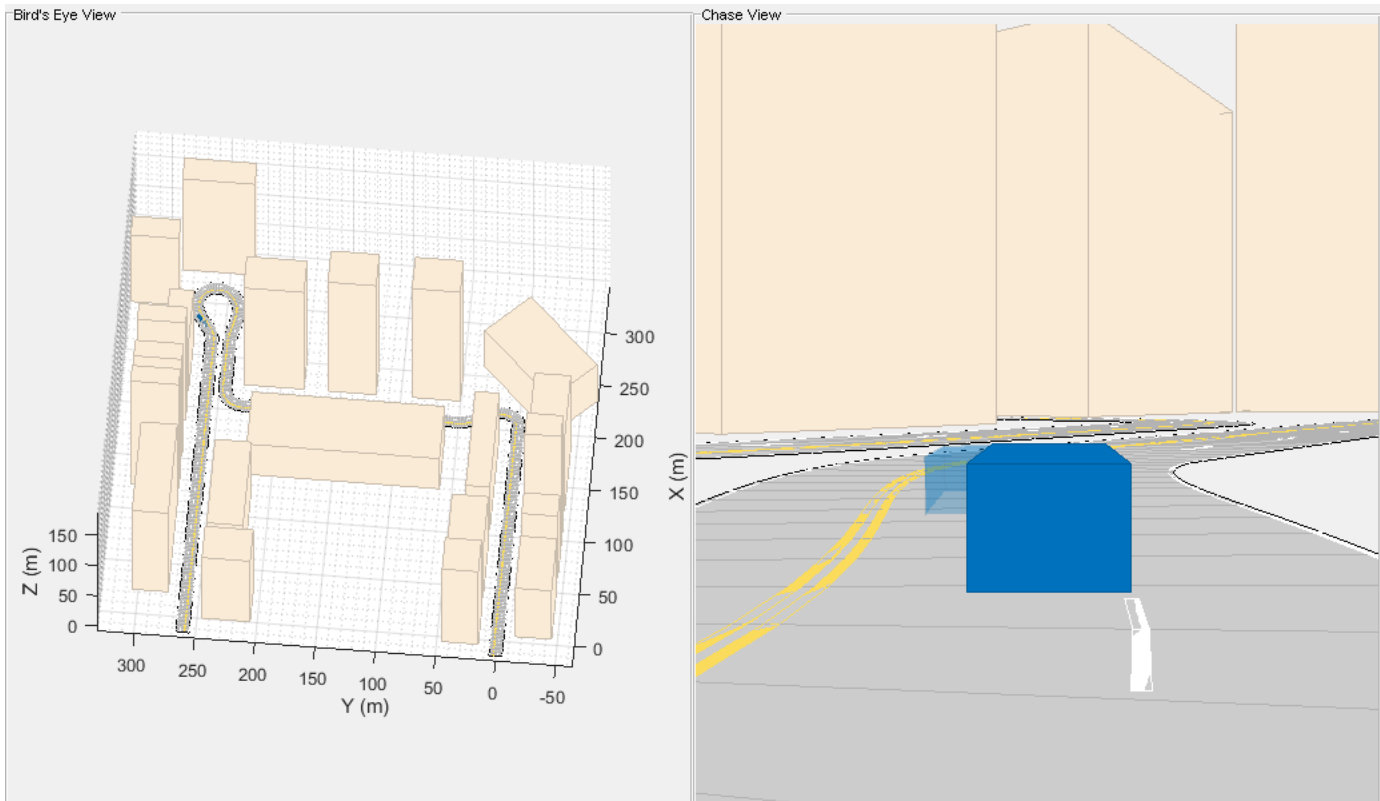
    cameraIdx = cameraIdx + 1;
end

[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
```



Plot the Results

Plot the ground truth vehicle trajectory, the visual odometry estimate, and the fusion filter estimate.

```
figure
if useV0
    plot3(pos(:,1), pos(:,2), pos(:,3), '-.', ...
          posV0(:,1), posV0(:,2), posV0(:,3), ...
          posEst(:,1), posEst(:,2), posEst(:,3), ...
          'LineWidth', 3)
    legend('Ground Truth', 'Visual Odometry (V0)', ...
          'Visual-Inertial Odometry (VI0)', '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
```



```

% Calculate visual odometry measurements.
posV0 = scaleV0*pos + drift;
orientV0 = 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, ...
        posV0, orientV0, ...
        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.
posV0 = zeros(numCameraSamples, 3);
orientV0 = 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.

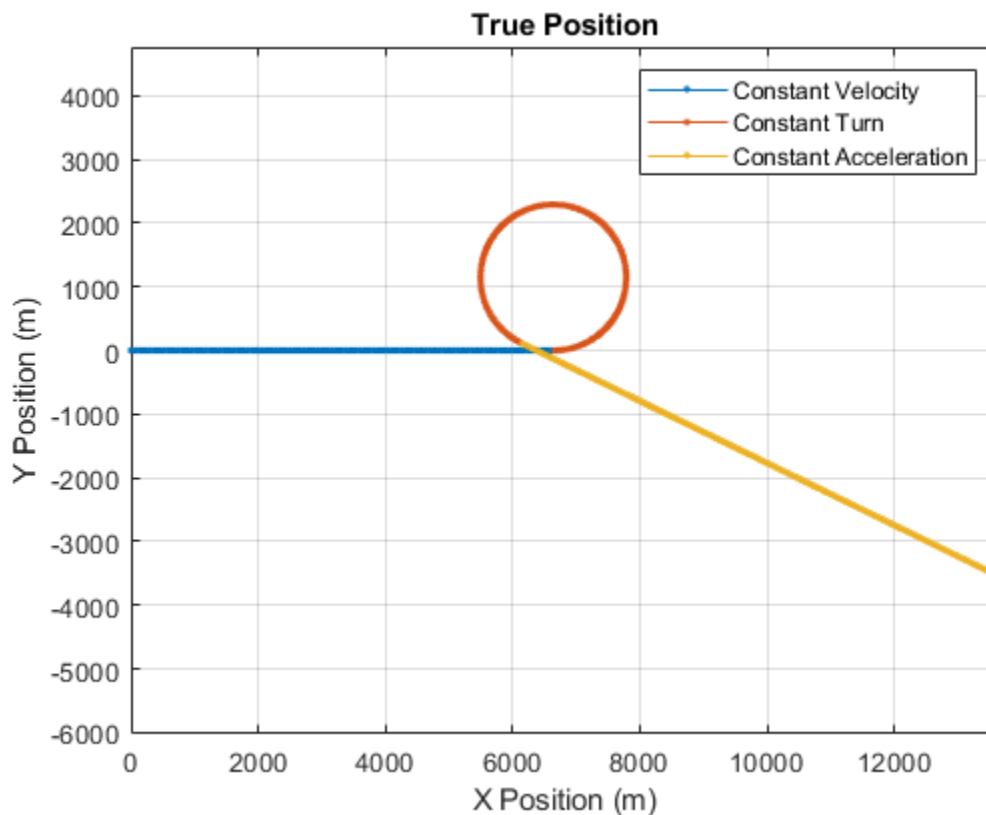
Tracking Maneuvering Targets

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/s².

```
[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;
```

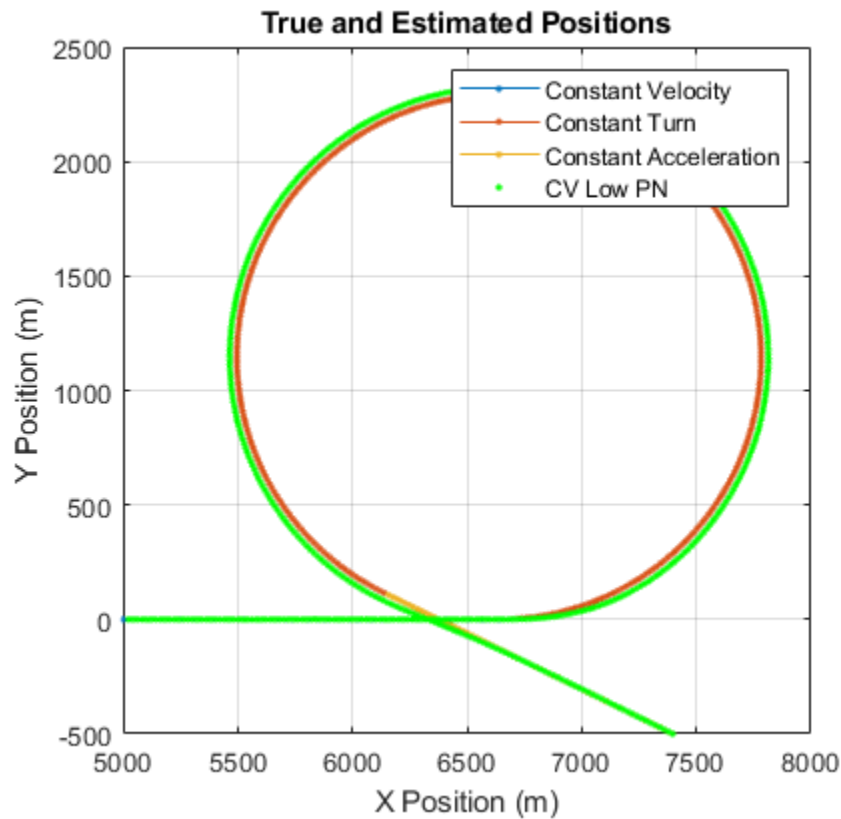
Track Using a Constant-Velocity Filter

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));
```

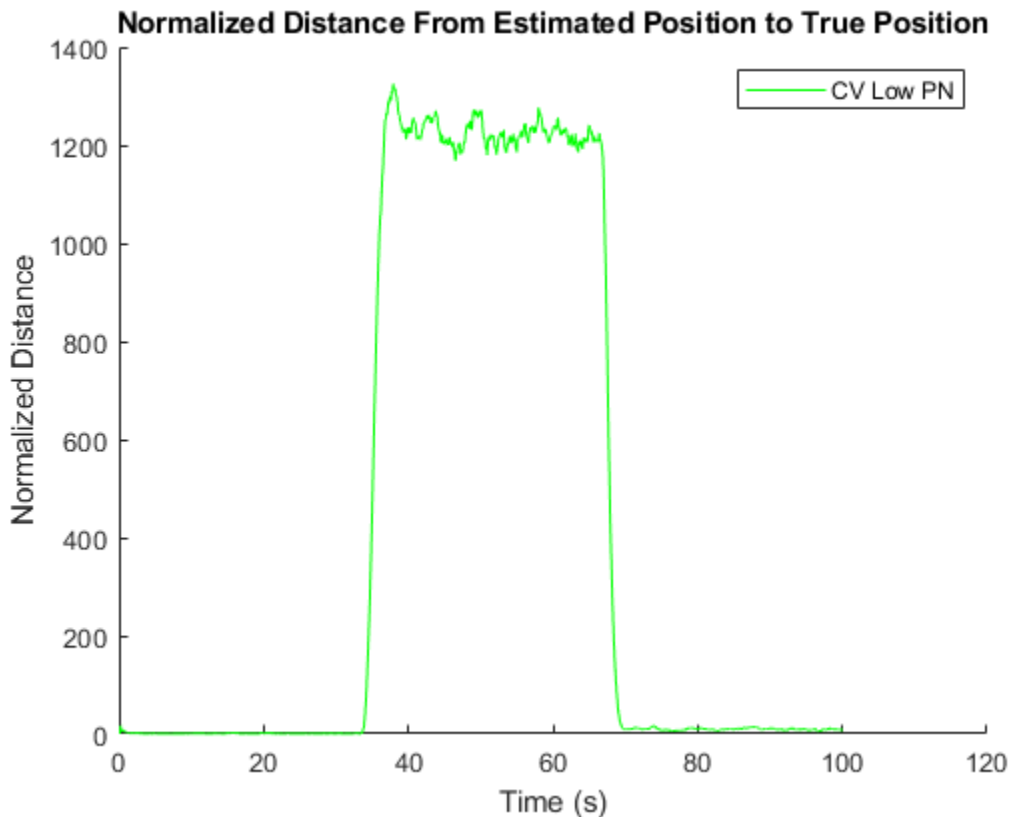
For each measurement, you predict the filter, calculate the distance of the predicted state from the true position, and correct the filter using the measurement to obtain the filtered estimate of the position.

```
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])
```



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.

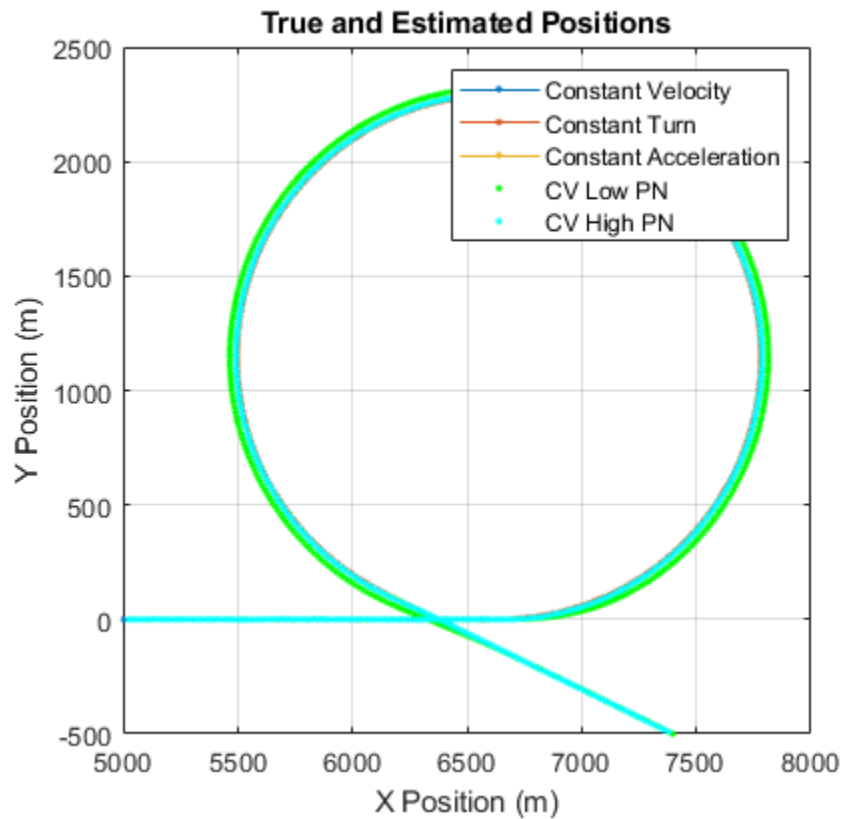
```
fig2 = figure;
hold on
plot((1:numSteps)*dt,dist,'g','DisplayName', 'CV Low PN')
title('Normalized Distance From Estimated Position to True Position')
xlabel('Time (s)')
ylabel('Normalized Distance')
legend
```



Increase the Process Noise

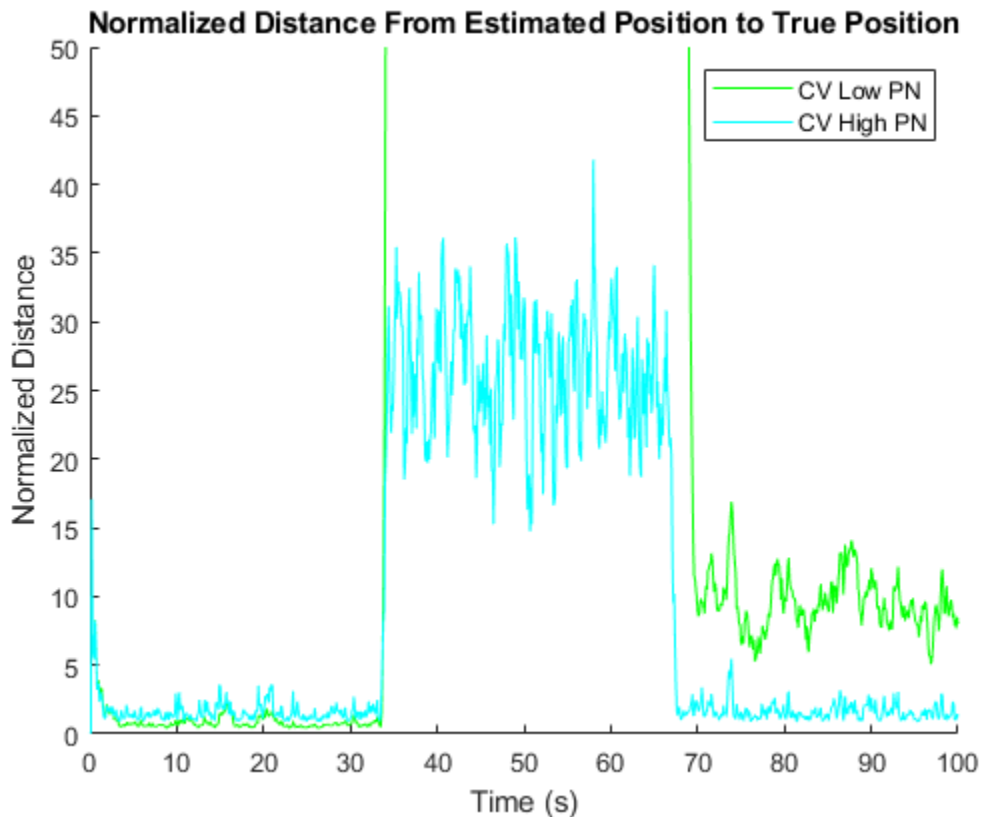
One possible solution is to increase the process noise. The process noise represents the unmodeled terms in the motion model. For a constant velocity model, these are the unknown acceleration terms. By increasing the process noise, you allow for larger uncertainty in the motion model, which causes the 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])
```



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])
```



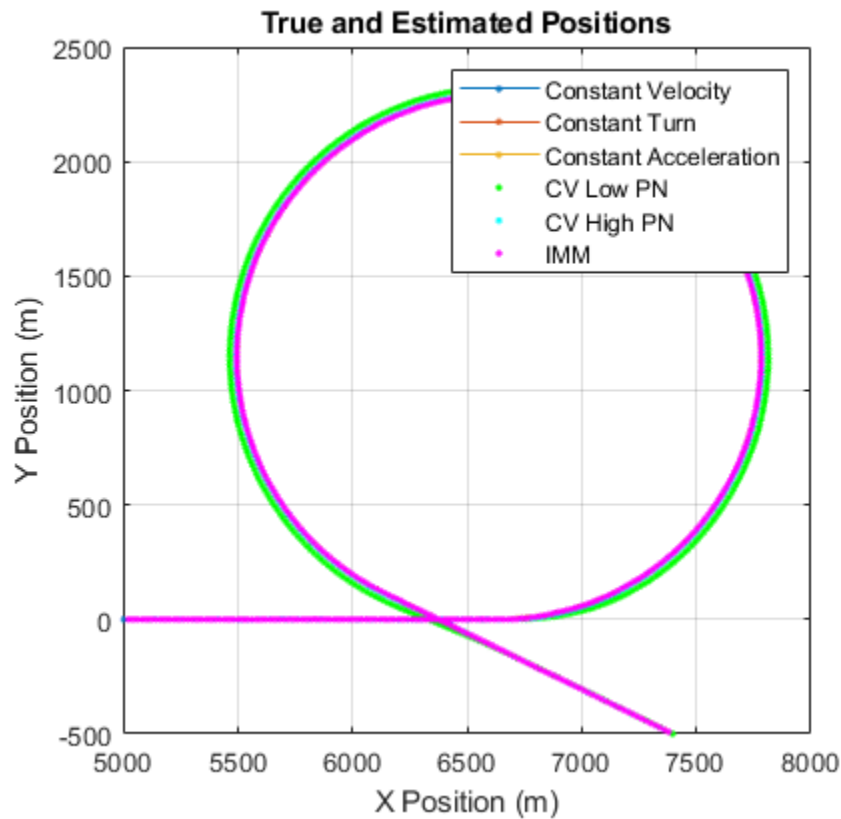
Use an Interacting Motion-Model Filter

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.

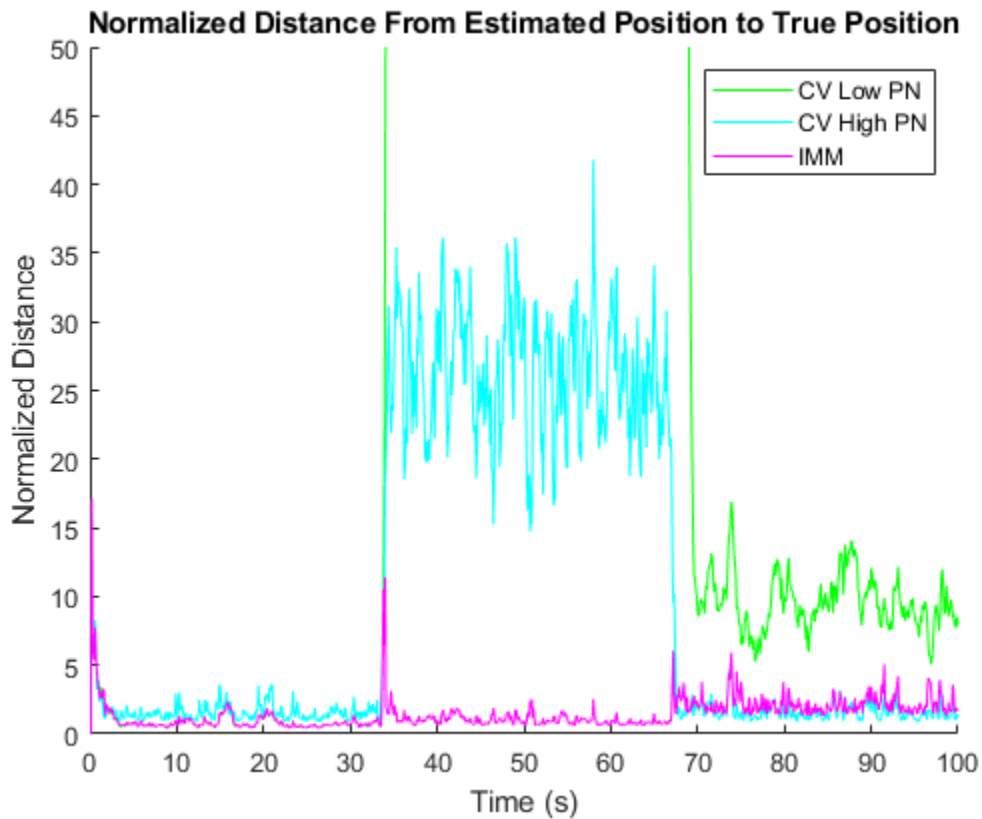
```
dist = zeros(1,numSteps);
estPos = zeros(3,numSteps);
modelProbs = zeros(3,numSteps);
modelProbs(:,1) = imm.ModelProbabilities;
for i = 2:size(measPos,2)
    predict(imm, dt);
    dist(i) = distance(imm,truePos(:,i)); % Distance from true position
    estPos(:,i) = positionSelector * correct(imm, measPos(:,i));
    modelProbs(:,i) = imm.ModelProbabilities;
end
figure(fig1)
plot(estPos(1,:),estPos(2:,:),'.m','DisplayName','IMM')
```

The tracking IMM filter is able to track the maneuvering target in all three parts of the motion.

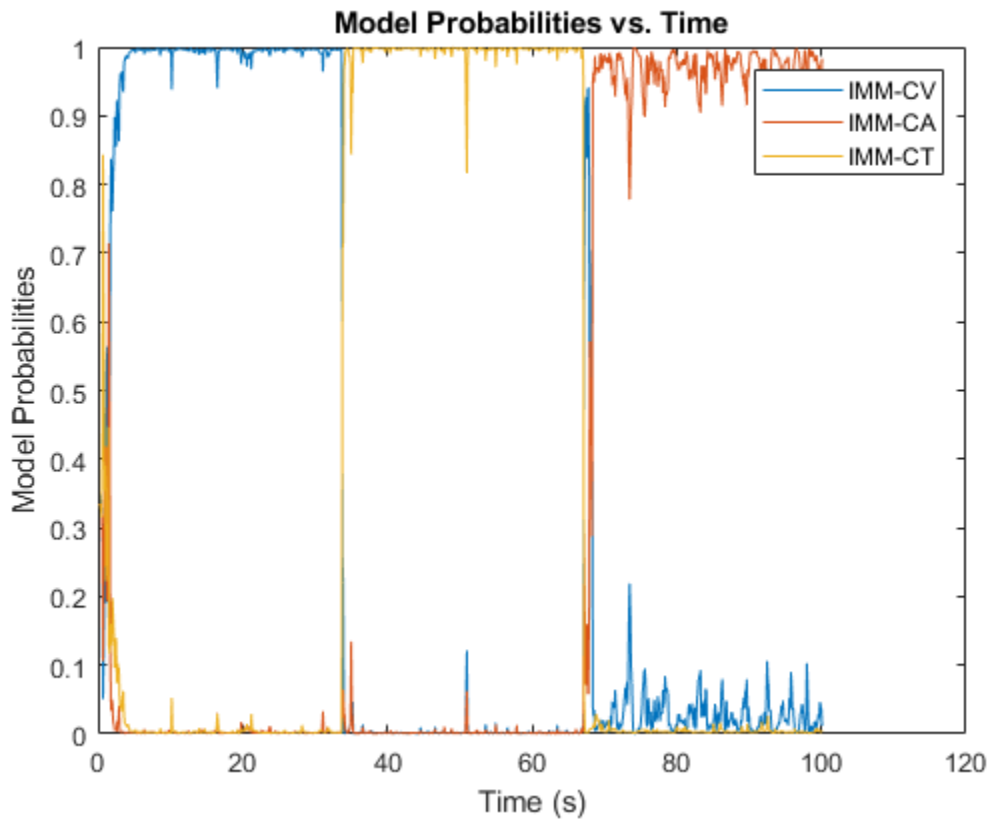
Examining the distance between the filter's predicted state and the true position, you see that the IMM filter is able to reduce the distance for all portions of the motion. In fact, the IMM filter is better at tracking the motion than both other constant velocity models used earlier.

```
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);
```

```

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 Generation” on page 6-215 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” on page 6-215 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
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, detectableTrackIDs);

        % 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',rowTimes));

        % Update display with detections, coverages, and tracks
        theaterDisplay(detBuffer, covcon, confirmedTracks, assignmentTable, truePoses, sensPlatIDs);

        % Empty buffers
        detBuffer = {};
        sensorConfigBuffer = [];
    end
end

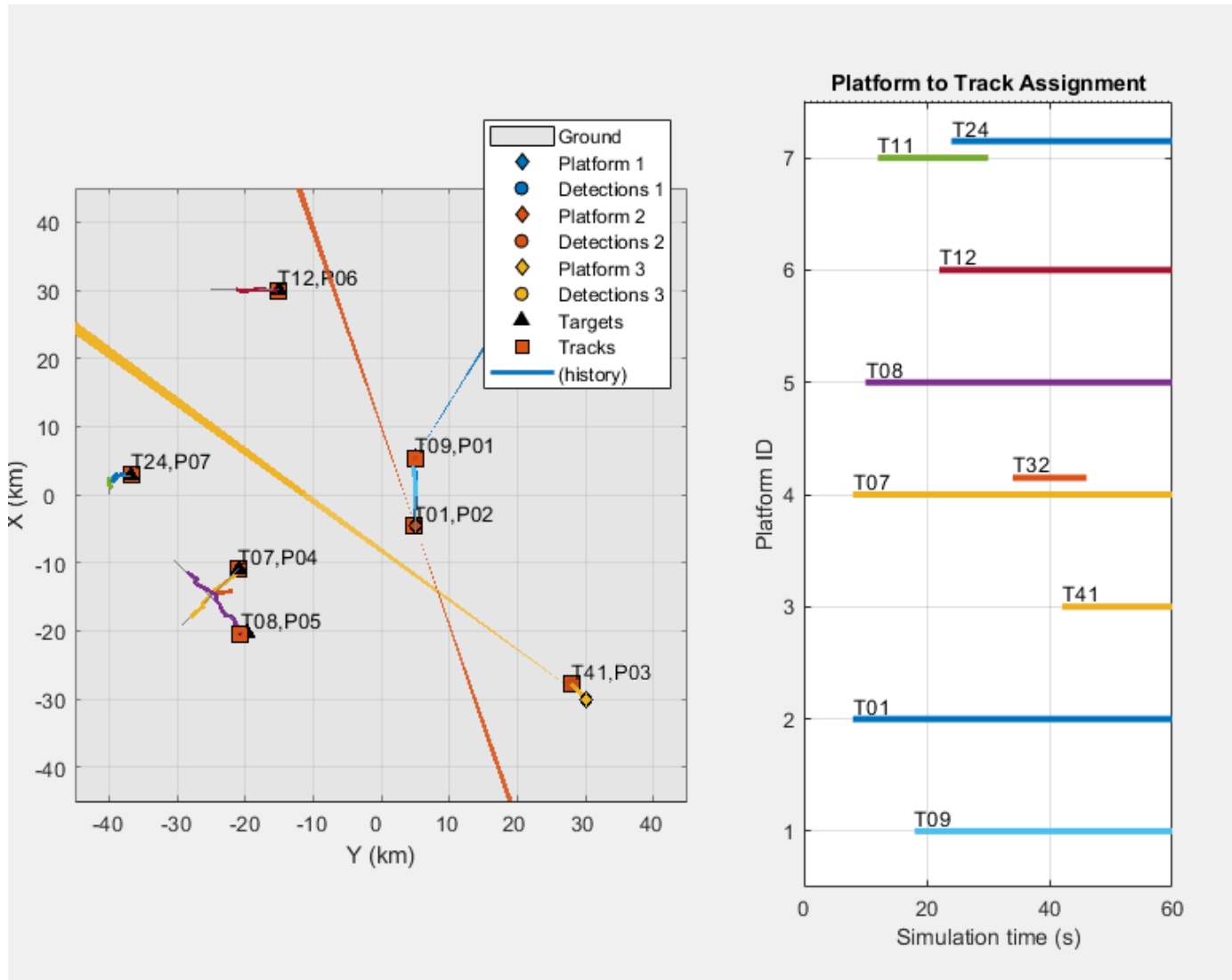
```

```

% 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*.

```

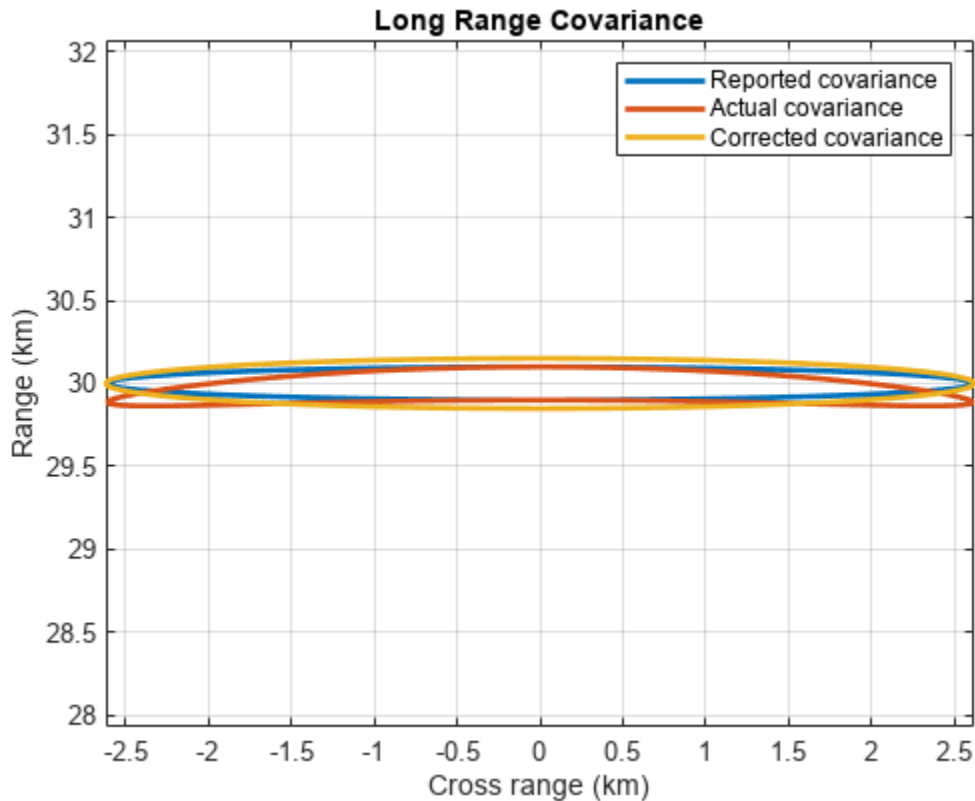
endTime = assignmentTable.Time(end);
assignmentTable(endTime,{'TrackID','AssignedTruthID','TotalLength','DivergenceCount','Redundancy

```

```
ans=9x6 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)
```

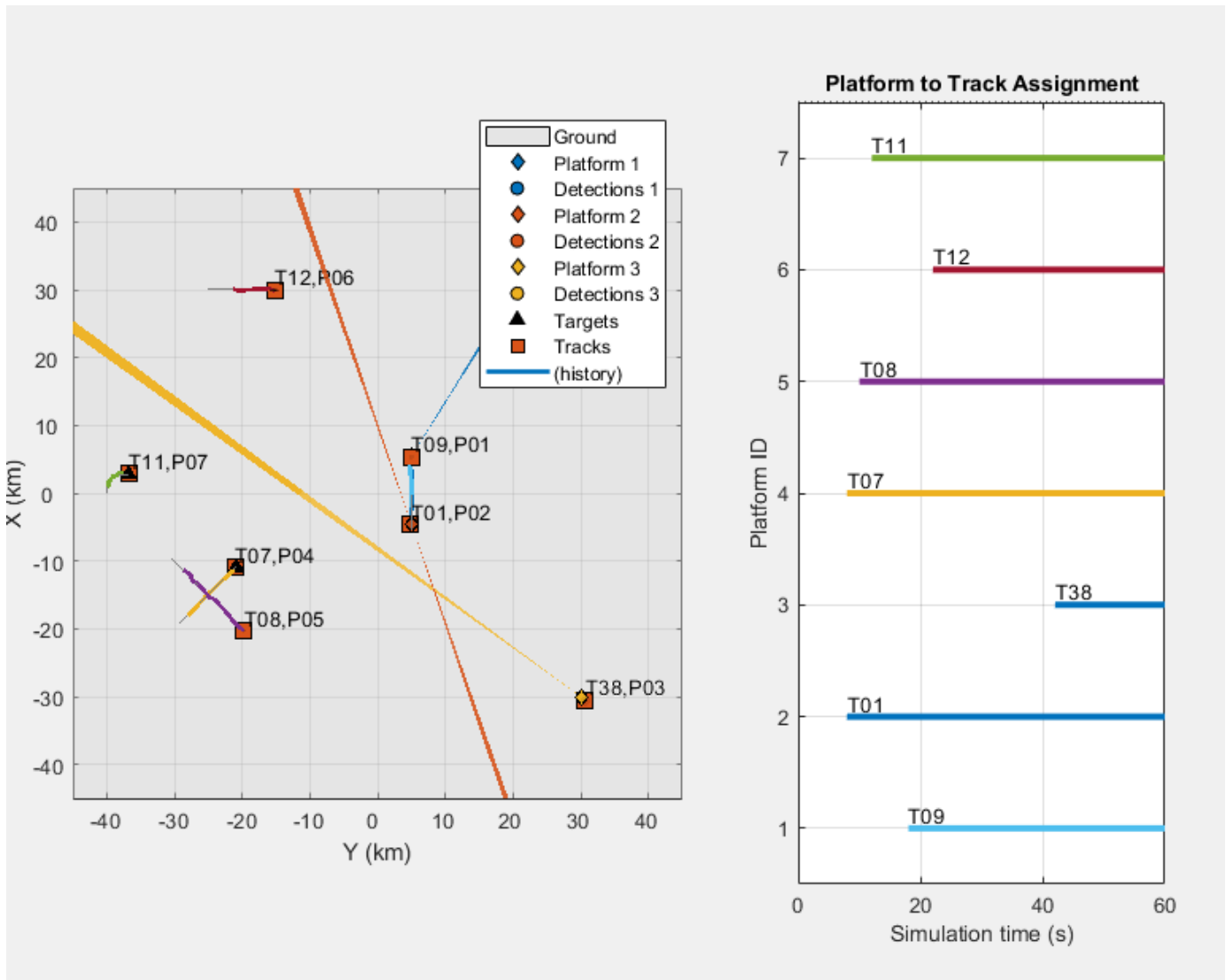



To account for the concave shape of the actual covariance at long ranges, the `longRangeCorrection` supporting function constrains the condition number of the reported measurement noise. The corrected measurement covariance shown above is constrained to a maximum condition number of 300. In other words, 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);
```



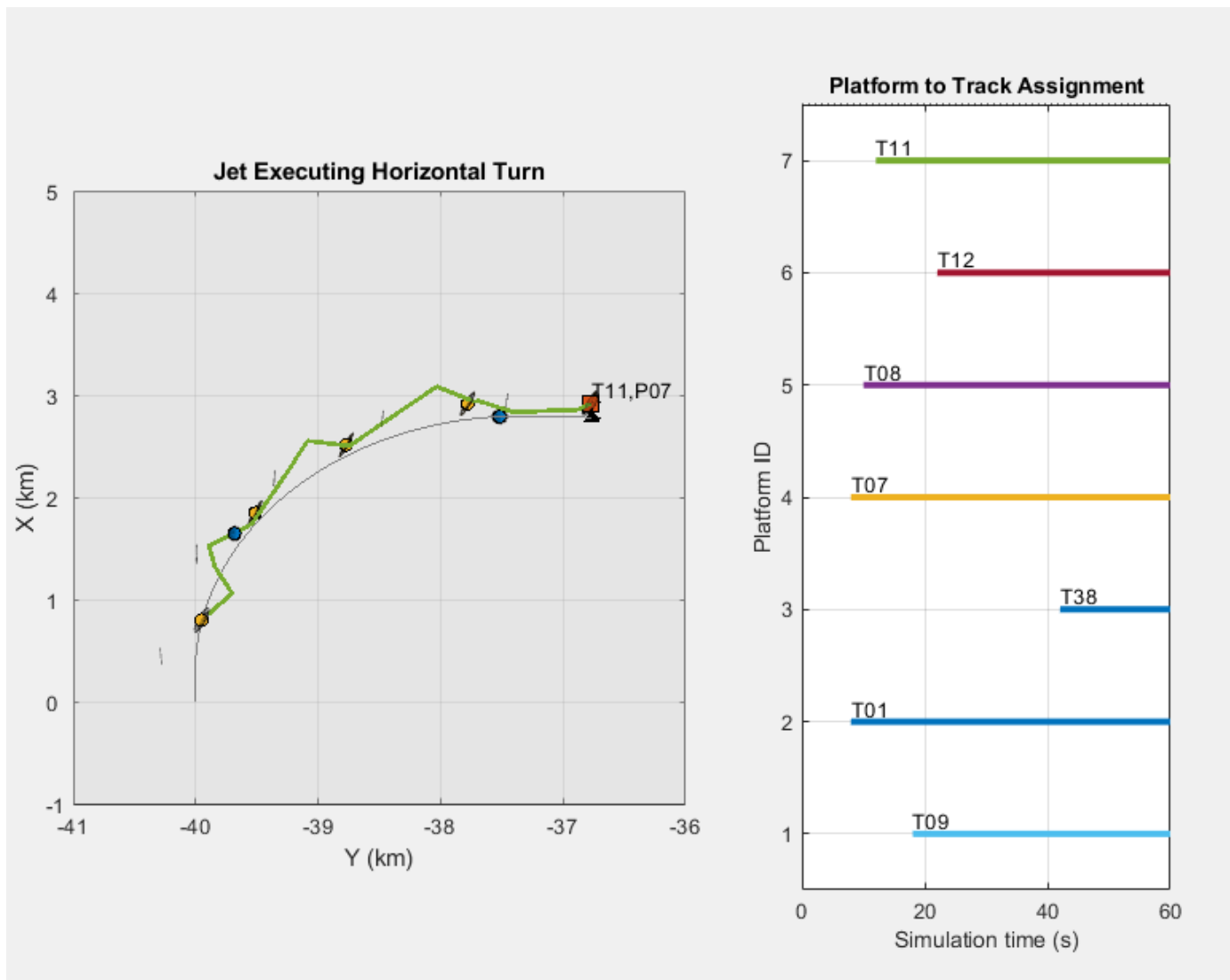
```
endTime = correctedAssignmentTable.Time(end);
correctedAssignmentTable(endTime,{'TrackID', 'AssignedTruthID', 'TotalLength', 'DivergenceCount', 'RedundancyCount'}
```

ans=7x6 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

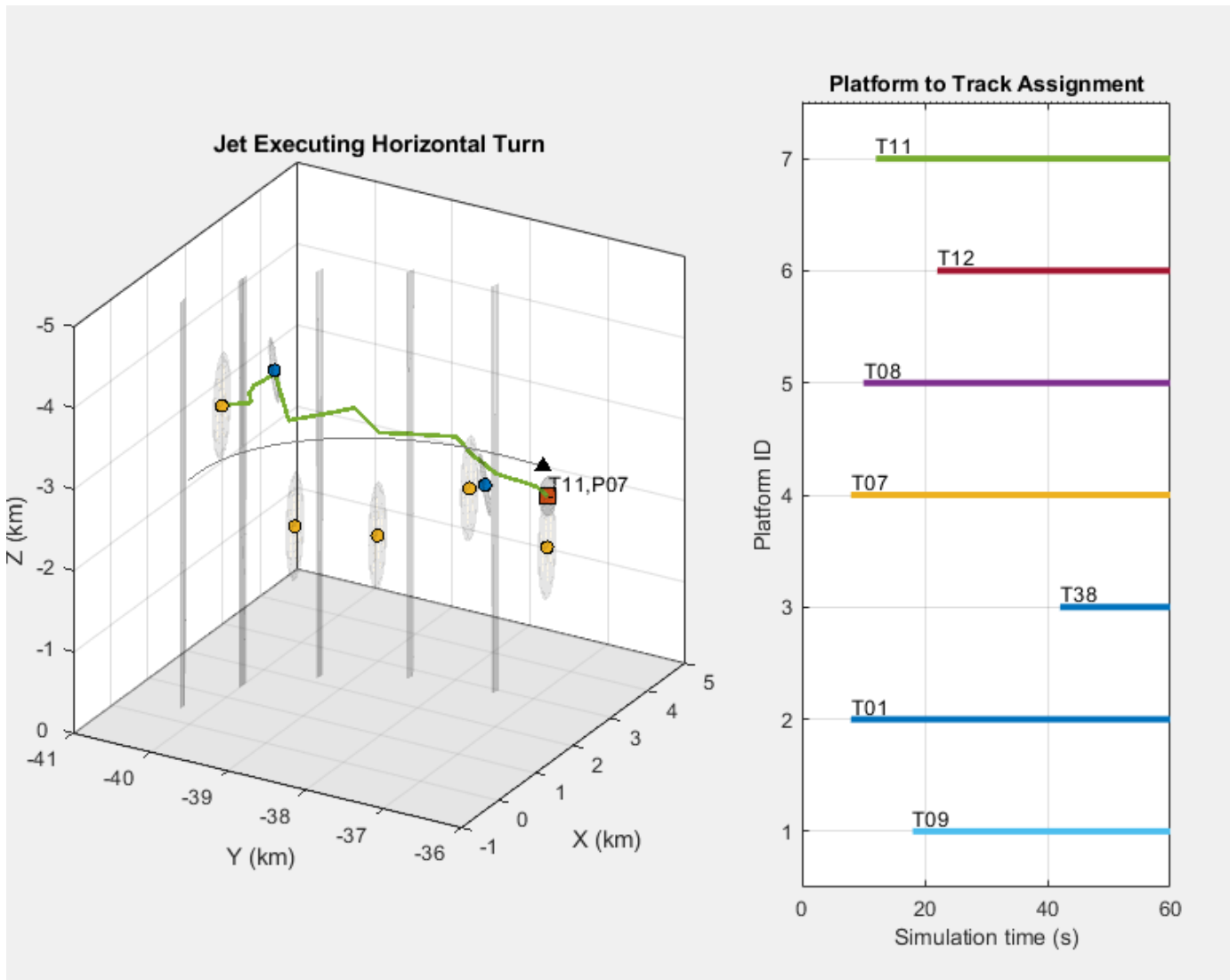
The preceding figure shows that by applying the long-range correction, no track-drops or multiple tracks are generated for the entire scenario. In this case, there is exactly one track for each platform detected by the surveillance network.

```
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')
```



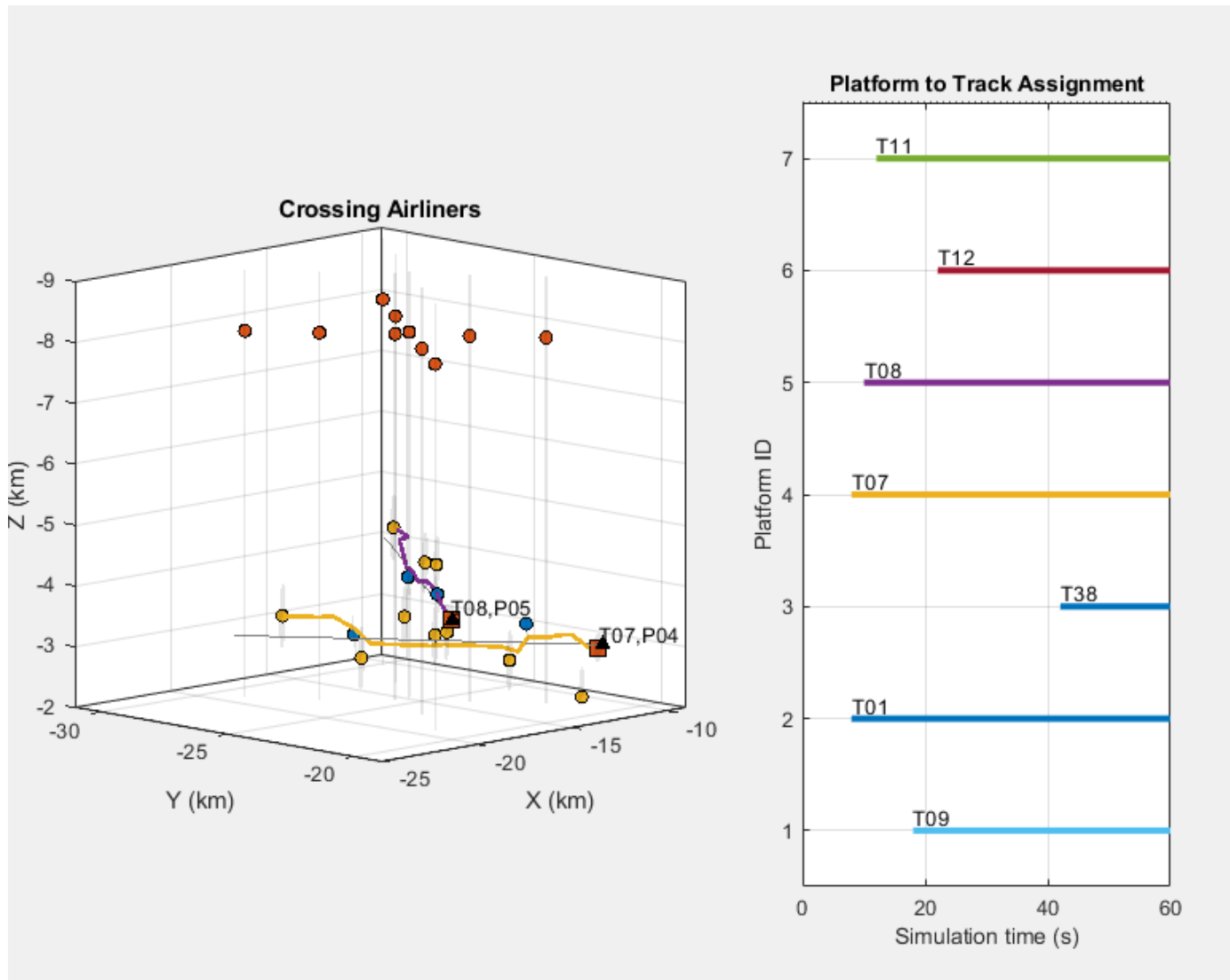
Zoom in the view in which the jet is executing a horizontal turn, the track follows the maneuvering target relatively well, even though the motion model used in this example is constant velocity. Tracking the maneuver could be further improved by using an interacting multiple-model (IMM) filter such as the trackingIMM filter.

```
view([-60 25])
```



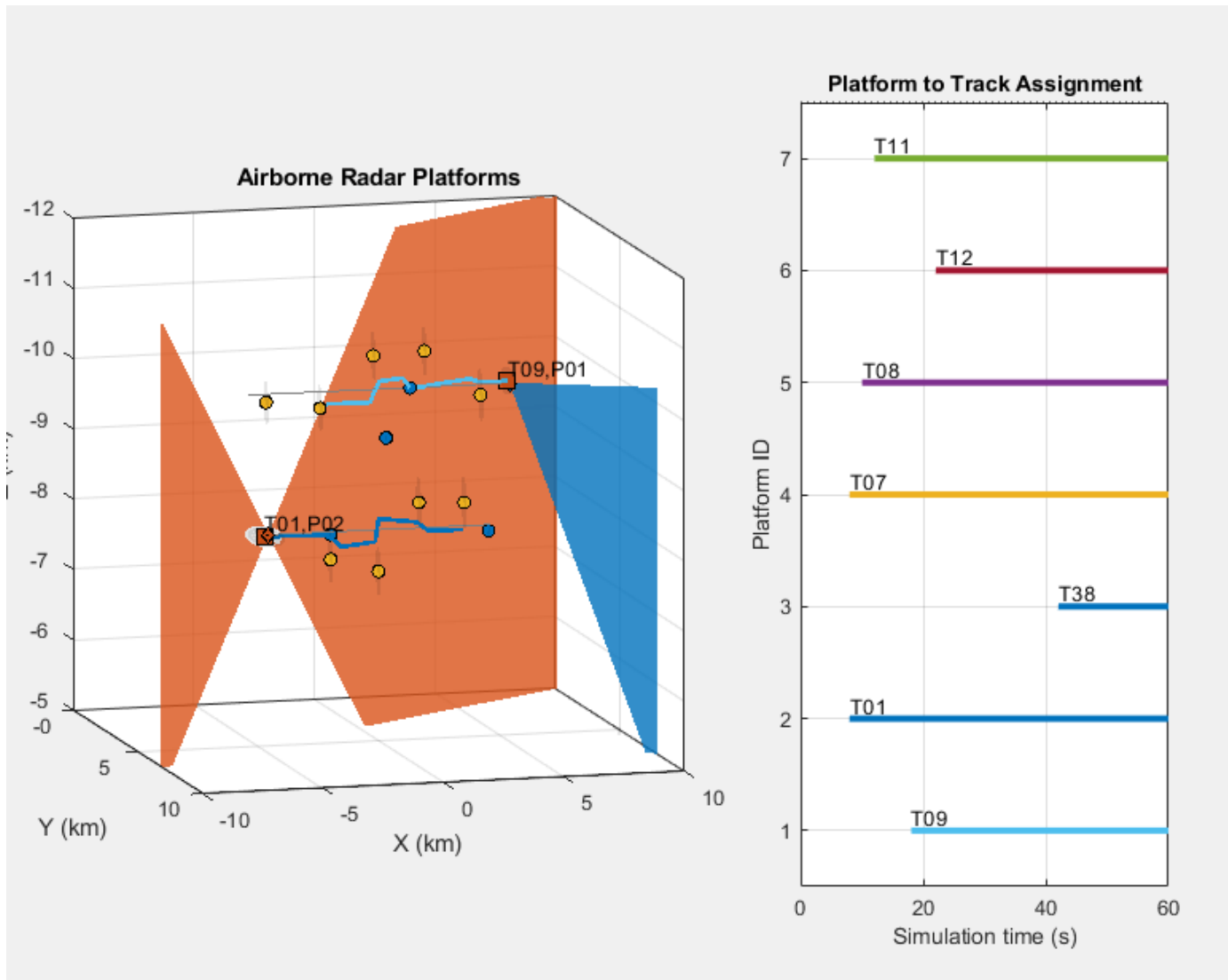
From another view in which the jet is executing a horizontal turn, you can see that the altitude is estimated correctly, despite the inaccurate altitude measurements from the sensors. One of the sensors does not report altitude at all, as seen by the large vertical ellipsoids, while the other two sensors underestimate their uncertainty in the altitude.

```
xlim([-25000 -9000]); ylim([-31000 -19000]); zlim([-9000 -2000]);
view([-45 10])
title('Crossing Airlines')
```



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);

% Airlines 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 radar sensor 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).FieldOfView(2);
                if inFov
                    isDetectable(iTrk) = inFov;
                    k = numsteps; %#ok<FXSET>
                    iSensor = numSensors; %#ok<FXSET>
                end
            end
        end
    end
end

```

```

        end
    end
end

end

trackIDs = [tracks.TrackID]';
trackIDs = trackIDs(isDetectable);
end

```

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 radar 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

```


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 `fusionRadarSensor`.

```
sensorIndex = 1; % Identifies originating sensor of each detection
radar = fusionRadarSensor(sensorIndex,'Rotator', ...
    'MountingLocation', [0 0 -5], ... % m
    'UpdateRate', 12.5, ... % Hz
```

```

'ReferenceRCS', 0, ... % dBsm
'ReferenceRange', 375e3, ... % m
'ScanMode', 'Mechanical and electronic', ...
'MechanicalAzimuthLimits', [0 360], ... % deg
'MechanicalElevationLimits', [0 0], ... % deg
'ElectronicAzimuthLimits', [0 0], ...
'ElectronicElevationLimits', [-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;

```

Note, to model a radar system that does not perform scanning in one of the angular dimensions the field of view in that dimension should be set to a value that is slightly larger than the value spanned by the corresponding mechanical scan limits. Thus, in this example the elevation field of view of the fusionRadarSensor object is set to 47.1 degrees, while according to the specification the elevation field of view of the modeled system is 47 degrees.

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 fusionRadarSensor.

```

% Create right facing radar by setting radar's yaw to 90 degrees.
sensorIndex = sensorIndex+1;

```

```

rightRadar = fusionRadarSensor(sensorIndex, 'Sector', ...
    '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', ...
    'ElectronicAzimuthLimits', [-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 `fusionRadarSensor`.

```

% Create an electronically scanning rectangular array radar.
sensorIndex = sensorIndex+1;
radar = fusionRadarSensor(sensorIndex, 'Raster', ...
    'MountingLocation', [0 0 -5], ...    % m
    'UpdateRate', 25, ...                % Hz
    'ReferenceRCS', 0, ...               % dBsm
    'ReferenceRange', 350e3, ...         % m
    'ScanMode', 'Electronic', ...
    'ElectronicAzimuthLimits', [-30 30], ... % deg
    'ElectronicElevationLimits', [-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

```

% 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
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.

```

rng(2018); % Set random seed for repeatable results

% Create a theaterPlot to show the true and measured positions of the detected targets and platform
theaterDisplay = helperMultiPlatDisplay(scene);

title('Multiplatform Radar Scenario');
legend('show');

% Show 3D view of the scenario.
view(-60,10);

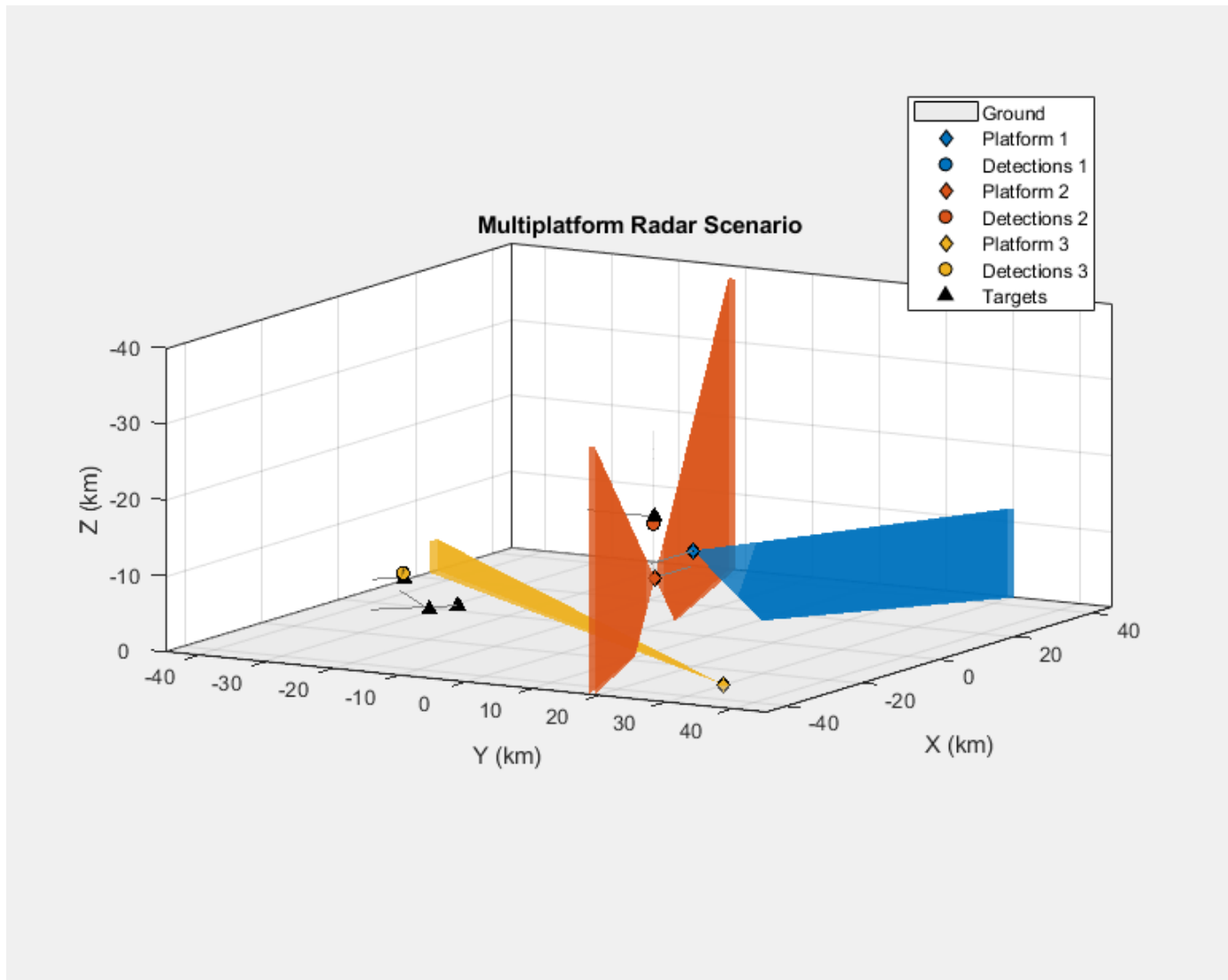
% Log all the detections
detLog = {};
timeLog = [];

while advance(scene)
    % Generate detections from radars on each platform.
    [dets,configs] = detect(scene);

    % Update display with current beam positions and detections.
    theaterDisplay(dets);

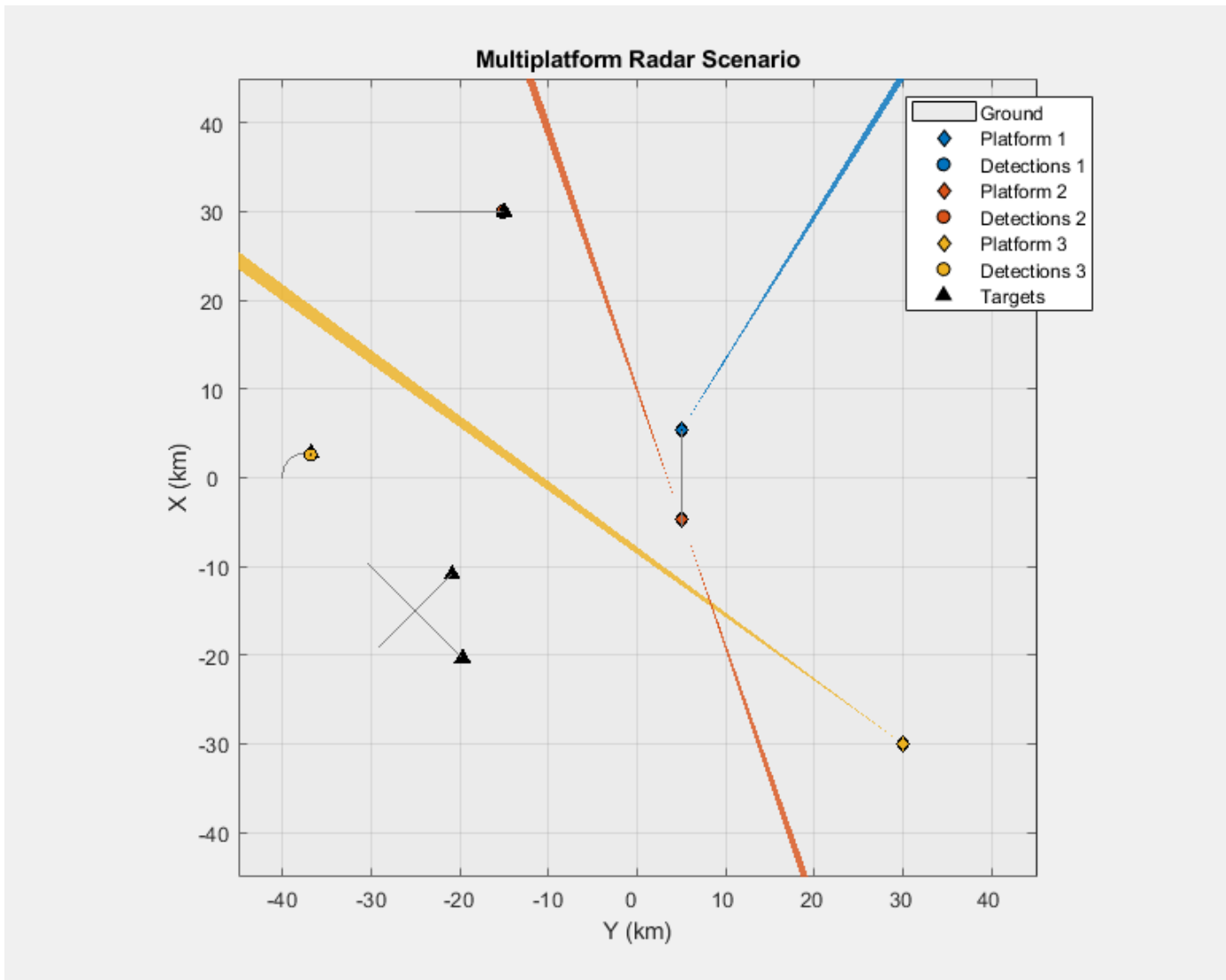
    % Log sensor data and ground truth.
    detLog = [detLog; dets]; %#ok<AGROW>
    timeLog = [timeLog; scene.SimulationTime];%#ok<AGROW>
end

```



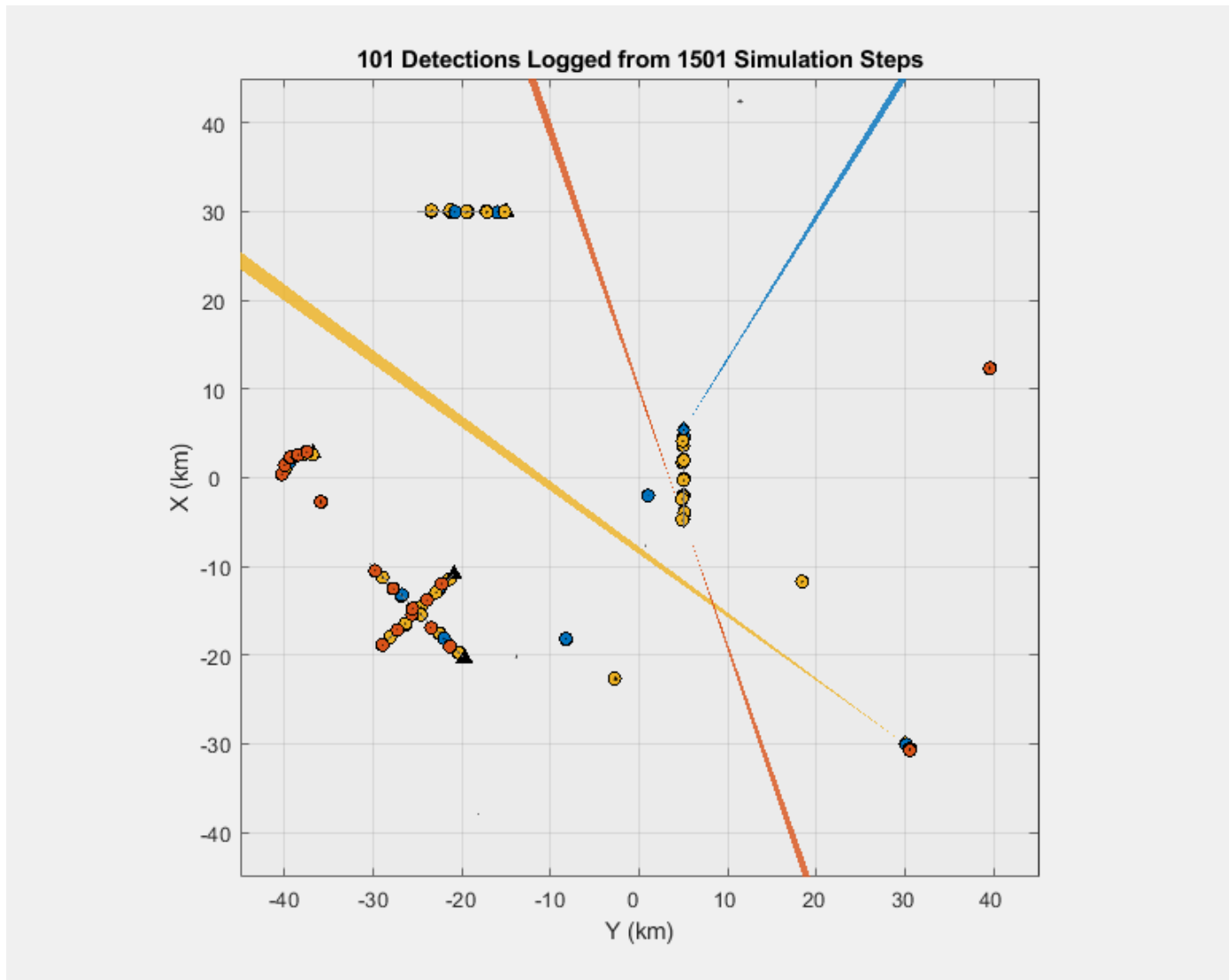
Notice the wide beams from the airborne platforms and the narrow beam from the ground-based radar executing a raster scan. You can visualize the ground truth trajectories in the 2D view below. The four targets are represented by triangles. Around 30 km on the x-axis is the airliner traveling east (left 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
```



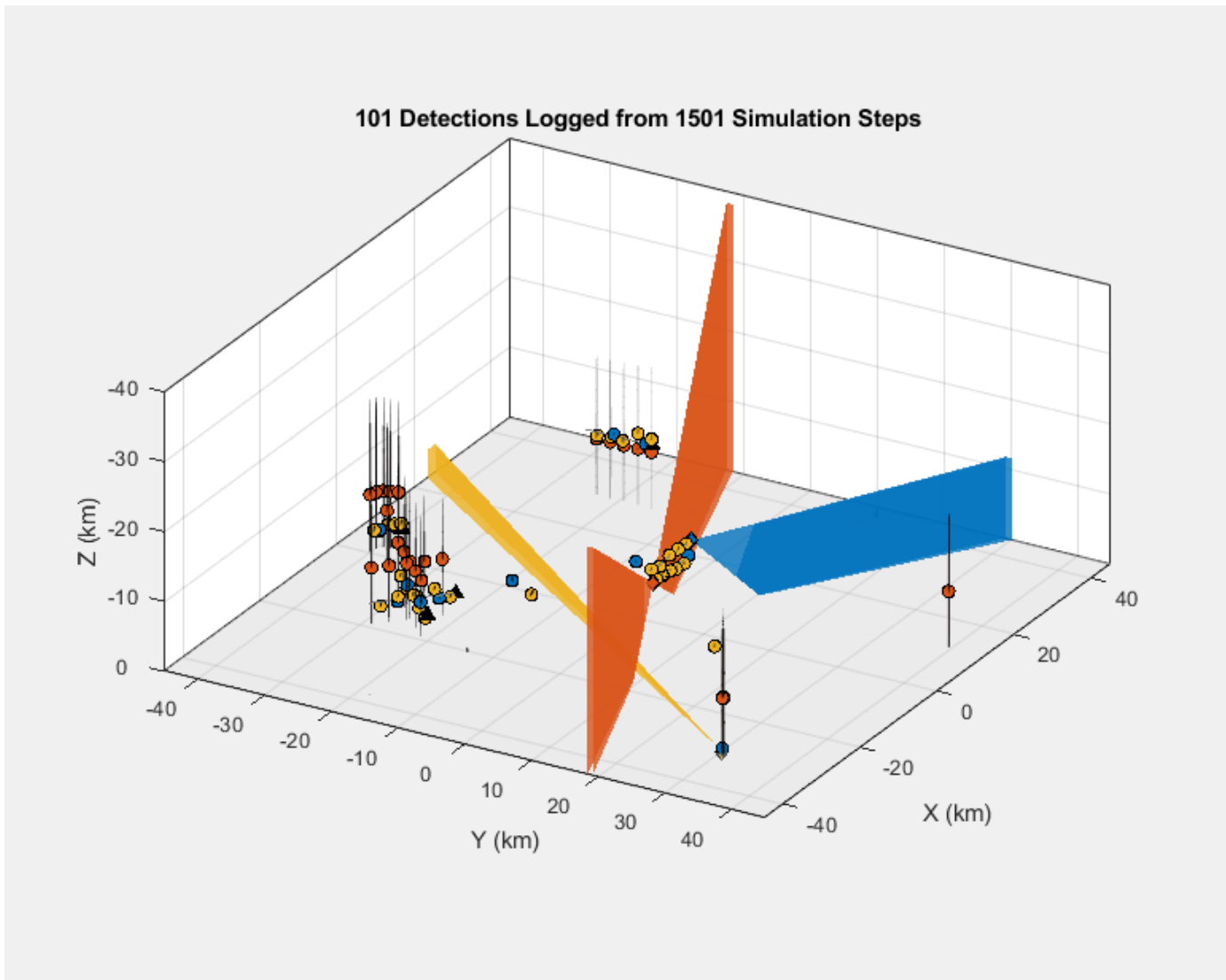
Plot the logged detections with their measurement uncertainties. Each color corresponds to the platform generating the detections. The legend from the previous display applies to all the following plots. Notice that the radars generate false alarms, which are detection far away from the target trajectories.

```
theaterDisplay(detLog);
title([num2str(numel(detLog)) ' Detections Logged from ' num2str(numel(timeLog)) ' Simulation Step ']);
legend('hide');
```



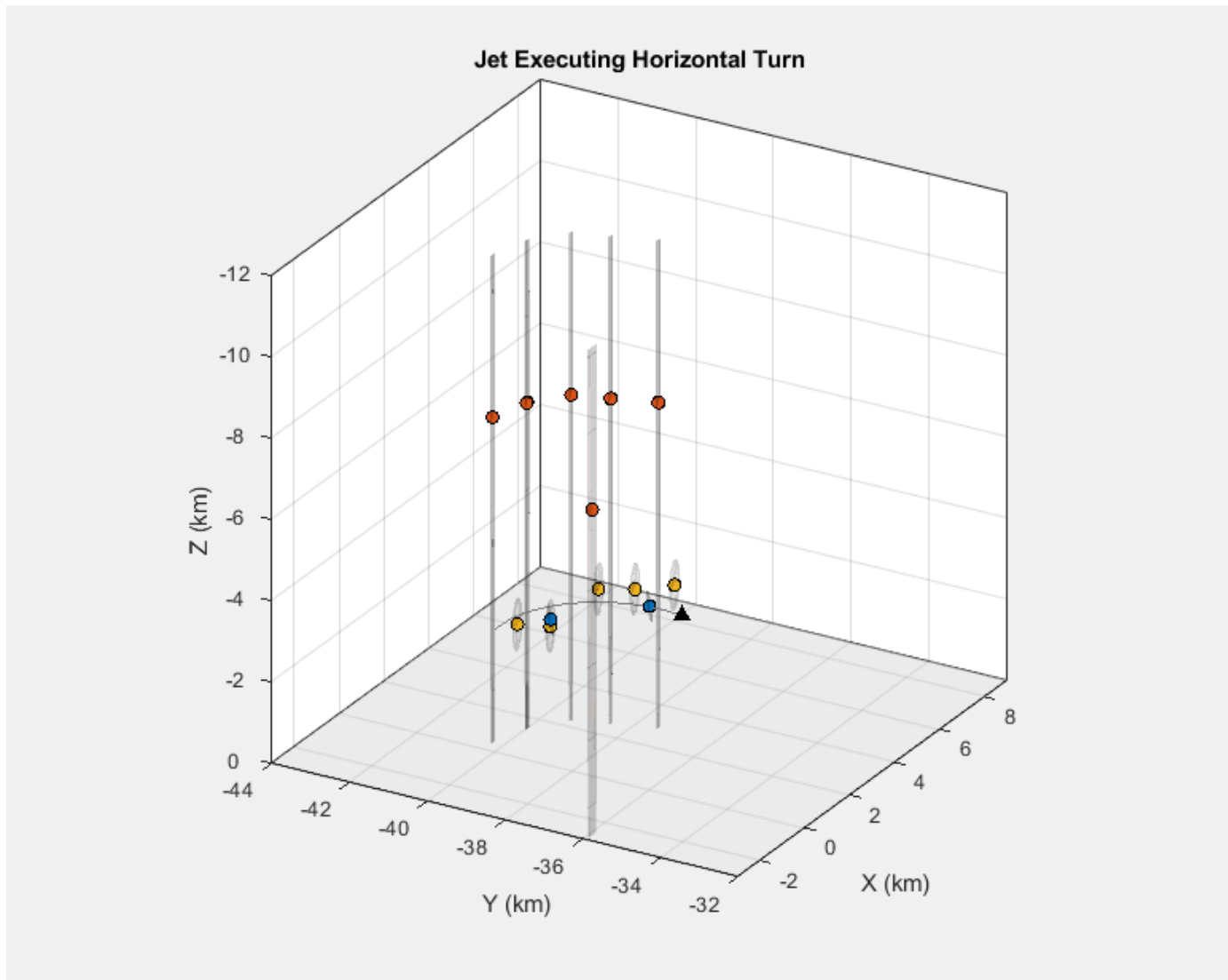
The following 3D view shows how these detections are distributed in elevation. For platforms with 3D sensors (the blue and yellow platforms), the detections closely follow the target trajectories. The 2D-view platform's detections (the red platform) are offset in elevation from the target trajectories 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
```

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.

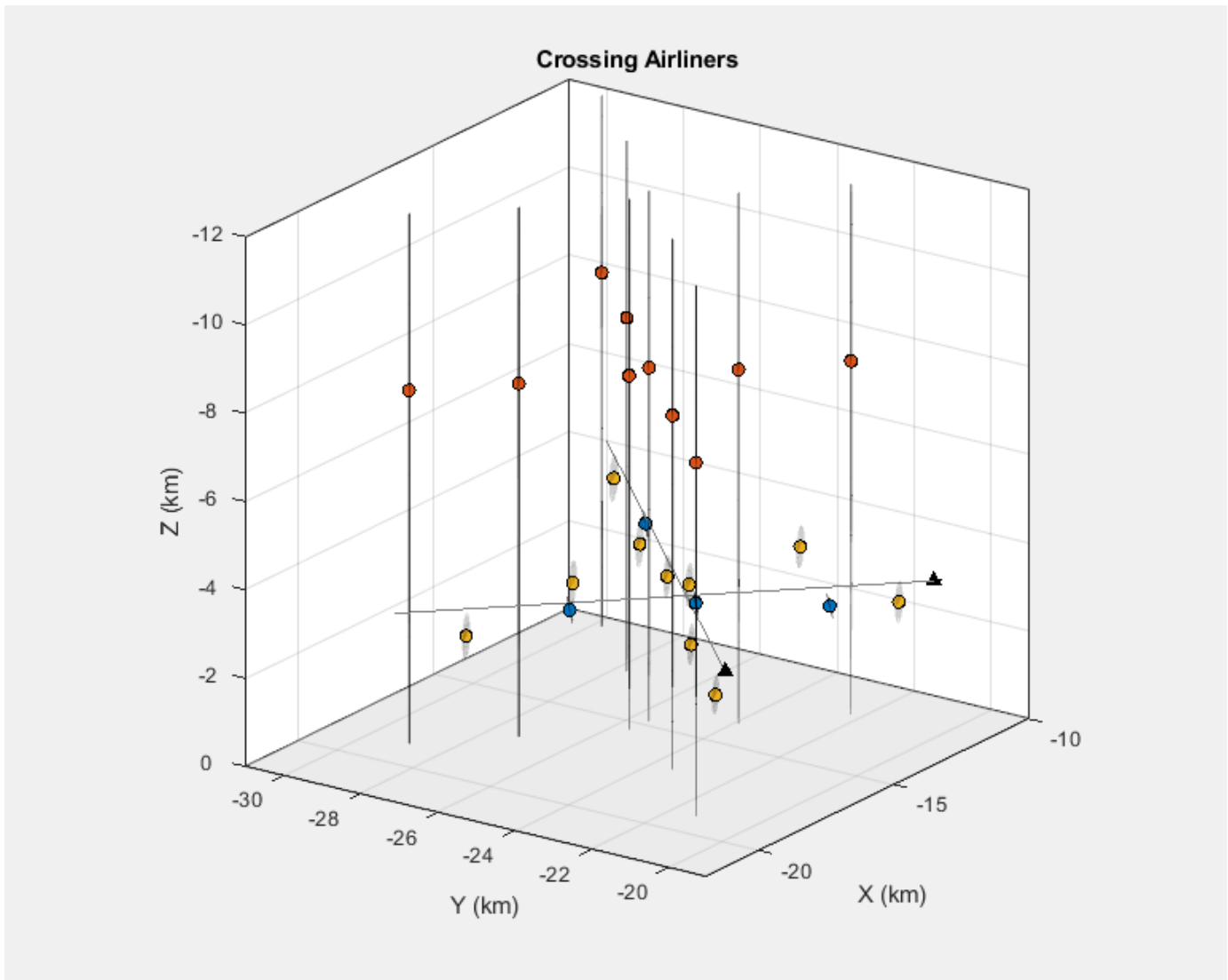
```
xlim([-3000 9000]); ylim([-44000 -32000]); zlim([-12000 0000]);
axis('square');
title('Jet Executing Horizontal Turn');
```



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.

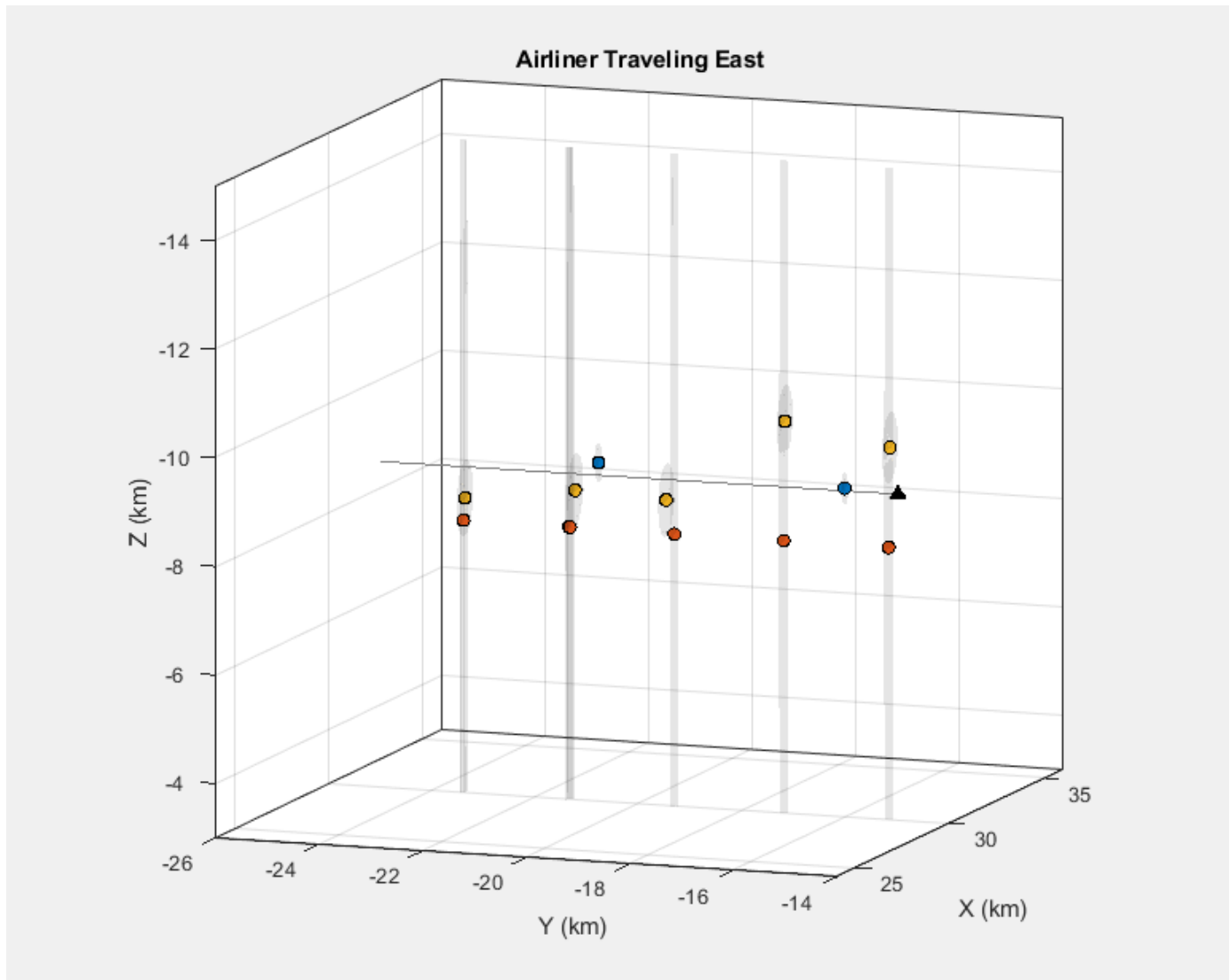
Zoom in on the two crossing airliners. The blue airborne radar with the rotating array generates the fewest number of detections (only 4 detections for these two targets), but these detections are the most precise (smallest ellipses). The small number of detections from this platform is due to its radar's 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');
```



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

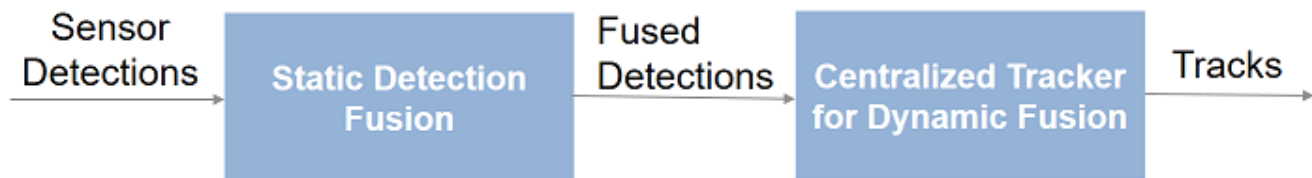
This example illustrates the tracking of objects using measurements from spatially-distributed and synchronous passive sensors. In the “Passive Ranging Using a Single Maneuvering Sensor” on page 6-241, you learned that passive measurements provide incomplete observability of a target's state and how 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:

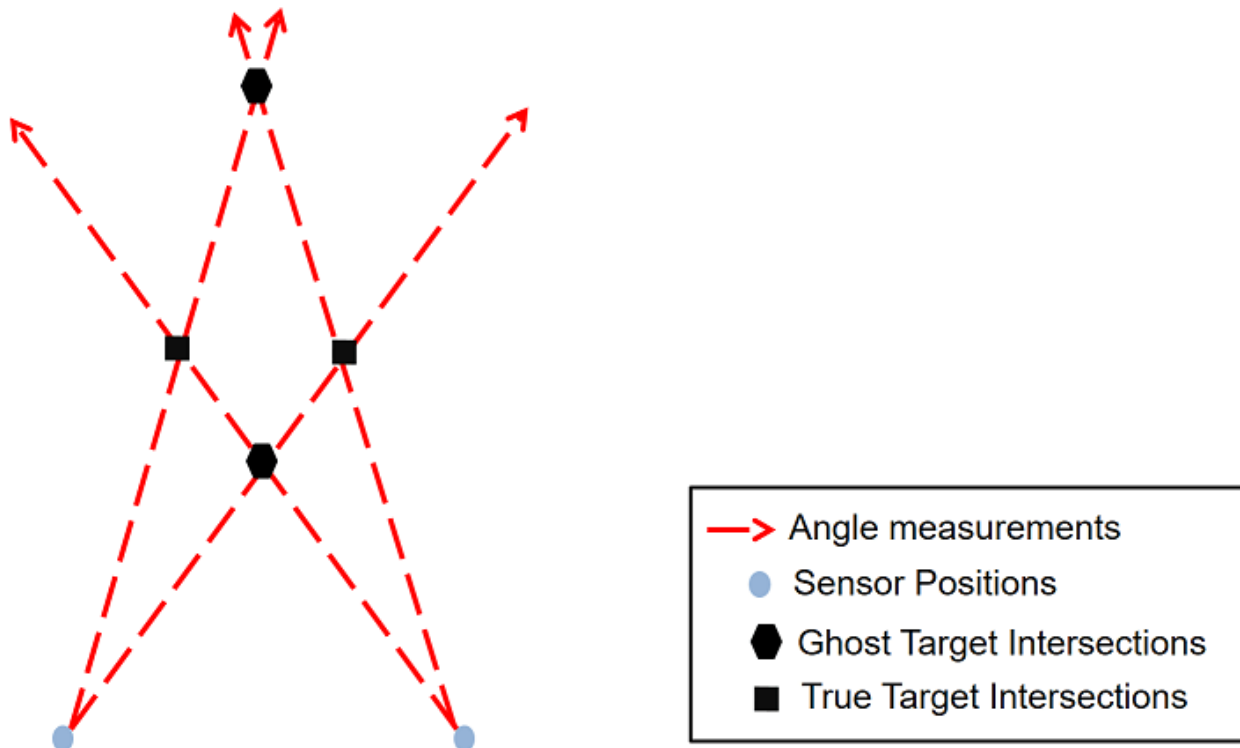
- Number of targets in the scenario
- Position and velocity of those targets

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:

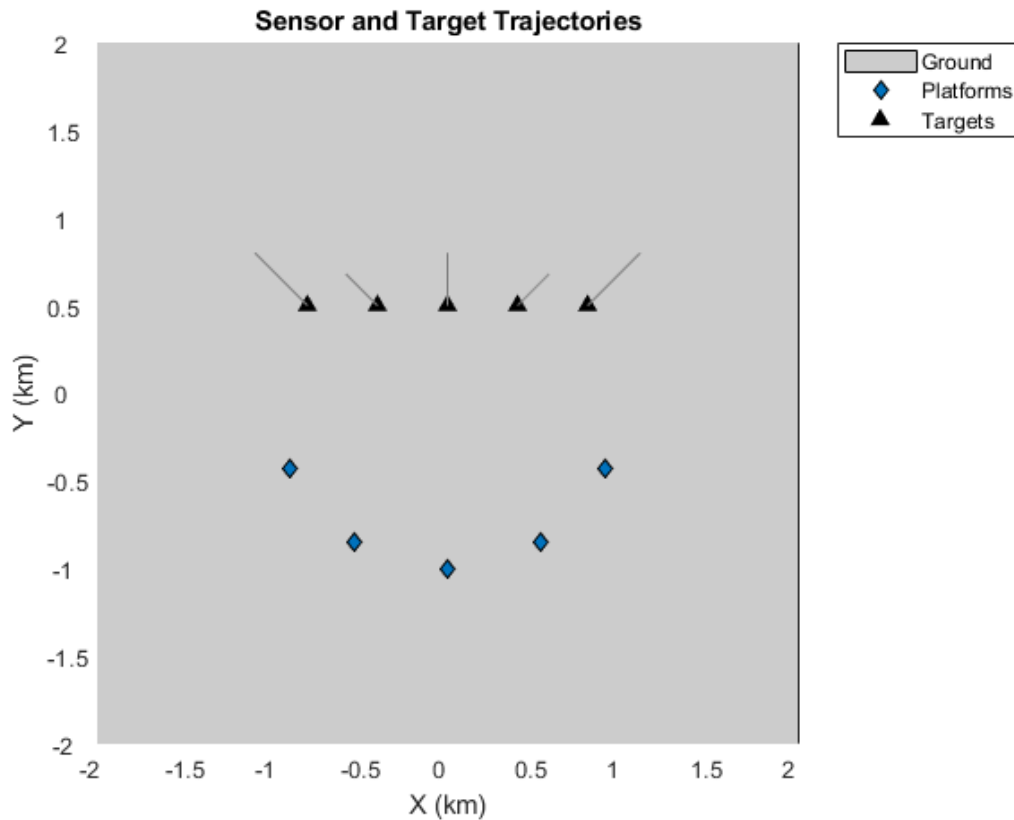


Therefore, one must use 3 or more sensors to reduce the problem of ghosting. Due to the presence of measurement noise and false measurements, it is difficult to eliminate the problem of ghosting completely. Ghost triangulations returned by static association are likely to be discarded by the dynamic 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 `fusionRadarSensor` 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 is added to measurements 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 the helper function `helperGenerateFusionScenarioData`.

```
[detectionBuffer,truthLog,theaterDisplay] = helperGenerateStaticFusionScenarioData;
showScenario(theaterDisplay);
```



```
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',45,...
    '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; % simulation time
dT = 1; % 1 Hz update rate of scenario.

% Loop through detections and track targets
for iter = 1:numel(detectionBuffer)

    % Truth information
    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);

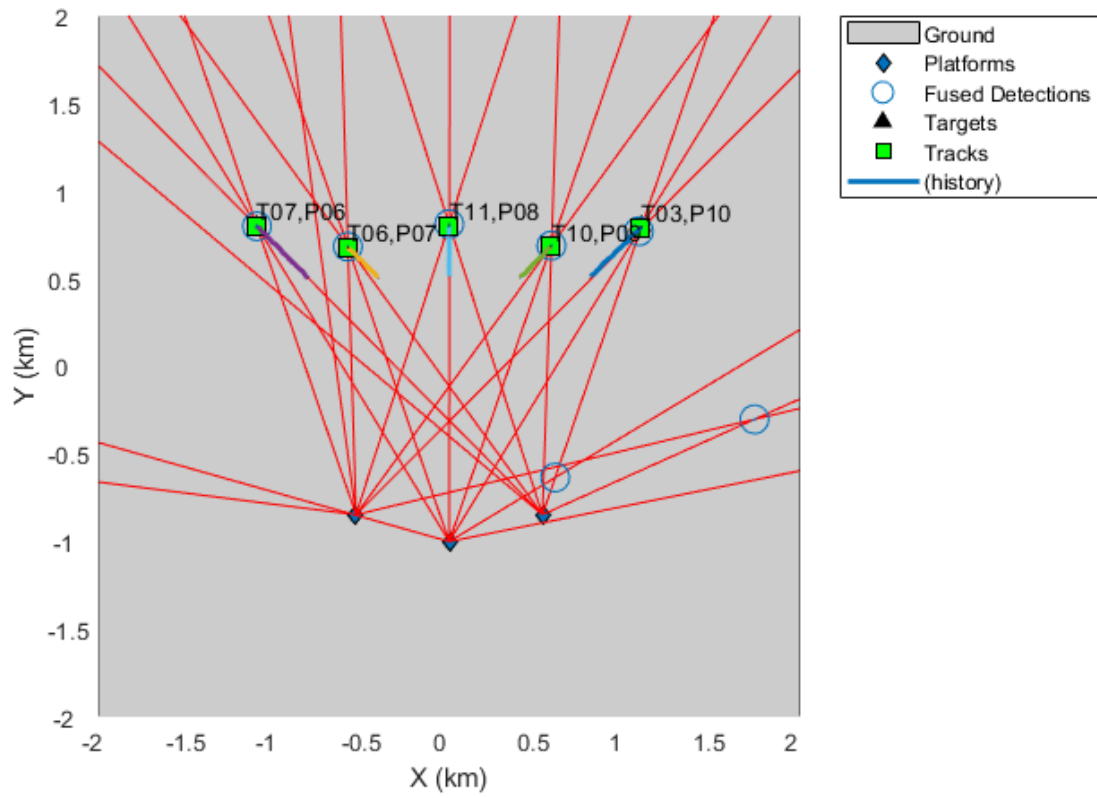
    % Run a tracker on fused detections
    confTracks = tracker(fusedDetections,time);

    % Update track and assignment metrics
    trackingMetrics(confTracks,targetPlatPoses);
    [trackIDs,truthIDs] = currentAssignment(trackingMetrics);
    errorMetrics(confTracks,trackIDs,targetPlatPoses,truthIDs);

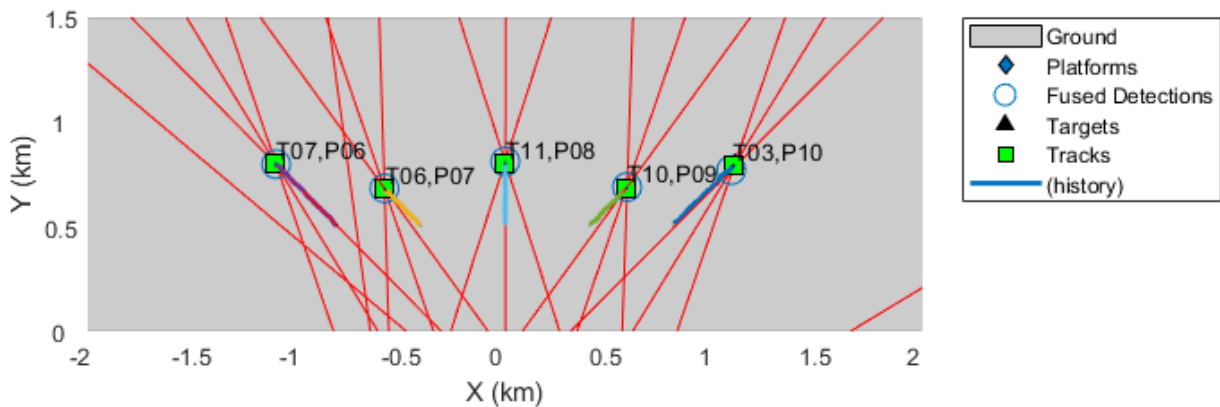
    % Update theater display
    detsToPlot = [noiseDetections(:);fusedDetections(:)];
    theaterDisplay(confTracks,detsToPlot,groundTruth);

    % Increment simulation time
    time = time + dT;
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 indicate good static association accuracy.

```
assignmentTable = trackMetricsTable(trackingMetrics);
assignmentTable(:, {'TrackID', 'AssignedTruthID', 'TotalLength', 'FalseTrackStatus'})
```

ans =

5×4 table

TrackID	AssignedTruthID	TotalLength	FalseTrackStatus
3	10	59	false
6	7	59	false
7	6	59	false
10	9	58	false
11	8	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));
```

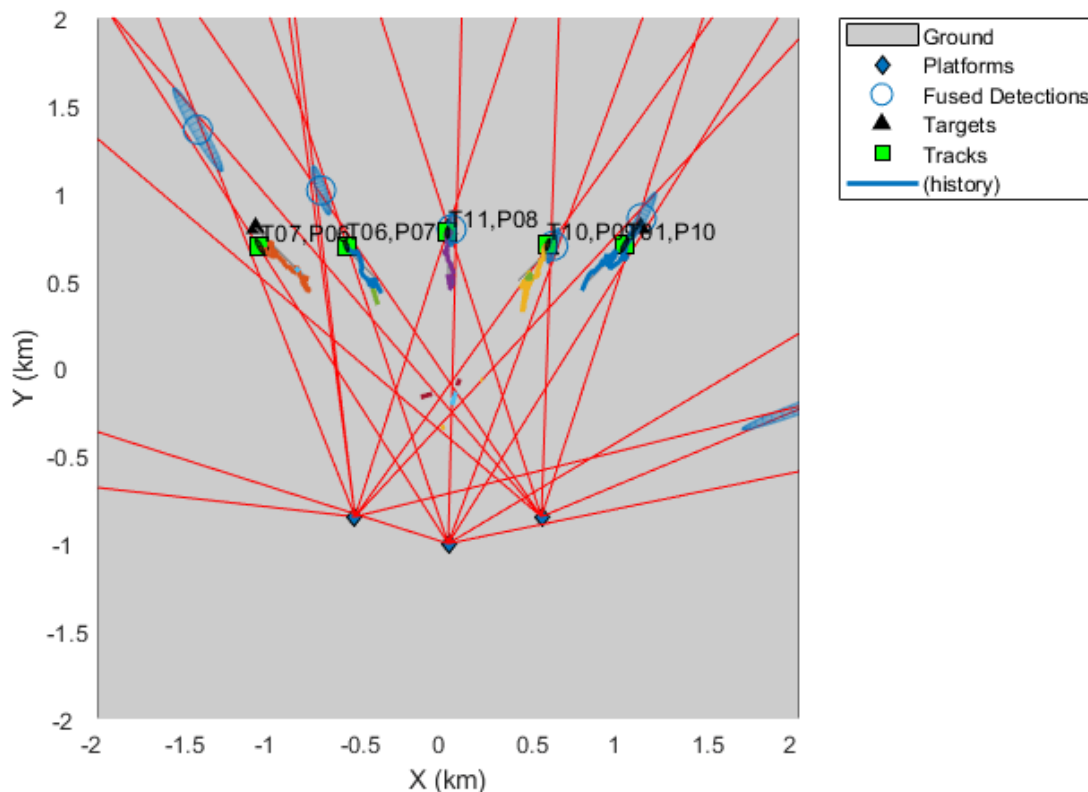
TrackID	posRMS	velRMS	posANEES	velANEES
3	6.8821	1.595	2.51	0.80396
6	3.9895	1.1149	1.6409	0.5416
7	5.8195	1.3356	1.9041	0.66745
10	4.2425	1.2514	1.6719	0.62374
11	3.6443	1.1453	1.375	0.55326

Effect of measurement accuracy

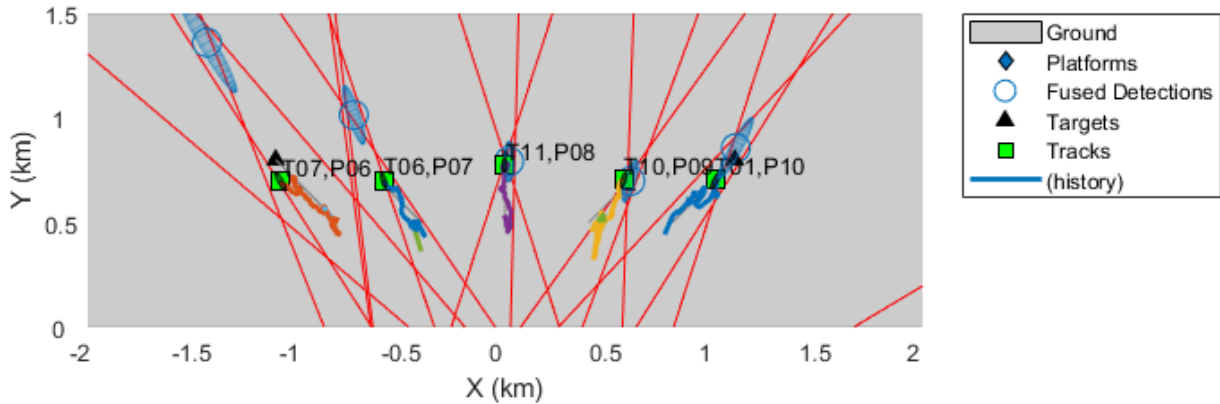
The fusion of passive detections to eliminate ghosting is highly dependent on the accuracy of passive measurements. As measurement noise increases, the distinction between ghost associations and true associations becomes less prominent, resulting in a significant drop in the accuracy of static association. 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,numSensors,
axes(theaterDisplay.TheaterPlot.Parent);
```



```
ylim([0 1.5]);
```



Note that a few 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 =
```

```
9×4 table
```

TrackID	AssignedTruthID	TotalLength	FalseTrackStatus
1	10	59	false
3	NaN	4	true
4	NaN	5	false
6	7	59	false
7	6	59	false
10	9	57	false
11	8	56	false
13	NaN	5	false
18	NaN	5	true

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	261.26	7.498	82.824	1.4568
7	54.822	3.226	3.9109	0.92307
8	50.606	4.6234	2.8907	1.1096
9	83.002	5.0335	7.1213	1.6252
10	206.17	7.0411	47.227	1.8917

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 “Get Started with MATLAB Coder” (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};

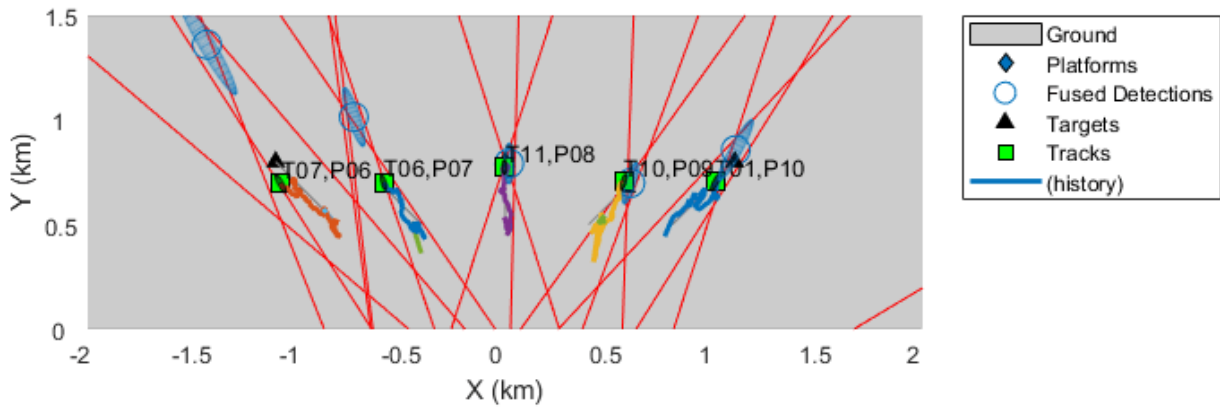
close(h);

Code generation successful.
```

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)]);

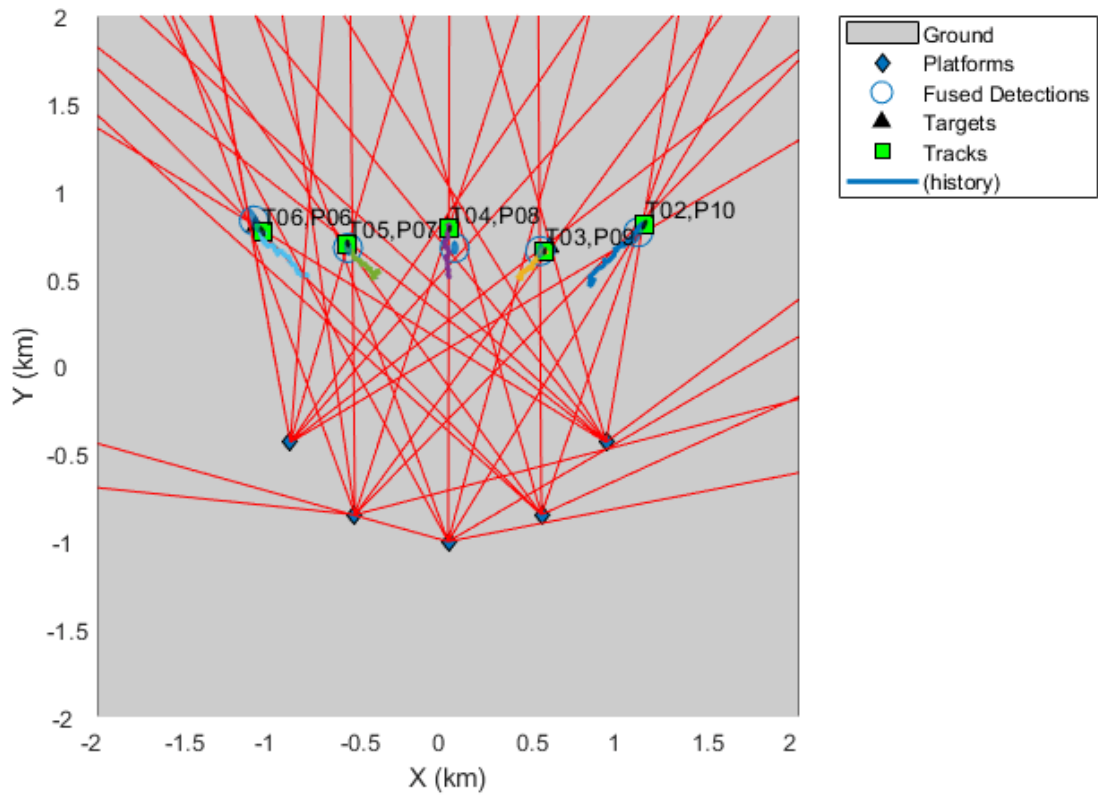
MATLAB Code Execution time = 37.9316
MEX Code Execution time = 0.4011
```



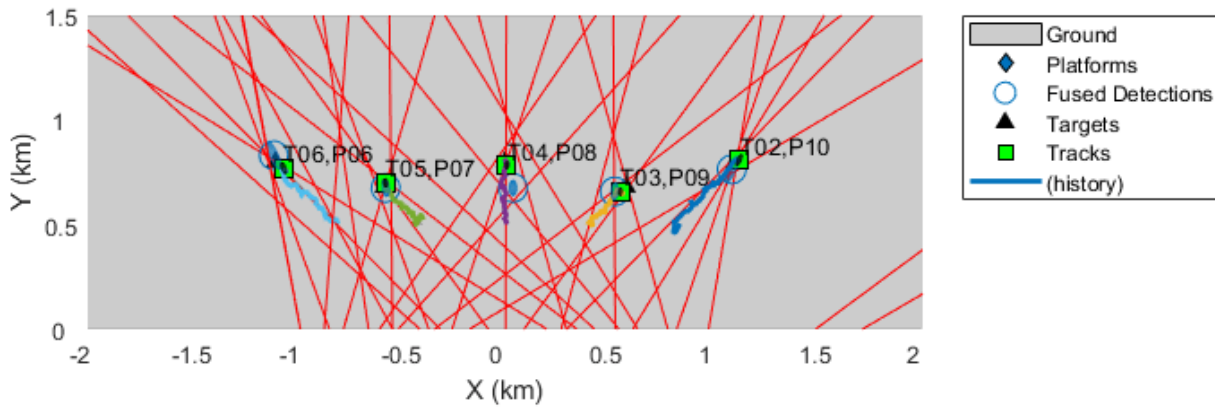
Track with Five Sensors

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,numSensors);
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 4 track drops in the low-accuracy three sensor simulation.

```
assignmentTable = trackMetricsTable(trackingMetrics);
assignmentTable(:, {'TrackID', 'AssignedTruthID', 'TotalLength', 'FalseTrackStatus'})
```

```
ans =
```

```
5x4 table
```

TrackID	AssignedTruthID	TotalLength	FalseTrackStatus
2	10	59	false
3	9	59	false
4	8	59	false
5	7	59	false
6	6	59	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	34.74	3.0009	3.0358	0.75358
7	16.415	2.7014	1.3547	0.53336
8	16.555	2.5768	1.5645	0.49951
9	16.361	2.5381	1.474	0.55633
10	26.137	4.0457	2.3739	1.0349

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 depends on 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).

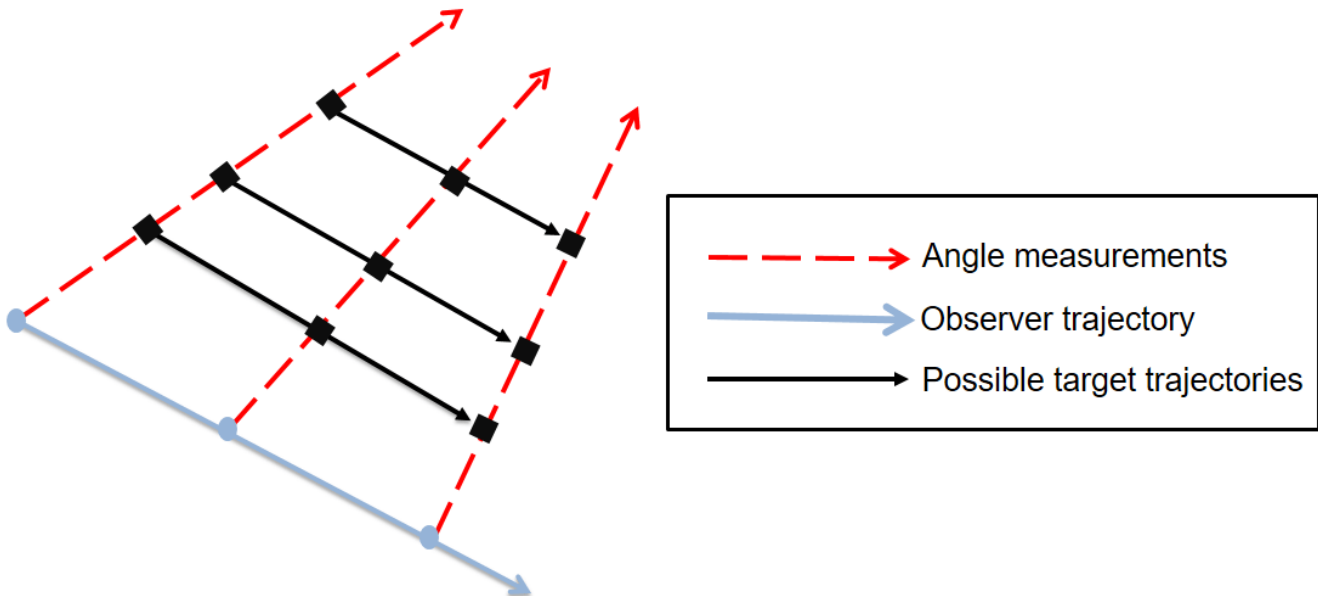
Passive Ranging Using a Single Maneuvering Sensor

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.

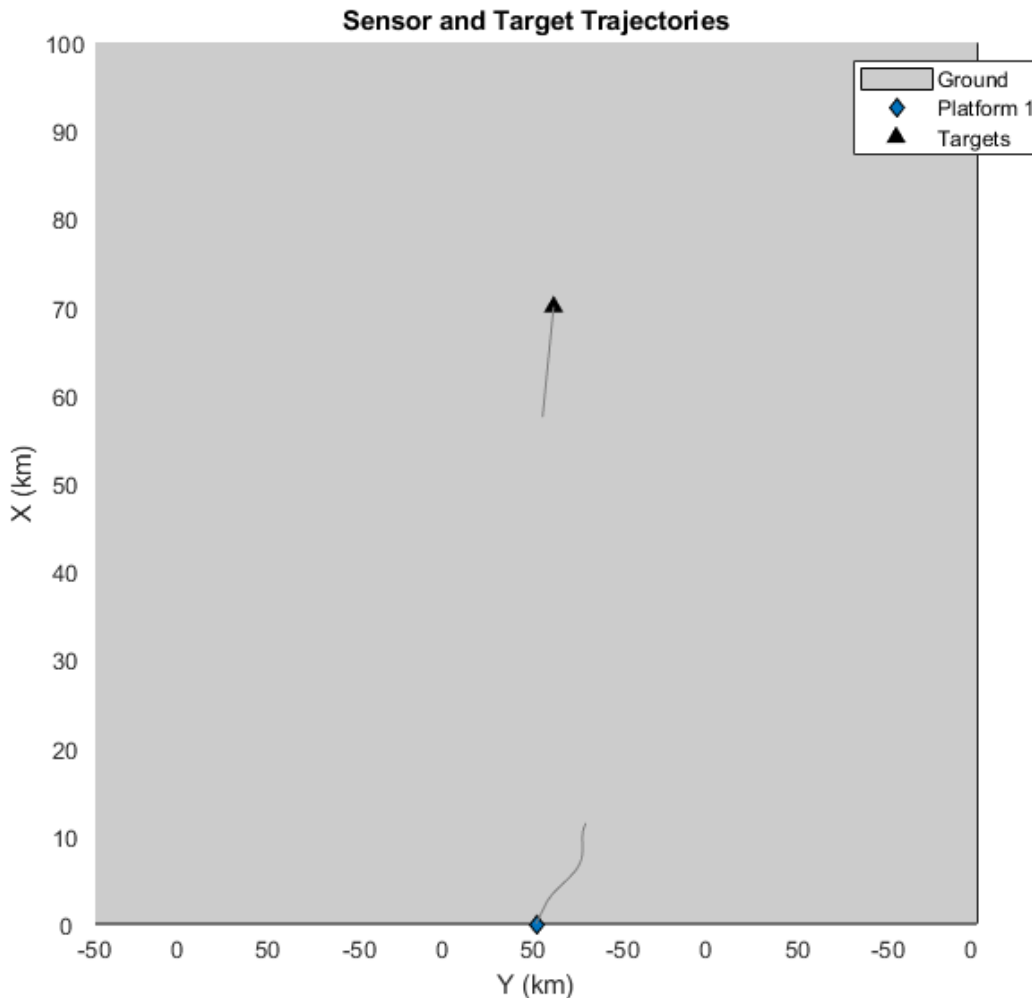
Define Scenario

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.

```

```

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);

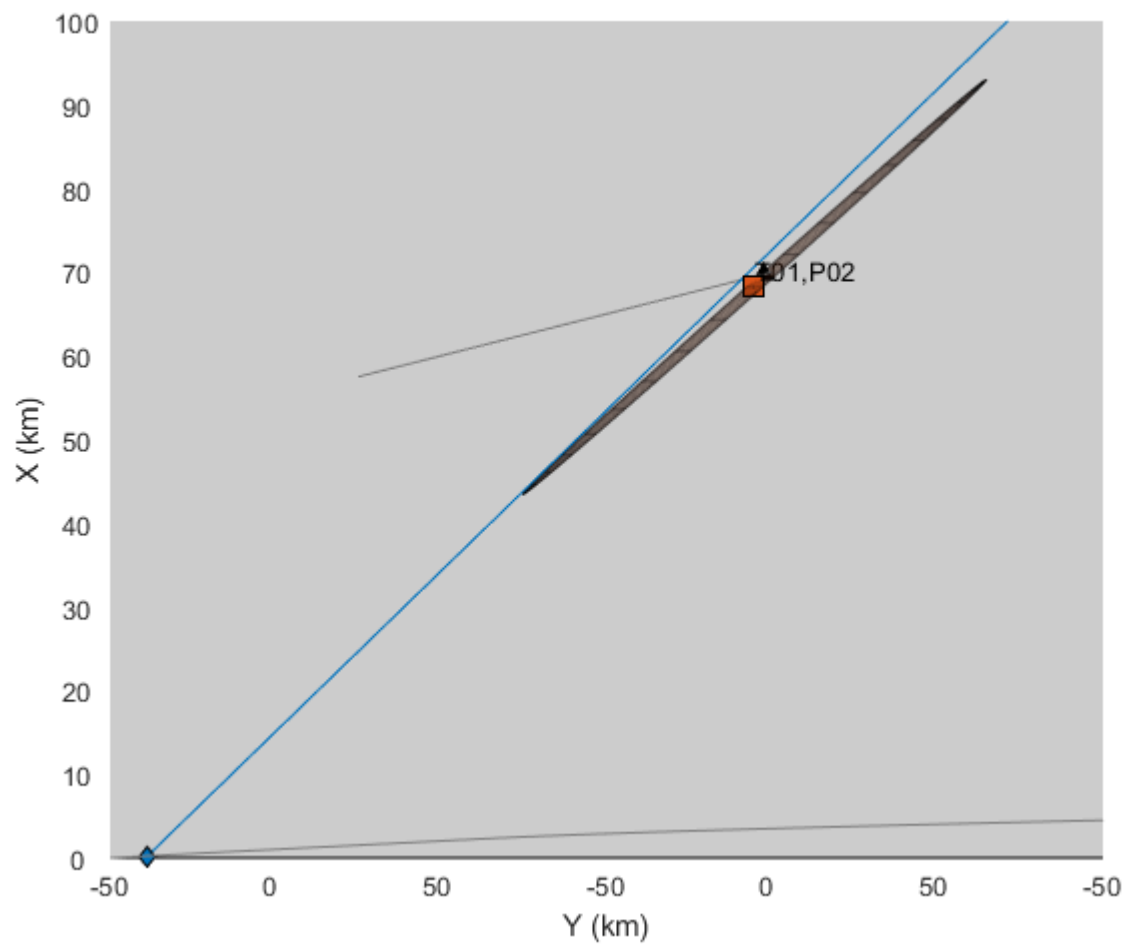
    % Pass detections to tracker
    if ~isempty(detections)
        tracks = tracker(detections,time);
    elseif isLocked(tracker)
        tracks = predictTracksToTime(tracker,'confirmed',time);
    end
    % Update error and assignment metrics
    tam(tracks,truths);
    [trackIDs, truthIDs] = currentAssignment(tam);
    tem(tracks,trackIDs,truths,truthIDs);

    % Update display
    theaterDisplay(tracks,detections,tracker);
end

```

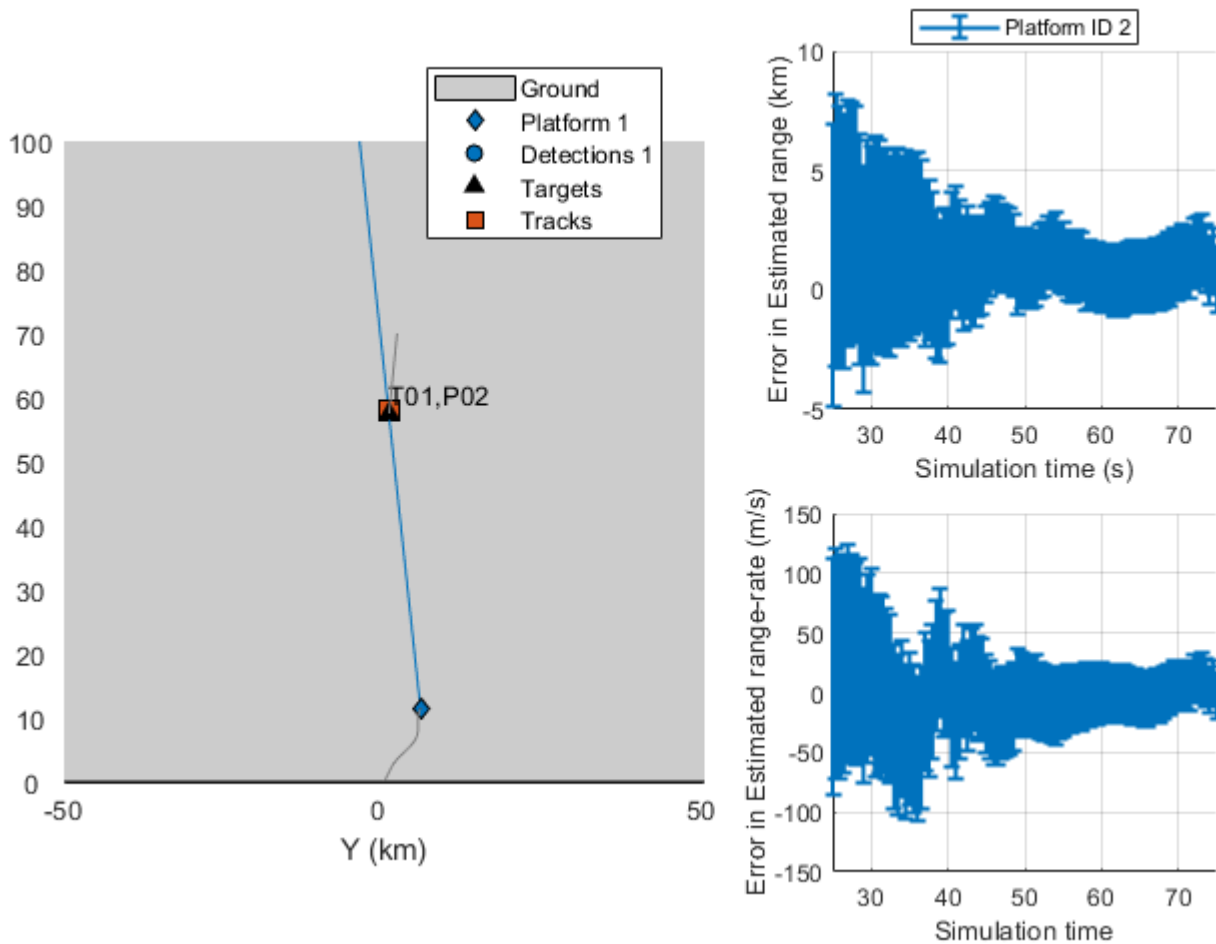
The track is initialized using a high covariance in range to account for the missing measurement. Notice the covariance in range of the track just after initialization in the theater plot. As angular estimates are fairly accurate, the target uncertainty is described by a thin covariance ellipse.

```
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 nonlinearities 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);
```

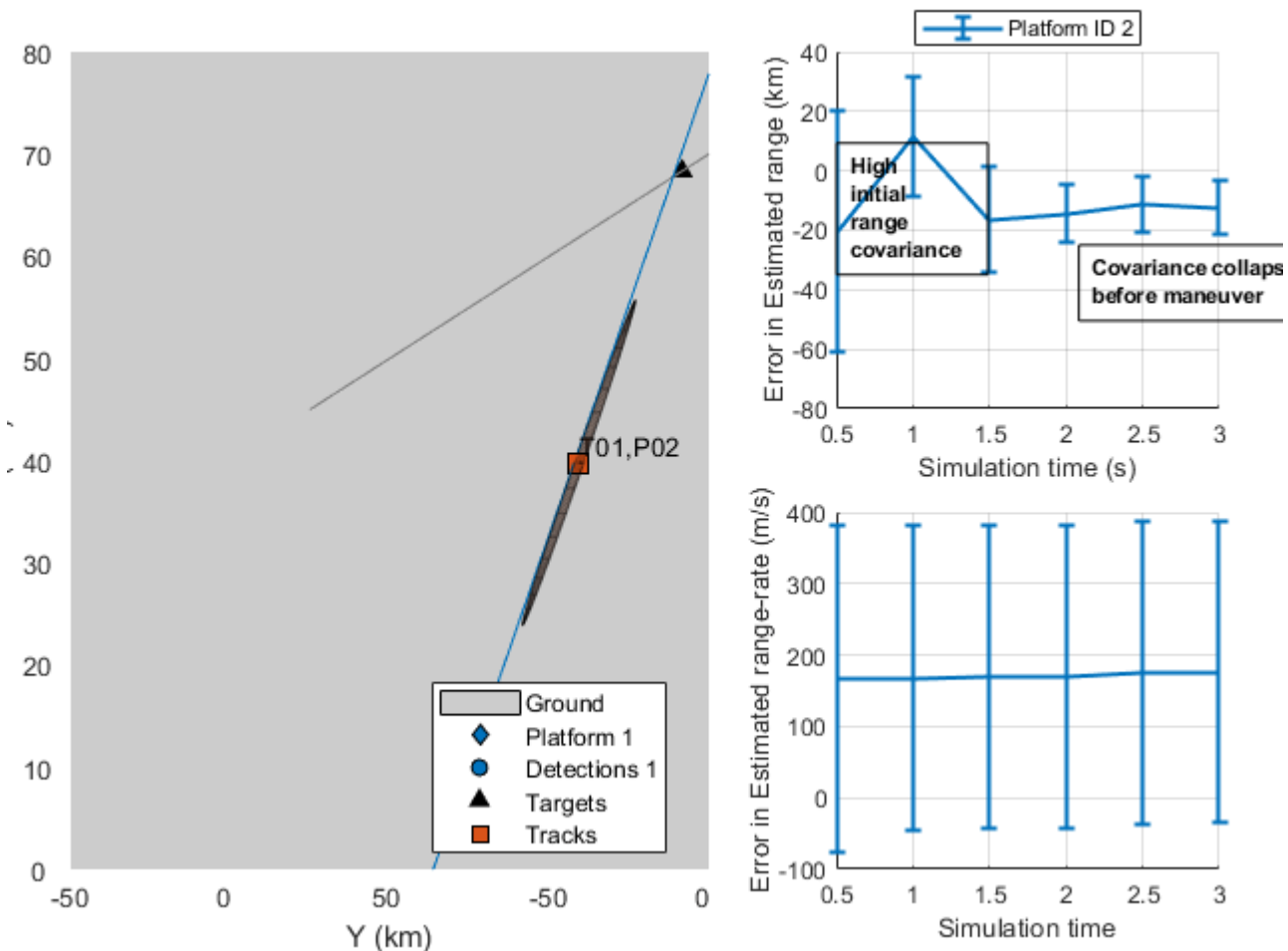
```

% Reset the theaterDisplay to capture snapshots with new scenario.
release(theaterDisplay);
theaterDisplay.GrabFigureFcn = @(fig,scene)helperGrabPassiveRangingDisplay(fig,scene,'CartEKF2')
helperRunPassiveRangingSimulation(scene,theaterDisplay,@initCartesianEKF);

```

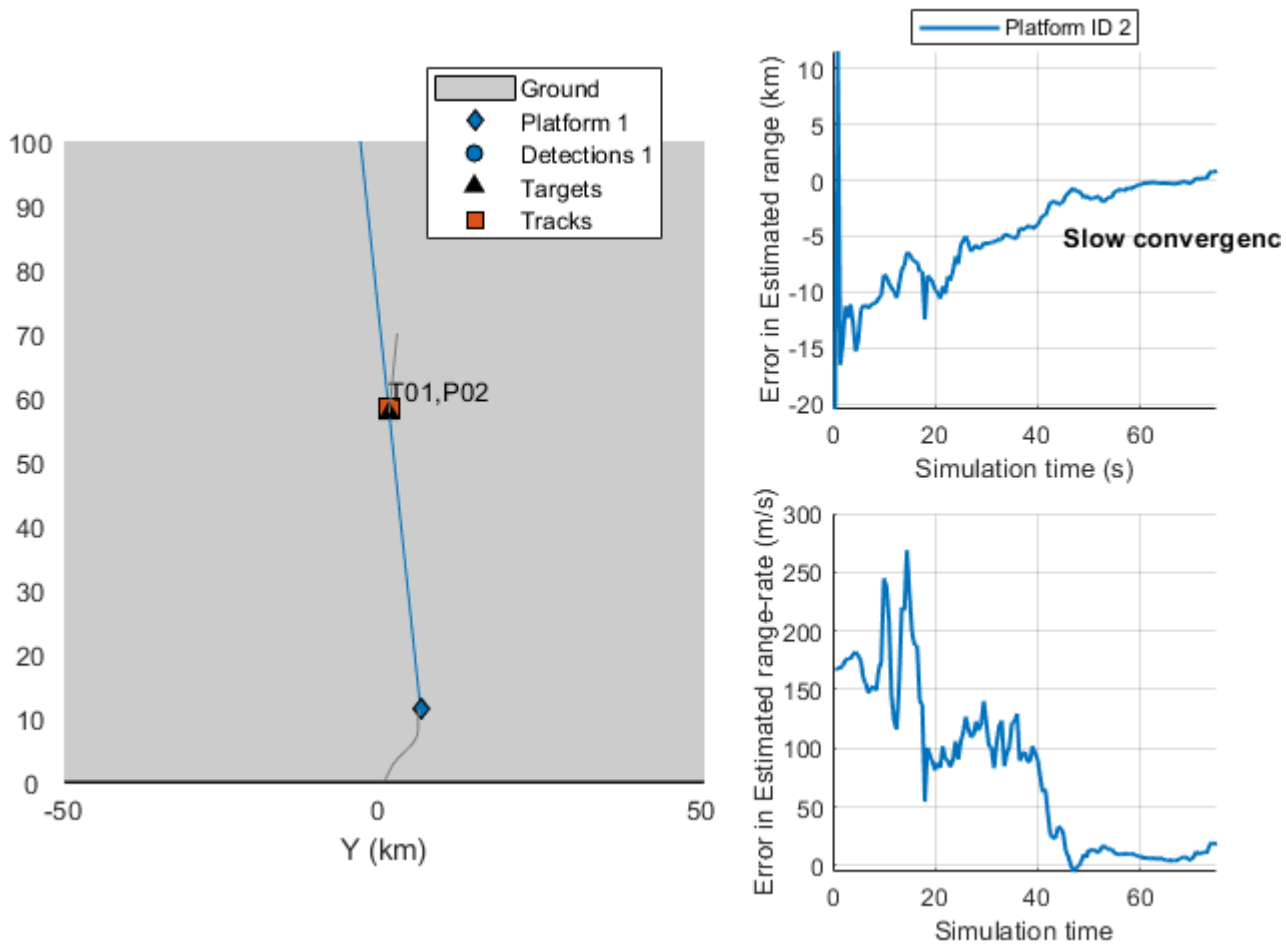
After 2 seconds, the covariance in the range has already collapsed, making the filter falsely confident about its estimate of the range. Notice that target lies outside the covariance ellipse of the track's state and so does the zero-error line in the range estimation plot.

```
showGrabs(theaterDisplay,3);
```



Because the filter remains confident about its 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.

$$\mathbf{x} = \mathbf{x}_t - \mathbf{x}_o$$

$$\dot{\mathbf{x}} = \dot{\mathbf{x}}_t - \dot{\mathbf{x}}_o$$

$$\dot{\mathbf{x}} = A\mathbf{x}_t - (A\mathbf{x}_o + \mathcal{O}(t^2) + \mathcal{O}(t^3) + \dots)$$

$$\dot{\mathbf{x}} = A(\mathbf{x}_t - \mathbf{x}_o) + \text{higher order terms}$$

```
% Set the same random seed to compare with the same detections
rng(2015);
```

```

% 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);
end

```

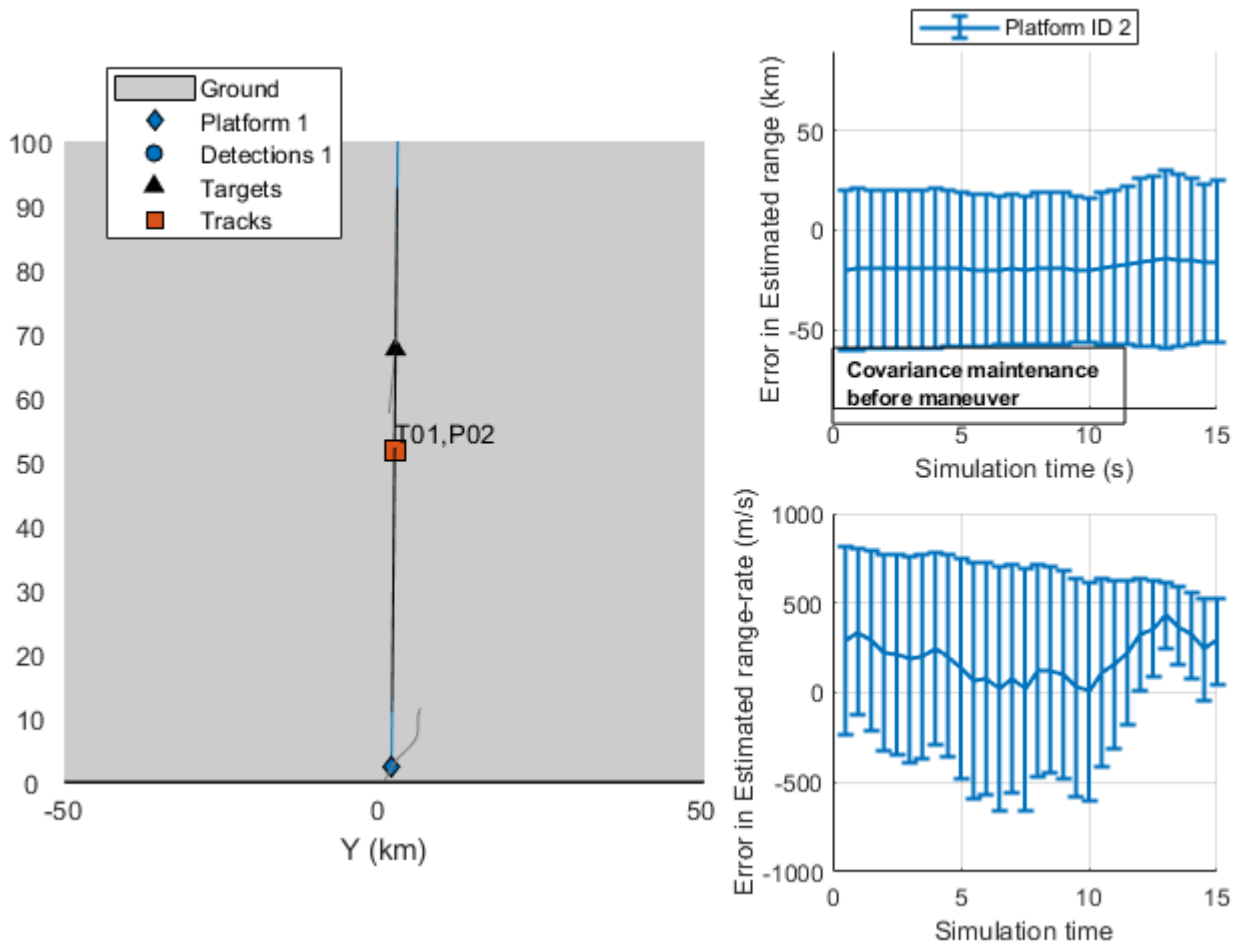
```

% Update display
theaterDisplay.NonLinearStateInput = currentPose.Position(:);
theaterDisplay(detections,tracks,tracker);
end

```

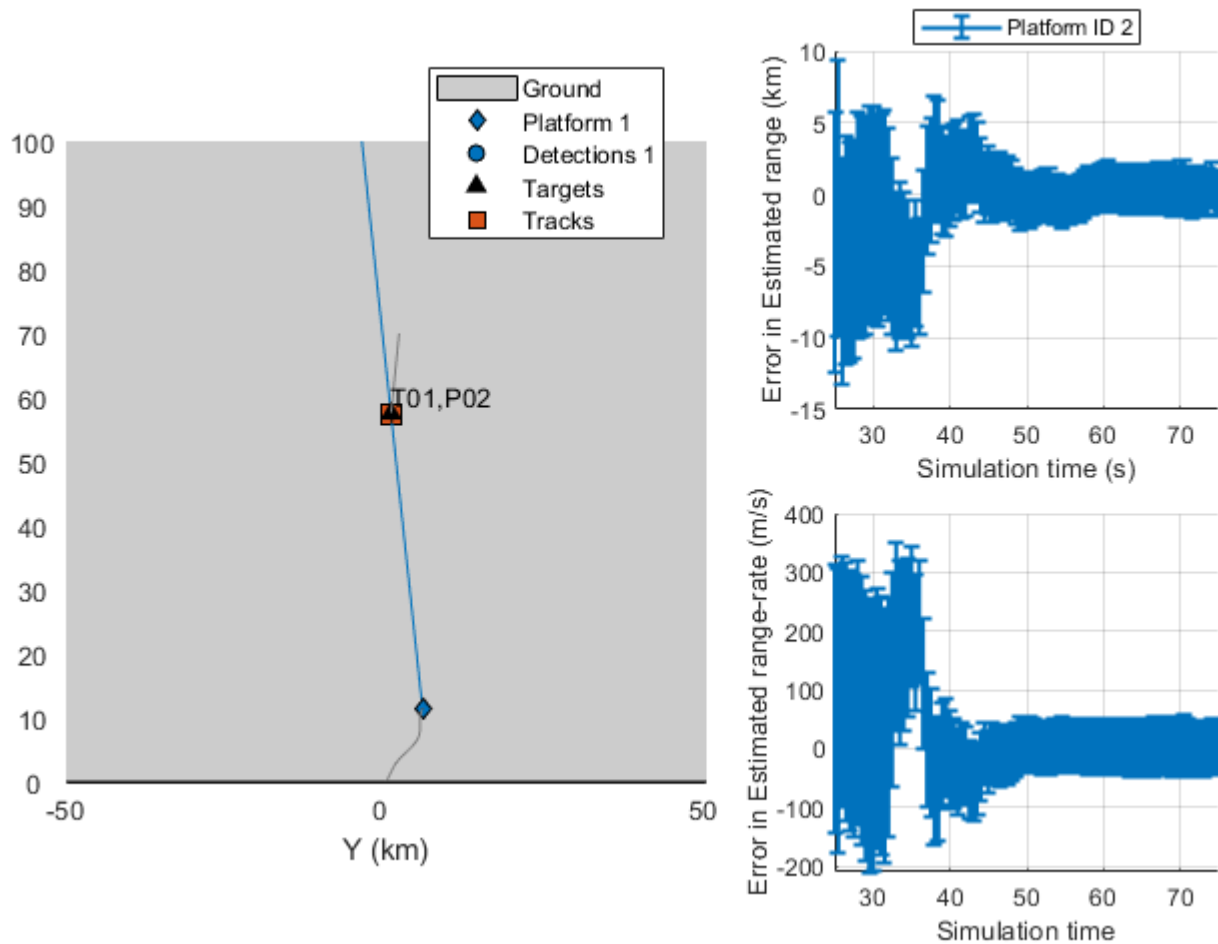
The MSC-EKF tries to maintain the covariance in range till the sensor has not made a maneuver. This is essentially due to decoupled observable and unobservable parts in the state.

```
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;
```

```

% 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

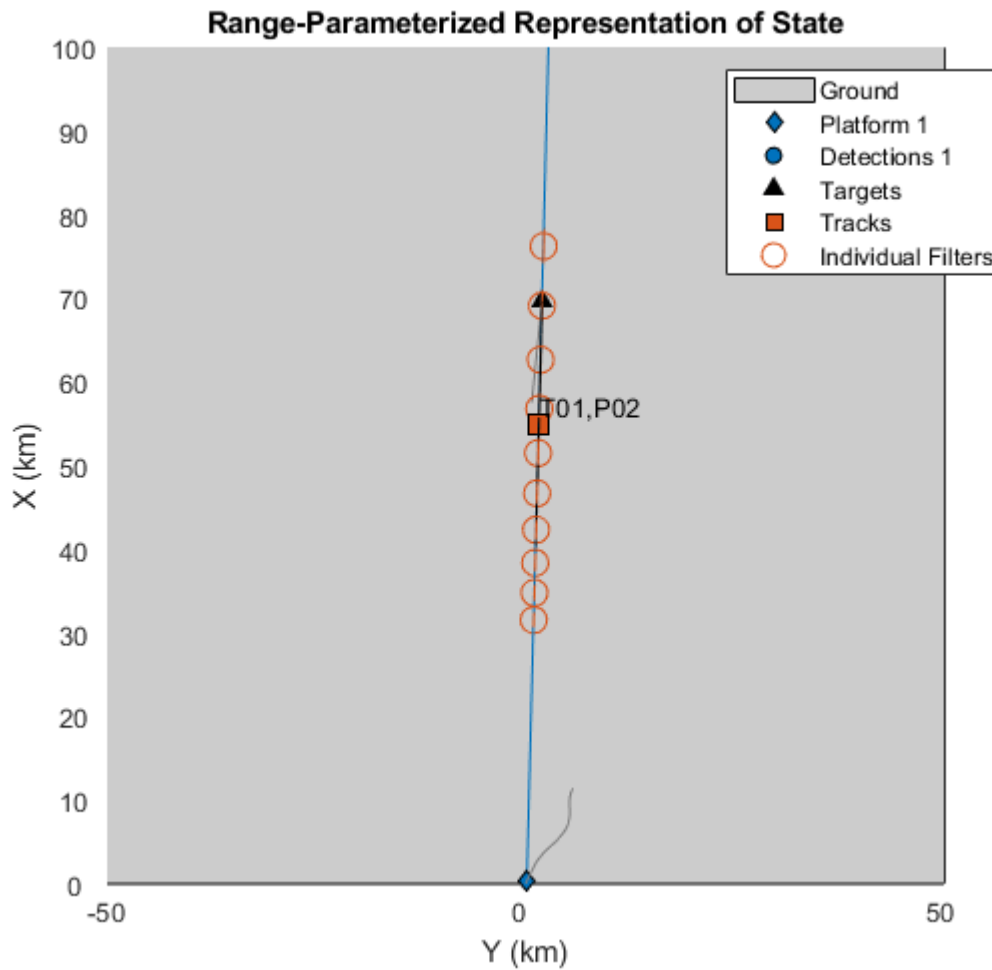
    % Update error and assignment metrics
    tam(tracks,truths);
    [trackIDs, truthIDs] = currentAssignment(tam);
    tem(tracks,trackIDs,truths,truthIDs);

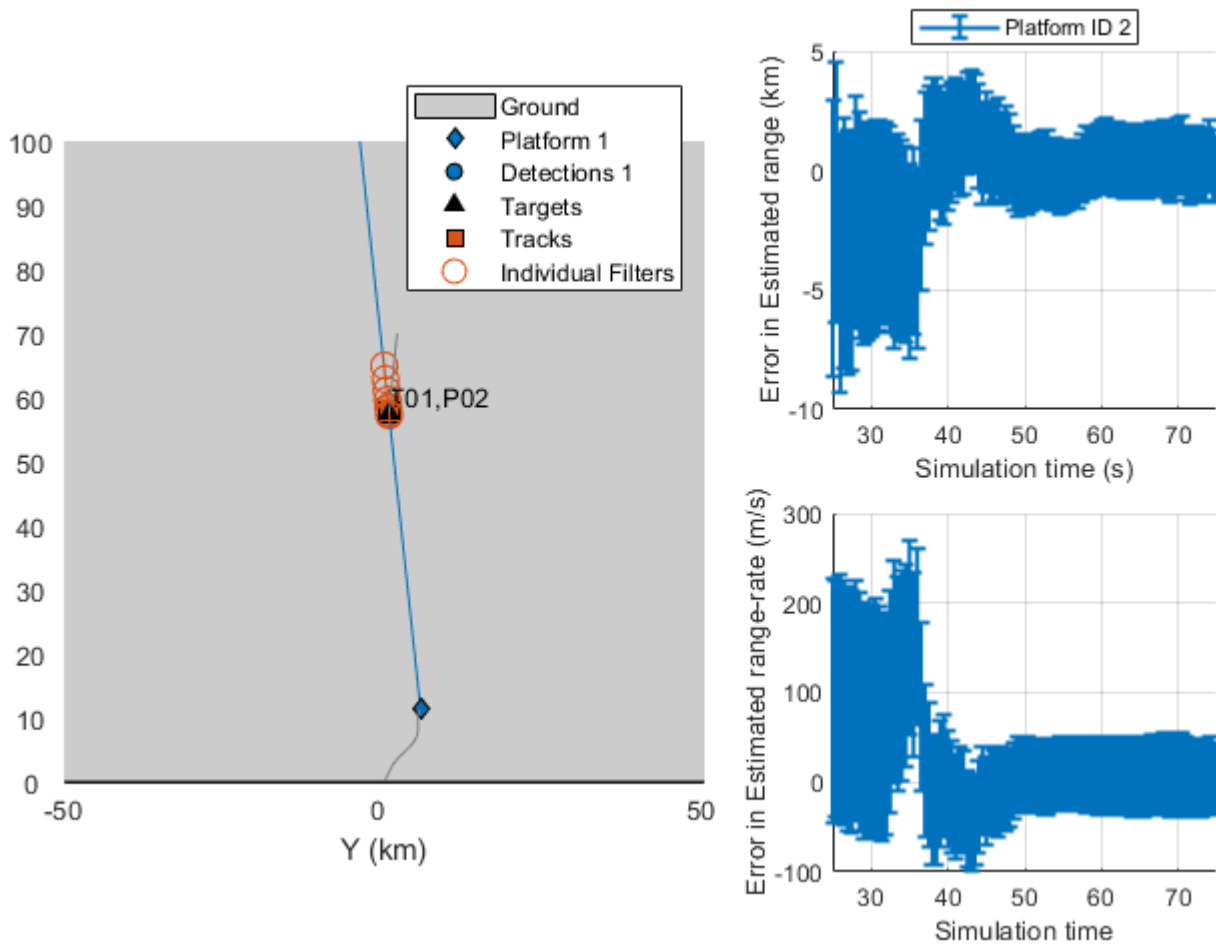
    % Update display
    theaterDisplay.NonLinearStateInput = currentPose.Position(:);
    theaterDisplay(detections,tracks,tracker);
end

```

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-parameterized 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-parameterized filter.

```
showGrabs(theaterDisplay,[7 8]);
```





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.

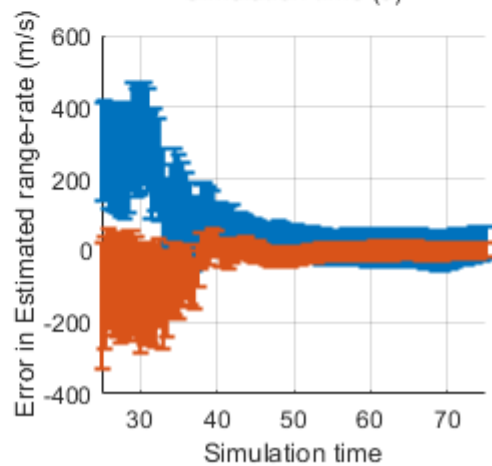
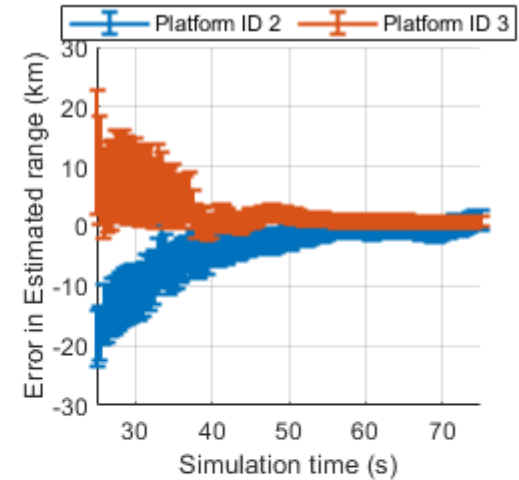
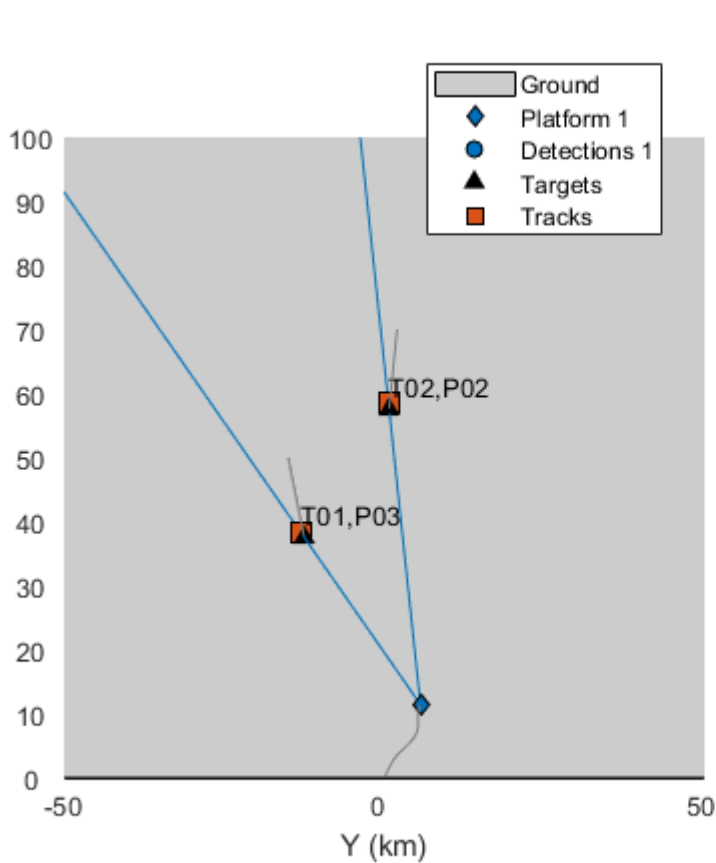
MSC-EKF

```
% 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);
```



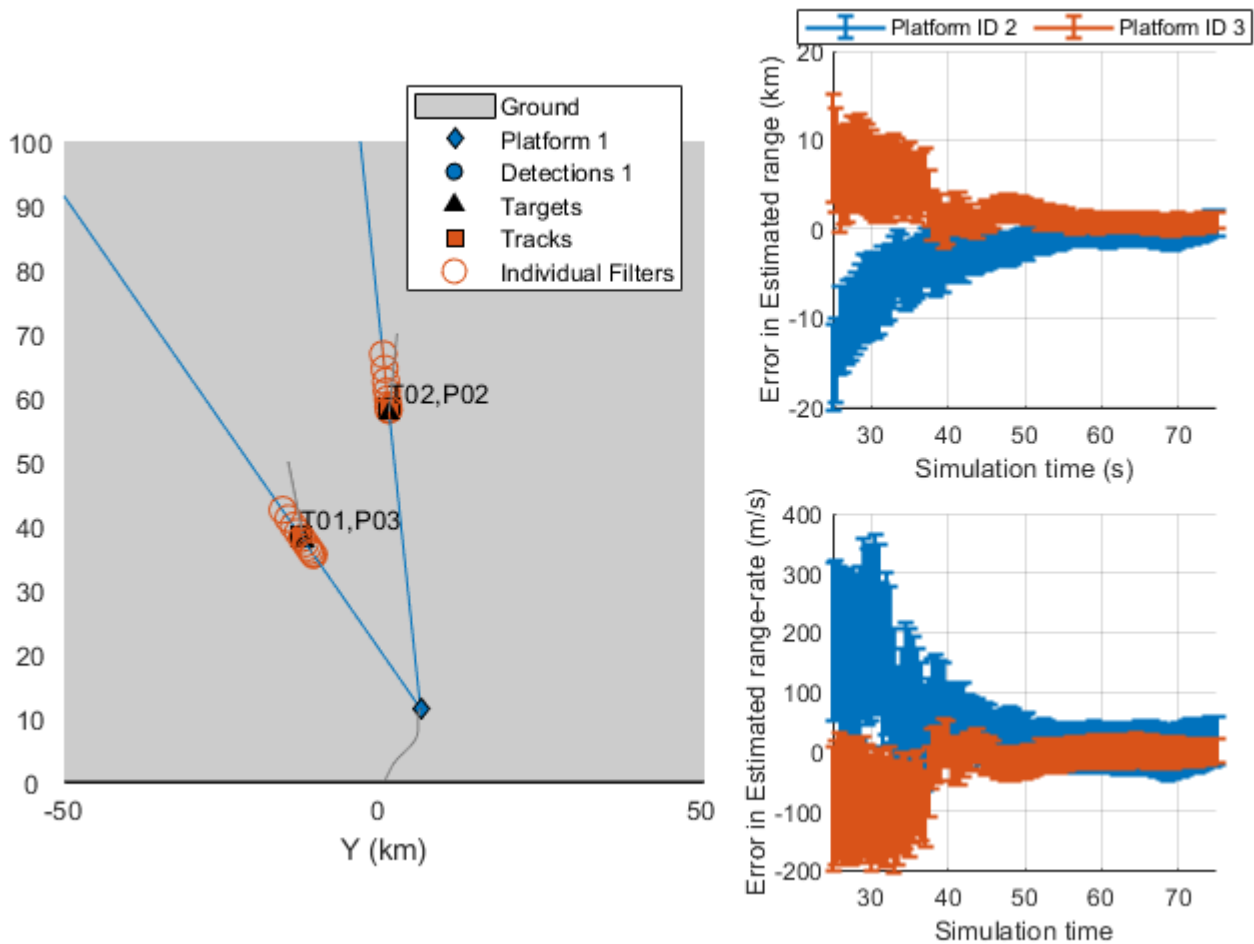
Range-parameterized MSC-EKF

```
% 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`, `initcvmsckf` 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;

```

```

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,2:2:end);
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';

```

```
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
```

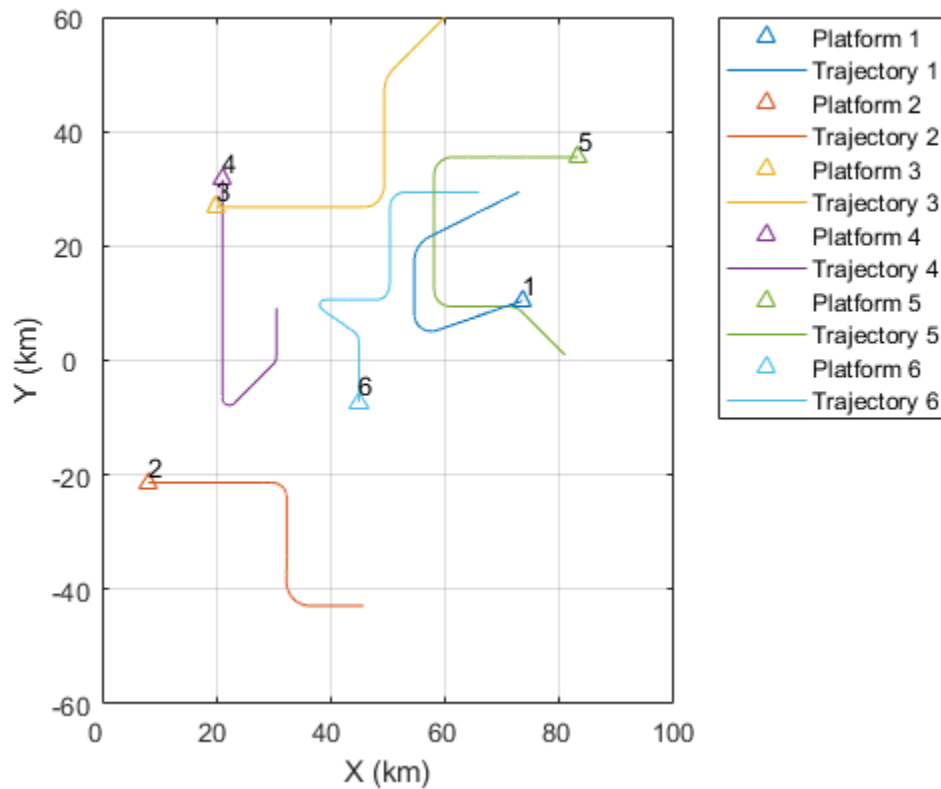
Trajectory Visualization

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));

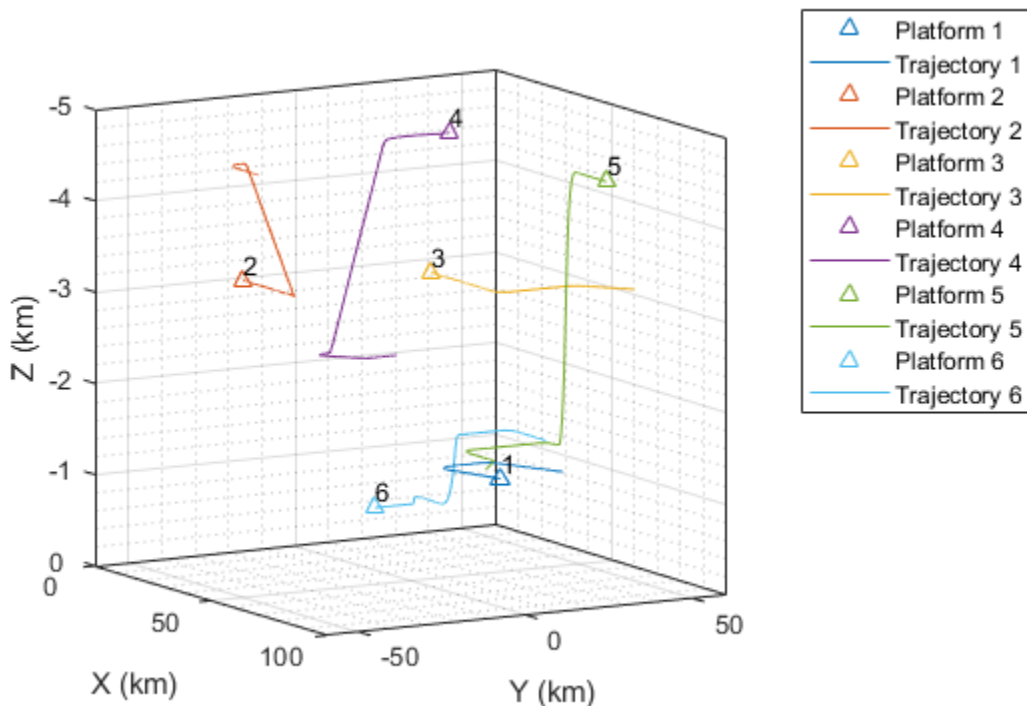
while advance(scene)
    % extract the pose of each of the six aircraft
    poses = platformPoses(scene);

    % 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');
```



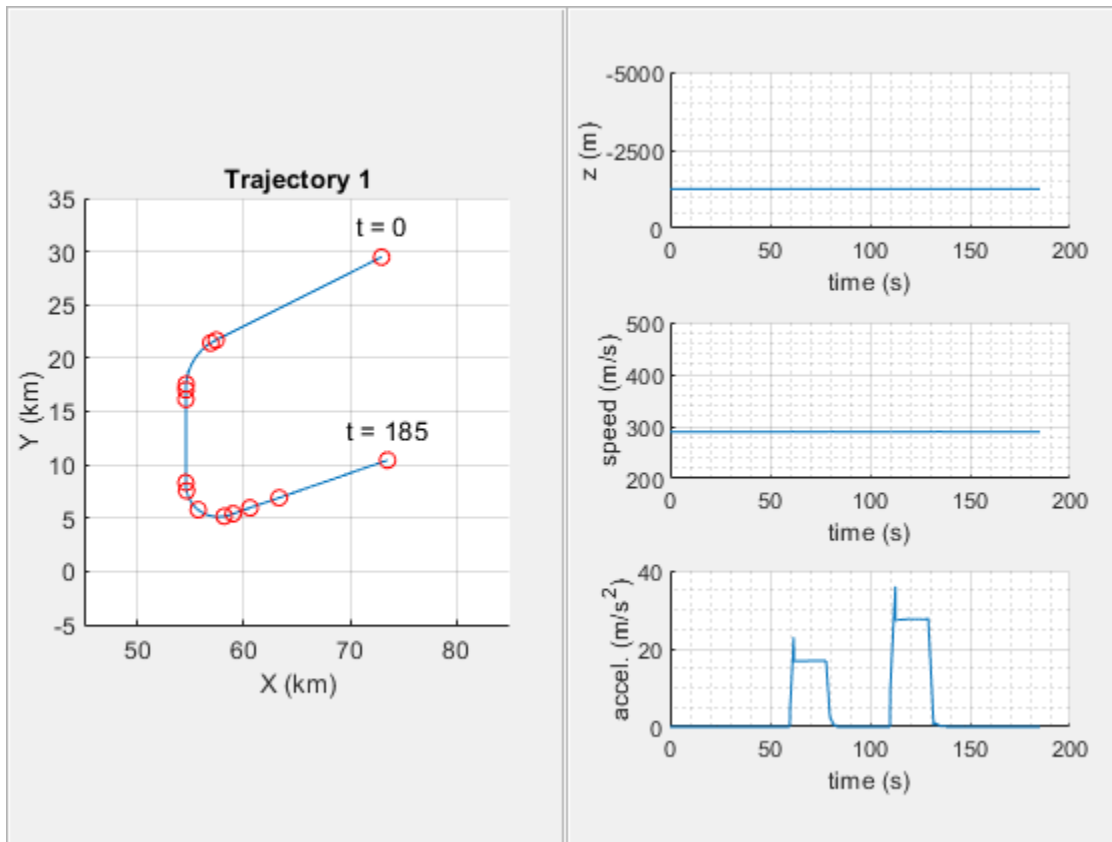
Trajectory 1

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.

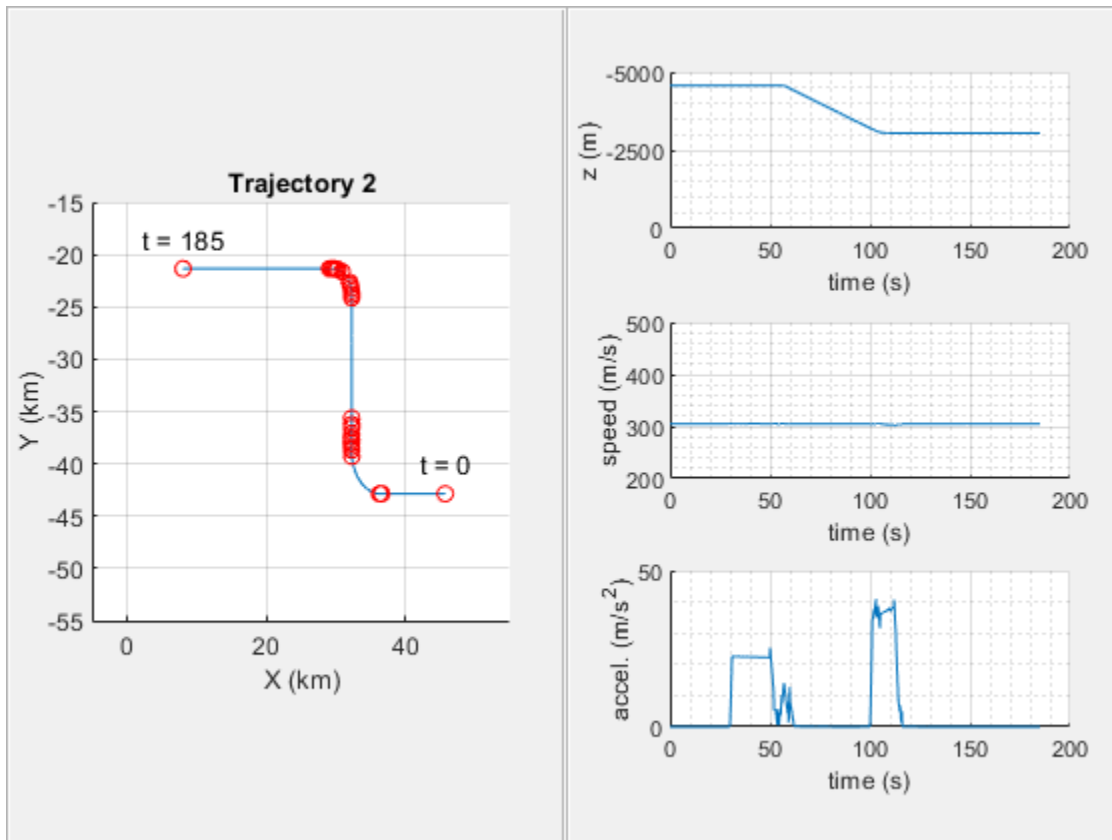
```
[time, position, velocity, acceleration] = cumulativeHistory(helperPlot);
helperTrajectoryViewer(1, time, position, velocity, acceleration, trajTable);
```

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.

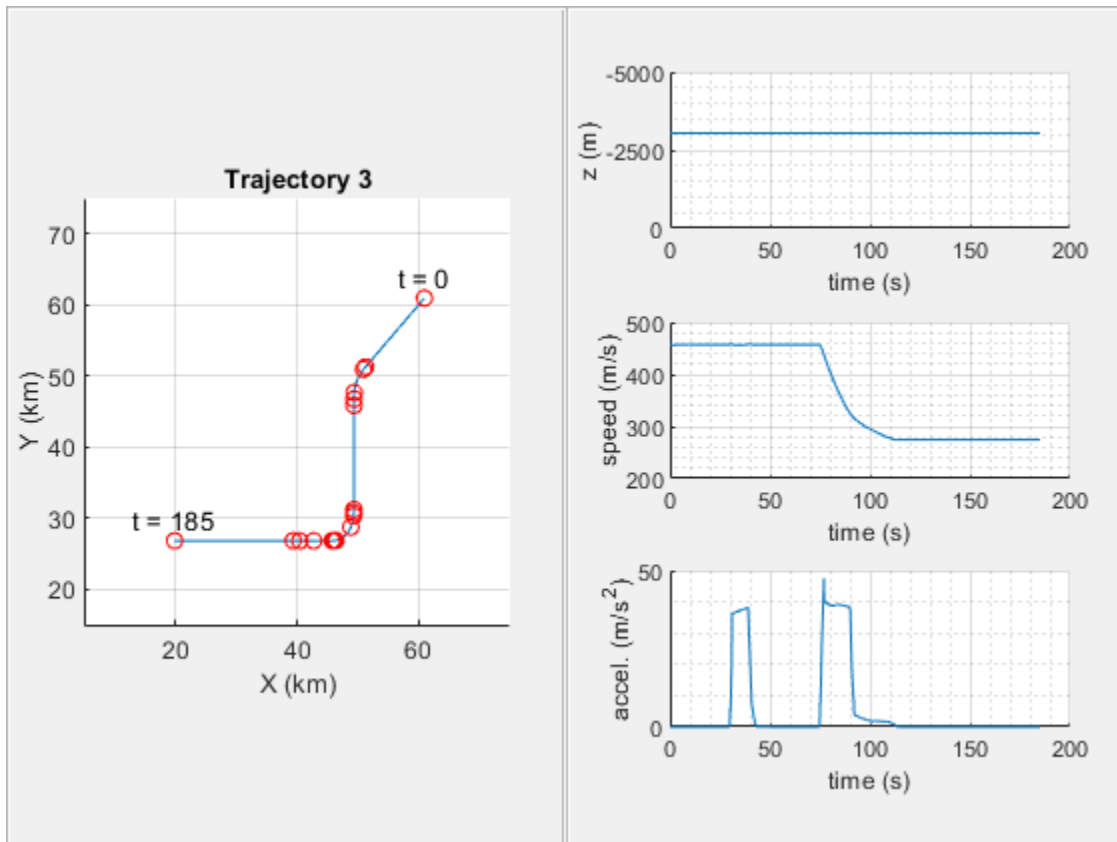
```
helperTrajectoryViewer(2, time, position, velocity, acceleration, trajTable);
```



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.

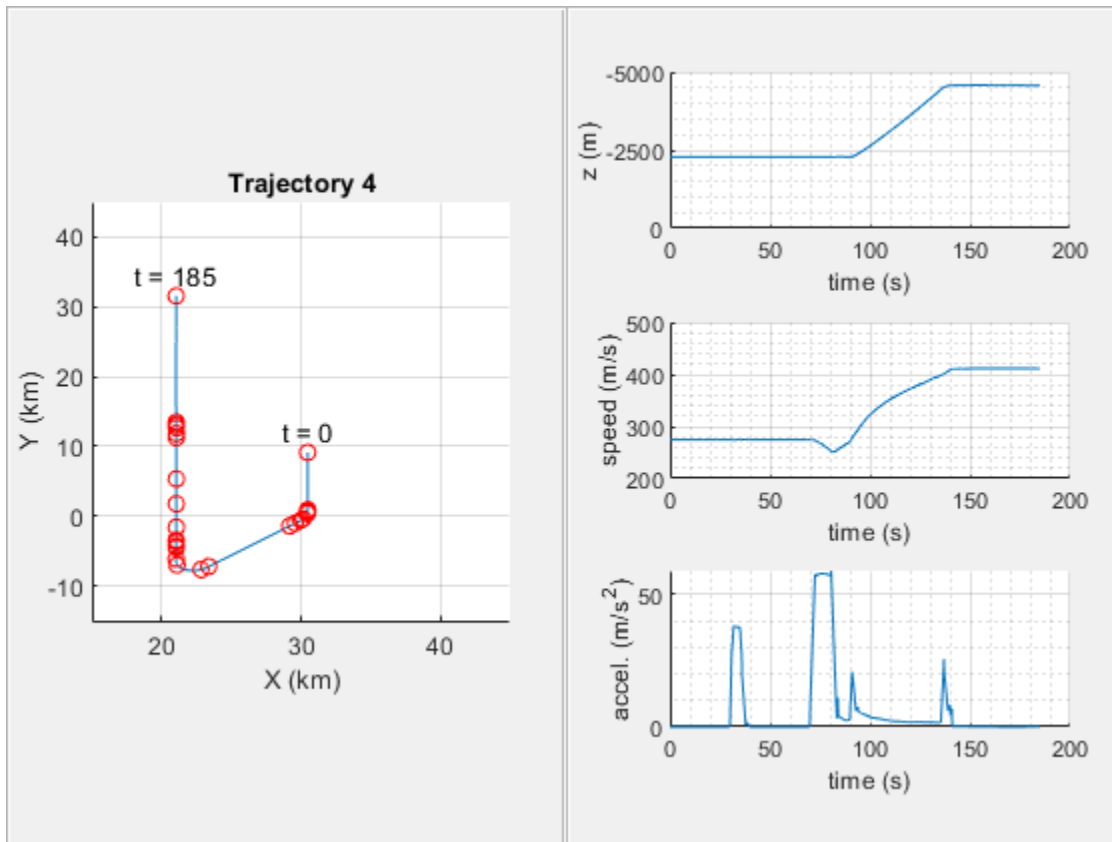
```
helperTrajectoryViewer(3, time, position, velocity, acceleration, trajTable);
```



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.

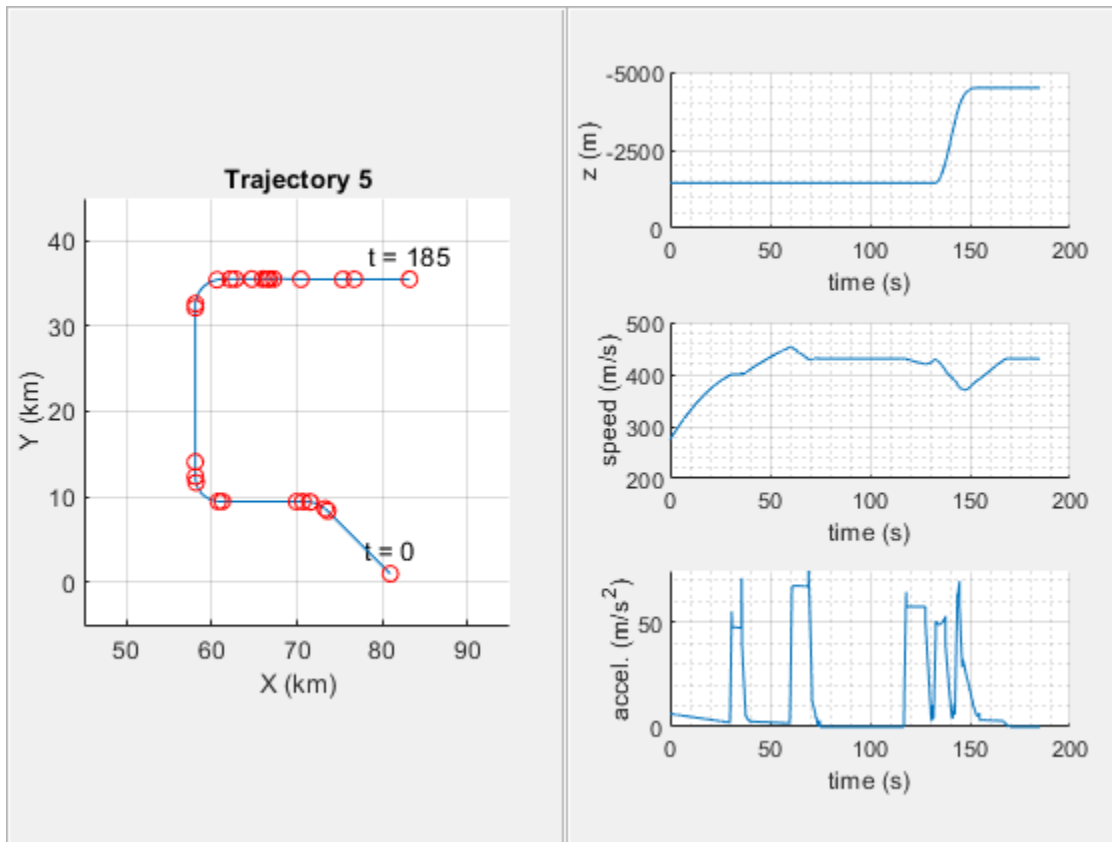
```
helperTrajectoryViewer(4, time, position, velocity, acceleration, trajTable);
```



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.

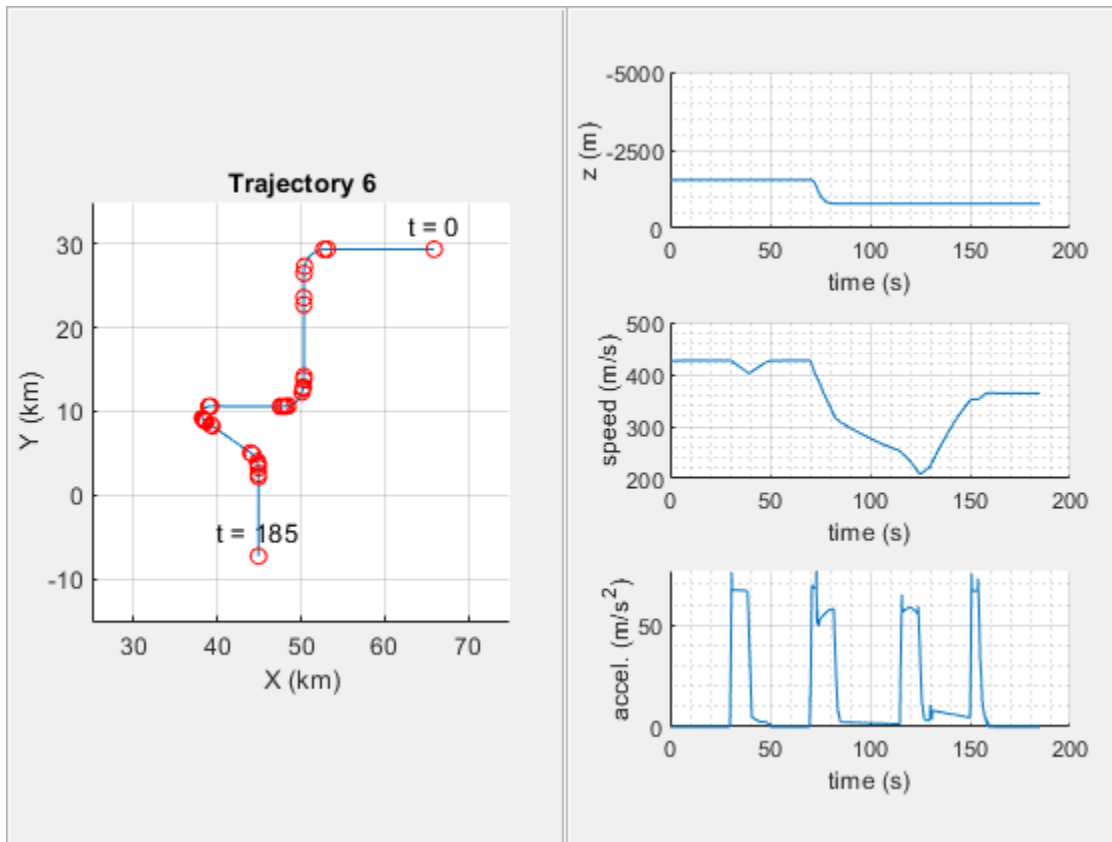
```
helperTrajectoryViewer(5, time, position, velocity, acceleration, trajTable);
```



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

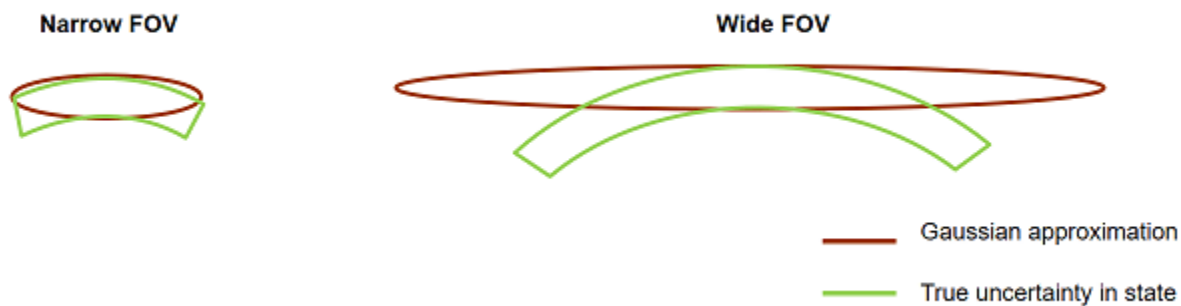
- 1 W.D. Blair, G. A. Watson, T. Kirubarajan, Y. Bar-Shalom, "Benchmark for Radar Allocation and Tracking in ECM." *Aerospace and Electronic Systems IEEE Trans on*, vol. 34. no. 4. 1998

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 6-203 example.

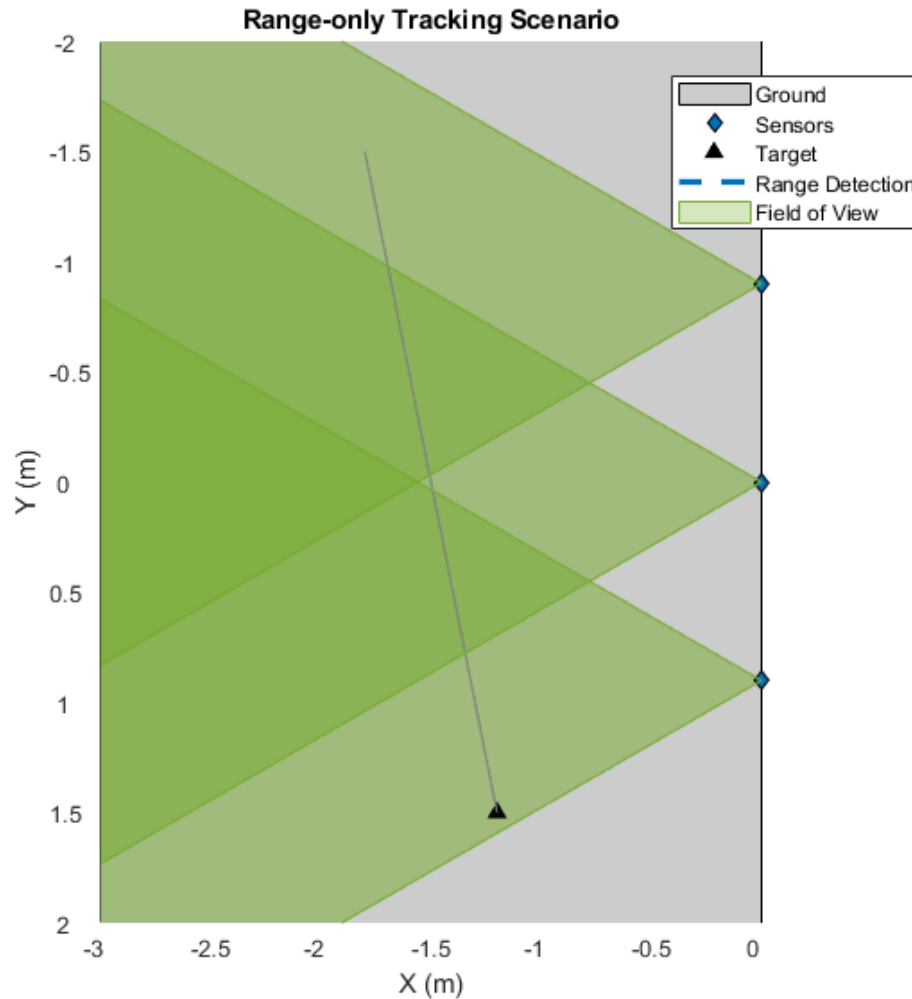
In this example, you will learn how to use a particle filter and a Gaussian-sum filter to represent the non-Gaussian uncertainty in state caused by range measurements from large FOV sensors.

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)
```



```
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);

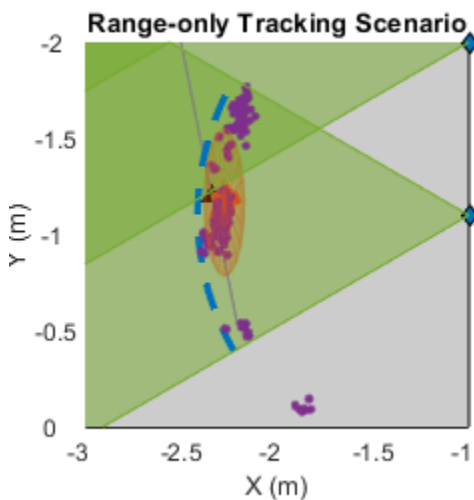
    % Pass detections to tracker
    if ~isempty(detections)
        [confTracks,~,allTracks] = tracker(detections,time);
    elseif isLocked(tracker)
        confTracks = predictTracksToTime(tracker,'confirmed',time);
    end

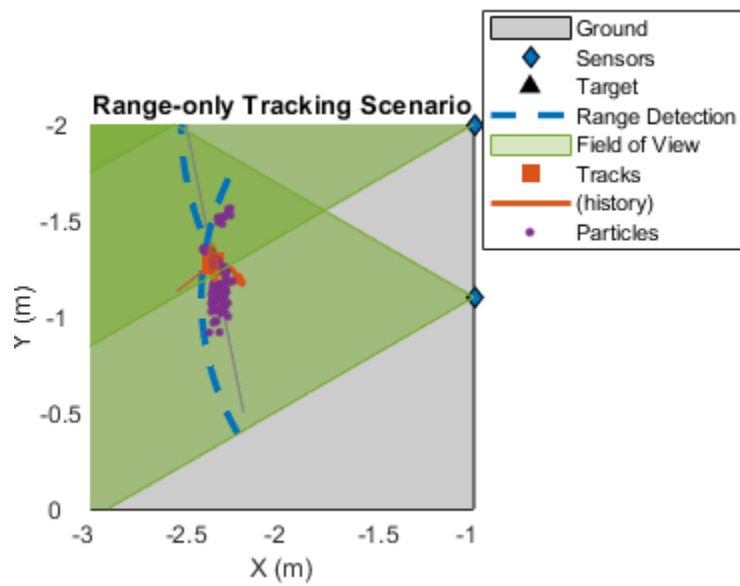
    % Update display
    theaterDisplay(detections,configs,confTracks,tracker);
end

```

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.

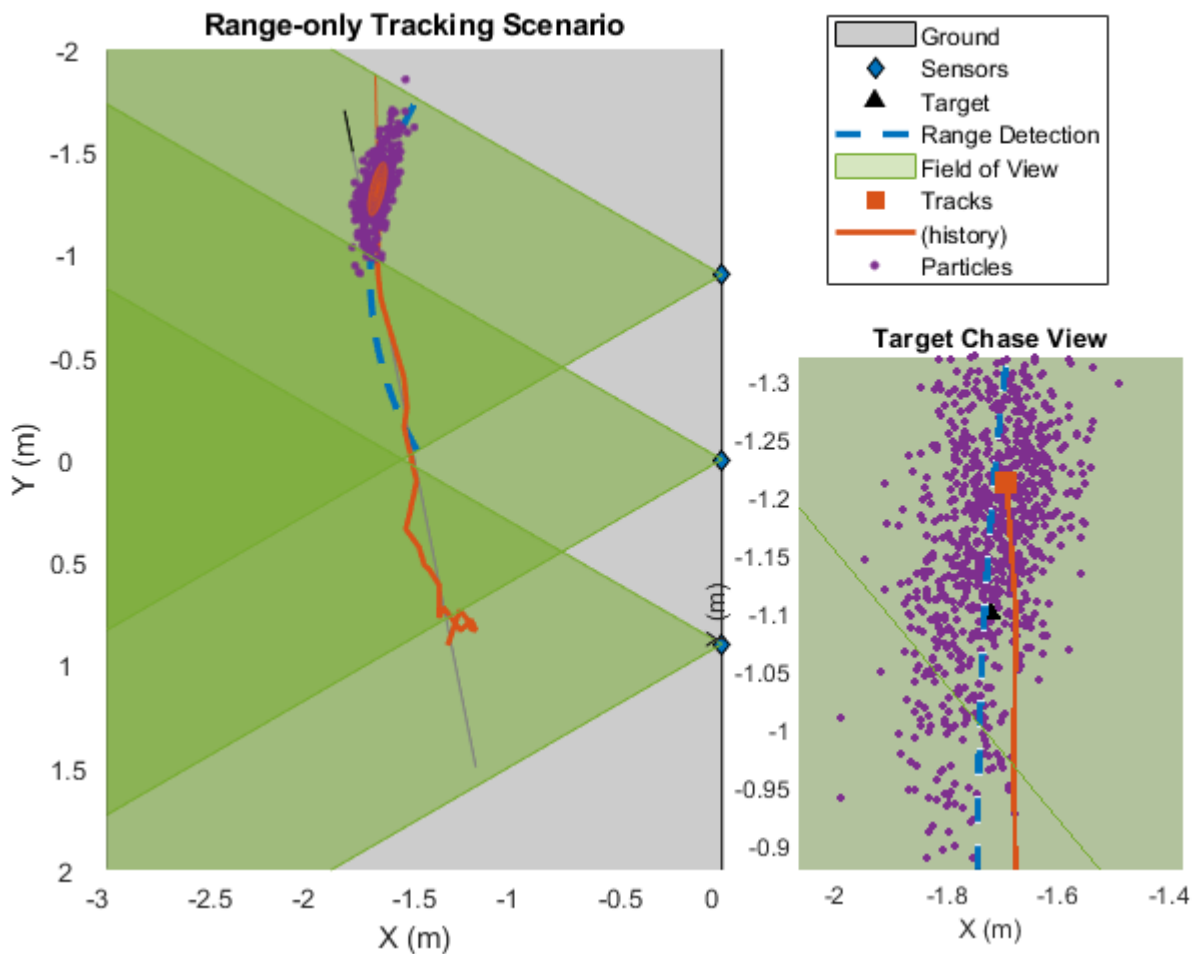
```
showGrabs(theaterDisplay,[1 2]);
```





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);

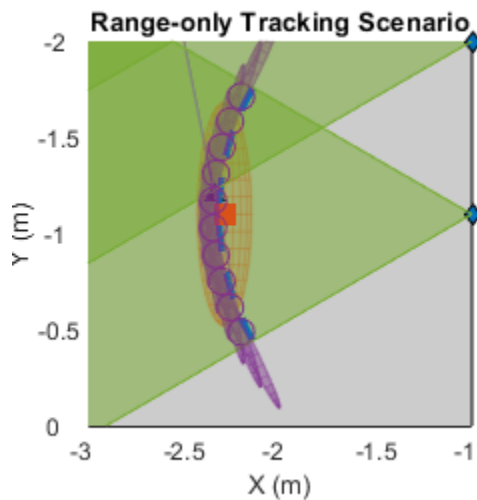
% Pass detections to tracker
if ~isempty(detections)
    [confTracks,~,allTracks] = tracker(detections,time);
elseif isLocked(tracker)
    confTracks = predictTracksToTime(tracker,'confirmed',time);
end

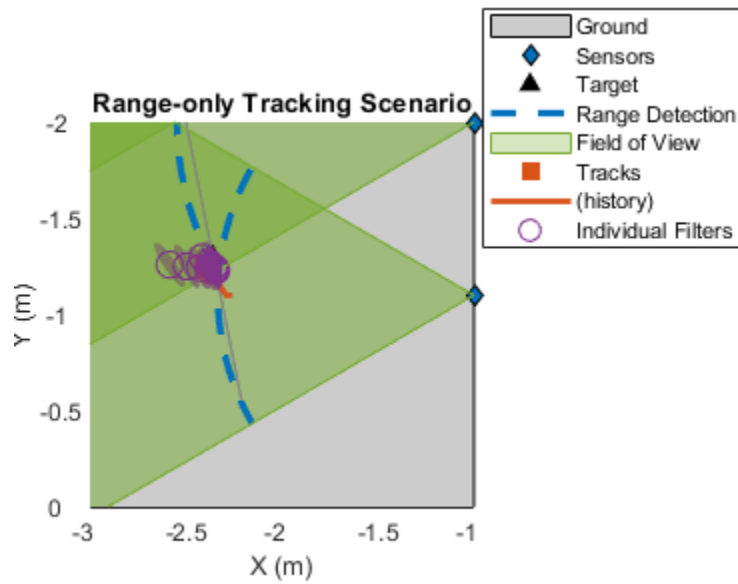
% Update display
theaterDisplay(detections,configs,confTracks,tracker);
end

```

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.

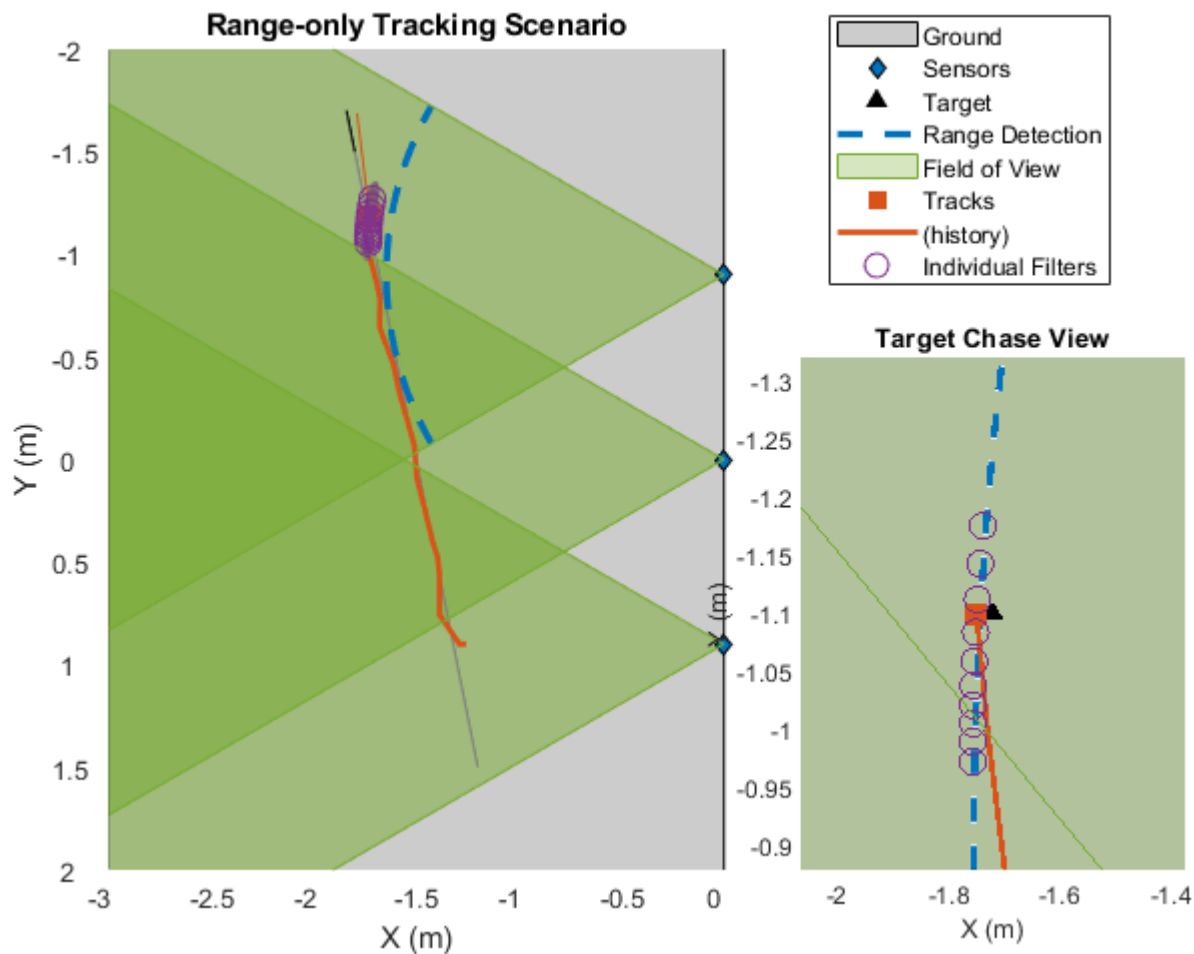
```
showGrabs(theaterDisplay,[4 5]);
```





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 shortcomings 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);
for i = 1:numel(config)
    % Populate the config correctly for plotting
    config(i).FieldOfView = observer.Sensors{i}.FieldOfView;
    config(i).MeasurementParameters(1).OriginPosition = observer.Sensors{i}.MountingLocation;
    config(i).MeasurementParameters(1).Orientation = rotmat(quaternion(observer.Sensors{i}.M
    config(i).MeasurementParameters(1).IsParentToChild = true;
end

% 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
```


Adaptive Tracking of Maneuvering Targets with Managed Radar

This example shows how to use radar resource management to efficiently track multiple maneuvering targets. Tracking maneuvering targets requires the radar to revisit the targets more frequently than tracking non-maneuvering targets. An interacting multiple model (IMM) filter estimates when the target is maneuvering. This estimate helps to manage the radar revisit time and therefore enhances the tracking. This example uses the Radar Toolbox™ for the radar model and Sensor Fusion and Tracking Toolbox™ for the tracking.

Introduction

Multifunction radars can search for targets, confirm new tracks, and revisit tracks to update the state. To perform these functions, a multifunction radar is often managed by a resource manager that creates radar tasks for search, confirmation, and tracking. These tasks are scheduled according to priority and time so that, at each time step, the multifunction radar can point its beam in a desired direction. The “Search and Track Scheduling for Multifunction Phased Array Radar” (Radar Toolbox) example shows a multifunction phased-array radar managed with a resource manager.

In this example, we extend the “Search and Track Scheduling for Multifunction Phased Array Radar” (Radar Toolbox) example to the case of multiple maneuvering targets. There are two conflicting requirements for a radar used to track maneuvering targets:

- 1 The number of targets and their initial location are typically not known in advance. Therefore, the radar must continuously search the area of interest to find the targets. Also, the radar needs to detect and establish a track on each target as soon as it enters the radar coverage area.
- 2 The time period of target maneuvering is unknown in advance. If it is known that targets are not maneuvering, the radar can revisit the targets infrequently. However, since the maneuver start and end times are unknown, the radar must revisit each track frequently enough to be able to recognize when the maneuver starts and ends.

The radar must balance between providing enough beams centered on the tracked targets and leaving enough time to search for new targets. One approach is to simply define a revisit rate on each tracked target regardless of its maneuvering status and leave the remaining time for new target searching. This radar management scheme is sometimes referred to as *Active Tracking* [1]. As more targets become tracked, the radar can either perform fewer search tasks or it can track each target less frequently. Clearly, if the number of targets is large, the radar can be overwhelmed.

Active tracking treats all the tracks in the same way, which makes it a *mode*-based resource management algorithm. A more sophisticated way to manage the radar is based on the properties of each track. For example, use track properties such as the size of state uncertainty covariance, whether the track is maneuvering, and how fast it is moving towards an asset the radar site protects. When such properties are used, the radar resource management is referred to as *Adaptive Tracking* [1].

In this example, you compare the results of Active Tracking and Adaptive Tracking when the radar adapts based on estimated track maneuver.

Define Scenario and Radar Model

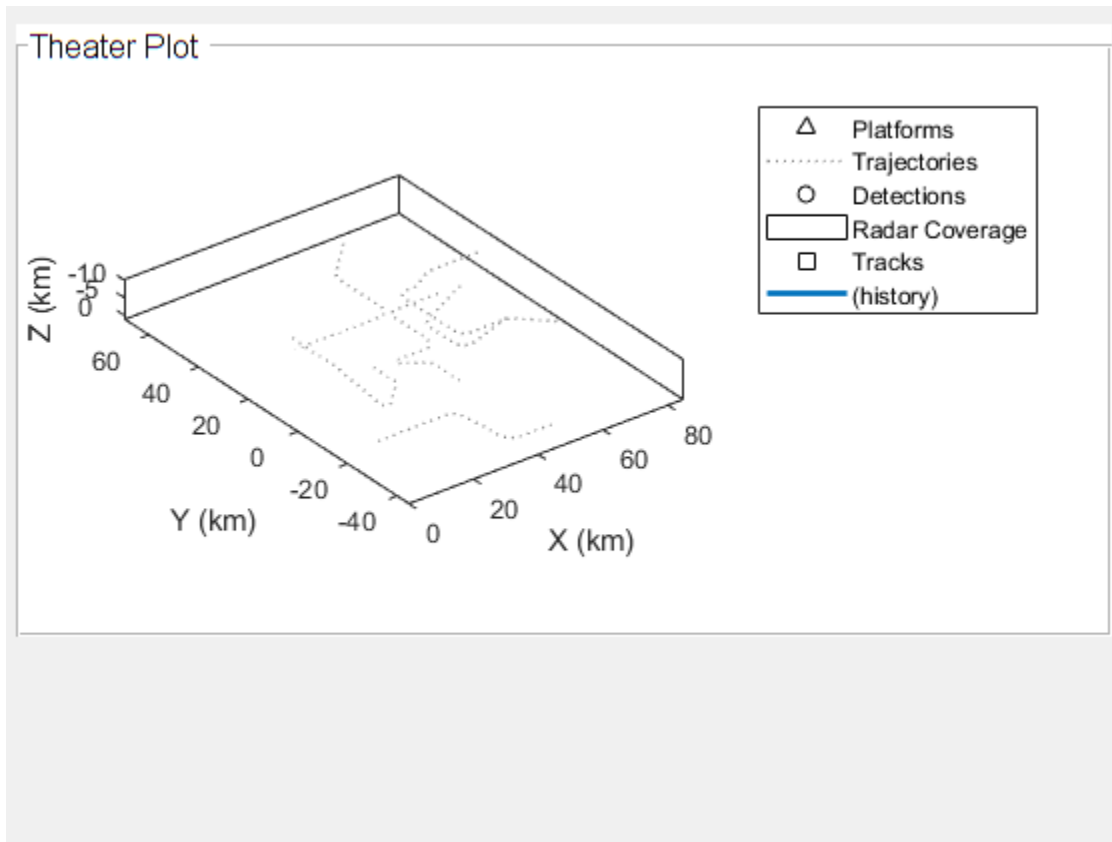
You define a scenario and a radar with an update rate of 20 Hz, which means that the radar has 20 beams per second allocated for either search, confirmation, or tracking. You load the benchmark trajectories used in the “Benchmark Trajectories for Multi-Object Tracking” on page 6-259 example.

There are six benchmark trajectories and you define a trajectory for each one. The six platforms in the figure follow non-maneuvering legs interspersed with maneuvering legs. You can view the trajectories in the figure.

```
% Create scenario
updateRate = 20;
scenario = trackingScenario('UpdateRate',updateRate);
% Add the benchmark trajectories
load('BenchmarkTrajectories.mat','-mat');
platform(scenario,'Trajectory',v1Trajectory);
platform(scenario,'Trajectory',v2Trajectory);
platform(scenario,'Trajectory',v3Trajectory);
platform(scenario,'Trajectory',v4Trajectory);
platform(scenario,'Trajectory',v5Trajectory);
platform(scenario,'Trajectory',v6Trajectory);

% Create visualization
f = figure;
mp = uipanel('Parent',f,'Title','Theater Plot','FontSize',12,...
    'BackgroundColor','white','Position',[.01 .25 .98 .73]);
tax = axes(mp,'ZDir','reverse');

% Visualize scenario
thp = theaterPlot('Parent',tax,'AxesUnits',['km',"km","km'],'XLimits',[0 85000],'YLimits',[-4500
plp = platformPlotter(thp, 'DisplayName', 'Platforms');
pap = trajectoryPlotter(thp, 'DisplayName', 'Trajectories', 'LineWidth', 1);
dtp = detectionPlotter(thp, 'DisplayName', 'Detections');
cvp = coveragePlotter(thp, 'DisplayName', 'Radar Coverage');
trp = trackPlotter(thp, 'DisplayName', 'Tracks', 'ConnectHistory', 'on', 'ColorizeHistory', 'on');
numPlatforms = numel(scenario.Platforms);
trajectoryPositions = cell(1,numPlatforms);
for i = 1:numPlatforms
    trajectoryPositions{i} = lookupPose(scenario.Platforms{i}.Trajectory,(0:0.1:185));
end
plotTrajectory(pap, trajectoryPositions);
view(tax,3)
```



A probabilistic radar model is defined using the `radarDataGenerator` System object™. Setting the 'ScanMode' property of this object to 'Custom' allows the resource manager to control the radar look angle. This enables scheduling of the radar for searching, confirming, and tracking targets. The radar is mounted on a new platform in the scenario.

```
radar = radarDataGenerator(1, ...
    'ScanMode', 'Custom', ...
    'UpdateRate', updateRate, ...
    'MountingLocation', [0 0 -15], ...
    'AzimuthResolution', 1.5, ...
    'ElevationResolution', 10, ...
    'HasElevation', true, ...
    'DetectionCoordinates', 'Sensor spherical');
platform(scenario, 'Position', [0 0 0], 'Sensors', radar);
```

Define Tracker

After the radar detects objects, it feeds the detections to a tracker, which performs several operations. The tracker maintains a list of tracks that are 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.

In this example, you use a tracker that associates the detections to the tracks using a global nearest neighbor (GNN) algorithm. To track the maneuvering targets, you define a

`FilterInitializationFcn` function that initializes an IMM filter. The `initMPARIMM` function uses two motion models: a constant-velocity model and a constant-turn rate model. The `trackingIMM` filter is responsible for estimating the probability of each model, which you can access from its `ModelProbabilities` property. In this example, you classify a target as maneuvering when the probability of the constant-turn rate model is higher than 0.6.

```
tracker = trackerGNN('FilterInitializationFcn',@initMPARIMM,...
    'ConfirmationThreshold',[2 3], 'DeletionThreshold',[5 5],...
    'HasDetectableTrackIDsInput',true,'AssignmentThreshold',150,...
    'MaxNumTracks',10,'MaxNumSensors',1);
posSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 1 0];
```

Radar Resource Management

This section only briefly outlines the radar resource management. For more details, see the “Adaptive Tracking of Maneuvering Targets with Managed Radar” (Radar Toolbox) example.

Search Tasks

In this example, you assign search tasks deterministically. A raster scan is used to cover the desired airspace. The azimuth scanning limits are set to [-90 60] degrees and the elevation limits to [-9.9 0] degrees. 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 radar's `AzimuthResolution` and `ElevationResolution` properties. Negative elevation angles mean that the radar points the beam from the horizon up.

```
AzimuthLimits = [-90 60];
ElevationLimits = [-9.9 0];
azscanspan = diff(AzimuthLimits);
numazscan = floor(azscanspan/radar.AzimuthResolution)+1;
azscanangles = linspace(AzimuthLimits(1),AzimuthLimits(2),numazscan)+radar.MountingAngles(1);
elscanspan = diff(ElevationLimits);
numelscan = floor(elscanspan/radar.ElevationResolution)+1;
elscanangles = linspace(ElevationLimits(1),ElevationLimits(2),numelscan)+radar.MountingAngles(2);
[elscangrid,azscangrid] = meshgrid(elscanangles,azscanangles);
scanangles = [azscangrid(:) elscangrid(:)].';
searchq = struct('JobType','Search','BeamDirection',num2cell(scanangles,1),...
    'Priority',1000,'WaveformIndex',1);
current_search_idx = 1;
```

Track Tasks

Unlike search tasks, track tasks cannot be planned in advance. Instead, the resource manager creates confirmation and tracking tasks based on the changing scenario. The main difference in this example from the “Adaptive Tracking of Maneuvering Targets with Managed Radar” (Radar Toolbox) example is that the `JobType` for each track task can be either `"TrackNonManeuvering"` or `"TrackManeuvering"`. The distinction between the two types of tracking tasks enables you to schedule tasks for each type of track at different revisit rates, making it an adaptive tracking algorithm. Similar to search tasks, tracking tasks are also managed in a job queue.

```
trackq = repmat(struct('JobType',[],'BeamDirection',[],'Priority',3000,'WaveformIndex',[],...
    'Time',[],'Range',[],'TrackID',[]), 10, 1);
num_trackq_items = 0;
```

Group search and tracking 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;
jobq.PositionSelector = posSelector;
% Keep a reset state of jobq
resetJobQ = jobq;

```

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 a confirmation or track task if its time to run has come. Otherwise, the radar picks a search task. To control the time to run tasks, you set the `managerPreferences` structure defined below. The highest revisit rate, which is equal to the radar update rate, is given to the `confirm` task to guarantee that a confirmation beam follows every new tentative track that exists. Similarly, you can control the revisit rate for non-maneuvering and maneuvering targets. In this case, you choose the values of 0.8 Hz and 4 Hz for non-maneuvering and maneuvering targets, respectively. Since there are 6 targets, the resource manager will use $6 \cdot 0.8 \approx 5$ updates per second on tracking targets if all of them are not maneuvering. Given the radar update rate is 20 Hz, the radar manager will perform approximately one target update in every four updates. Thus, the radar will be spending about 75% of the time in the search mode and 25% of the time in the tracking mode. When new tracks are initialized and when the tracker considers that the targets are maneuvering, the resource manager allocates more tracking beams at the expense of search beams.

The Analyze Results section shows the results of other options.

```

managerPreferences = struct(...
    'Type', {"Search", "Confirm", "TrackNonManeuvering", "TrackManeuvering"}, ...
    'RevisitRate', {0, updateRate, 0.8, 4}, ...
    'Priority', {1000, 3000, 1500, 2500});

```

Run the Scenario

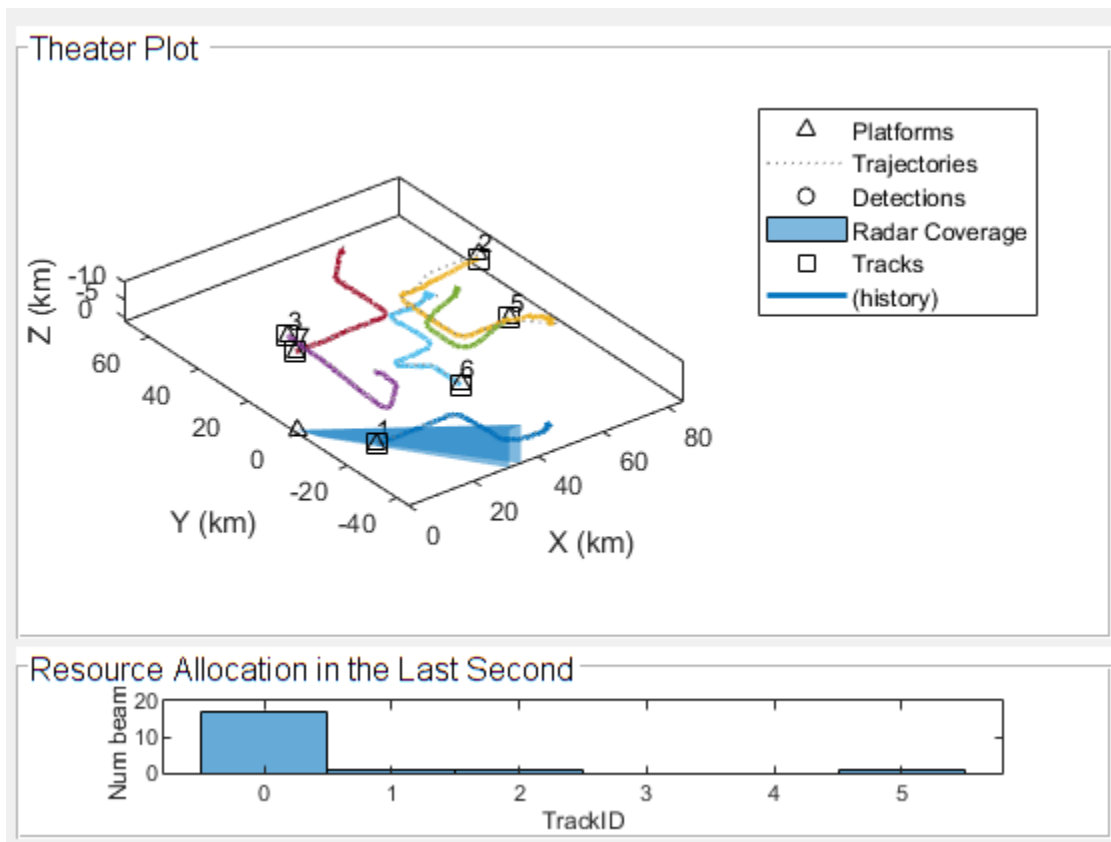
During the simulation, the radar beam is depicted by blue or purple colors representing search and track-related beams, respectively. You can also see the distribution of tasks between search and specific tracks for the last second of simulation using the Resource Allocation in the Last Second panel at the bottom of the figure. In the Theater Plot part of the figure, you can see the history of each track and compare it to the trajectory.

```

% Create a radar resource allocation display
rp = uipanel('Parent',f,'Title','Resource Allocation in the Last Second','FontSize',12,...
    'BackgroundColor','white','Position',[.01 0.01 0.98 0.23]);
rax = axes(rp);

% Run the scenario
allocationType = helperAdaptiveTrackingSim(scenario, thp, rax, tracker, resetJobQ, managerPreferences);

```



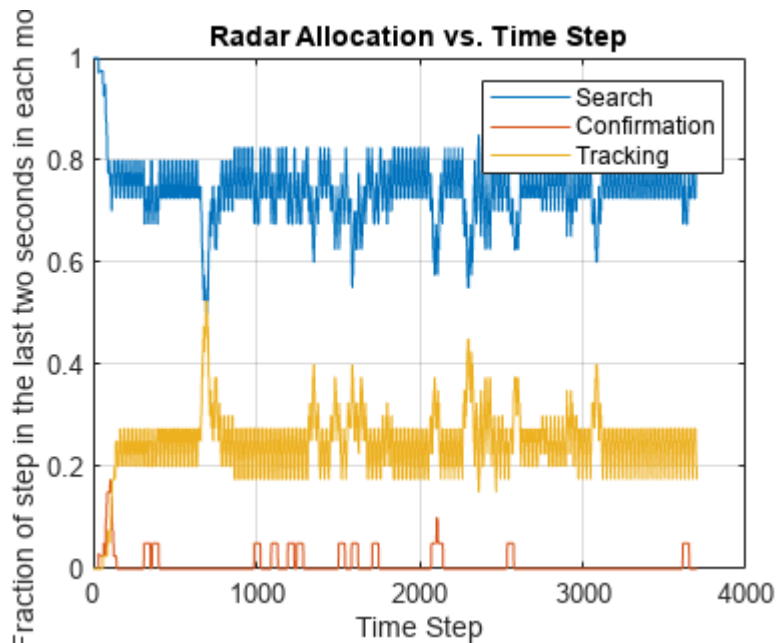
Analyze Results

Analyze the radar task load and its division between search, confirmation, and tracking jobs. The graph shows that most of the time the radar allocates about 75% search jobs and 25% tracking jobs, which is as expected when the targets are not maneuvering. When the targets are maneuvering, the resource manager adapts to allocate more tracking jobs. When more tracks are maneuvering at the same time, there are more tracking jobs, as seen near the 700th time step. The confirmation jobs occupy very little of the radar time because the tracker is configured to confirm tracks after two detection associations in three attempts. Therefore, the confirmation or rejection of tentative tracks is swift.

```

numSteps = numel(allocationType);
allocationSummary = zeros(3,numSteps);
for i = 1:numSteps
    for jobType = 1:3
        allocationSummary(jobType,i) = sum(allocationType(1,max(1,i-2*updateRate+1):i)==jobType);
    end
end
figure;
plot(1:numSteps,allocationSummary(:,1:numSteps))
title('Radar Allocation vs. Time Step');
xlabel('Time Step');
ylabel('Fraction of step in the last two seconds in each mode');
legend('Search','Confirmation','Tracking');
grid on

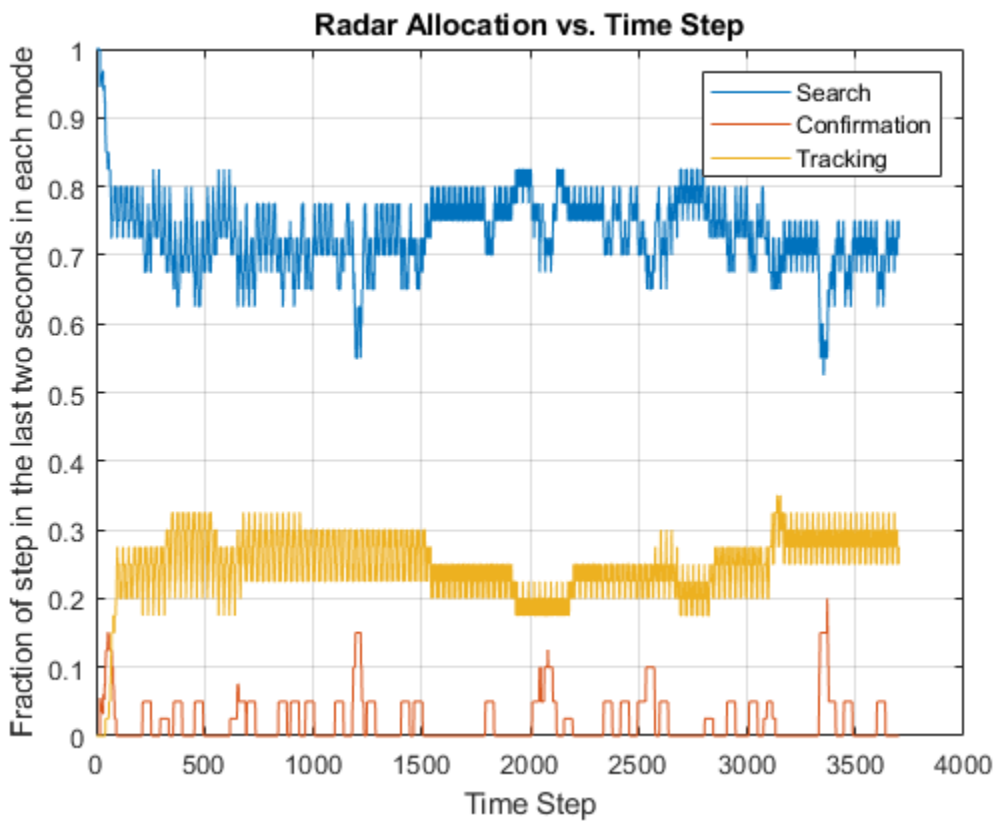
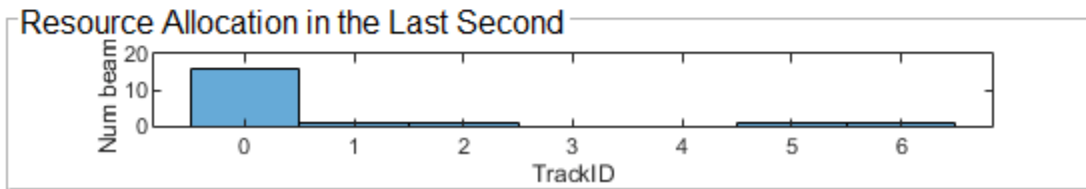
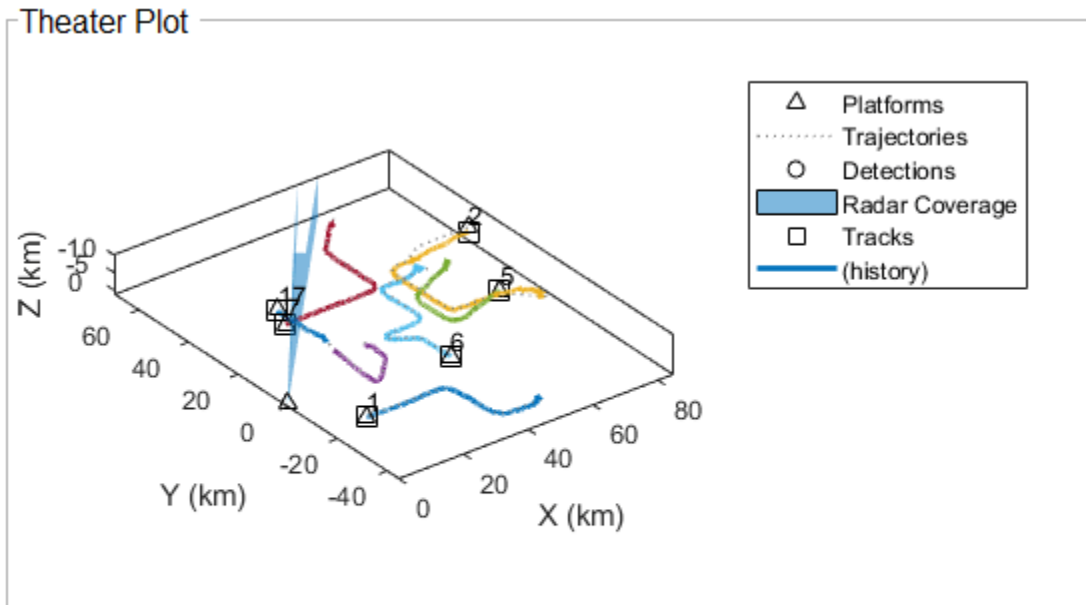
```



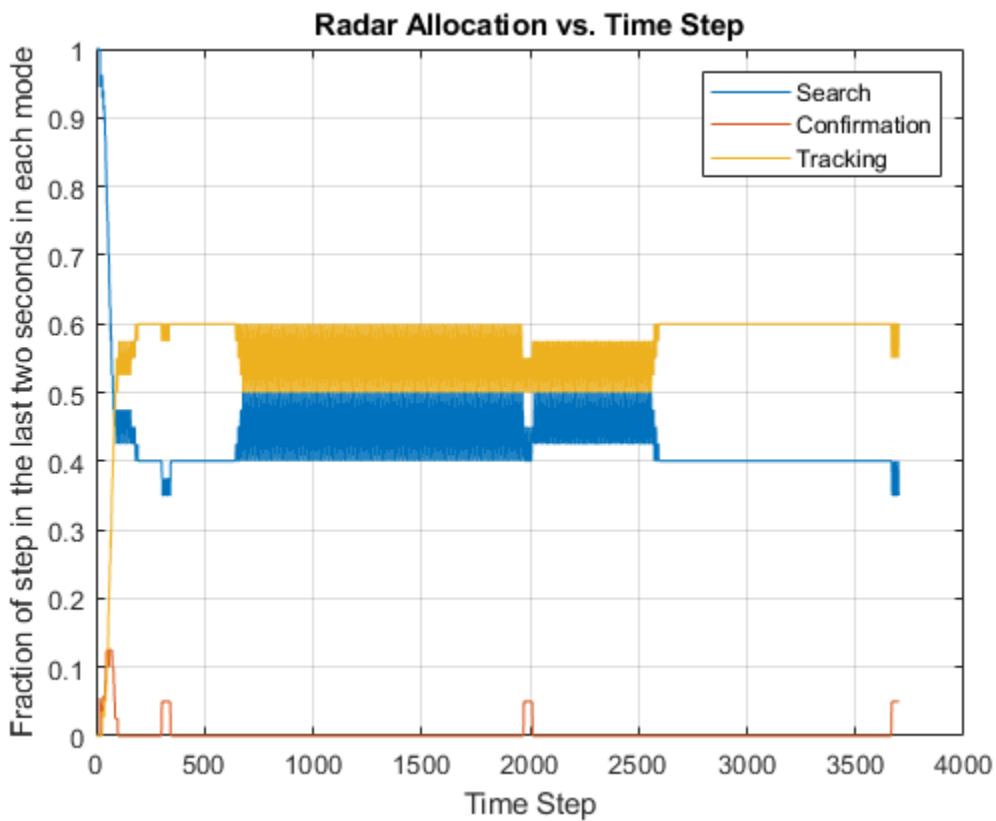
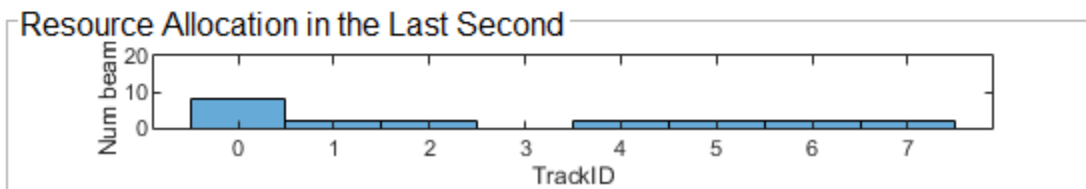
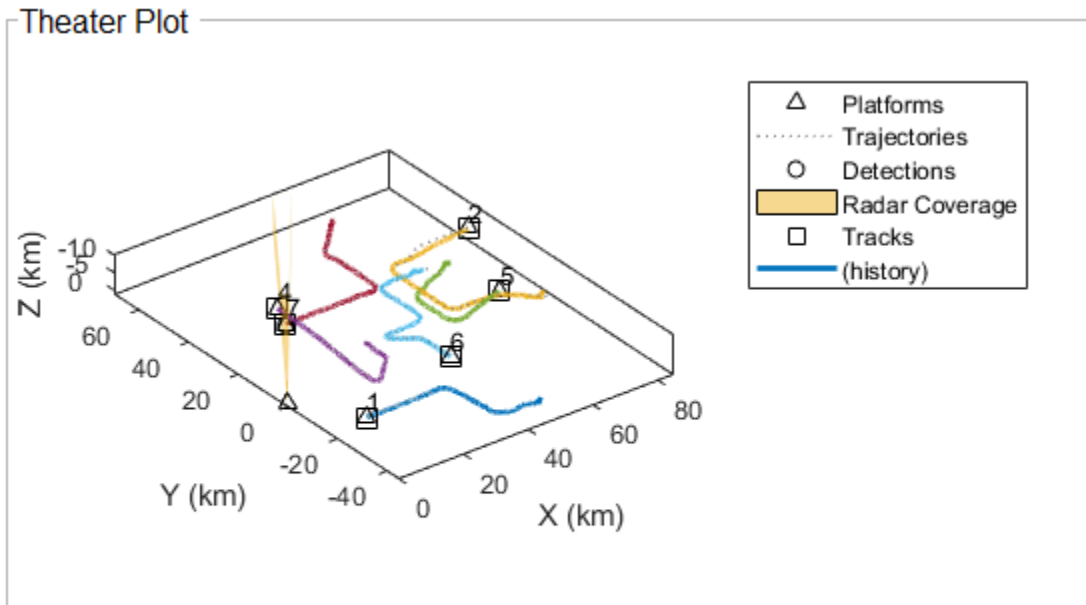
Compare this result with the result of Active Tracking at 0.8 Hz track revisit rate. The following figures show the tracking results and radar allocation graph for the Active Tracking case. The tracking results show that some tracks were lost and broken, but the radar resource allocation graph shows a similar 75% search and 25% tracking task division as in the case of the Adaptive Tracking. You can obtain these results by executing the code sample below.

```
clearData(plp);
clearData(dtp);
clearData(trp);
reset(tracker);
restart(scenario);
```

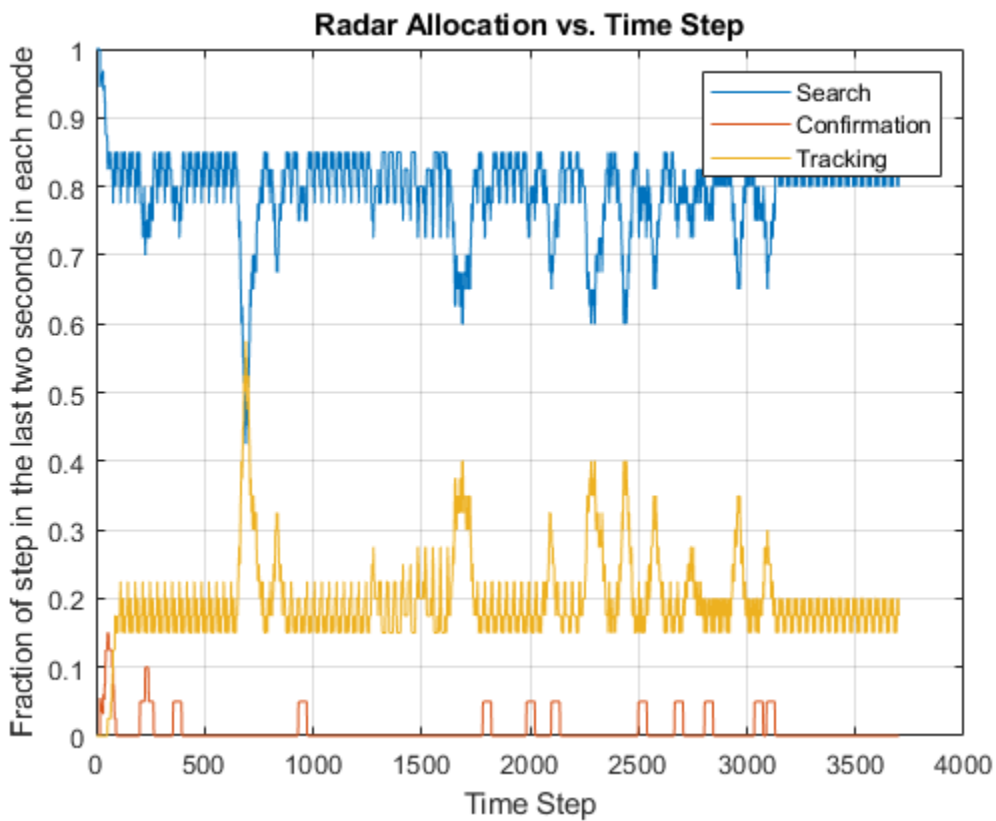
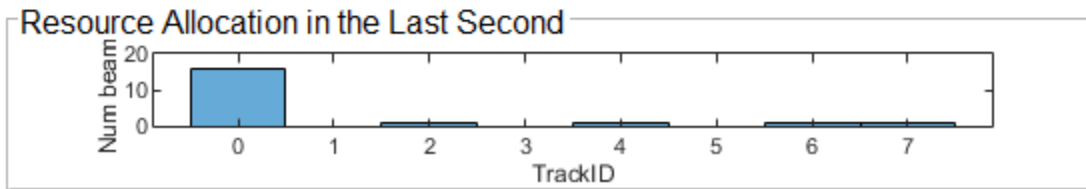
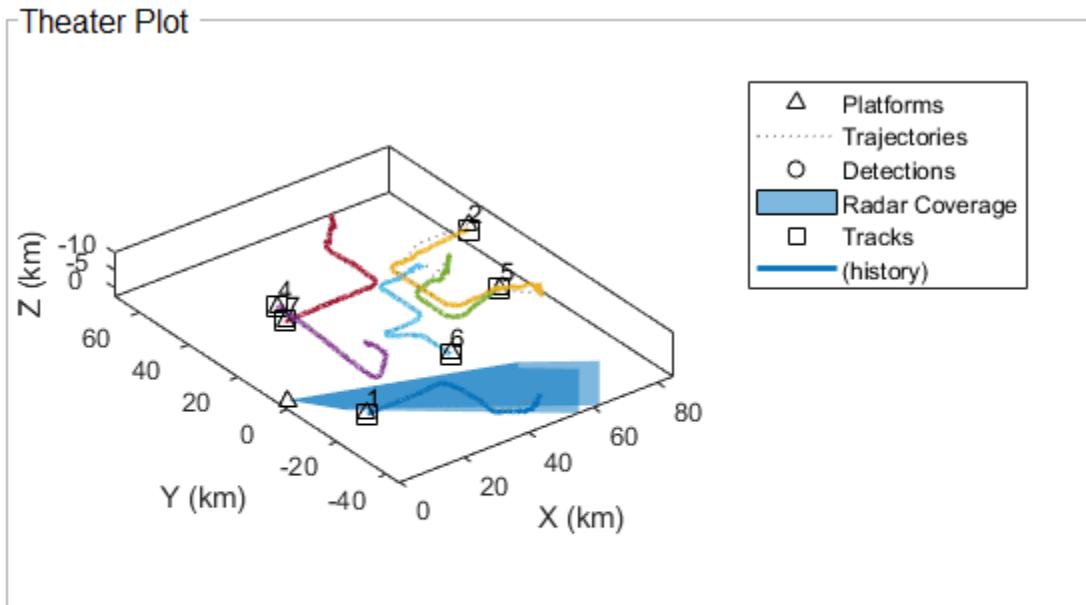
```
% Modify manager preferences to 0.8 Hz revisit rate for both non-maneuvering and maneuvering targets
managerPreferences = struct(...
    'Type', {"Search", "Confirm", "TrackNonManeuvering", "TrackManeuvering"}, ...
    'RevisitRate', {0, updateRate, 0.8, 0.8}, ...
    'Priority', {1000, 3000, 1500, 2500});
% Run the scenario
allocationType = helperAdaptiveTrackingSim(scenario, thp, rax, tracker, resetJobQ, managerPreferences);
```



Clearly, a higher tracking revisit rate is needed for Active Tracking. The following two graphs show that increasing the track revisit rate to 2 Hz improves the tracking of maneuvering targets. However, the cost is that the radar dedicates more than 50% of its time to tracking tasks even when the tracks are not maneuvering. If the number of targets were greater, the radar would become overwhelmed.



The previous results show that it is enough to revisit the maneuvering targets at a rate of 2 Hz. However, can Adaptive Tracking be used to lower the revisit rate of non-maneuvering targets beyond 0.8 Hz? The following graphs present results for 0.6 Hz and 4 Hz for non-maneuvering and maneuvering targets, respectively. With this setting, the radar resource allocation allows for 80%-85% of the time in search mode, leaving the radar with capacity to search and track even more targets.



Summary

This example shows how to use the combination of tracking and radar resource management to adapt the revisit rate for maneuvering tracks. Adaptive tracking allows you to select revisit rate that is appropriate for each target type and maneuvering status. As a result, the radar becomes more efficient and can track a larger number of maneuvering targets.

References

[1] Charlish, Alexander, Folker Hoffmann, Christoph Degen, and Isabel Schlangen. "The Development From Adaptive to Cognitive Radar Resource Management." *IEEE Aerospace and Electronic Systems Magazine* 35, no. 6 (June 1, 2020): 8-19. <https://doi.org/10.1109/MAES.2019.2957847>.

Supporting Functions

getCurrentRadarTask

Returns the radar task that is used for pointing the radar beam.

```
type('getCurrentRadarTask.m')

function [currentjob,jobq] = getCurrentRadarTask(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;
```

helperAdaptiveRadarSim

Runs the simulation

```

type('helperAdaptiveTrackingSim.m')

function allocationType = helperAdaptiveTrackingSim(scenario, thp, rax, tracker, resetJobQ, mana)
% Initialize variables
radar = scenario.Platforms{end}.Sensors{1};
updateRate = radar.UpdateRate;
resourceAllocation = nan(1,updateRate);
h = histogram(rax,resourceAllocation,'BinMethod','integers');
xlabel(h.Parent,'TrackID')
ylabel(h.Parent,'Num beams');
numSteps = updateRate * 185;
allocationType = nan(1,numSteps);
currentStep = 1;
restart(scenario);
reset(tracker);
% Return to a reset state of jobq
jobq = resetJobQ;

% Plotters and axes
plp = thp.Plotters(1);
dtp = thp.Plotters(3);
cvp = thp.Plotters(4);
trp = thp.Plotters(5);

ax = gca;
colors = ax.ColorOrder;

% For repeatable results, set the random seed and revert it when done
s = rng(2020);
oc = onCleanup(@( ) rng(s));

% Main loop
tracks = {};

while advance(scenario)
    time = scenario.SimulationTime;

    % Update ground truth display
    poses = platformPoses(scenario);
    plotPlatform(plp, reshape([poses.Position],3,[]));

    % Point the radar based on the scheduler current job
    [currentJob,jobq] = getCurrentRadarTask(jobq,time);
    currentStep = currentStep + 1;
    if currentStep > updateRate
        resourceAllocation(1:end-1) = resourceAllocation(2:updateRate);
    end
    if strcmpi(currentJob.JobType,'Search')
        detectableTracks = zeros(0,1,'uint32');
        resourceAllocation(min([currentStep,updateRate])) = 0;
        allocationType(currentStep) = 1;
        cvp.Color = colors(1, :);
    else
        detectableTracks = currentJob.TrackID;
        resourceAllocation(min([currentStep,updateRate])) = currentJob.TrackID;
    end
end

```

```

    if strcmpi(currentJob.JobType,'Confirm')
        allocationType(currentStep) = 2;
        cvp.Color = colors(2, :);
    else
        allocationType(currentStep) = 3;
        cvp.Color = colors(3, :);
    end
end
ra = resourceAllocation(~isnan(resourceAllocation));
h.Data = ra;
h.Parent.YLim = [0 updateRate];
h.Parent.XTick = 0:max(ra);
radar.LookAngle = currentJob.BeamDirection;
plotCoverage(cvp, coverageConfig(scenario));

% Collect detections and plot them
detections = detect(scenario);
if isempty(detections)
    meas = zeros(0,3);
else
    dets = [detections{:}];
    meassph = reshape([dets.Measurement],3,[]);
    [x,y,z] = sph2cart(deg2rad(meassph(1)),deg2rad(meassph(2)),meassph(3));
    meas = (detections{1}.MeasurementParameters.Orientation*[x;y;z]+detections{1}.Measurement);
end
plotDetection(dtp, meas);

% Track and plot tracks
if isLocked(tracker) || ~isempty(detections)
    [confirmedTracks,tentativeTracks,~,analysisInformation] = tracker(detections, time, detector);
    pos = getTrackPositions(confirmedTracks,jobq.PositionSelector);
    plotTrack(trp,pos,string([confirmedTracks.TrackID]));

    tracks.confirmedTracks = confirmedTracks;
    tracks.tentativeTracks = tentativeTracks;
    tracks.analysisInformation = analysisInformation;
    tracks.PositionSelector = jobq.PositionSelector;
end

% Manage resources for next jobs
jobq = manageResource(detections,jobq,tracker,tracks,currentJob,time,managerPreferences);
end

```

initMPARIMM

Initializes the IMM filter used by the tracker.

```

type('initMPARIMM.m')

function imm = initMPARIMM(detection)

cvekf = initcvekf(detection);
cvekf.StateCovariance(2,2) = 1e6;
cvekf.StateCovariance(4,4) = 1e6;
cvekf.StateCovariance(6,6) = 1e6;

```

```

ctekf = initctekf(detection);
ctekf.StateCovariance(2,2) = 1e6;
ctekf.StateCovariance(4,4) = 1e6;
ctekf.StateCovariance(7,7) = 1e6;
ctekf.ProcessNoise(3,3) = 1e6; % Large noise for unknown angular acceleration

imm = trackingIMM('TrackingFilters', {cvekf;ctekf}, 'TransitionProbabilities', [0.99, 0.99]);

```

manageResources

Manages the job queue and creates new tasks based on the tracking results.

```
type('manageResource.m')
```

```
function jobq = manageResource(detections,jobq,tracker,tracks,current_job,current_time,managerPre
```

```

trackq          = jobq.TrackQueue;
num_trackq_items = jobq.NumTrackJobs;
if ~isempty(detections)
    detection = detections{1};
else
    detection = [];
end

% 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)
            % A search task can still find a track we already have. Define
            % a confirmation task only if it's a tentative track. There
            % could be more than one if there are false alarms. Create
            % confirm jobs for tentative tracks created at this update.

            numTentative = numel(tracks.tentativeTracks);
            for i = 1:numTentative
                if tracks.tentativeTracks(i).Age == 1 && tracks.tentativeTracks(i).IsCoasted == 0
                    trackid = tracks.tentativeTracks(i).TrackID;
                    job = revisitTrackJob(tracker, trackid, current_time, managerPreferences, 'Confirm');
                    num_trackq_items = num_trackq_items+1;
                    trackq(num_trackq_items) = job;
                end
            end
        end

    case 'Confirm'
        % For a confirm job, if the track ID is within the tentative
        % tracks, it means that we need to run another confirmation job
        % regardless of having a detection. If the track ID is within the
        % confirmed tracks, it means that we must have gotten a detection,
        % and the track passed the confirmation logic test. In this case we
        % need to schedule a track revisit job.
        trackid = current_job.TrackID;
        tentativeTrackIDs = [tracks.tentativeTracks.TrackID];
        confirmedTrackIDs = [tracks.confirmedTracks.TrackID];
        if any(trackid == tentativeTrackIDs)
            job = revisitTrackJob(tracker, trackid, current_time, managerPreferences, 'Confirm',

```



```

        num_trackq_items = num_trackq_items+1;
        trackq(num_trackq_items) = job;
    elseif any(trackid == confirmedTrackIDs)
        job = revisitTrackJob(tracker, trackid, current_time, managerPreferences, 'TrackNonM
        num_trackq_items = num_trackq_items+1;
        trackq(num_trackq_items) = job;
    end

    otherwise % Covers both types of track jobs
        % For track job, if the track hasn't been dropped, update the track
        % and schedule a track job corresponding to the revisit time
        % regardless of having a detection. In the case when there is no
        % detection, we could also predict and schedule a track job sooner
        % so the target is not lost. This would require defining another
        % job type to control the revisit rate for this case.

        trackid = current_job.TrackID;
        confirmedTrackIDs = [tracks.confirmedTracks.TrackID];
        if any(trackid == confirmedTrackIDs)
            jobType = 'TrackNonManeuvering';
            mdlProbs = getTrackFilterProperties(tracker, trackid, 'ModelProbabilities');
            if mdlProbs{1}(2) > 0.6
                jobType = 'TrackManeuvering';
            end

            job = revisitTrackJob(tracker, trackid, current_time, managerPreferences, jobType, t
            num_trackq_items = num_trackq_items+1;
            trackq(num_trackq_items) = job;
        end
    end

end

jobq.TrackQueue = trackq;
jobq.NumTrackJobs = num_trackq_items;
end

function job = revisitTrackJob(tracker, trackID, currentTime, managerPreferences, jobType, posit
    types = [managerPreferences.Type];
    inTypes = strcmpi(jobType,types);
    revisitTime = 1/managerPreferences(inTypes).RevisitRate + currentTime;
    predictedTrack = predictTracksToTime(tracker,trackID,revisitTime);

    xpred = getTrackPositions(predictedTrack,positionSelector);

    [hipred,thetapred,rpred] = cart2sph(xpred(1),xpred(2),xpred(3));
    job = struct('JobType',jobType,'Priority',managerPreferences(inTypes).Priority,...
        'BeamDirection',rad2deg([hipred thetapred]),'WaveformIndex',1,'Time',revisitTime,...
        'Range',rpred,'TrackID',trackID);

end

```

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 6-2 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 `fusionRadarSensor` 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.

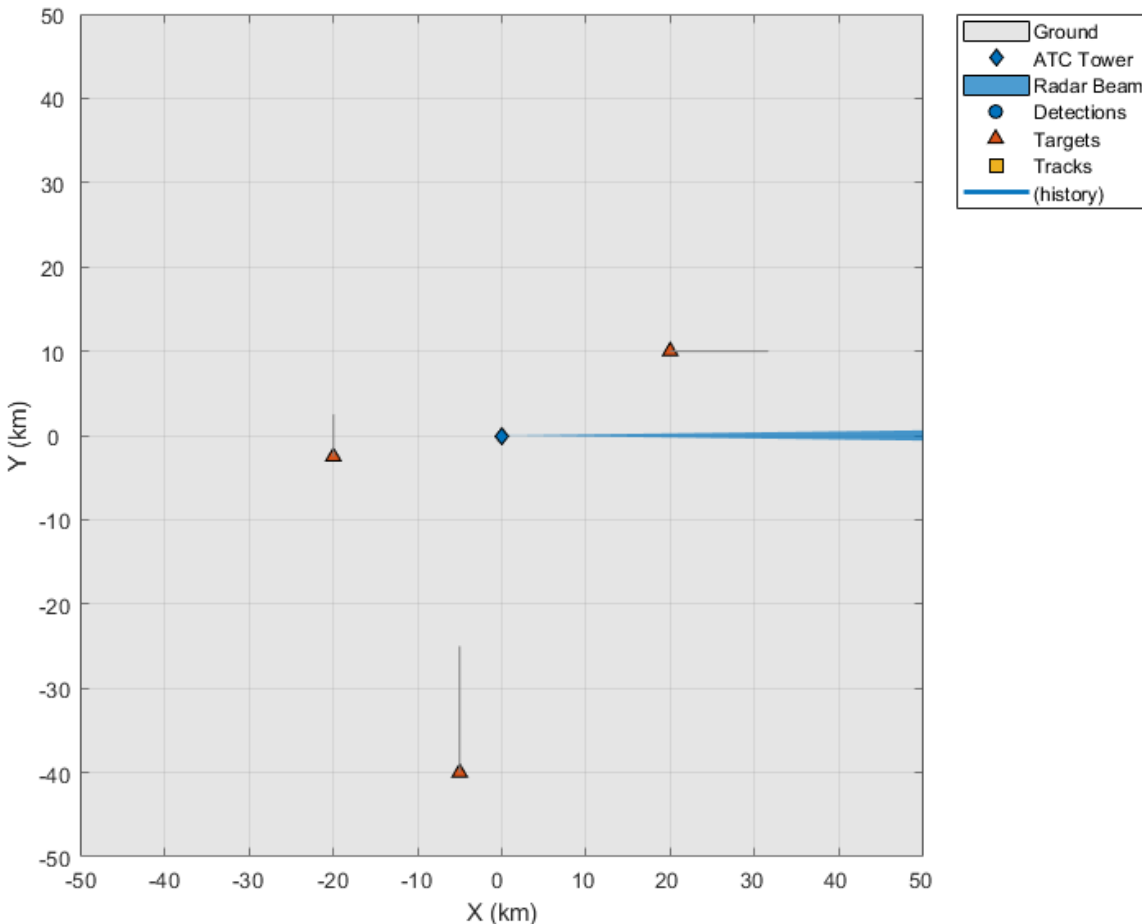
```
% If a previous tracker is defined, clear it.
clear tracker_kernel

% 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.
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)

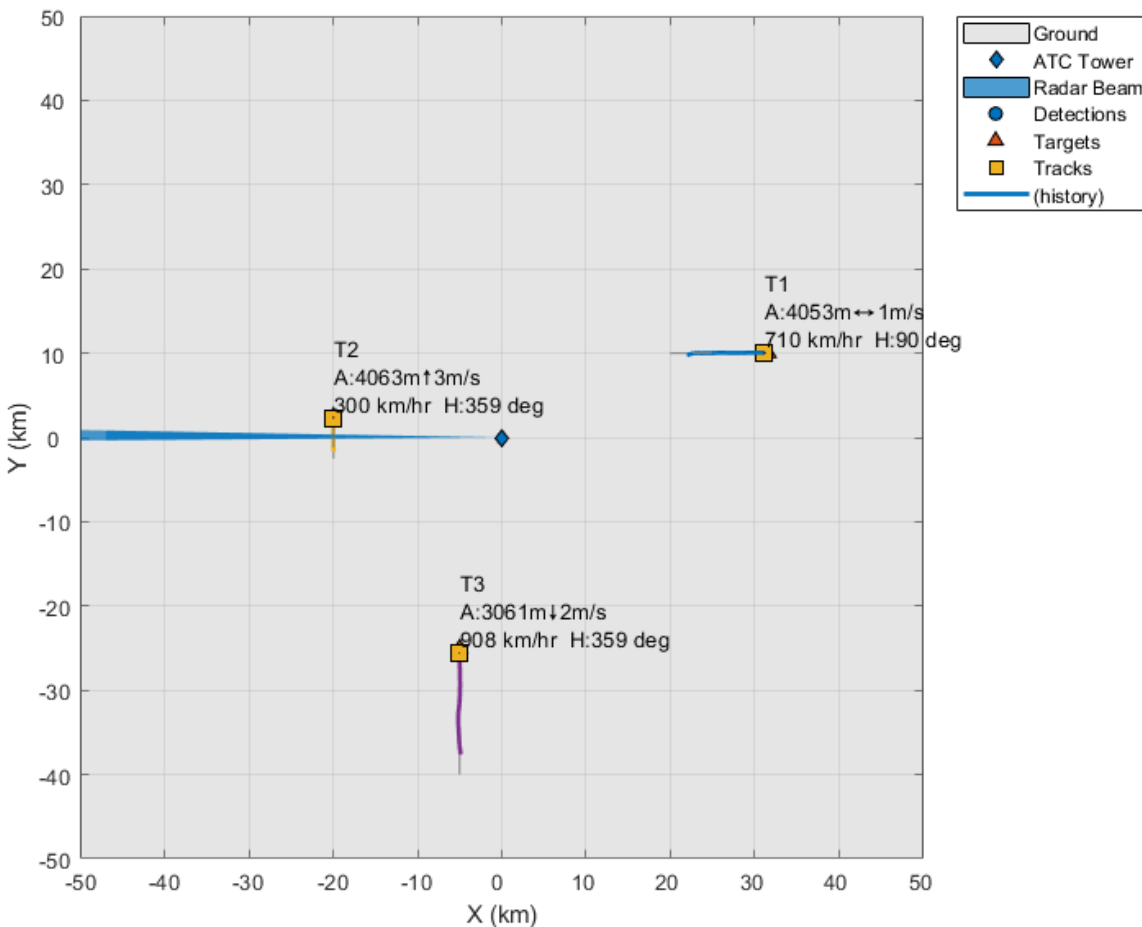
    % Generate detections on targets in the radar's current field of view.
    [dets,config] = detect(scenario);

    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 done');
% Generate code.
try
    codegen tracker_kernel -args compInputs;
    close(h)
catch ME
    close(h)
    throw(ME)
end
```

Code generation successful.

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)

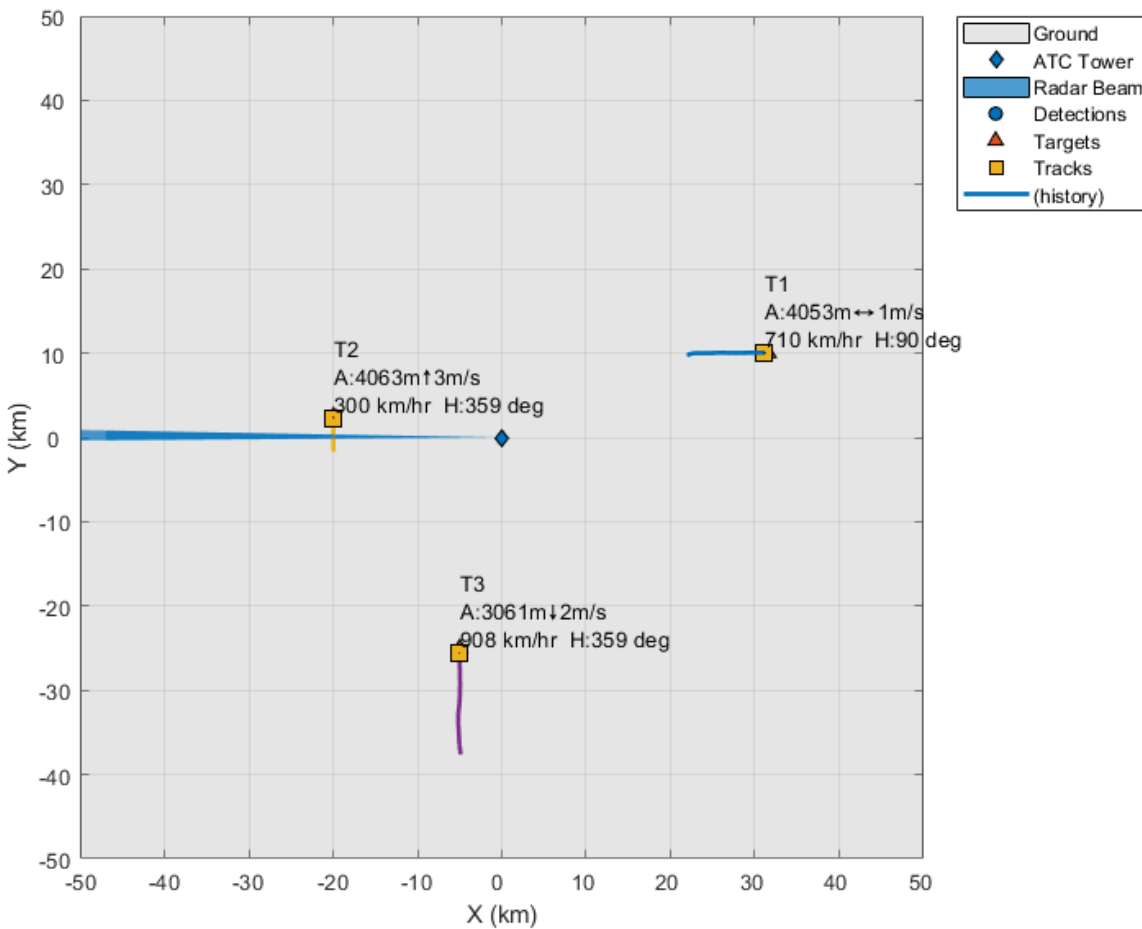
    % Generate detections on targets in the radar's current field of view.
    [dets, config] = detect(scenario);

    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.SimulationTime, scenario);
        runTimesMex(index) = toc; % Gather MEX run time data

        % Clear scan buffer for next scan.
        scanBuffer = {};
    end

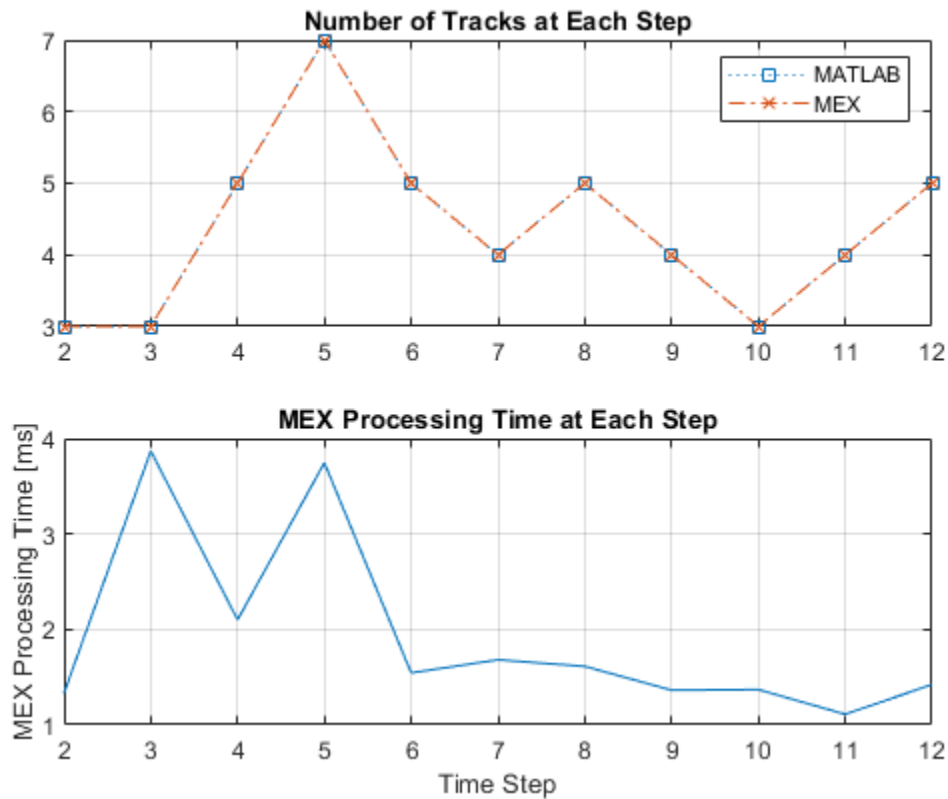
    % Update display with current beam position, buffered detections, and
    % track positions.
    helperTrackerCGExample('Update Display', theater, scenario, tower, scanBuffer, tracks);
end
```



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

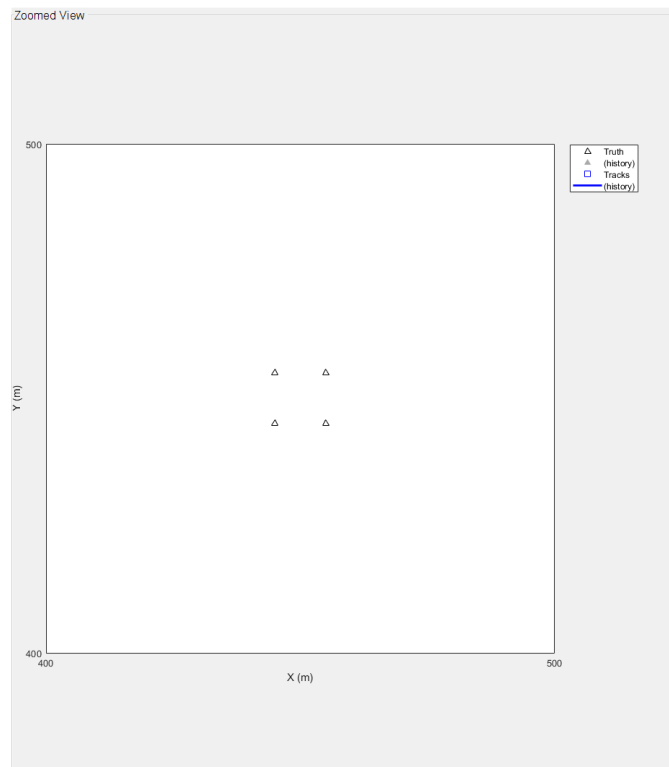
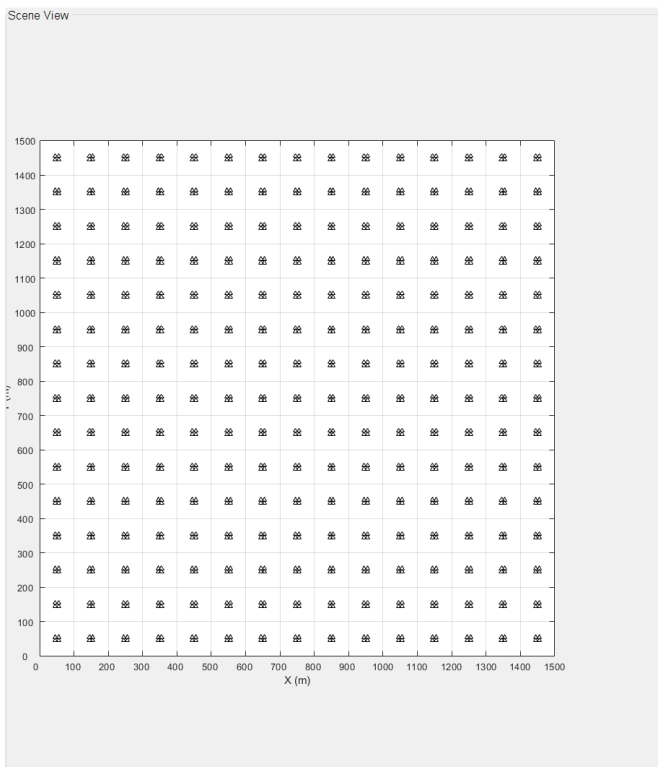
In many applications, trackers are required to track hundreds or thousands of objects. Increasing the number of tracks maintained by a tracker is a challenge, caused by the computational complexity of the algorithm at the core of every tracker. In particular, two common stages in the tracker update step 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.

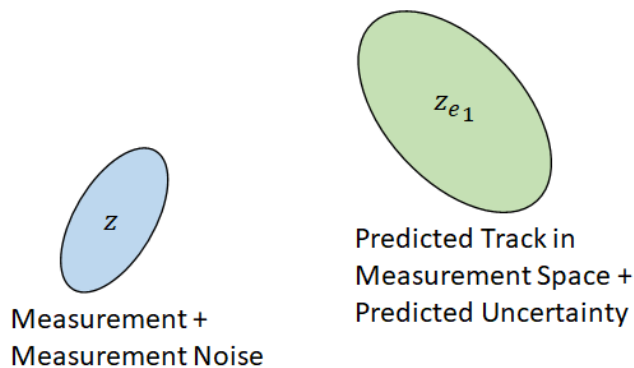
```
[platforms, tp, zoomedtp] = createPlatforms;
```



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: $d^2 \leq d_{max}^2$

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.

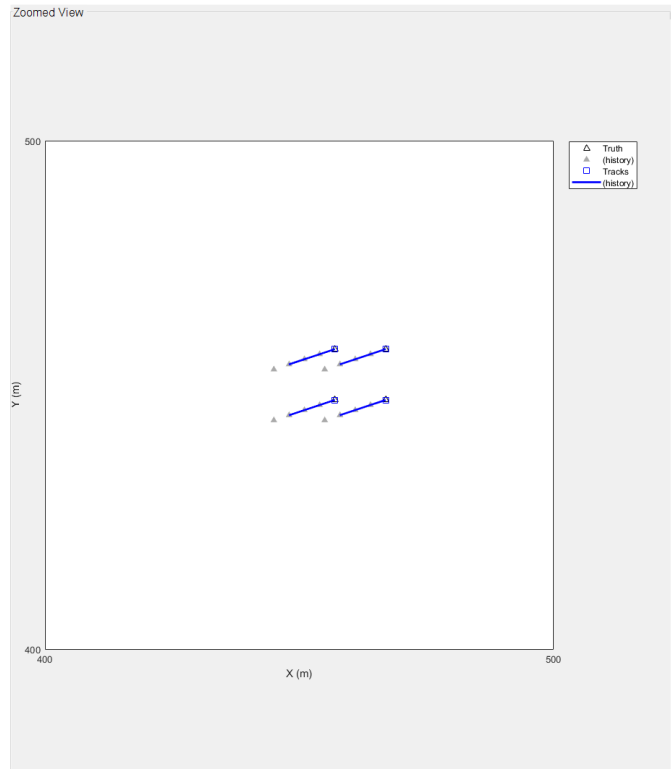
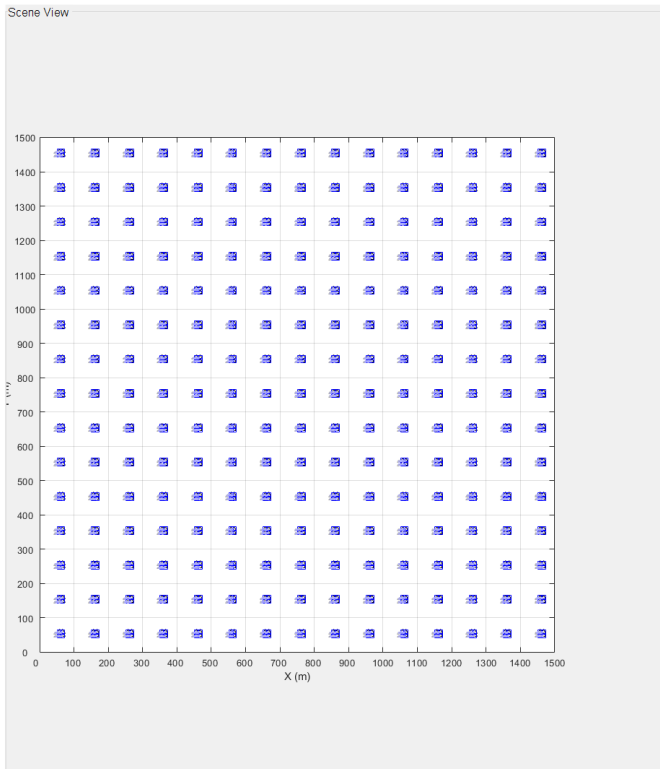
Define a tracker that can track up to 1000 tracks. The tracker uses the default constant velocity extended Kalman filter, and its state is defined as [x;vx;y;vy;z;vz], which is used by the `positionSelector` below to get the position components.

```
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.

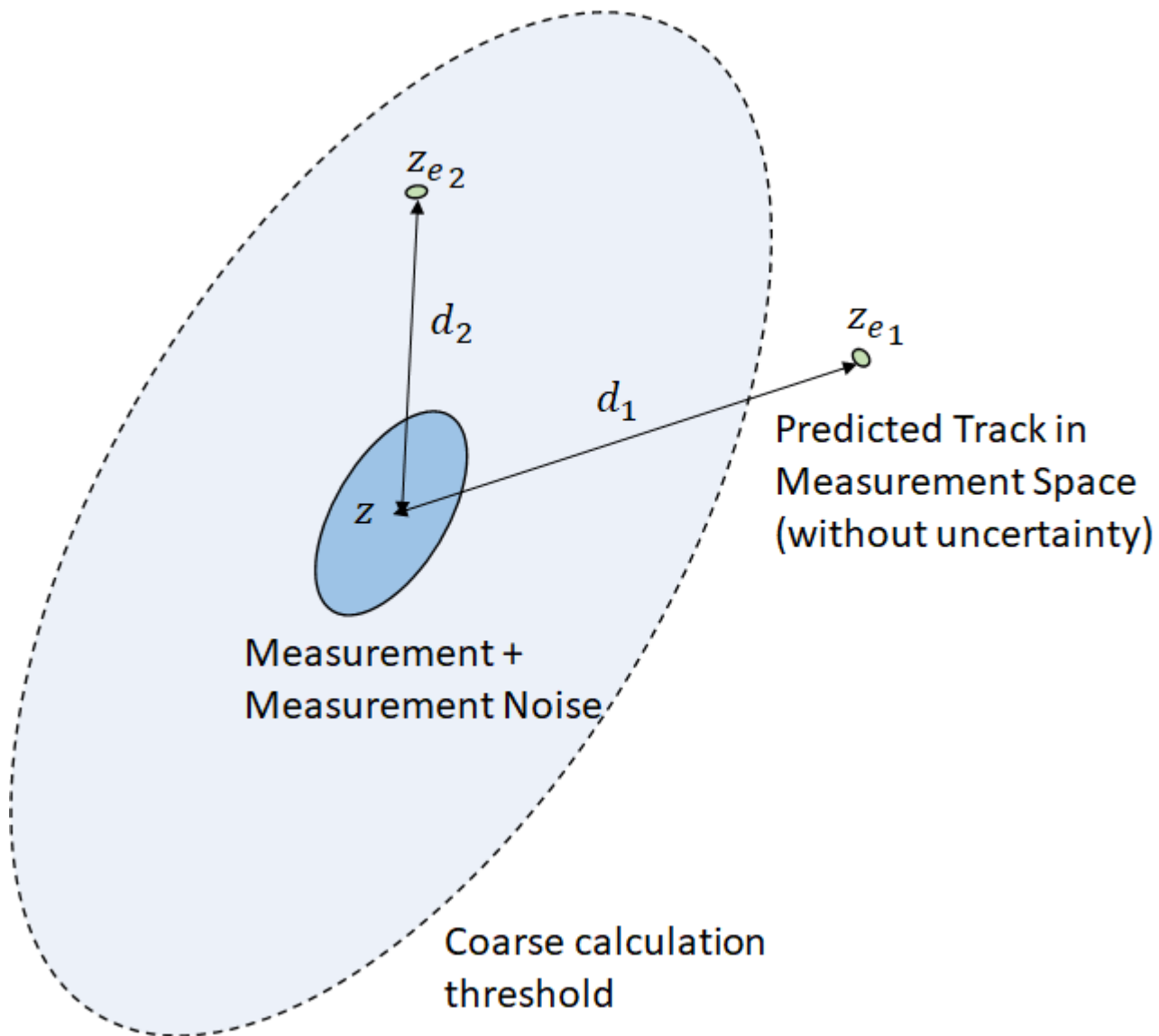
```
cm = info.CostMatrix;
disp("Cost matrix has " + numel(cm) + " elements.");
disp("But the number of finite values is " + numel(cm(isfinite(cm))) + newline)
```

```
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 C_2 are calculated accurately. The coarse cost calculation is depicted in the image below. A

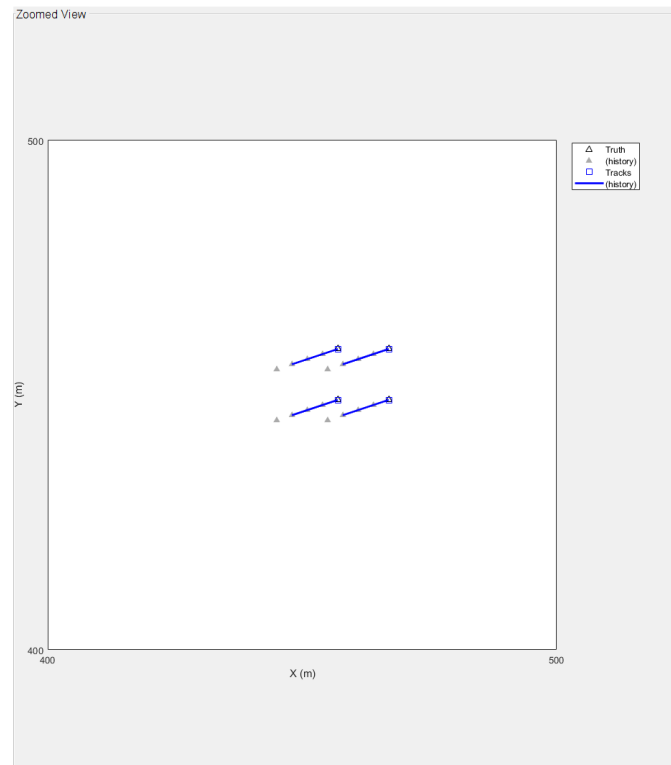
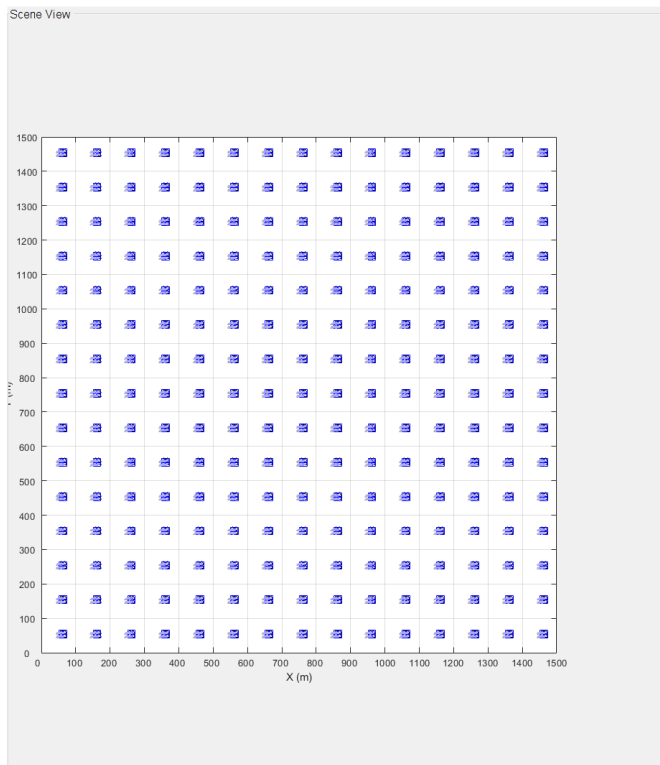
detection is represented by its measurement z and measurement noise R . Two tracks are predicted to the time of the detection and projected to the measurement space, depicted by the points ze_1 and ze_2 . Note that the track uncertainty is not projected to the measurement space, which allows us to vectorize 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      MaxDivergenceLength
```

Truth assignment metrics summary:				
TotalNumTruths	NumMissingTruths	MaxEstablishmentLength	MaxBreakCount	
900	0	0	0	0
900	0	1	0	

Use an External Cost Calculation

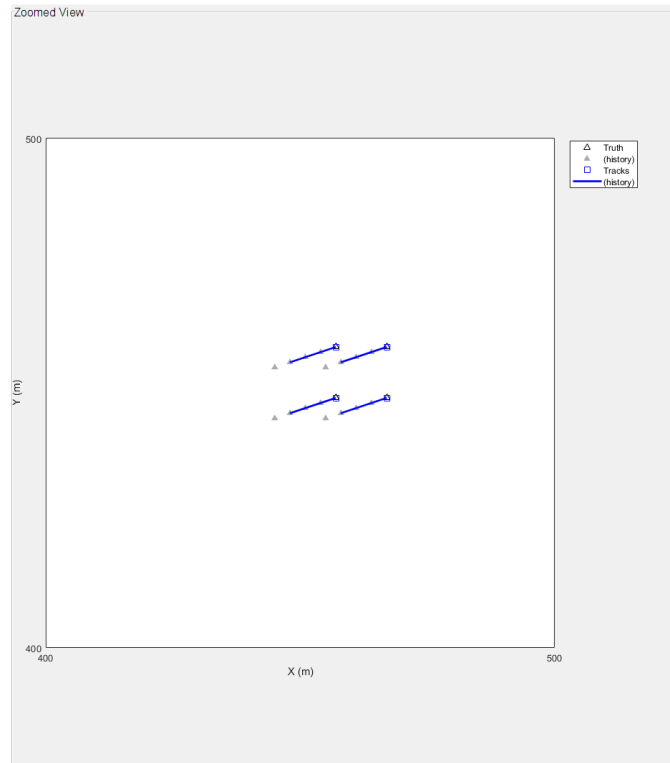
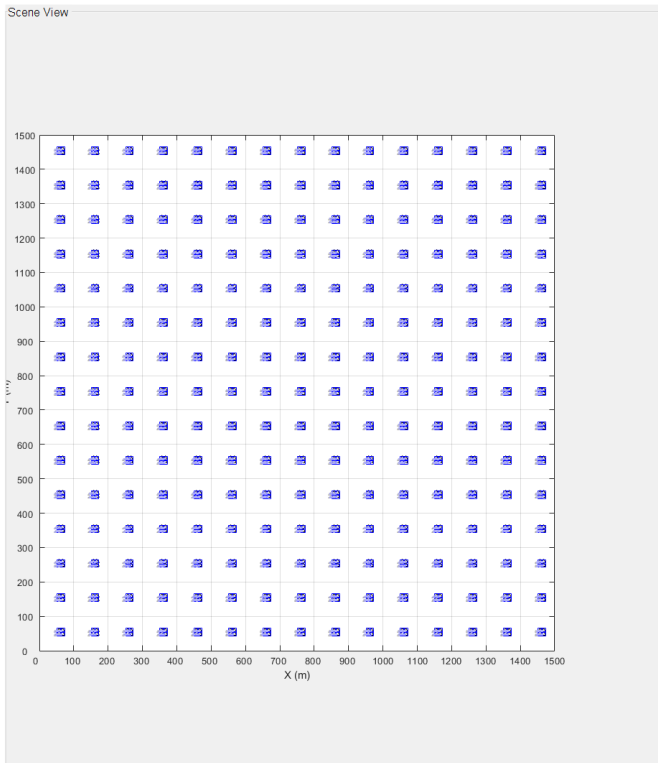
Another way to control the time it takes to calculate the cost assignment is by using your own assignment cost calculation instead of the default the tracker uses.

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

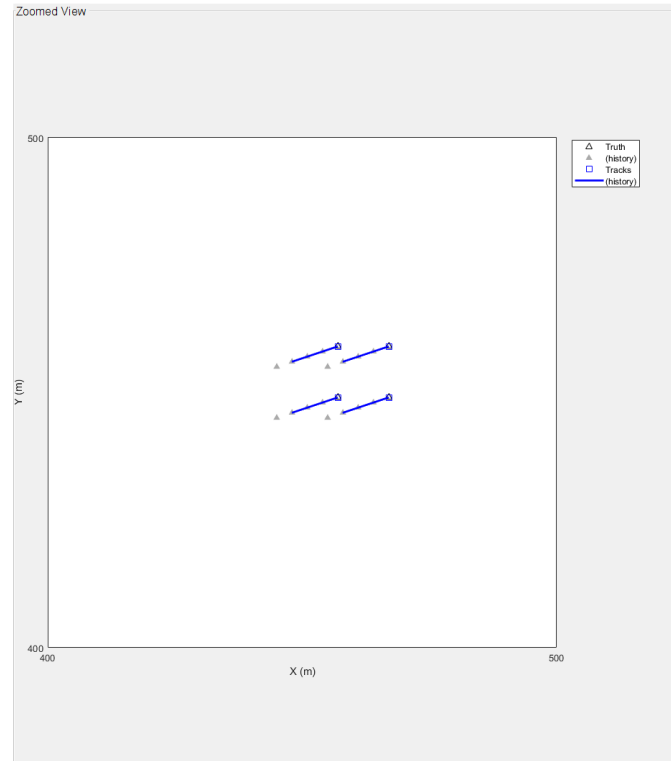
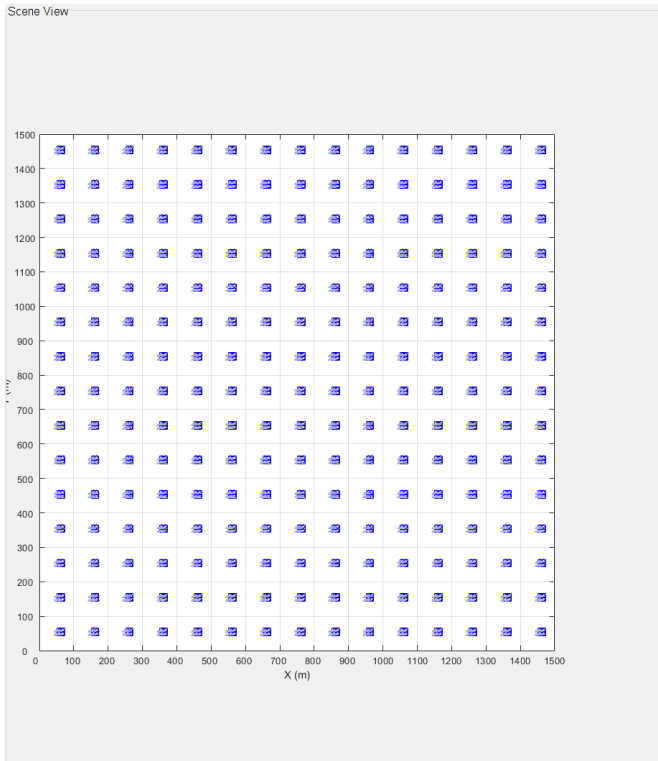
Another option to try is using a different GNN assignment algorithm that may be more efficient in finding the assignment by modifying the Assignment property of the tracker.

```
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.
```

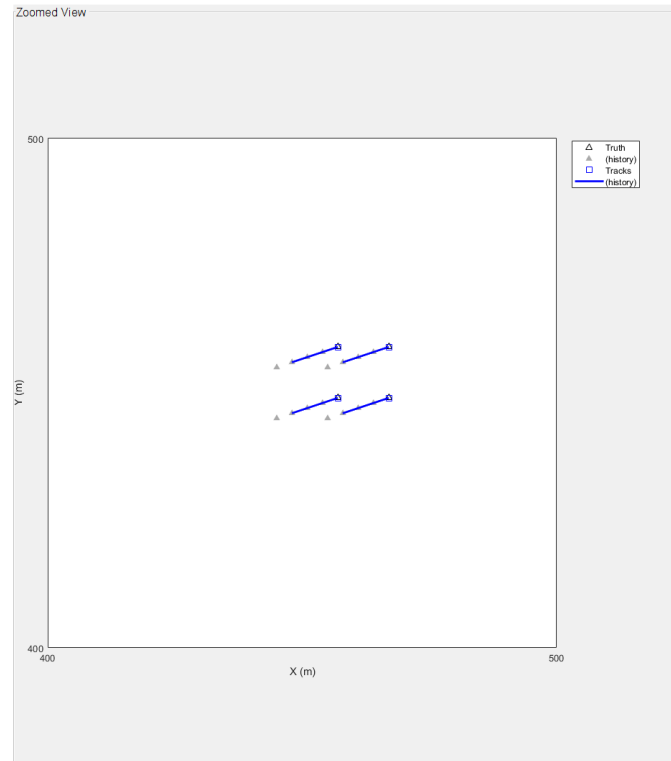
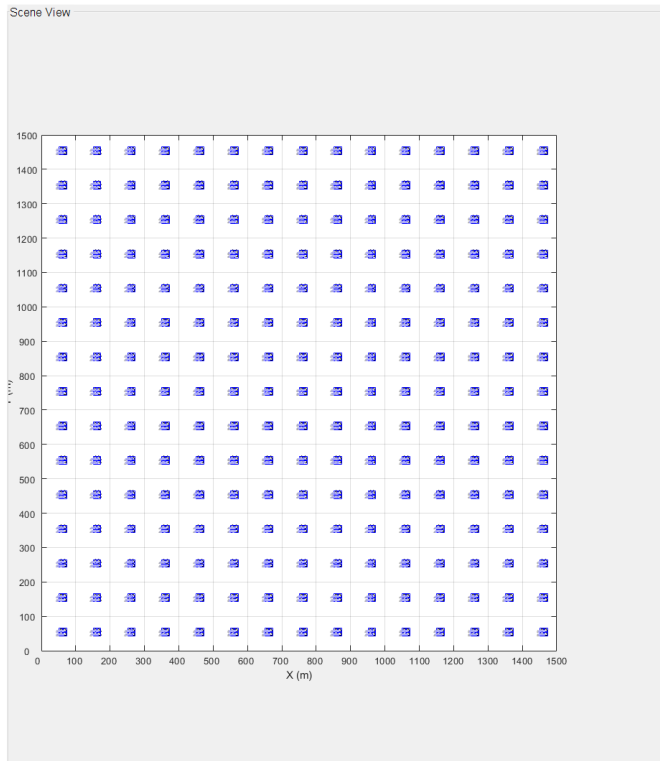
The Jonker-Volgenant algorithm performs the assignment in the second step faster relative to the default Munkres algorithm.

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.

```
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, 'HistoryDepth', 10);
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
end
```

```

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', 'MarkerSize', 8, 'HistoryDepth', 10);
plotTrack(zoomedpp, reshape([platforms.Position], 3, []));
trackPlotter(zoomedtp, 'DisplayName', 'Tracks', 'MarkerEdgeColor', 'b', 'MarkerSize', 8, 'HistoryDepth', 10);
end

```

runTracker

This function runs the tracker, updates the plotters, and gathers track assignment metrics.

```

function [trkSummary,truSummary,info] = runTracker(platforms,tracker,positionSelector,tp,zoomedtp)
% 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

```

```

    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
end

```

```

    % 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.

```

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.

```

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, `tTrackerGNN`, 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.

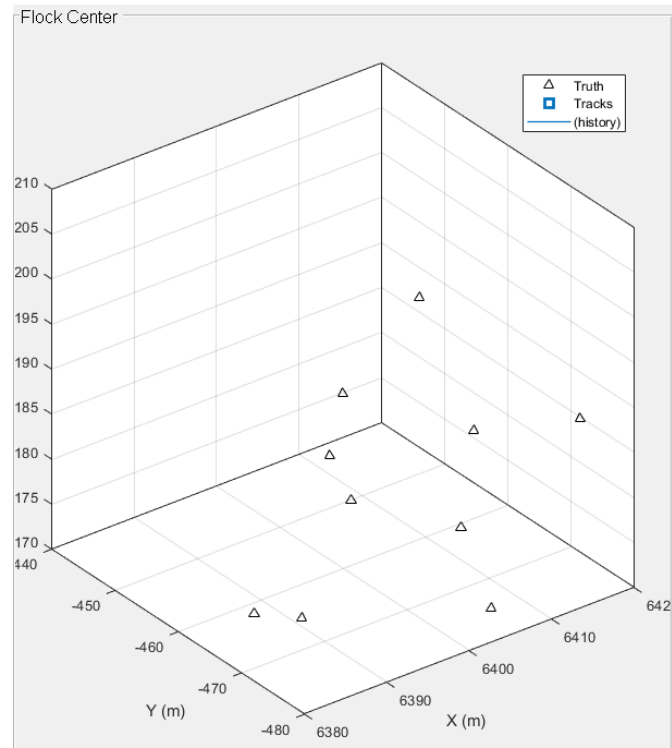
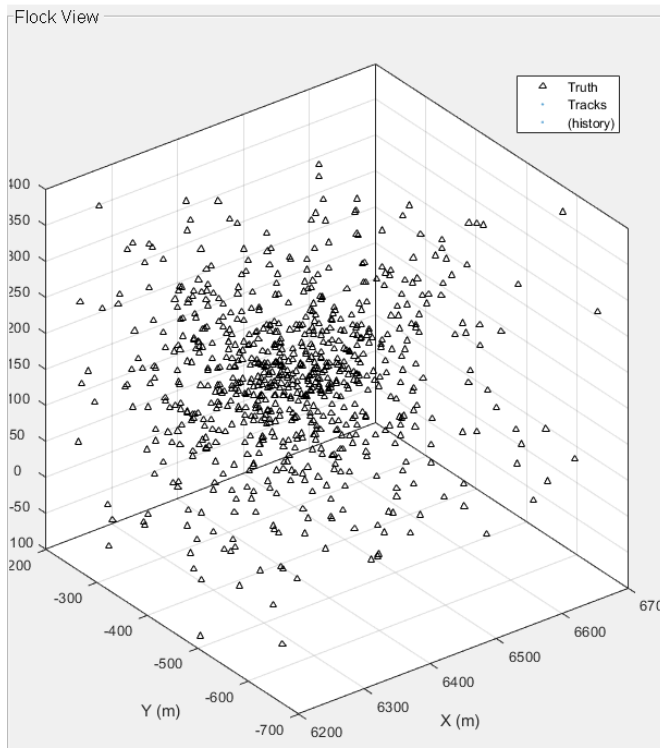
Tracking such a large and dense flock presents two challenges:

- 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), "CollisionAvidanceWeight", 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 6-303.

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 6-296, 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

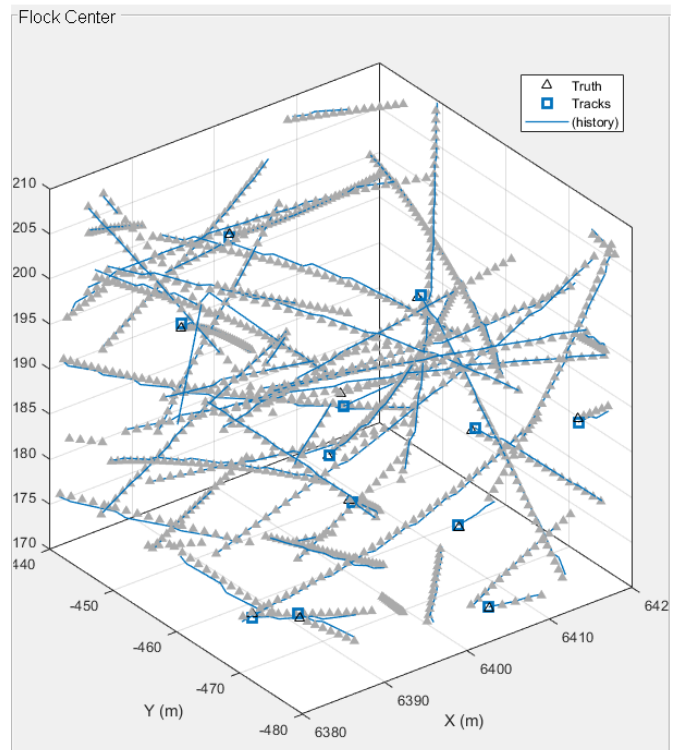
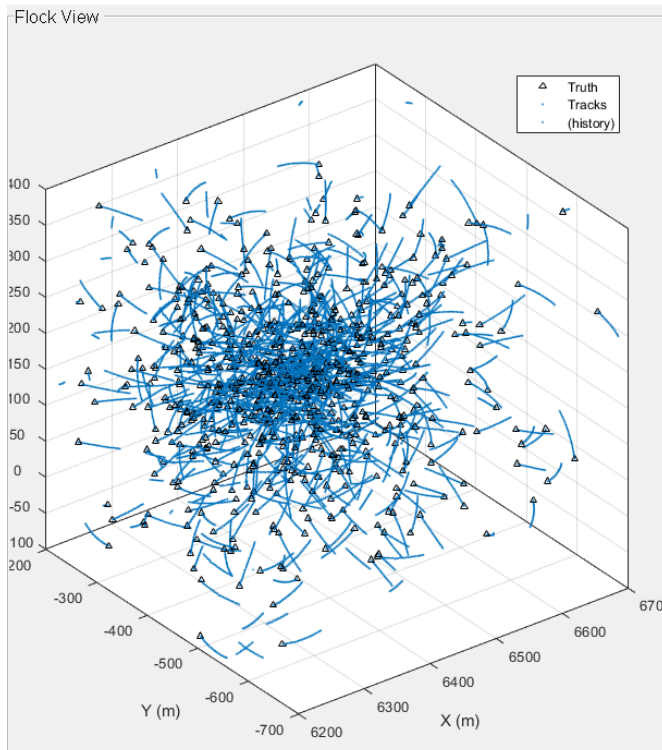
Next, you run the scenario and track the flock.

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.

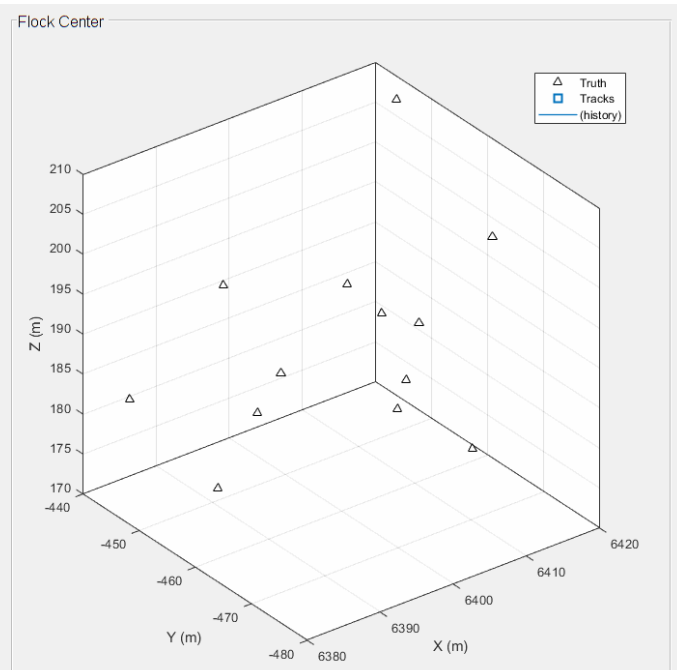
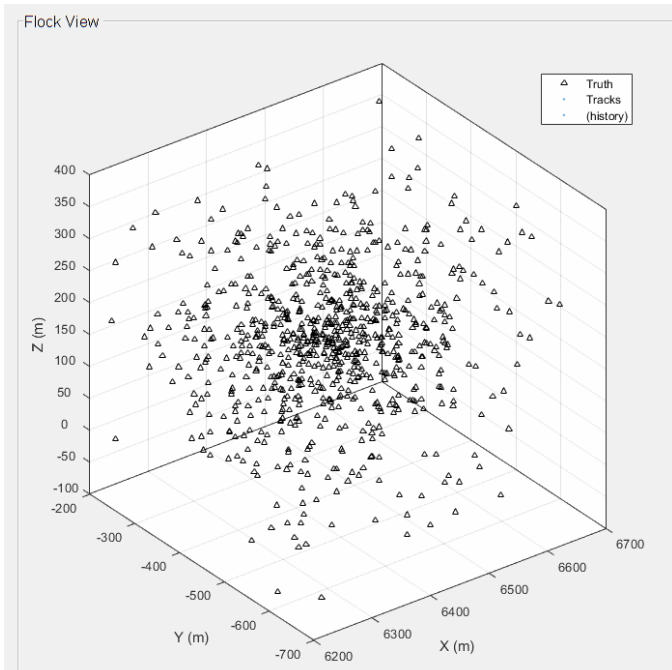
```
clear flockTracker_kernel
positionSelector = [1 0 0 0 0 0 0; 0 0 1 0 0 0 0; 0 0 0 0 0 1 0];
trackIDs = zeros(0,1,'uint32');
trax = zeros(0,3);
bounds = inf(3,2);
alltrax = zeros(size(allx));
allIDs = repmat({},1,numSteps);
trup2 = tp2.Plotters(1);
trap2 = tp2.Plotters(2);
trup2.HistoryDepth = 2*trap2.HistoryDepth;
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
```

6 Featured Examples



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,"ConnectHistory",...
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);

```

```

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

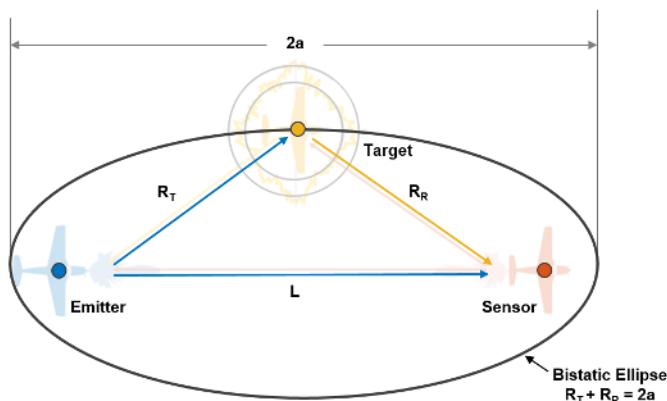
This example shows you how to simulate bistatic range-only detections using four sensor-emitter pairs. In addition, this example demonstrates how to localize and track multiple targets using bistatic range-only measurements.

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 = fusionRadarSensor(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.
```

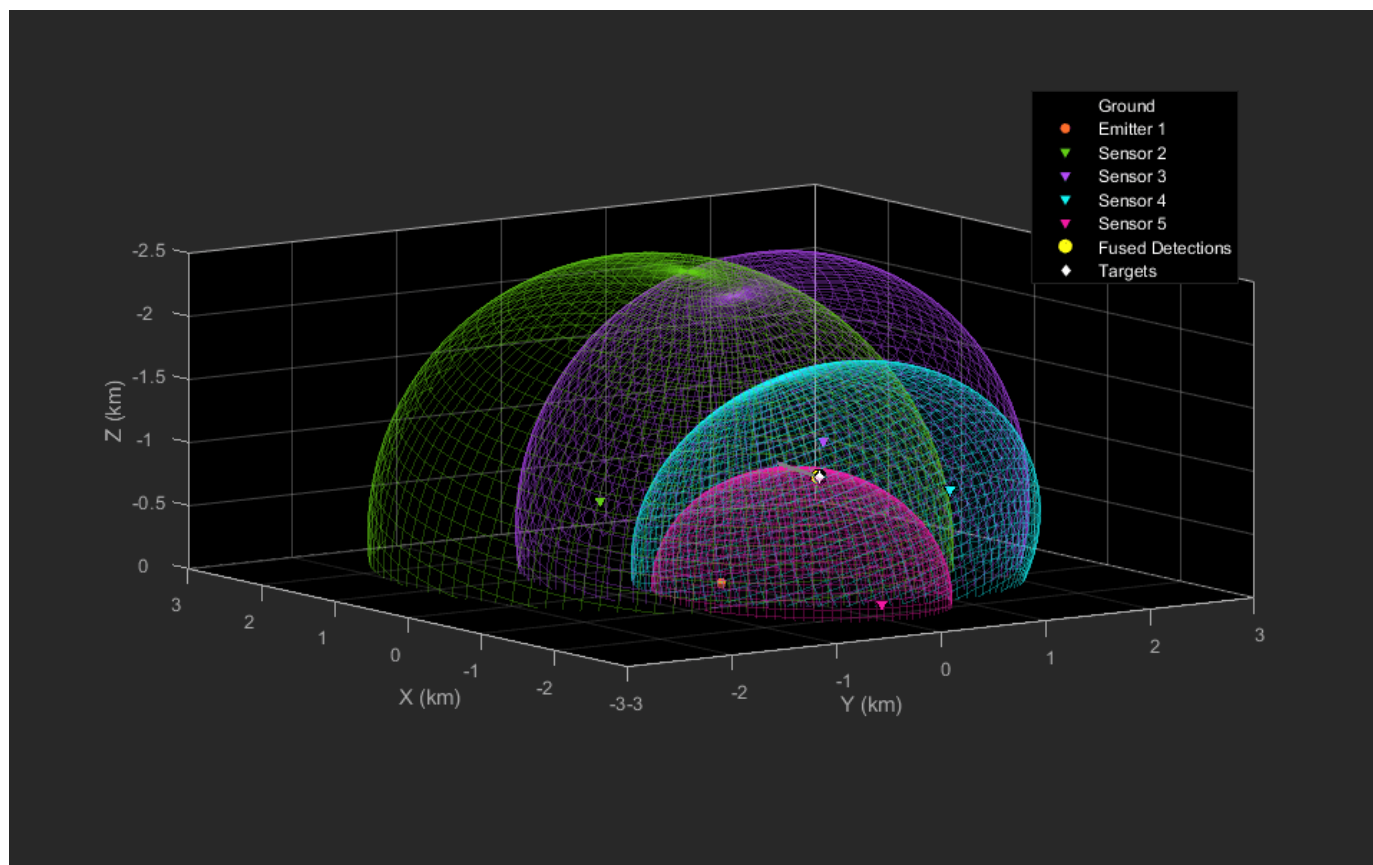
```
detections = detectBistaticTargetRange(scene,time,emitterIdx,true);

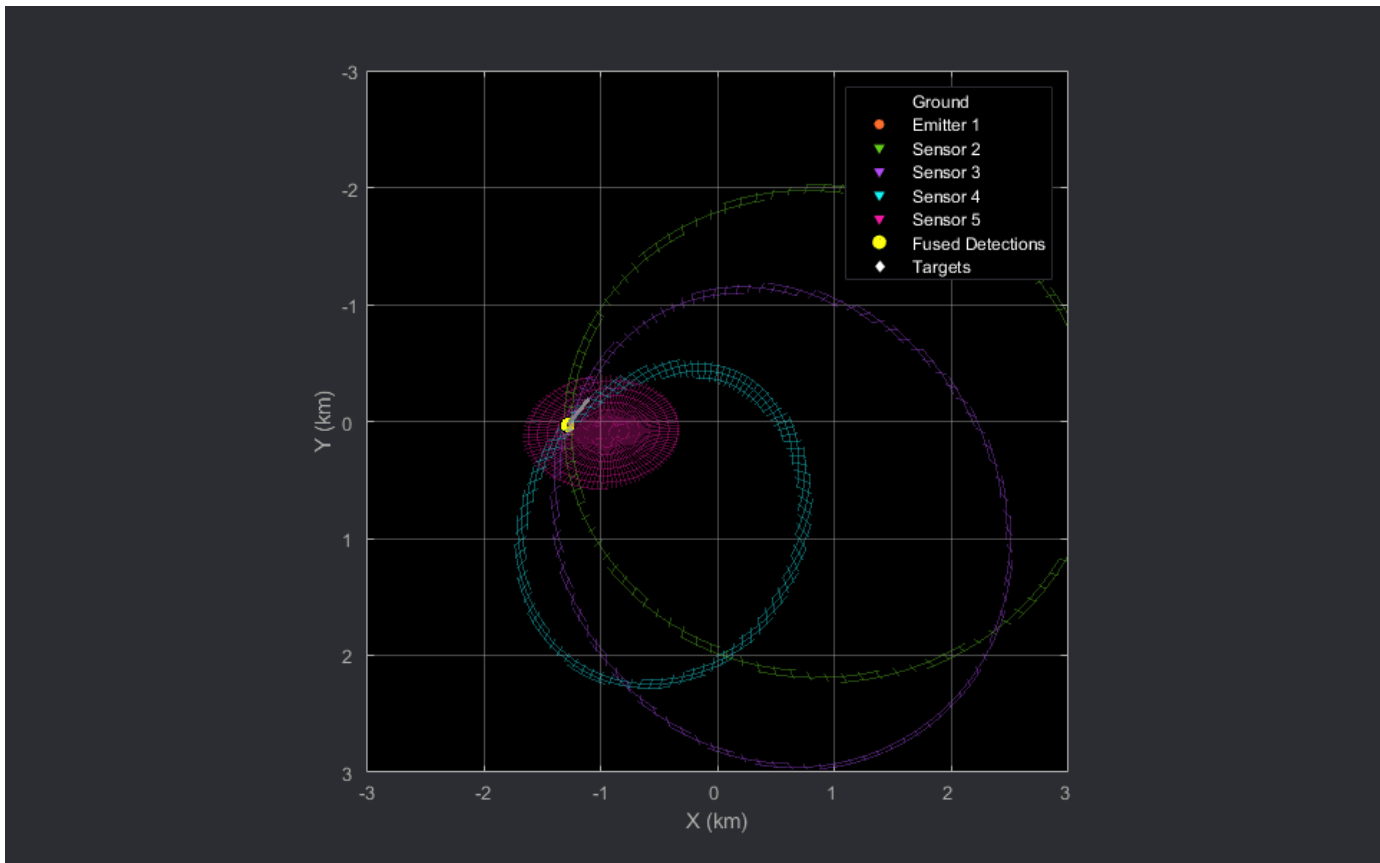
% Triangulate detections to estimate target position.
[position, covariance] = helperBistaticRangeFusion(detections);

% 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 by 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);
```

```
% Reproducible geometry
rng(2021);

% 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;

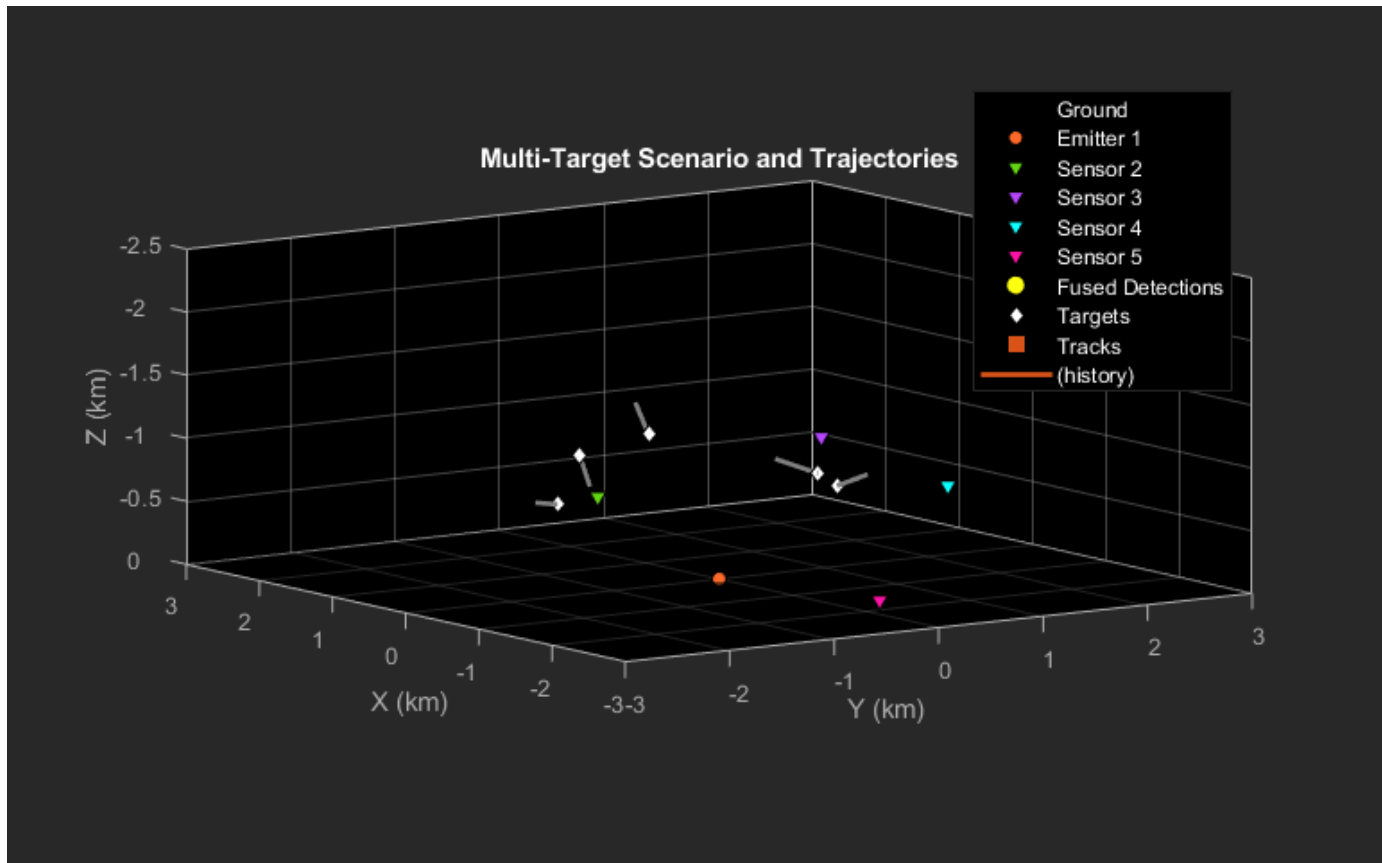
% Reset the display.
release(theaterDisplay);

% 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 bistatic
    'MeasurementFcn','helperBistaticMeas',... % Set measurement function for reporting bistatic
    'DetectionProbability',0.99 ... % Probability of detecting the target
);

% Define a GNN tracker.
tracker = trackerGNN('AssignmentThreshold',100);
```

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

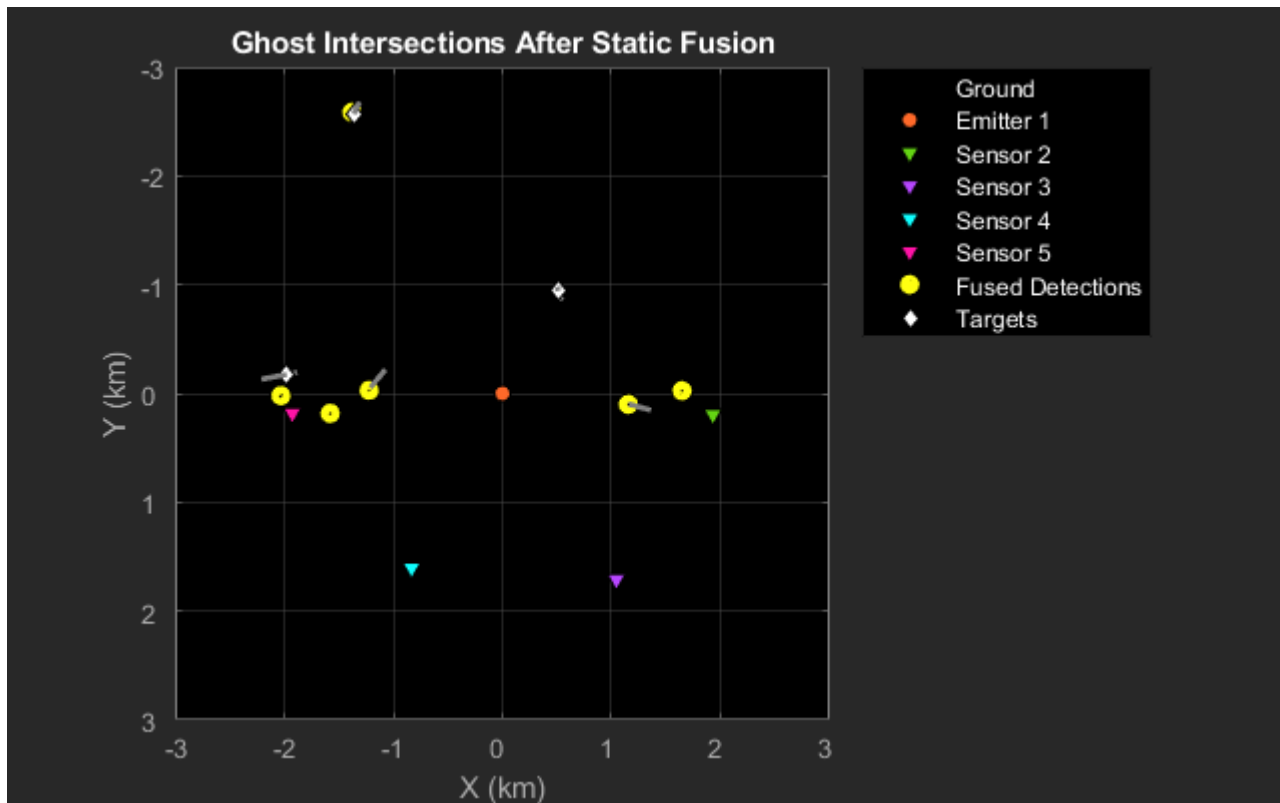
```

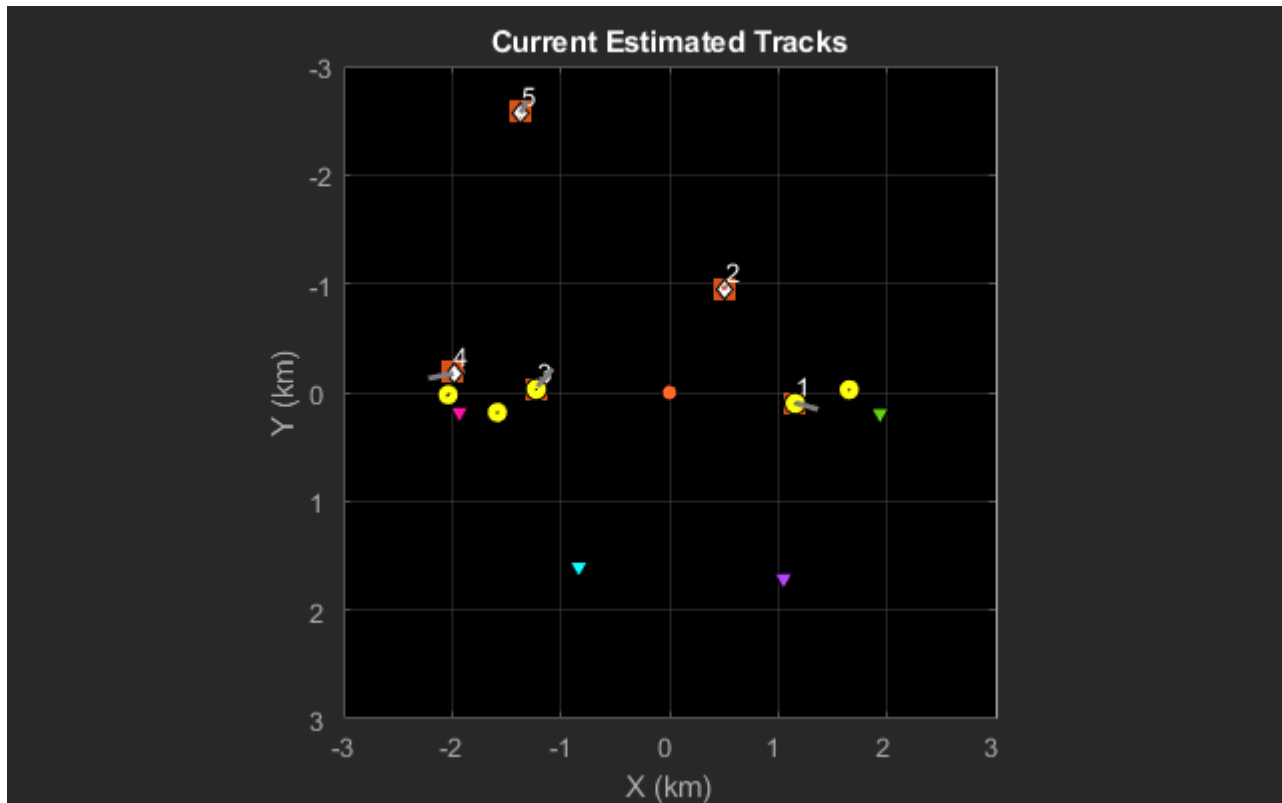
Results

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.

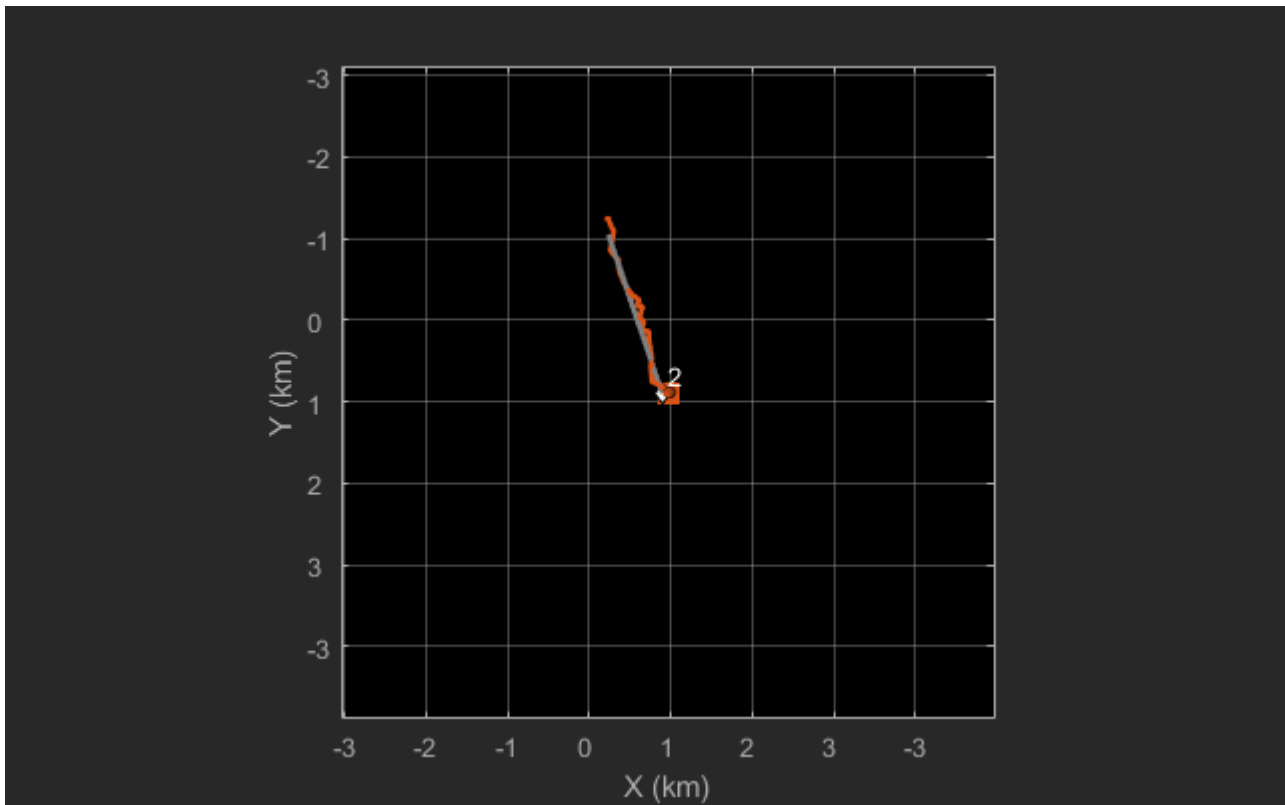
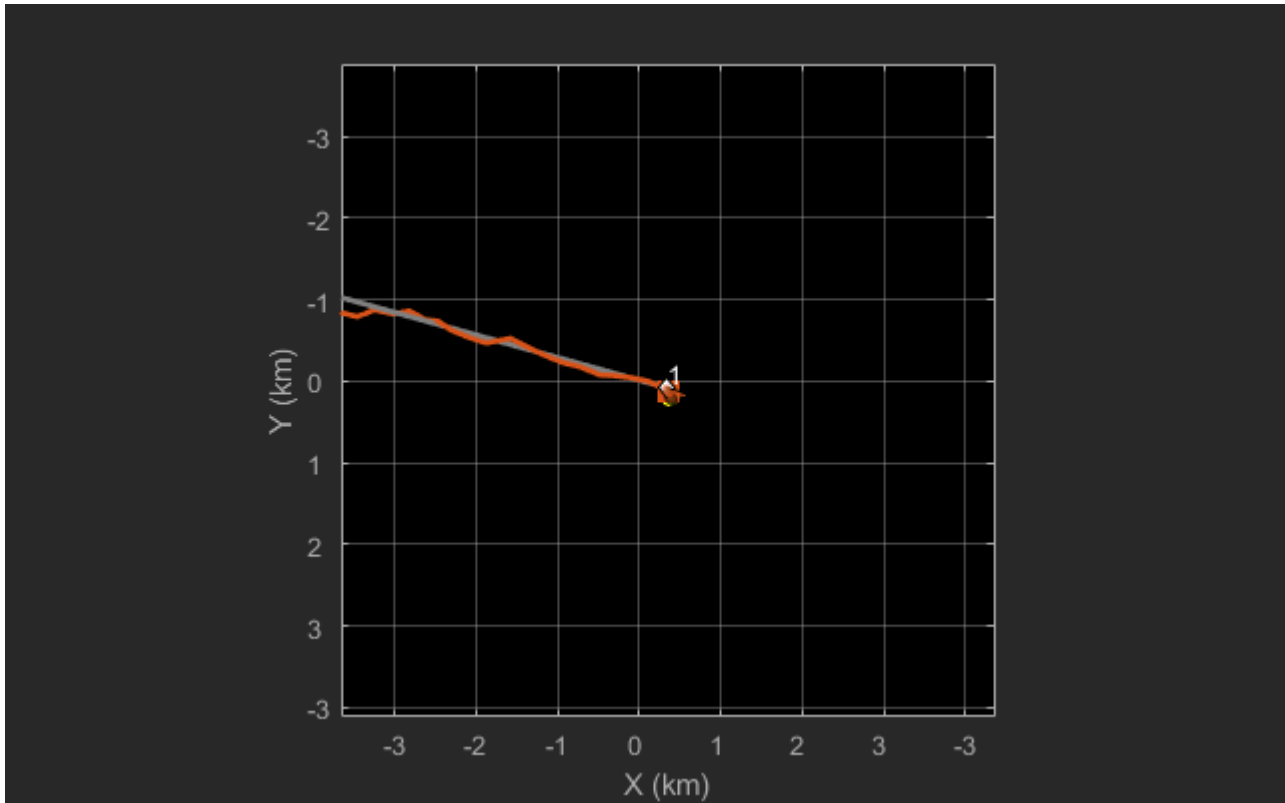
```
showGrabs(theaterDisplay,1);
```





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 measurement 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 nargin > 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 measur
    detections{iD}.MeasurementNoise = detections{iD}.MeasurementNoise(end,end); %#ok<AGROW> %
    detections{iD}.MeasurementParameters(1).HasAzimuth = false; %#ok<AGROW> % Update measure
    detections{iD}.MeasurementParameters(1).HasElevation = false; %#ok<AGROW> % Update measur
    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
    posscope = 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

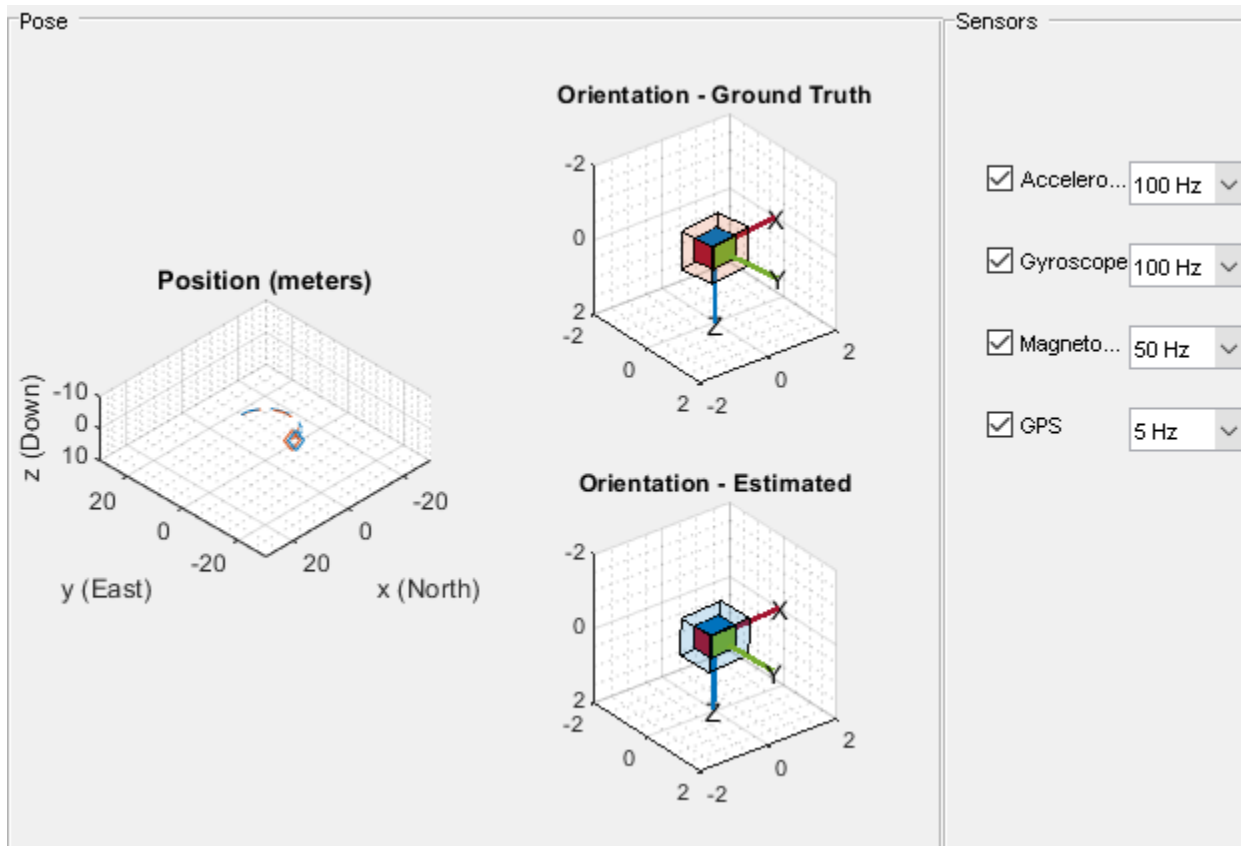
        fusionfilt.fusemag(mag(ii,:), Rmag);
    end

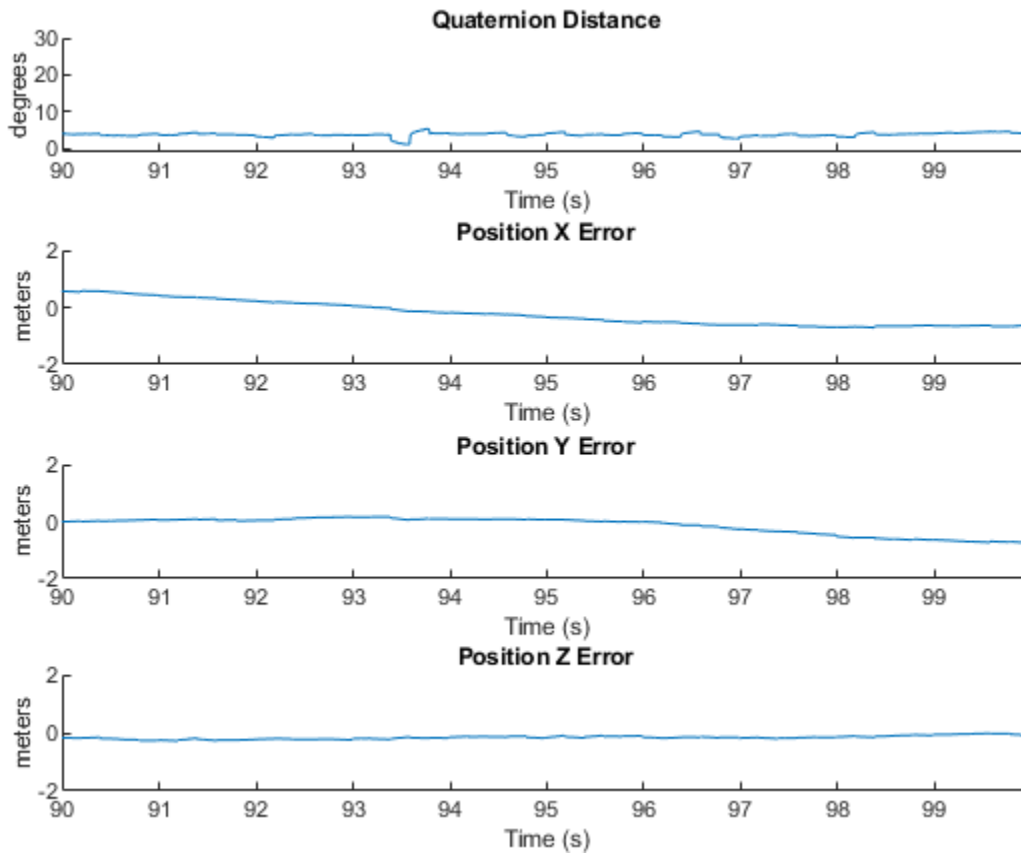
    % Fuse GPS
    if (f.UserData.GPS) && mod(ii, fix(Fs/f.UserData.GPSSampleRate)) == 0
        fusionfilt.fusegps(lla(ii,:), Rpos, gpsvel(ii,:), Rvel);
    end

    % 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,:);
    errslope(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.

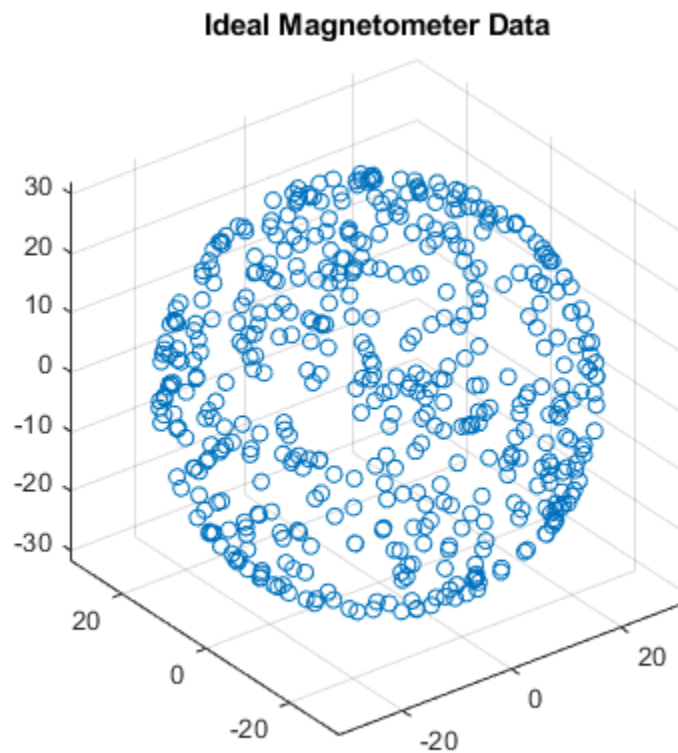
In order to be useful for heading and orientation computation, typical low cost MEMS magnetometers need to be calibrated to compensate for environmental noise and manufacturing defects.

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');
```

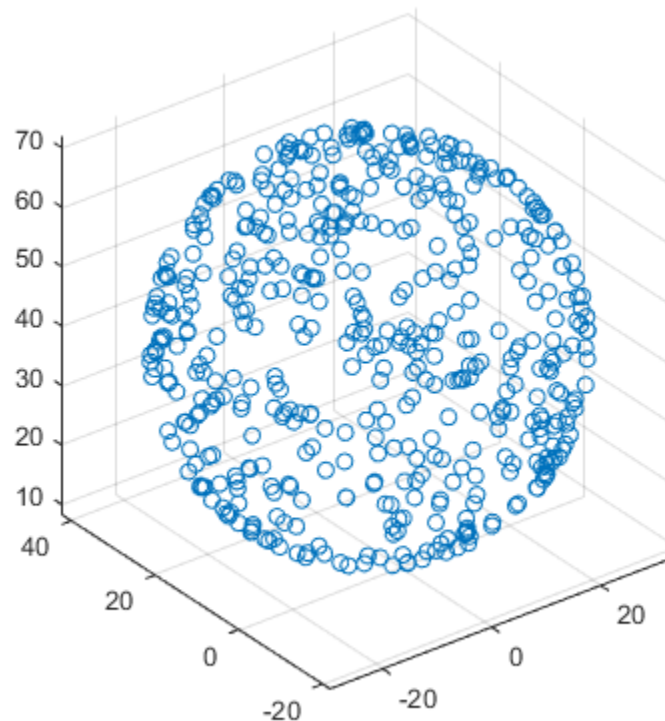


Hard Iron Effects

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

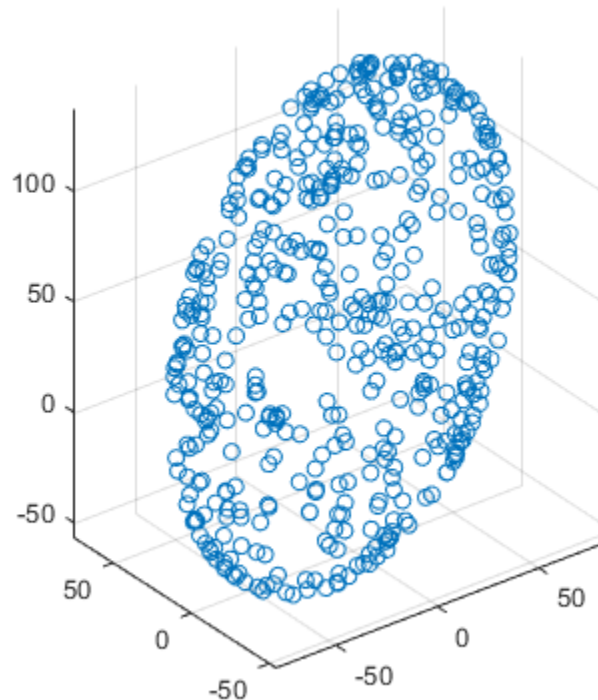
Soft iron effects are more subtle. They arise from objects near the sensor which distort the surrounding magnetic field. These have the effect of stretching and tilting the sphere of ideal measurements. The resulting measurements lie on an ellipsoid.

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.

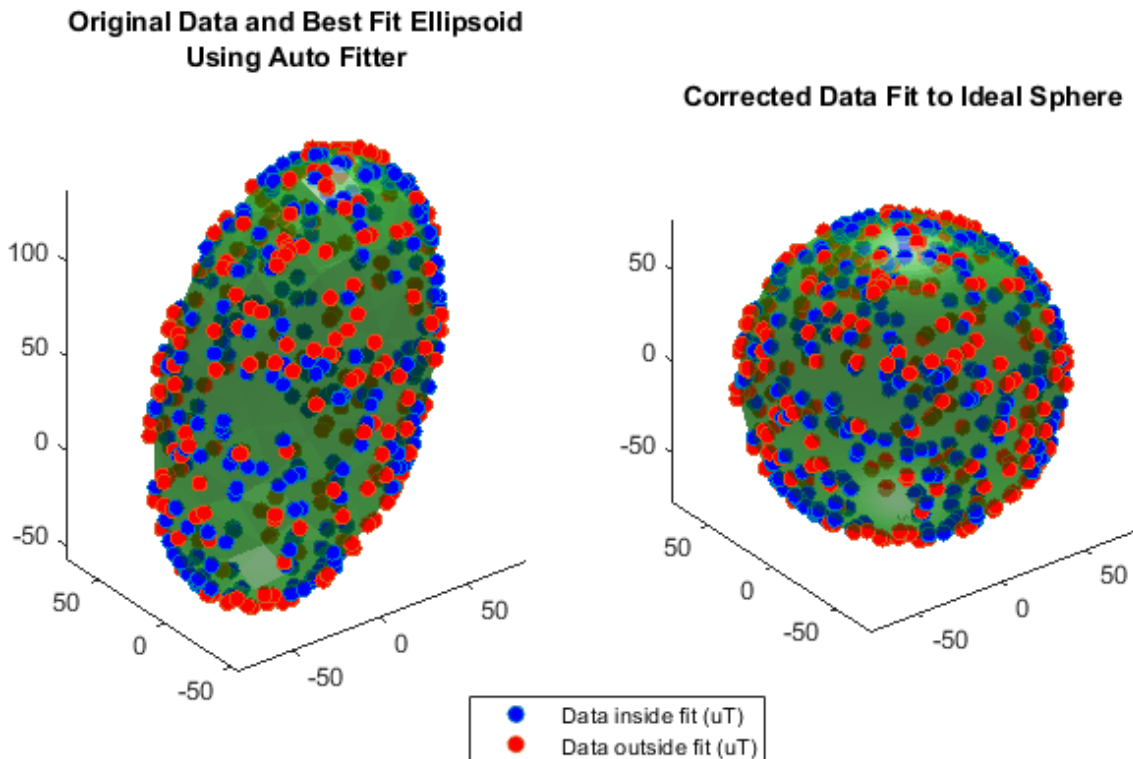
```
imu.Magnetometer.NoiseDensity = 0.08;
for ii=1:numel(q)
    imu.MagneticField = soft(ii,:);
    [~,x(ii,:)] = imu(acc(ii,:),av(ii,:),q(ii));
end
```

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');
```



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
```

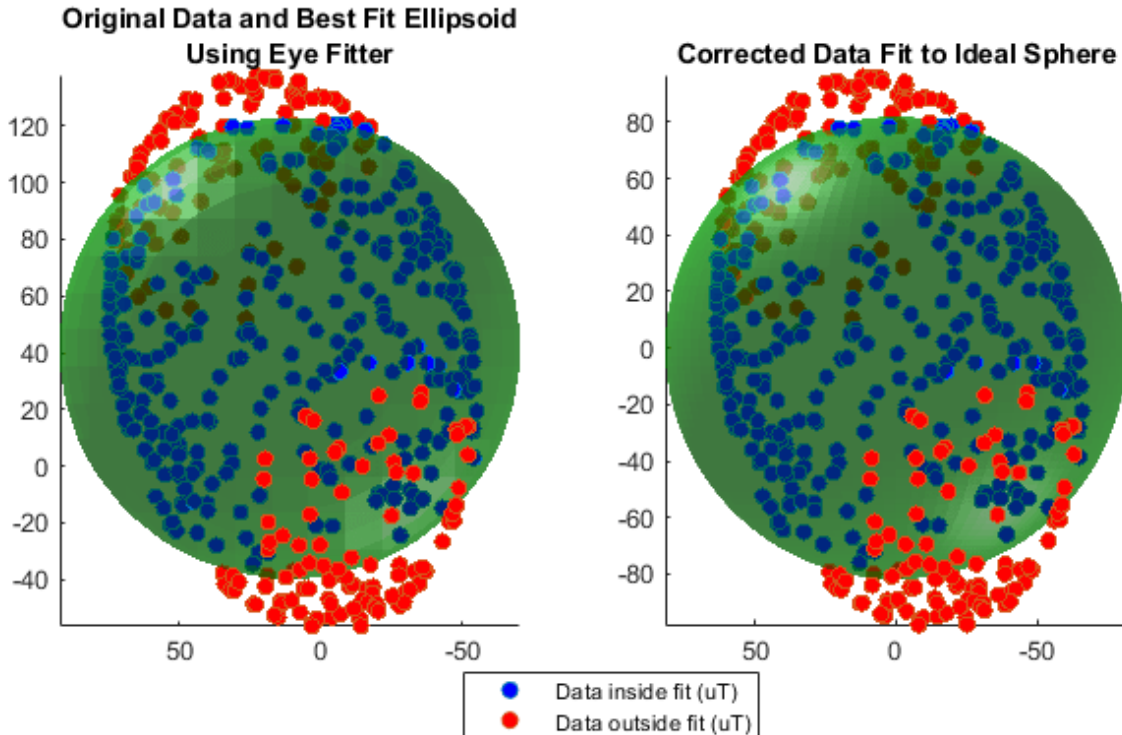
You can run the individual solvers if only some defects need to be corrected or to achieve a simpler correction computation.

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');
```

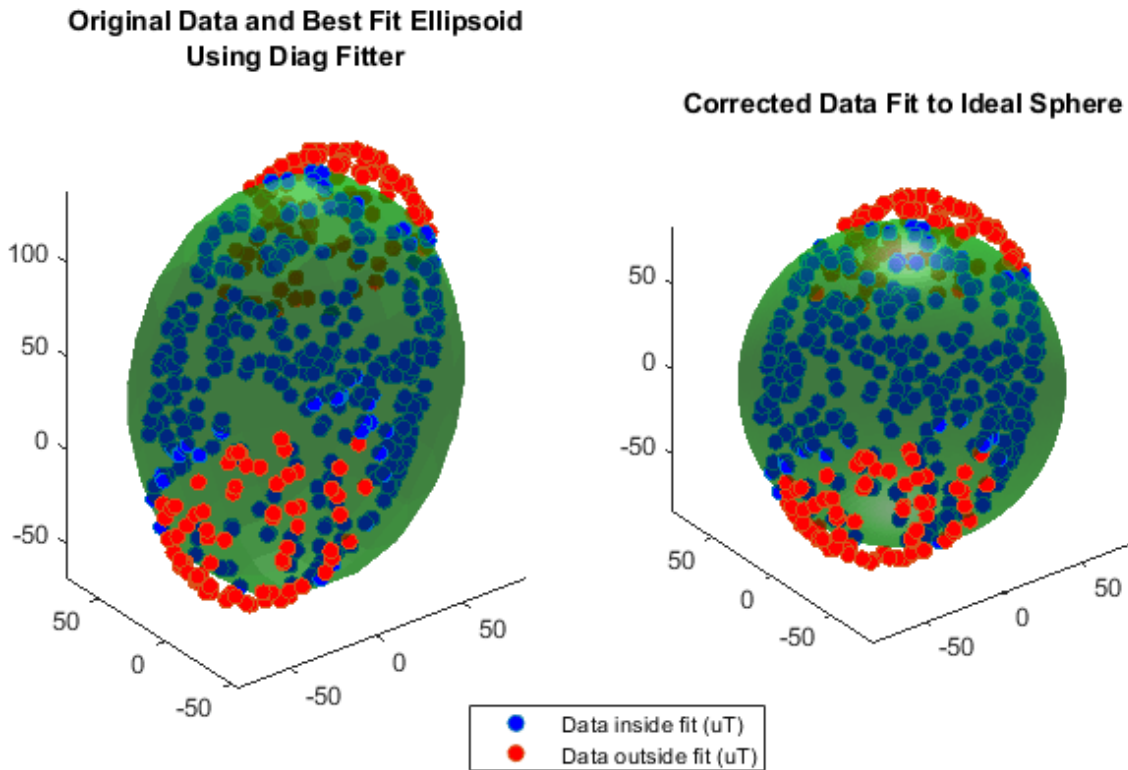
```
view(ax1,[-1 0 0]);
view(ax2,[-1 0 0]);
```



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');
```



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,mfspoer] = 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
```

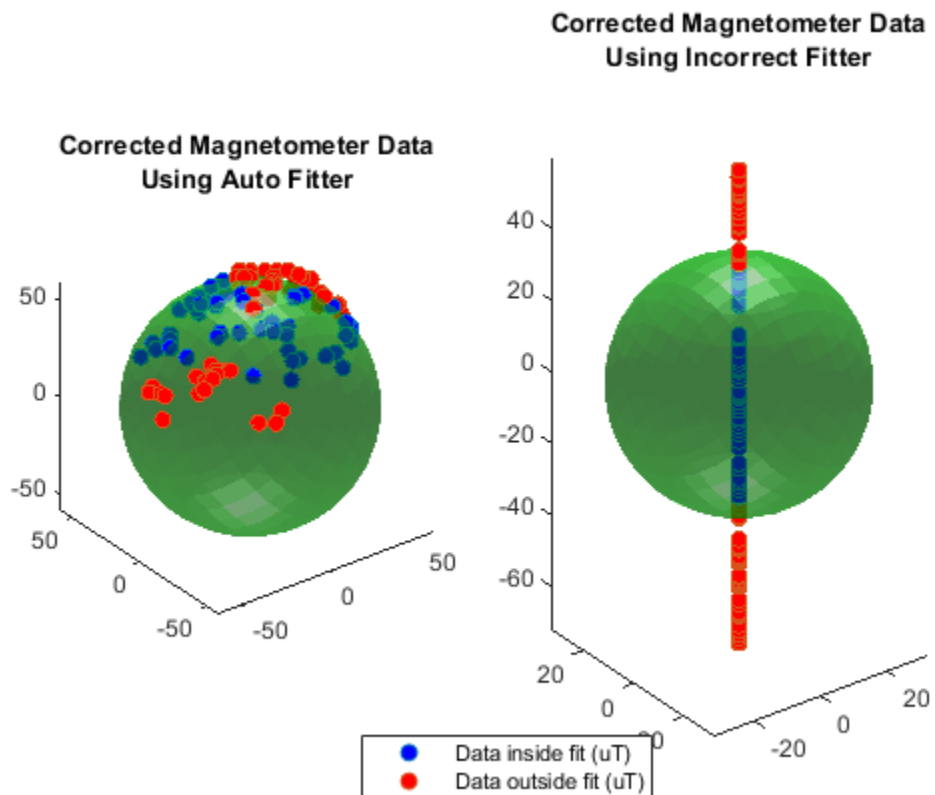

Using the 'auto' fit option avoids this problem and finds a simpler A matrix which is real, symmetric, and positive definite. Calling magcal with the 'auto' option string is the same as calling without any option string.

```
[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,mfspoer,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 examples shows 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 6-383. 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

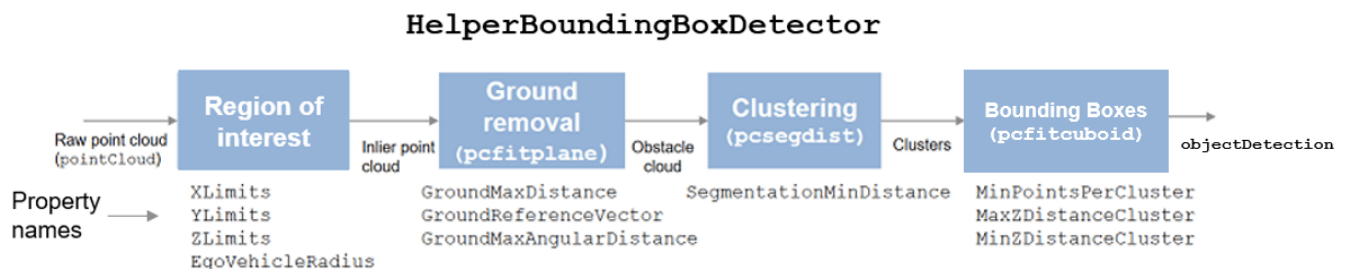
Due to high resolution capabilities of the lidar sensor, each scan from the sensor contains a large number of points, commonly known as a point cloud. This raw data must be preprocessed to extract objects of interest, such as cars, cyclists, and pedestrians. In this example, you use a classical segmentation 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/TrackVehiclesUsingLidarExampleData.zip';
    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:

$$\mathbf{x} = [x_{kin} \ \theta \ l \ w \ h]$$

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, and height of the cuboid are modeled as 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:

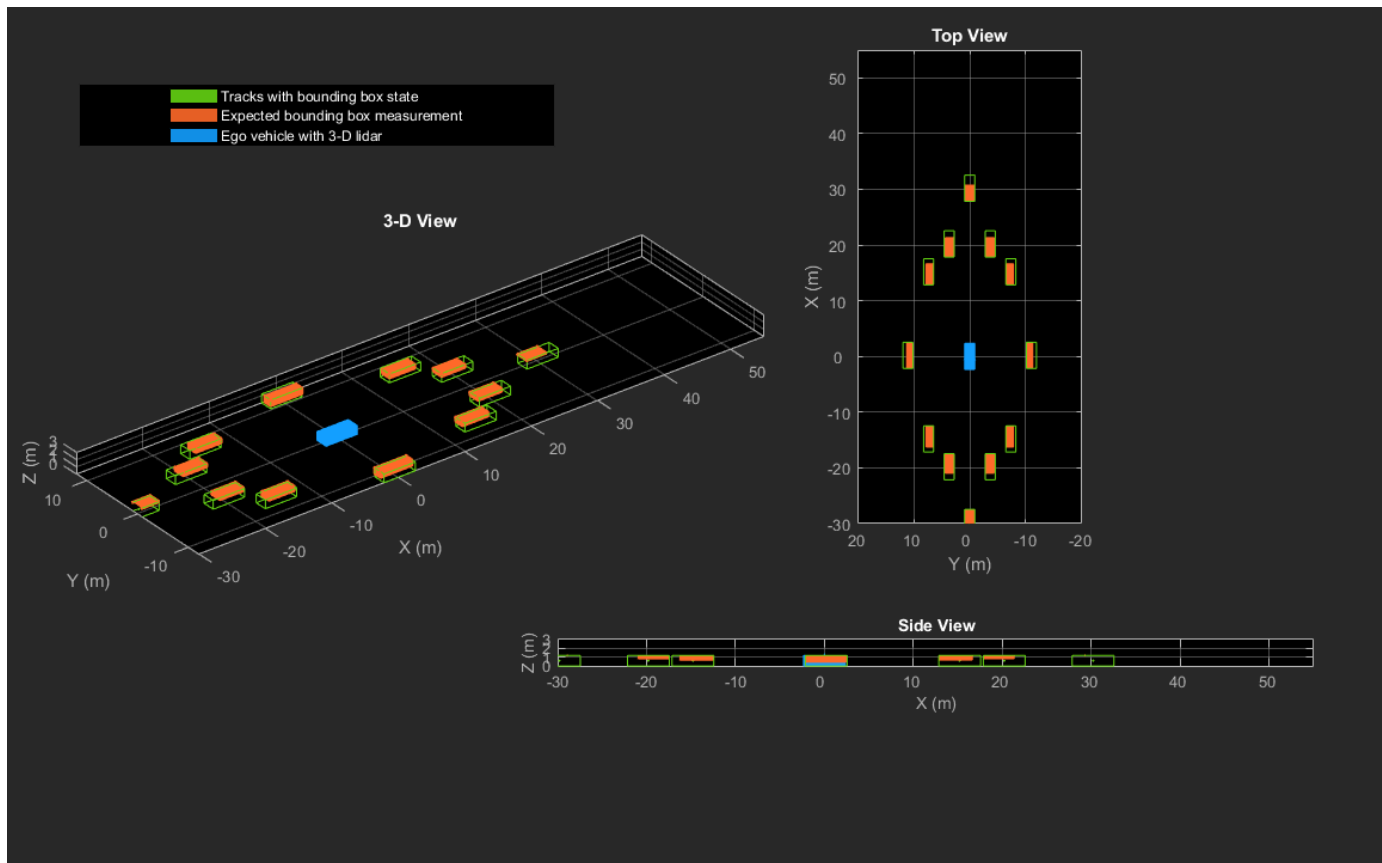
$$\mathbf{x}_{cv} = [x \ \dot{x} \ y \ \dot{y} \ z \ \dot{z} \ \theta \ l \ w \ h]$$

$$x_{ct} = [x \ \dot{x} \ y \ \dot{y} \ \dot{\theta} \ z \ \dot{z} \ \theta \ l \ w \ h]$$

For information about their state transition, refer to the `helperConstvelCuboid` and `helperConstturnCuboid` functions used in this example.

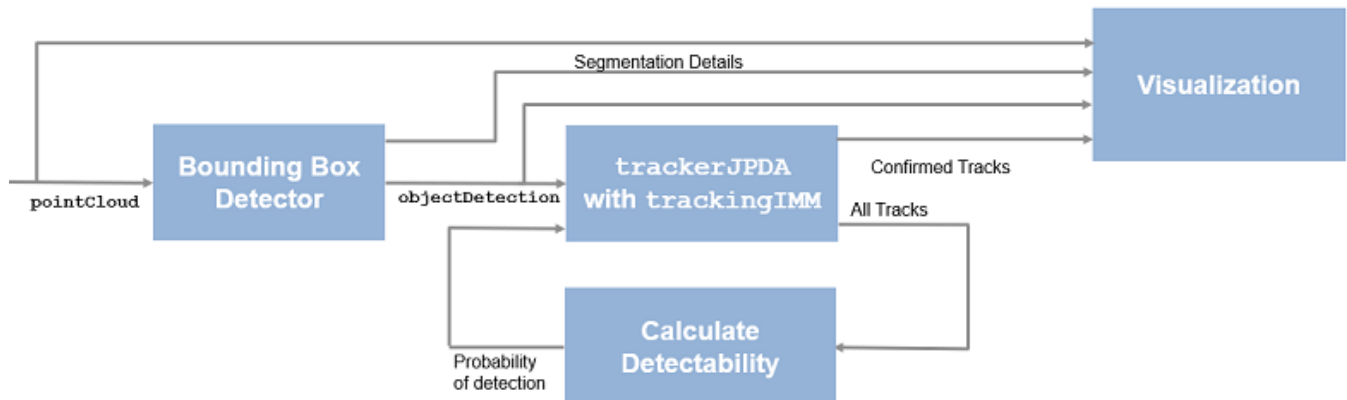
The `helperCvmeasCuboid` and `helperCtmeasCuboid` measurement models describe how the sensor perceives the constant velocity and constant turn-rate states respectively, and they return bounding box measurements. Because the state contains information about size of the target, the measurement model includes 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.



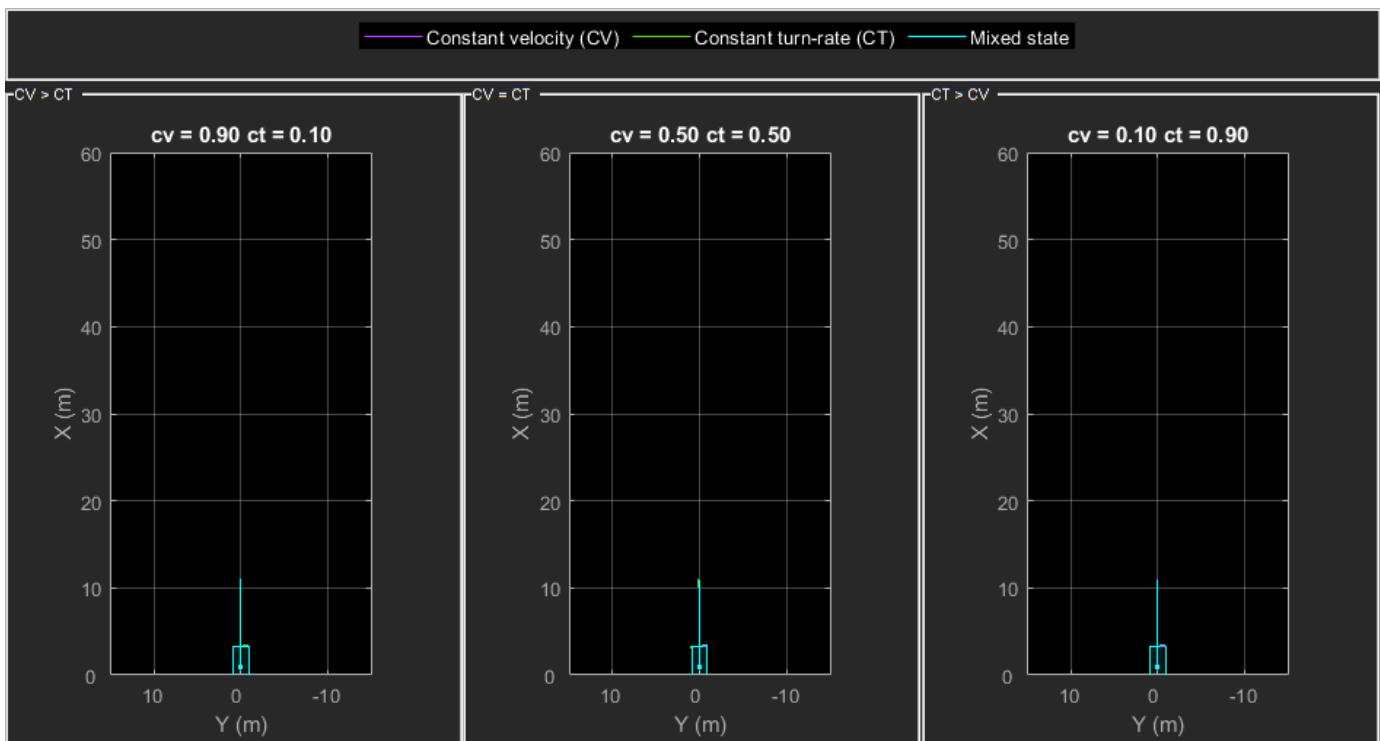
Set Up Tracker and Visualization

The image below shows the complete workflow to obtain a list of tracks from a pointCloud input.

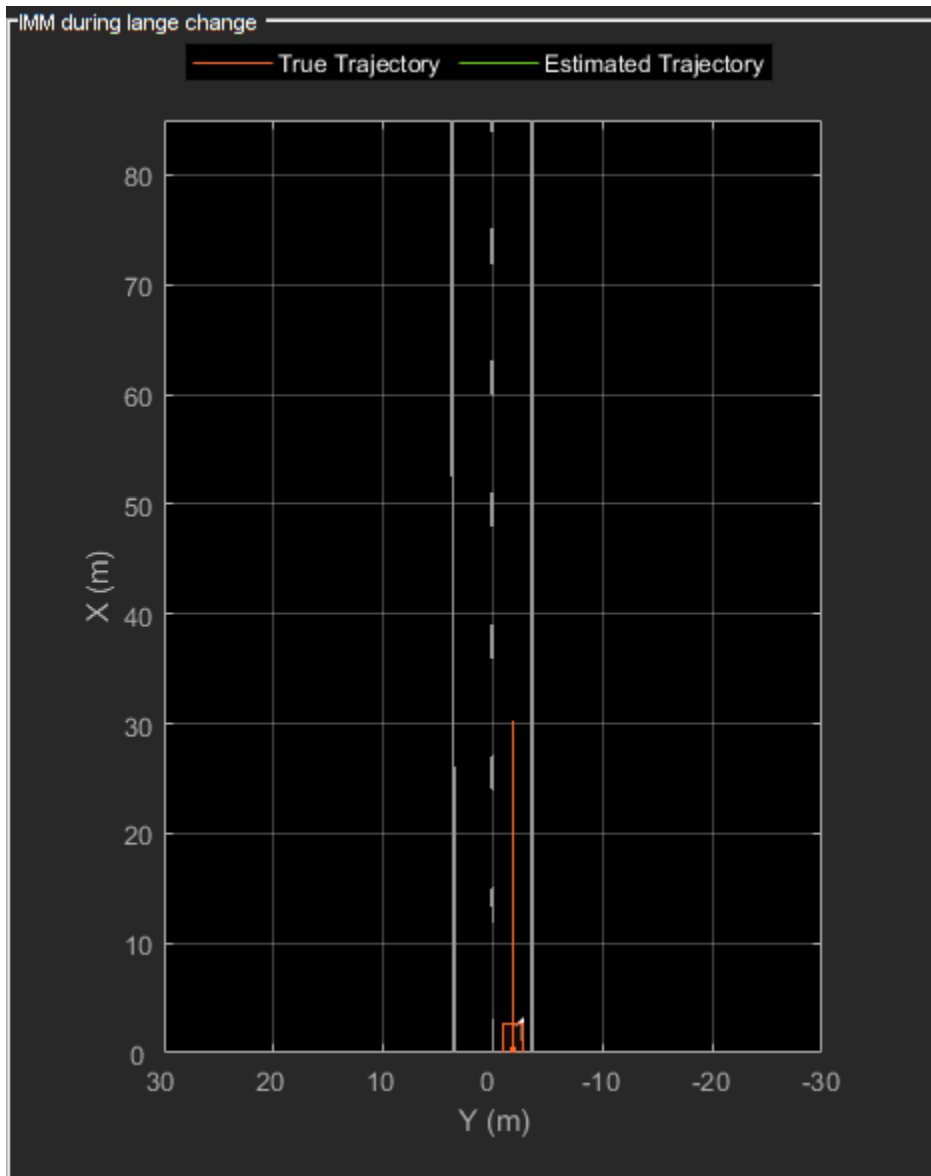


Now, set up the tracker and the visualization used in the example.

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]);

```

```
% Pass detections to track.
[confirmedTracks,tentativeTracks,allTracks,info] = tracker(detections,time,detectableTracksI
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

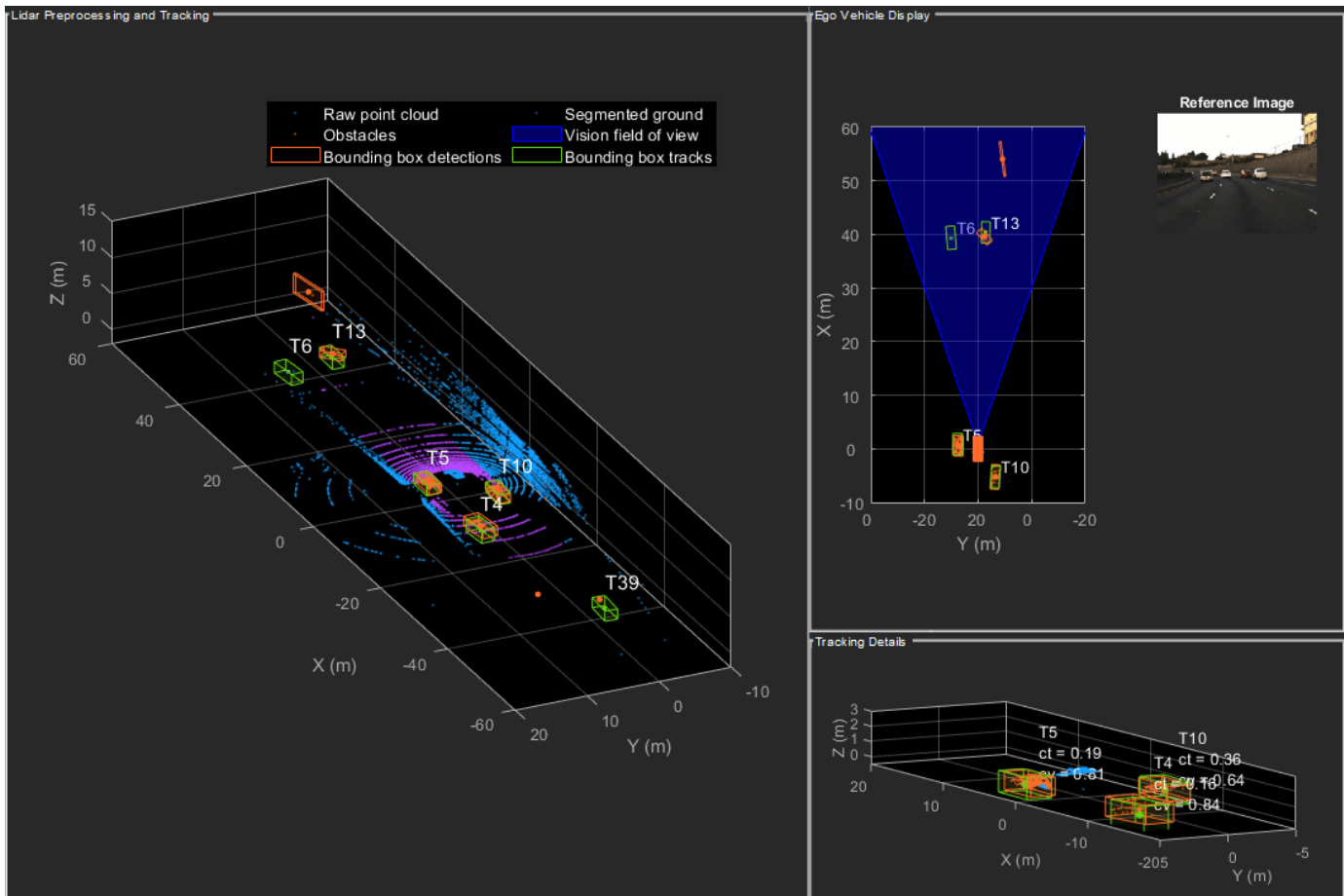
% Update display
if isValid(displayObject.PointCloudProcessingDisplay.ObstaclePlotter)
    % Get current image scan for reference image
    currentImage = imageData{i};

    % 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

% 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};

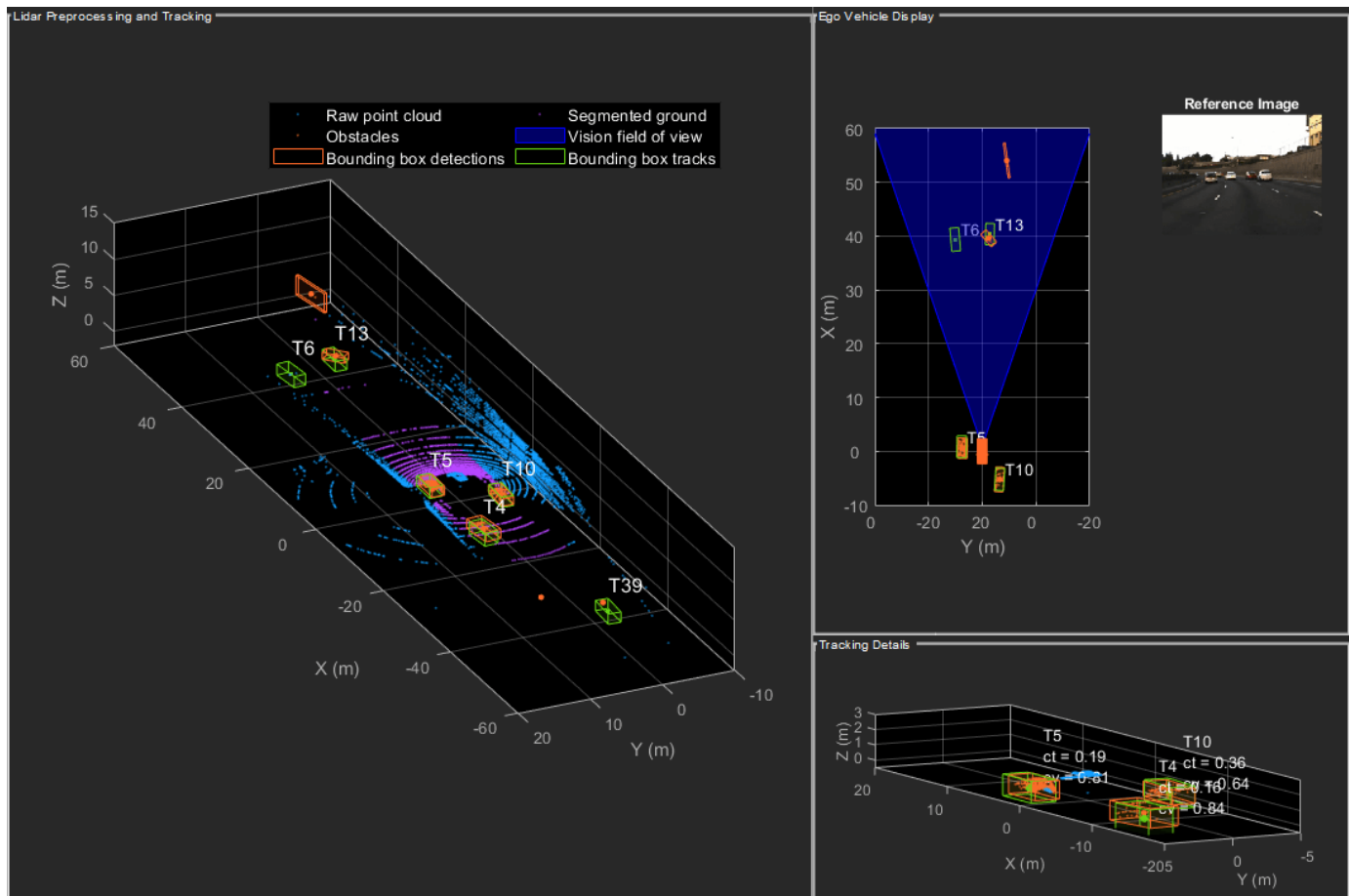
% Create configuration for MEX generation
cfg = coder.config('mex');

% Replace cfg with the following to generate static library and perform
% software-in-the-loop simulation. This requires an 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

```

Code generation successful.



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)));

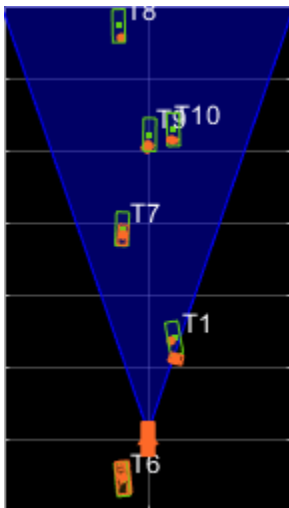
1
```

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.

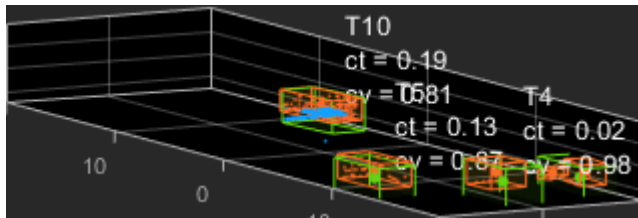
Track Maintenance



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,

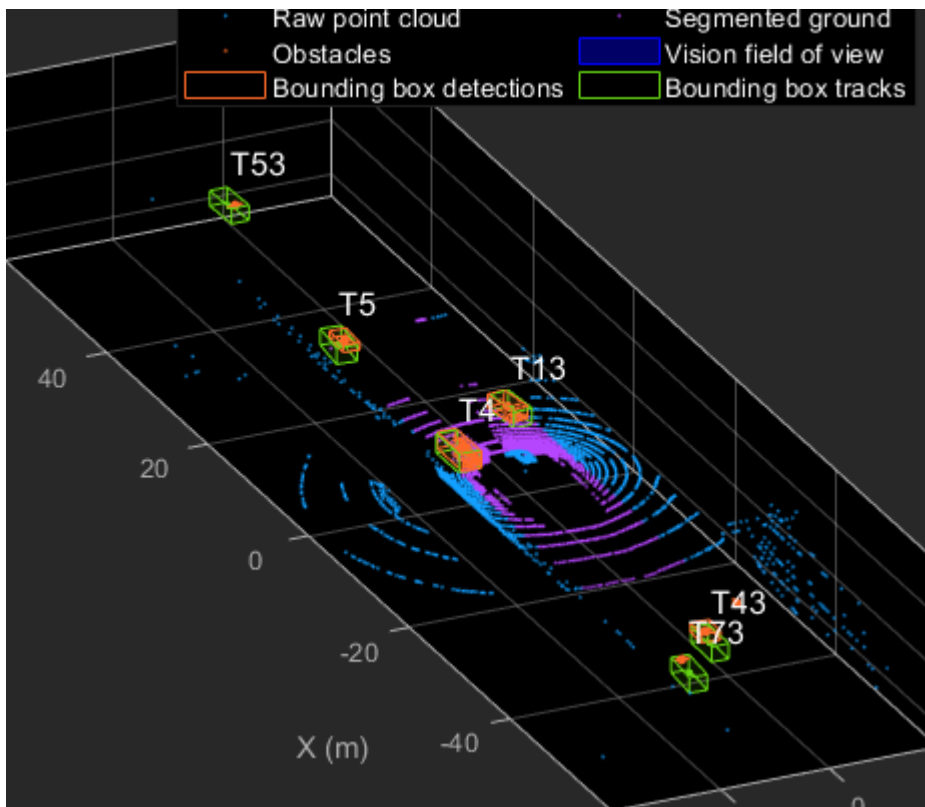
track T9 is lost because the tracked vehicle was missed (not detected) for a long time by the sensor. Also, notice that the tracked objects are able to maintain their shape and kinematic center by positioning the detections onto the visible portions of the vehicles. For example, as Track T7 moves forward, 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 to 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

This section highlights the code from some important supporting files used in this example. The complete list of supporting files can be found in the current working directory after opening the example in MATLAB.

```
% *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.
%
% <include>helperLidarModel.m</include>
%
```

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.

    % 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
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,minZDistance)
% 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 = pcfitecuboid(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
bboxes = bboxes(:,isValidCluster);
end

function [ptCloudOut,obstacleIndices,groundIndices] = removeGroundPlane(ptCloudIn,maxGroundDist,referenceVector,maxAngularDist)
% This method removes the ground plane from point cloud using
% pcfiteplane.
[~,groundIndices,outliers] = pcfiteplane(ptCloudIn,maxGroundDist,referenceVector,maxAngularDist);
ptCloudOut = select(ptCloudIn,outliers);
obstacleIndices = currentIndices(outliers);
groundIndices = currentIndices(groundIndices);
end

function [ptCloudOut,indices,croppedIndices] = cropPointCloud(ptCloudIn,xLim,yLim,zLim,egoVehicleLocation)
% 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.

```

function [detections,obstacleIndices,groundIndices,croppedIndices,...
    confirmedTracks, modelProbs] = mexLidarTracker(ptCloudLocations,time)

persistent detectorModel tracker detectableTracksInput currentNumTracks

if isempty(detectorModel) || isempty(tracker) || isempty(detectableTracksInput) || isempty(currentNumTracks)

    % 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 noise
        '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
end

```

helperCalcDetectability

The function calculates 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
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 `fusionRadarSensor`.

```
% Create surveillance radar.
sensor = fusionRadarSensor(SensorIndex = 1, ...
    ScanMode = '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, ...                     % Report INS information
    TargetReportFormat = 'Detections', ... % Detections without clustering
    RangeLimits = [0 15e3], ...           % m
    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 fusionRadarSensor]}
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.

% 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);

% 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);

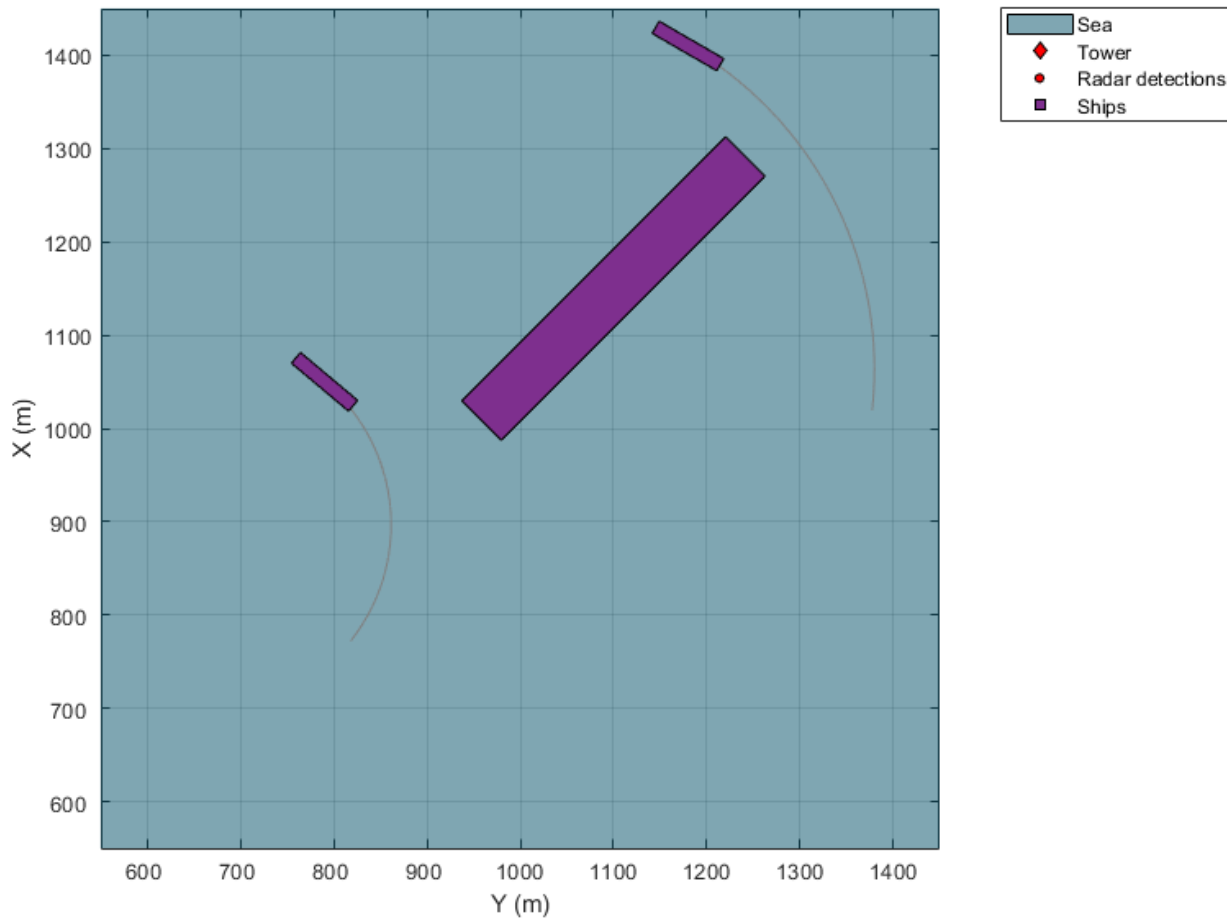
% Create the large ship's trajectory, traveling at a constant heading at 10 knots.
speed = 10; % knots
initYaw = -135; % degs
initPos = [1150 1100 0];

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`.

```

%Create a trackingSensorConfiguration from the platform.
sensorConfig = trackingSensorConfiguration(tower, SensorTransformFcn = @ctmeas);

% Set FilterInitializationFcn
sensorConfig{1}.FilterInitializationFcn = @(varargin)filterInitFcn(varargin{:},sensorConfig{1}.S

% Set DetectionProbability for trackingSensorConfiguration
sensorConfig{1}.DetectionProbability = 0.99;

% % Noise covariance corresponding to a resolution cell of the radar.
resolutionNoise = diag((sensorConfig{1}.SensorResolution/2).^2);

% Create a PHD tracker using the trackingSensorConfiguration.
tracker = trackerPHD(SensorConfigurations = sensorConfig, ...
    HasSensorConfigurationsInput = true, ...
    PartitioningFcn = @(x)partitionDetections(x,2,6), ...
    ExtractionThreshold = 0.75,...
    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;

% Set random seed for repeatable results.
rng(2019,'twister');

% 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);

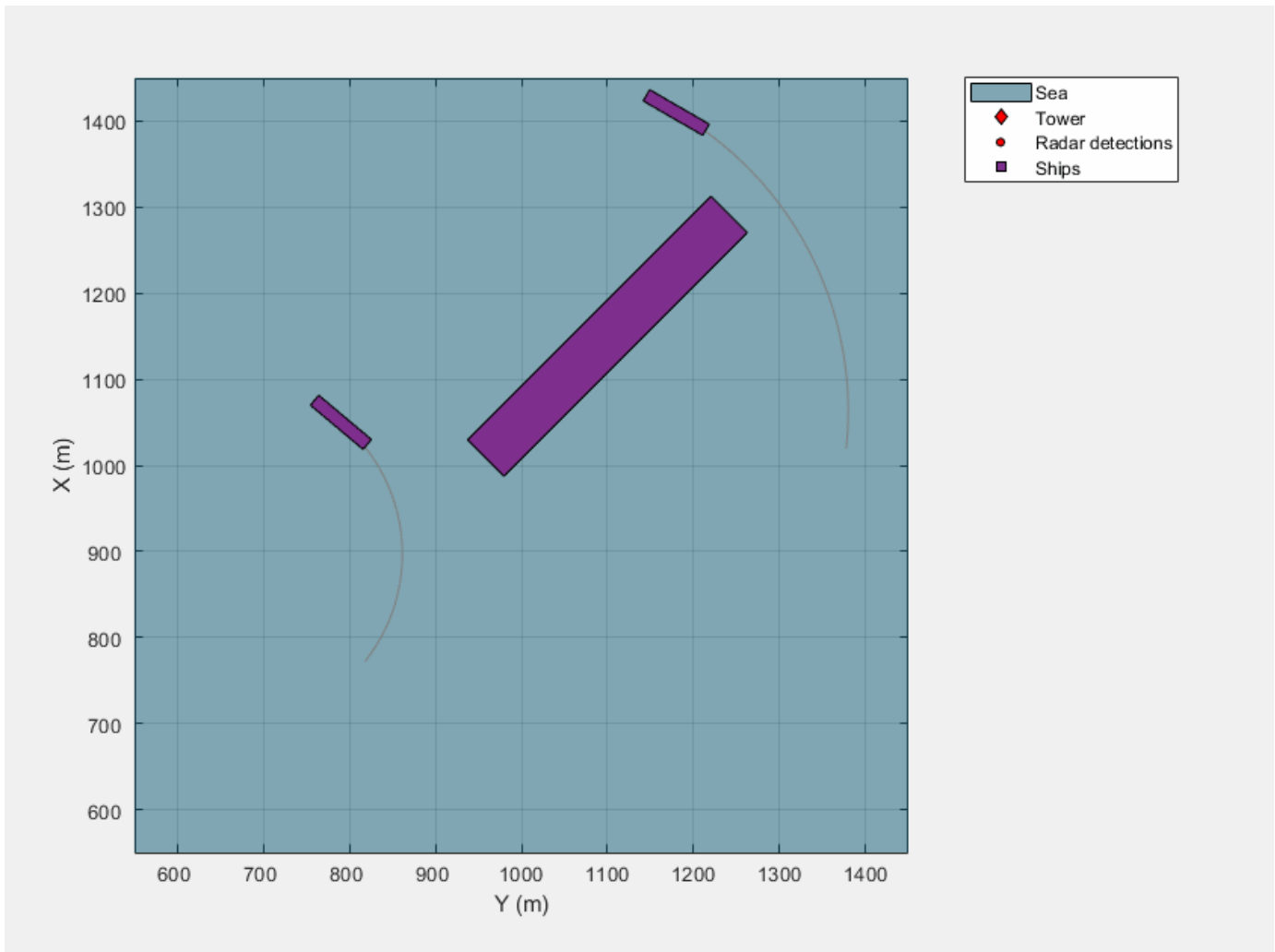
    % Update measurement noise of detections to match radar's resolution.
    dets = updateMeasurementNoise(dets,resolutionNoise);

    % Update tracker.
    tracks = tracker(dets,config,time);

    % Update display with current beam position, detections, and track positions.
    theaterDisplay(dets,config,tracks);

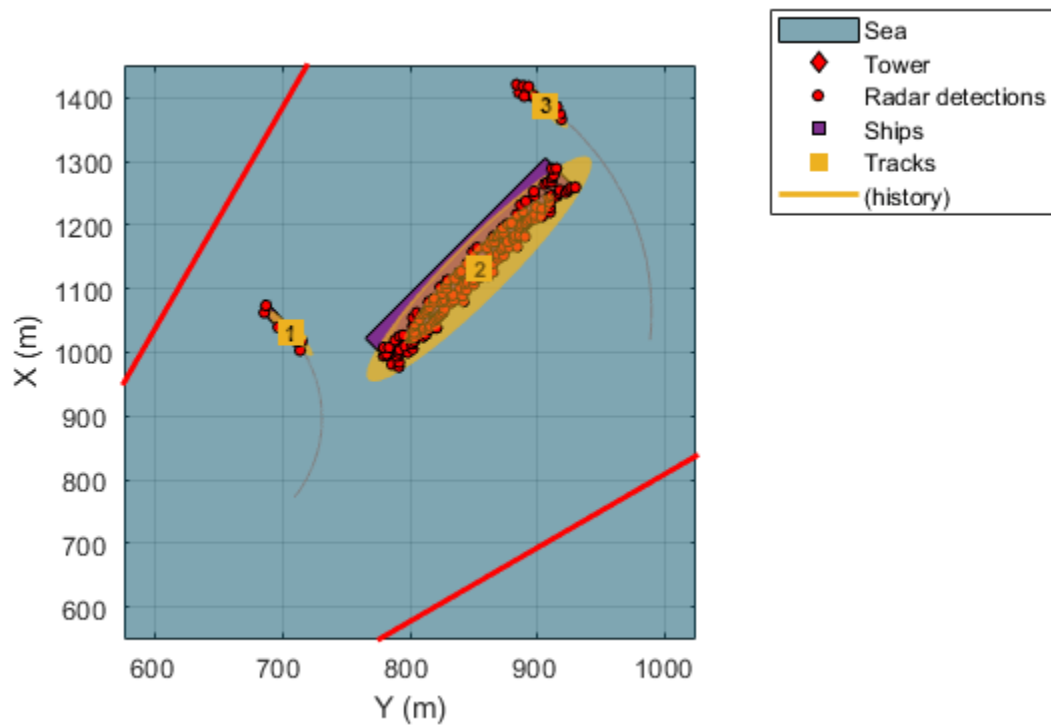
    % Take snapshot.
    snapFigure(theaterDisplay,any(time==snapTimes));
end
writeMovie(theaterDisplay);

```

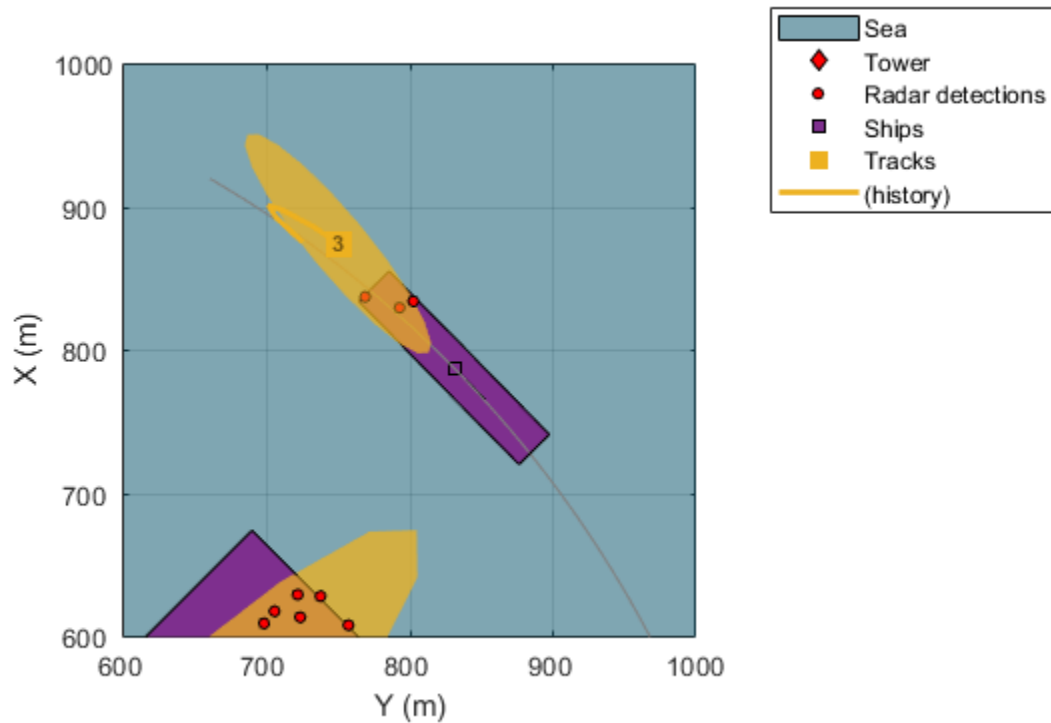
The following figure shows the radar detections, shown as red dots, and the estimated track locations, shown as yellow squares annotated with the track ID, and the estimated tracked object's extent, shown as a yellow ellipse. The radar tower is located at the origin, (0,0), which is not shown in the 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)
```



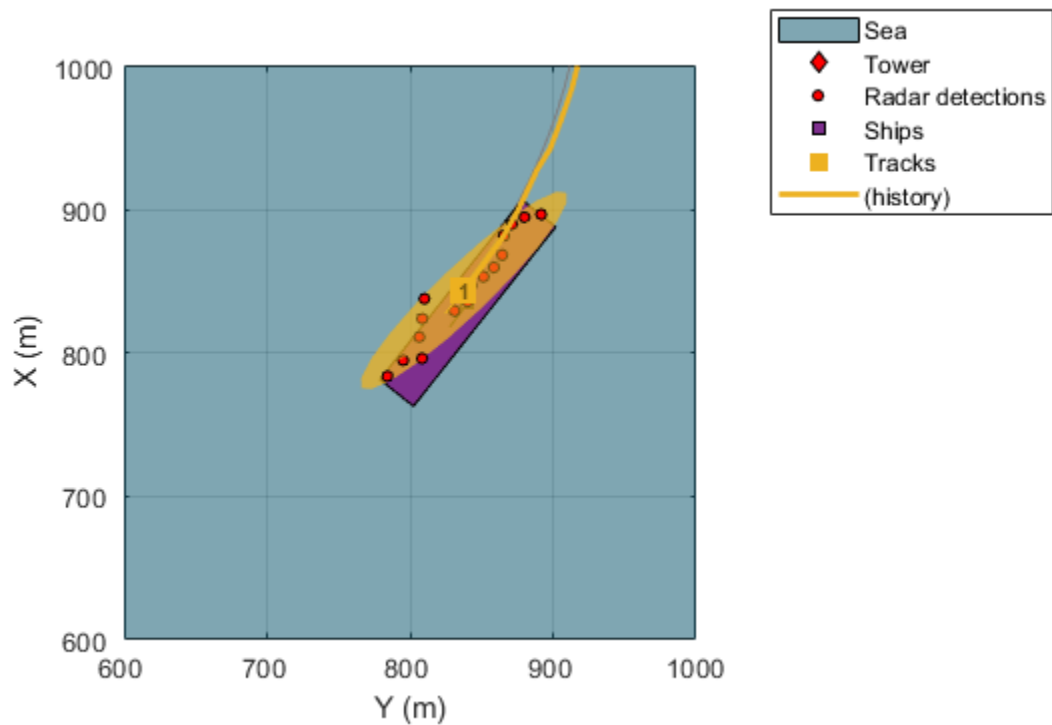
Because the ships are modeled as extended objects and not point targets, detections of a ship can be occluded by the presence of another ship between the ship and the radar. This is shown in the following figure. In this case, the smaller ship at the top of the figure is not detected by the radar. The 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]);
```



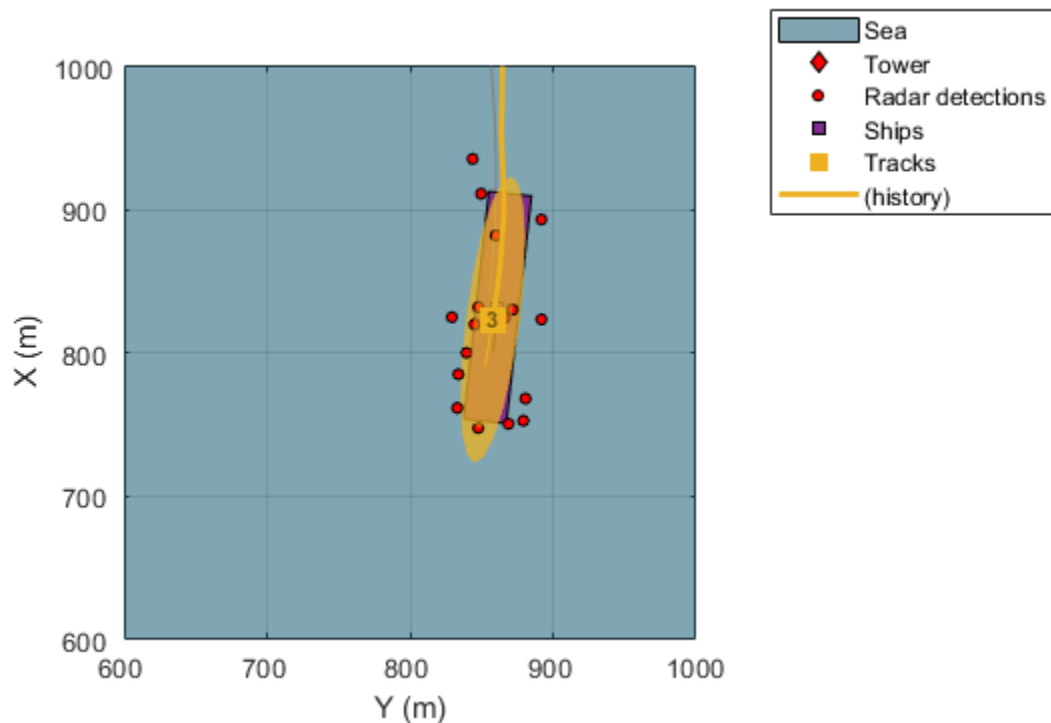
The following figure shows the PHD estimate of the smaller ship located closest to the radar in the scenario. You can verify the PHD estimate of the ship's location lies near the center of the ship and the estimated size is reasonably close to the actual size of the ship, indicated by the overlap of 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	98.986	17.426	16.321
2	473.55	55.782	8.8344
3	101	18.791	17.39

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

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 50]);

% 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 numComps]);
    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 numComps]);
    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 6-352 MATLAB® example.

Setup

The lidar data used in this example is available at the following link: <https://ssd.mathworks.com/supportfiles/lidar/data/TrackVehiclesUsingLidarExampleData.zip>

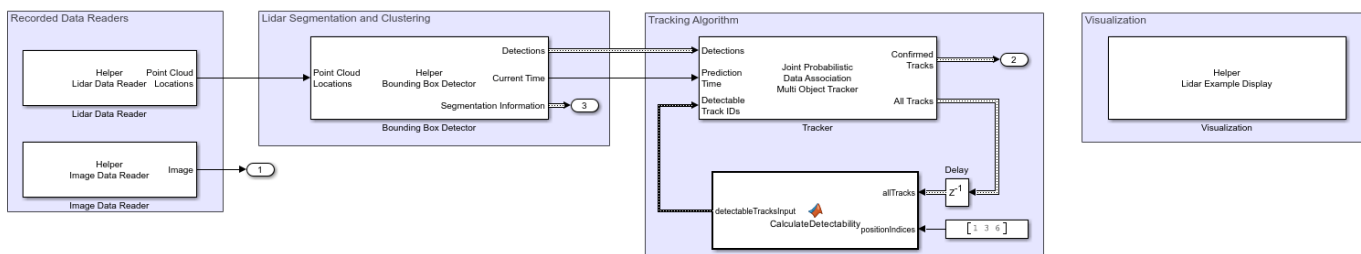
Download the data files into the current working folder. If you want to place the files in a different folder, change the directory name in the subsequent instructions.

```
% Load the data if unavailable.
if ~exist('lidarData_1.mat','file')
    dataUrl = 'https://ssd.mathworks.com/supportfiles/lidar/data/TrackVehiclesUsingLidarExampleData.zip';
    datasetFolder = fullfile(pwd);
    unzip(dataUrl,datasetFolder);
end
```

Overview of the Model

```
load_system('TrackVehiclesSimulinkExample');
set_param('TrackVehiclesSimulinkExample','SimulationCommand','update');
open_system('TrackVehiclesSimulinkExample');
```

Track Vehicles Using Lidar Data: From Point Cloud To Track List



info

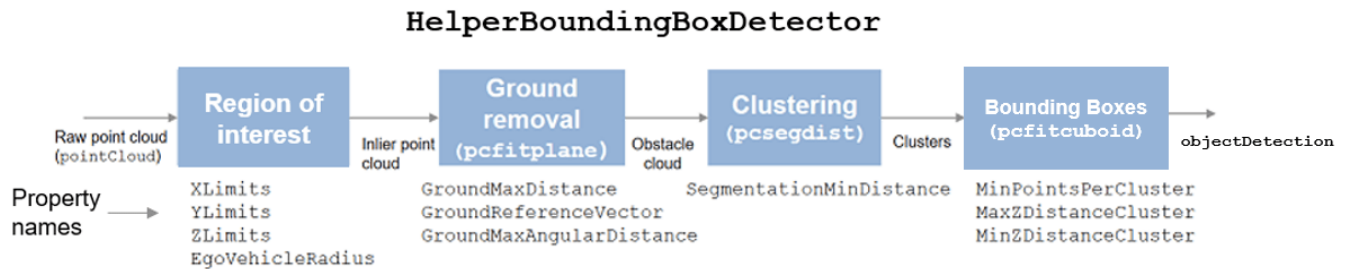
Copyright 2019-2020 The MathWorks, Inc.

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

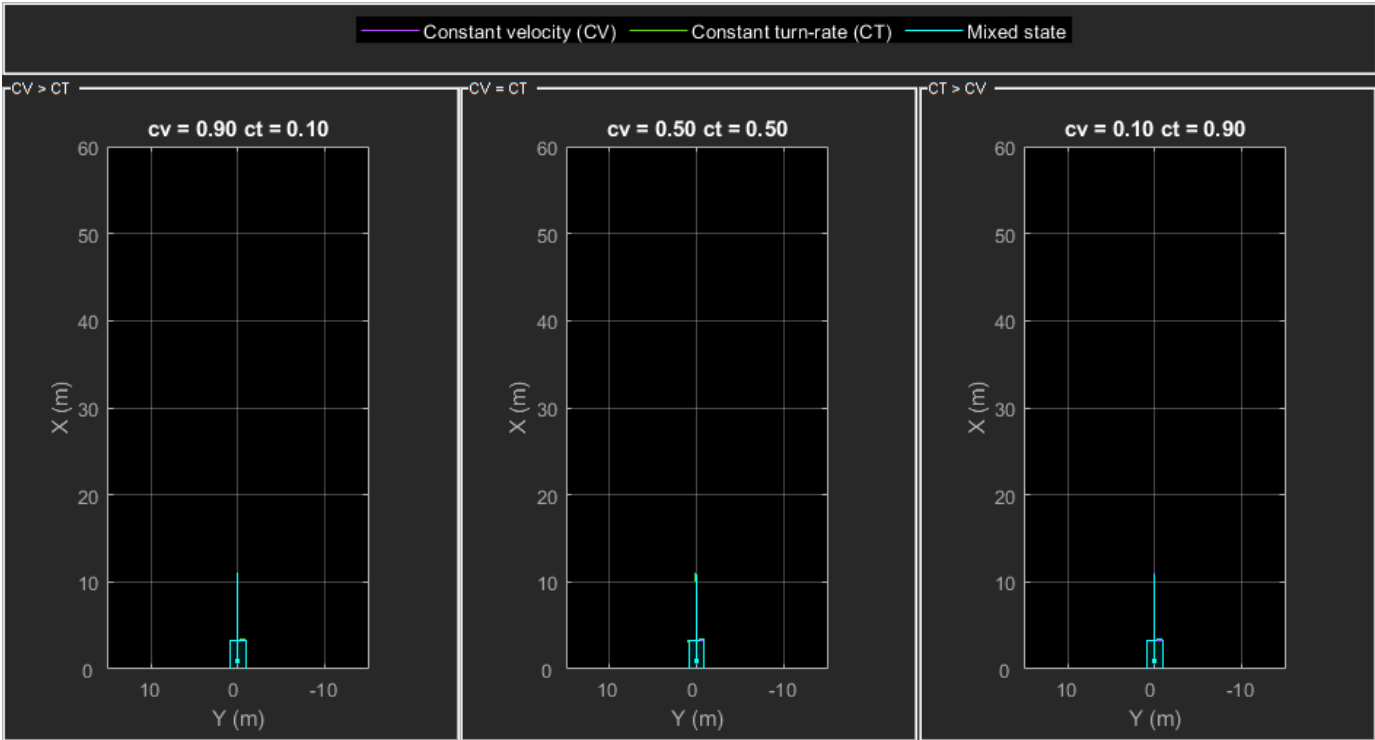
As described earlier, the raw data from sensor contains a large number of points. This raw data must be preprocessed to extract objects of interest, such as cars, cyclists, and pedestrian. The preprocessing is done using the Bounding Box Detector block. The Bounding Box Detector is also implemented as 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.



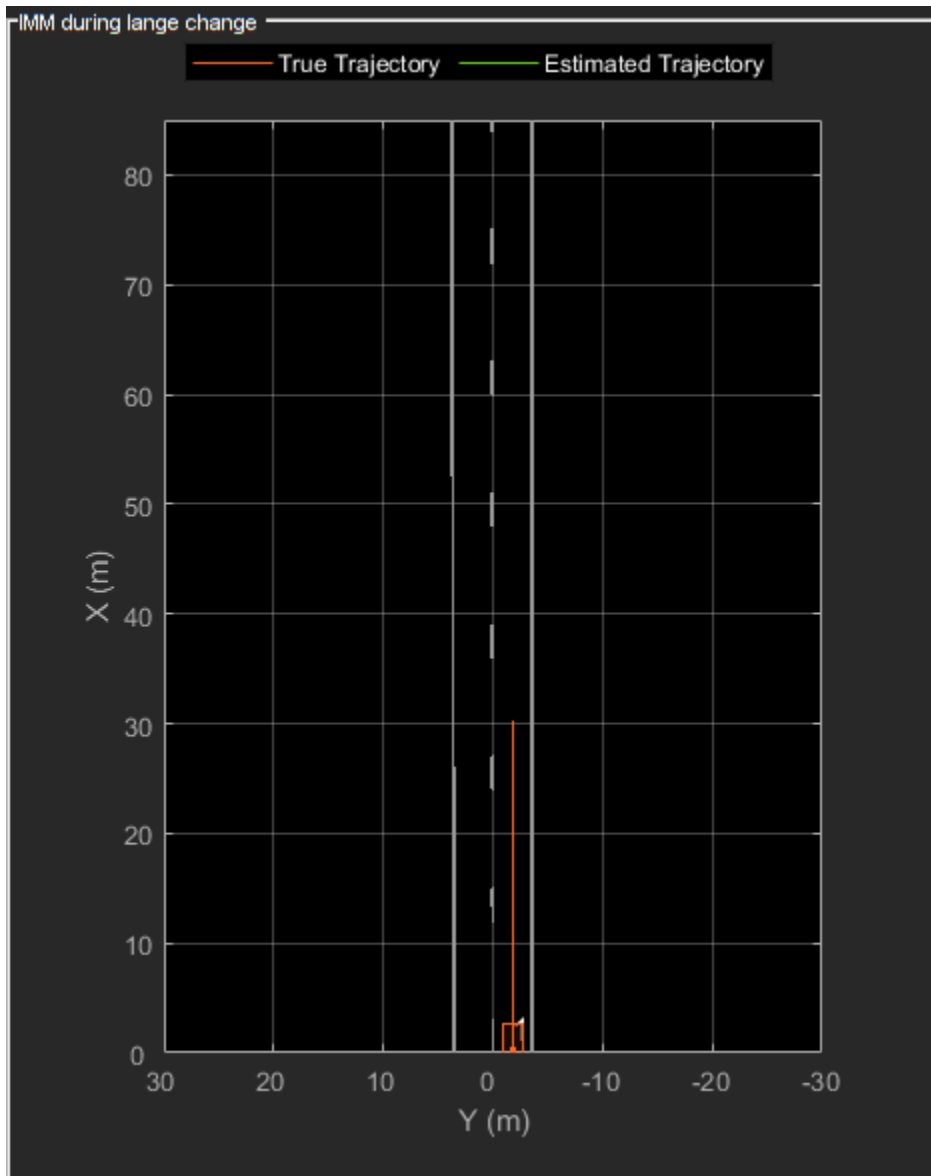
The block outputs the detections and segmentation information as `Simulink.Bus` (Simulink) objects named `detectionBus` and `segmentationBus`. These buses are created in the base workspace using helper function `helperCreateDetectorBus` specified in the `PreLoadFcn` callback. See “Model Callbacks” (Simulink) 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

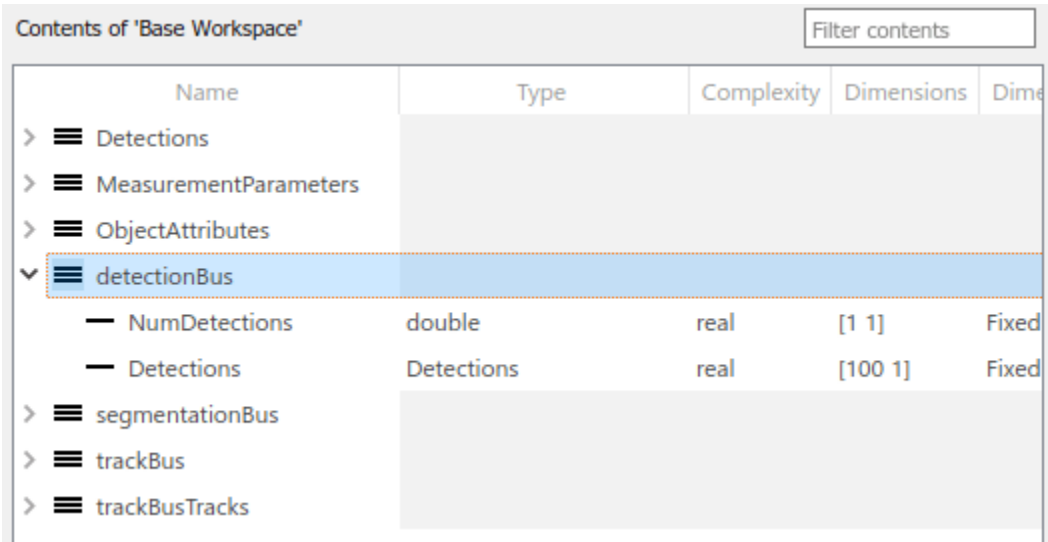
The Visualization block is also implemented using the MATLAB System block and is defined using `HelperLidarExampleDisplayBlock`. The block uses `RunTimeObject` parameter of the blocks to display their outputs. See "Access Block Data During Simulation" (Simulink) for further information on how to access block outputs during simulation.

Detections and Tracks Bus Objects

As described earlier, the inputs and outputs of different blocks are defined by bus objects. You can visualize the structure of each bus using the Type 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`.



Name	Type	Complexity	Dimensions	Dimensionality
> Detections				
> MeasurementParameters				
> ObjectAttributes				
▼ detectionBus				
— NumDetections	double	real	[1 1]	Fixed
— Detections	Detections	real	[100 1]	Fixed
> segmentationBus				
> trackBus				
> trackBusTracks				

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.

Contents of 'Base Workspace' Filter contents

Name	Type	Complexity	Dimensions	Dimensionality
▼ Detections				
— SensorIndex	double	real	[1 1]	Fixed
— Time	double	real	[1 1]	Fixed
— Measurement	double	real	[7 1]	Fixed
— MeasurementNoise	double	real	[7 7]	Fixed
— ObjectClassID	double	real	[1 1]	Fixed
— MeasurementParameters	MeasurementParameters	real	1	Fixed
— ObjectAttributes	ObjectAttributes	real	1	Fixed
> MeasurementParameters				
> ObjectAttributes				
> detectionBus				
> segmentationBus				
> trackBus				
> trackBusTracks				

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".

Contents of 'Base Workspace' Filter contents

Name	Type	Complexity	Dimensions	Dimensionality
> Detections				
> MeasurementParameters				
> ObjectAttributes				
> detectionBus				
> segmentationBus				
▼ trackBus				
— NumTracks	double	real	[1 1]	Fixed
— Tracks	Bus: trackBusTracks	real	[100 1]	Fixed
> trackBusTracks				

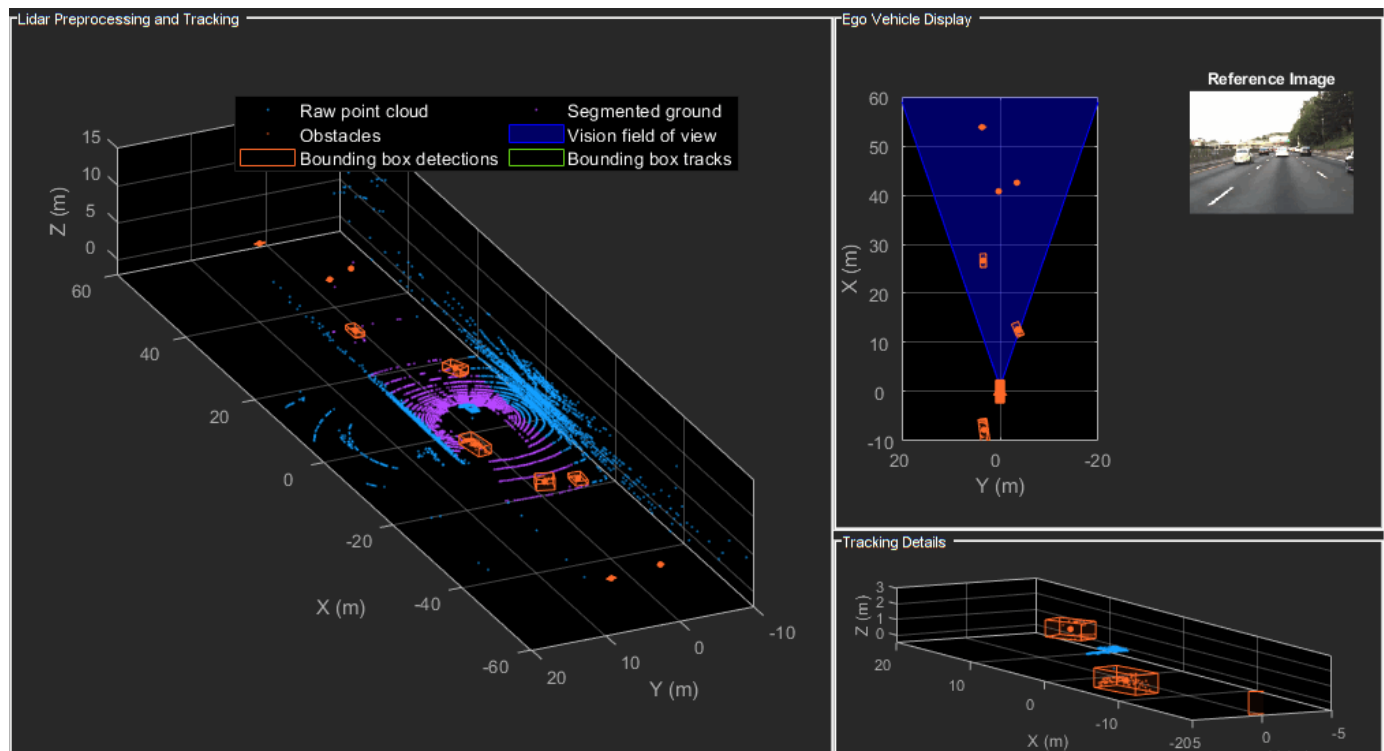
The second element Tracks is a bus object defined by trackBusTracks. This bus is automatically created by the tracker block by using the bus name specified as the prefix. Notice that the structure of the bus is similar to the objectTrack class.

Contents of 'Base Workspace' Filter contents

Name	Type	Complexity	Dimensions	Dimensions
> ≡ Detections				
> ≡ MeasurementParameters				
> ≡ ObjectAttributes				
> ≡ detectionBus				
> ≡ segmentationBus				
> ≡ trackBus				
▼ ≡ trackBusTracks				
— TrackID	uint32	real	[1 1]	Fixed
— BranchID	uint32	real	[1 1]	Fixed
— SourceIndex	uint32	real	[1 1]	Fixed
— UpdateTime	double	real	[1 1]	Fixed
— Age	uint32	real	[1 1]	Fixed
— State	double	real	[11 1]	Fixed
— StateCovariance	double	real	[11 11]	Fixed
— ObjectClassID	double	real	[1 1]	Fixed
— TrackLogic	Enum: trackLogicType	real	[1 1]	Fixed
— TrackLogicState	boolean	real	[1 10]	Fixed
— IsConfirmed	boolean	real	[1 1]	Fixed
— IsCoasted	boolean	real	[1 1]	Fixed
— IsSelfReported	boolean	real	[1 1]	Fixed
≡ ObjectAttributes	Bus: ObjectAttributes	real	[20 1]	Fixed

Results

The detector and tracker algorithm is configured exactly as the “Track Vehicles Using Lidar: From Point Cloud to Track List” on page 6-352 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 6-168 MATLAB® example.

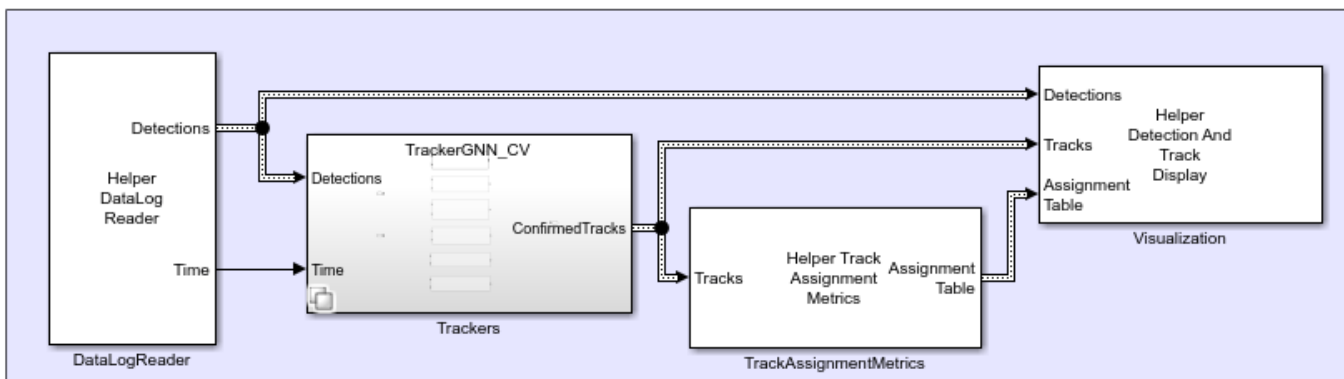
Introduction

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 6-168. The detection and time data from this scenario were then saved to the scenario file CloselySpacedData.mat.

Track Closely Spaced Targets Under Ambiguity



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

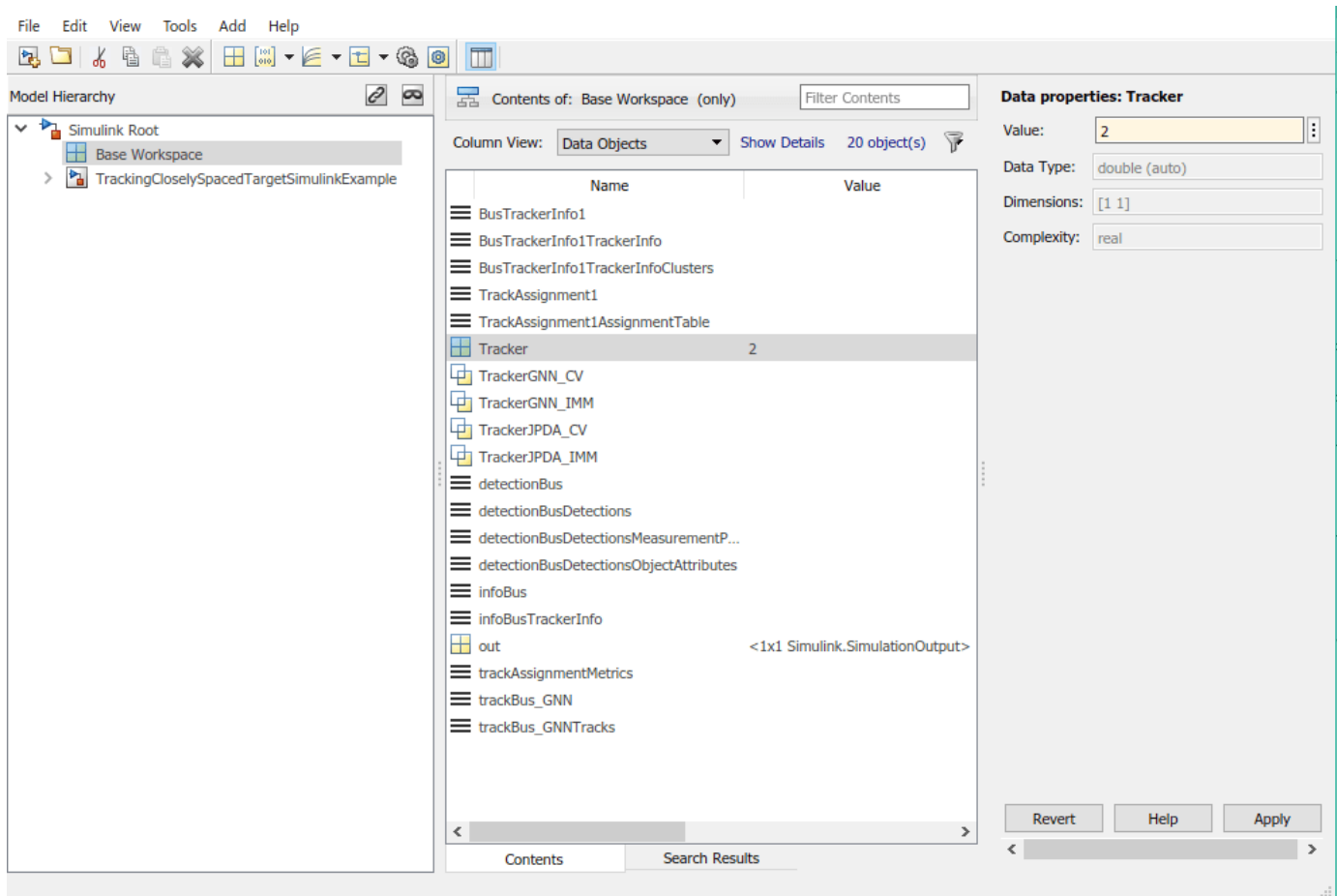
The Trackers block is a variant subsystem block, which has six subsystems defined internally. Each subsystem is composed of one of the three trackers and one of the two motion models.

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 `initcvkf` 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 Type 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.

Contents of 'Base Workspace' Filter contents

Name	Type	Complexity	Dimensions	Dimensi
▼ detectionBus				
> detectionBus	Bus: detectionBusDetections	<i>real</i>	[1 3]	<i>Fixed</i>
— NumDetections	double	real	[1 1]	Fixed
> detectionBusDetections				
> detectionBusDetectionsMeasu...				
> detectionBusDetectionsObject...				
> trackAssignmentMetrics				
> trackAssignmentMetricsAssig...				
> trackBus_GNN				
> trackBus_GNNTracks				

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.

Contents of 'Base Workspace' Filter contents

Name	Type	Complexity	Dimensions	Dimensi
> detectionBus				
▼ detectionBusDetections				
— Time	double	real	[1 1]	Fixed
— Measurement	double	real	[3 1]	Fixed
— MeasurementNoise	double	real	[3 3]	Fixed
— SensorIndex	double	real	[1 1]	Fixed
— ObjectClassID	double	real	[1 1]	Fixed
> MeasurementParameters	Bus: detectionBusDetectionsMeasu...	<i>real</i>	[1 1]	<i>Fixed</i>
> ObjectAttributes	Bus: detectionBusDetectionsObject...	<i>real</i>	[1 1]	<i>Fixed</i>
> detectionBusDetectionsMeasu...				
> detectionBusDetectionsObject...				
> trackAssignmentMetrics				
> trackAssignmentMetricsAssig...				
> trackBus_GNN				
> trackBus_GNNTracks				

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.

Contents of 'Base Workspace' Filter contents

Name	Type	Complexity	Dimensions	Dimensi
> ≡ detectionBus				
> ≡ detectionBusDetections				
> ≡ detectionBusDetectionsMeasu...				
> ≡ detectionBusDetectionsObject...				
> ≡ trackAssignmentMetrics				
> ≡ trackAssignmentMetricsAssig...				
√ ≡ trackBus_GNN				
— NumTracks	double	real	[1 1]	Fixed
> ≡ Tracks	Bus: trackBus_GNNTracks	real	[20 1]	Fixed
> ≡ trackBus_GNNTracks				

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.

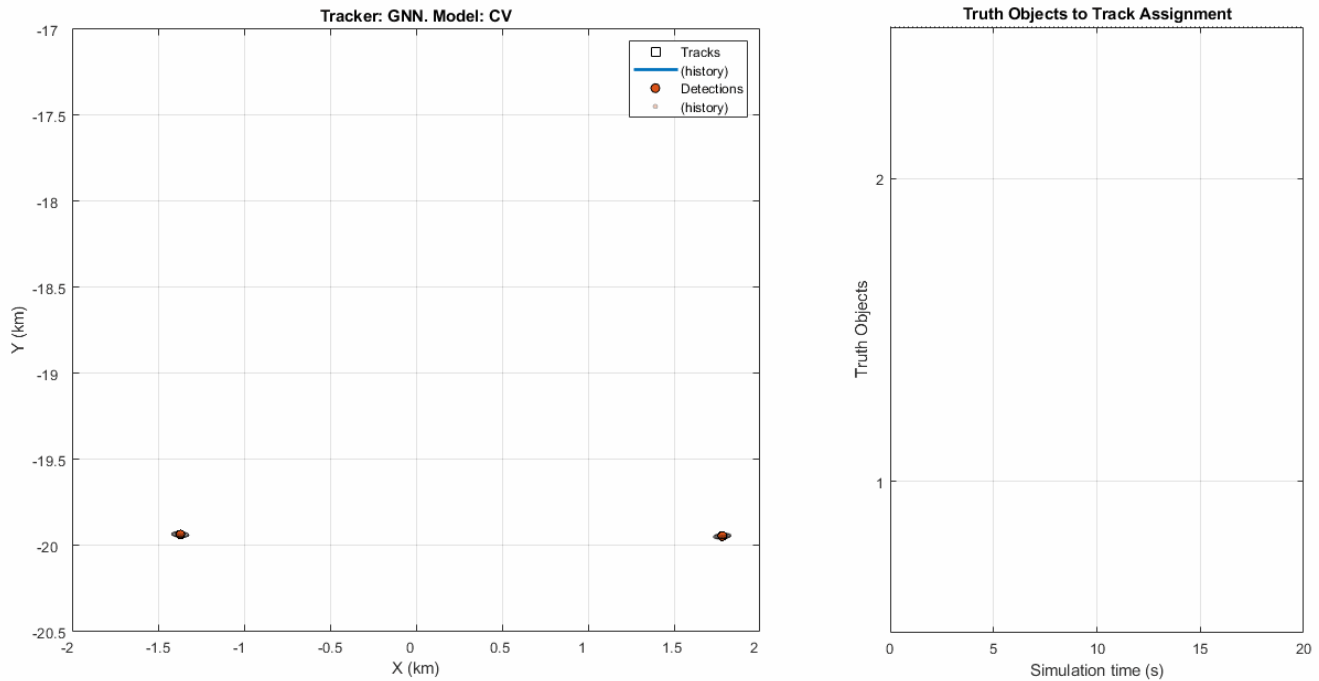
Contents of 'Base Workspace'		Filter contents			
Name	Type	Complexity	Dimensions	Dimensi	
> ≡ detectionBus					
> ≡ detectionBusDetections					
> ≡ detectionBusDetectionsMeasu...					
> ≡ detectionBusDetectionsObject...					
> ≡ trackAssignmentMetrics					
> ≡ trackAssignmentMetricsAssig...					
> ≡ trackBus_GNN					
▼ ≡ trackBus_GNNTracks					
— TrackID	uint32	real	[1 1]	Fixed	
— BranchID	uint32	real	[1 1]	Fixed	
— SourceIndex	uint32	real	[1 1]	Fixed	
— UpdateTime	double	real	[1 1]	Fixed	
— Age	uint32	real	[1 1]	Fixed	
— State	double	real	[6 1]	Fixed	
— StateCovariance	double	real	[6 6]	Fixed	
— ObjectClassID	double	real	[1 1]	Fixed	
— TrackLogic	Enum: trackLogicType	real	[1 1]	Fixed	
— TrackLogicState	double	real	[1 2]	Fixed	
— IsConfirmed	boolean	real	[1 1]	Fixed	
— IsCoasted	boolean	real	[1 1]	Fixed	
— IsSelfReported	boolean	real	[1 1]	Fixed	
> ≡ ObjectAttributes	Bus: detectionBusDetectionsObject...	<i>real</i>	[1 1]	<i>Fixed</i>	

Results

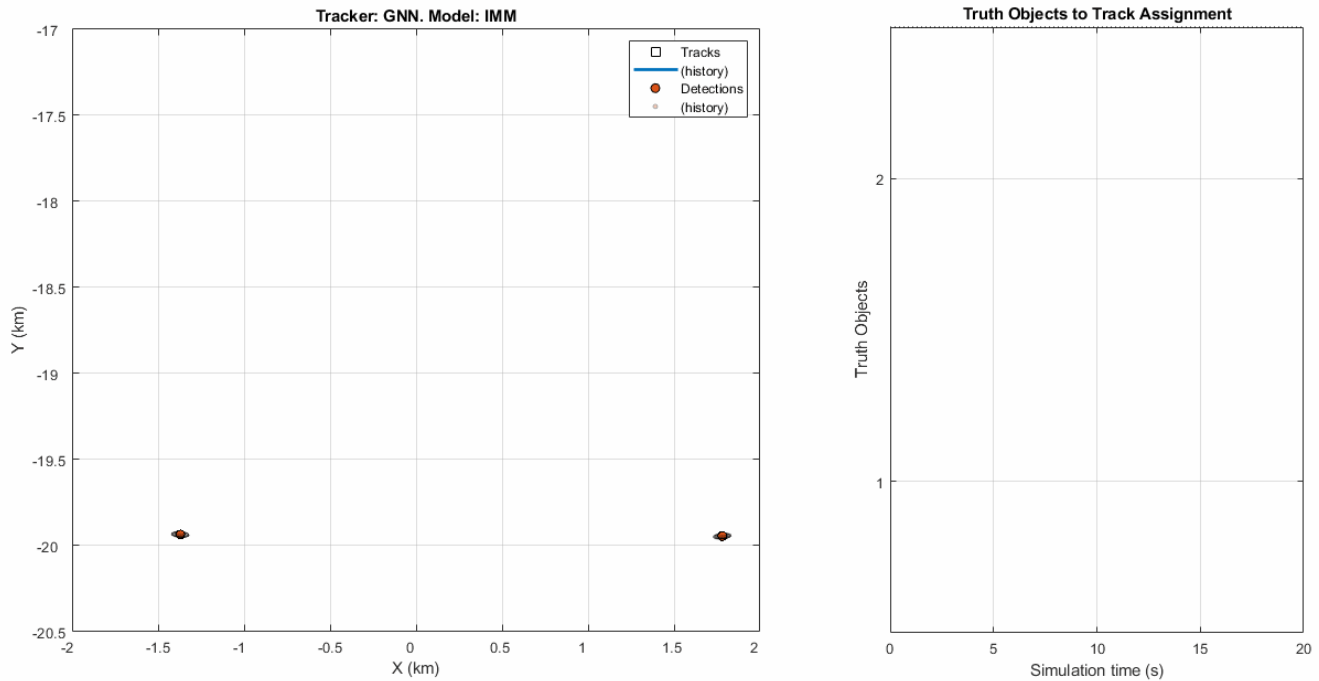
Tracker configuration in the Trackers variant subsystem block is similar to that of the “Tracking Closely Spaced Targets Under Ambiguity” on page 6-168 MATLAB® example.

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.

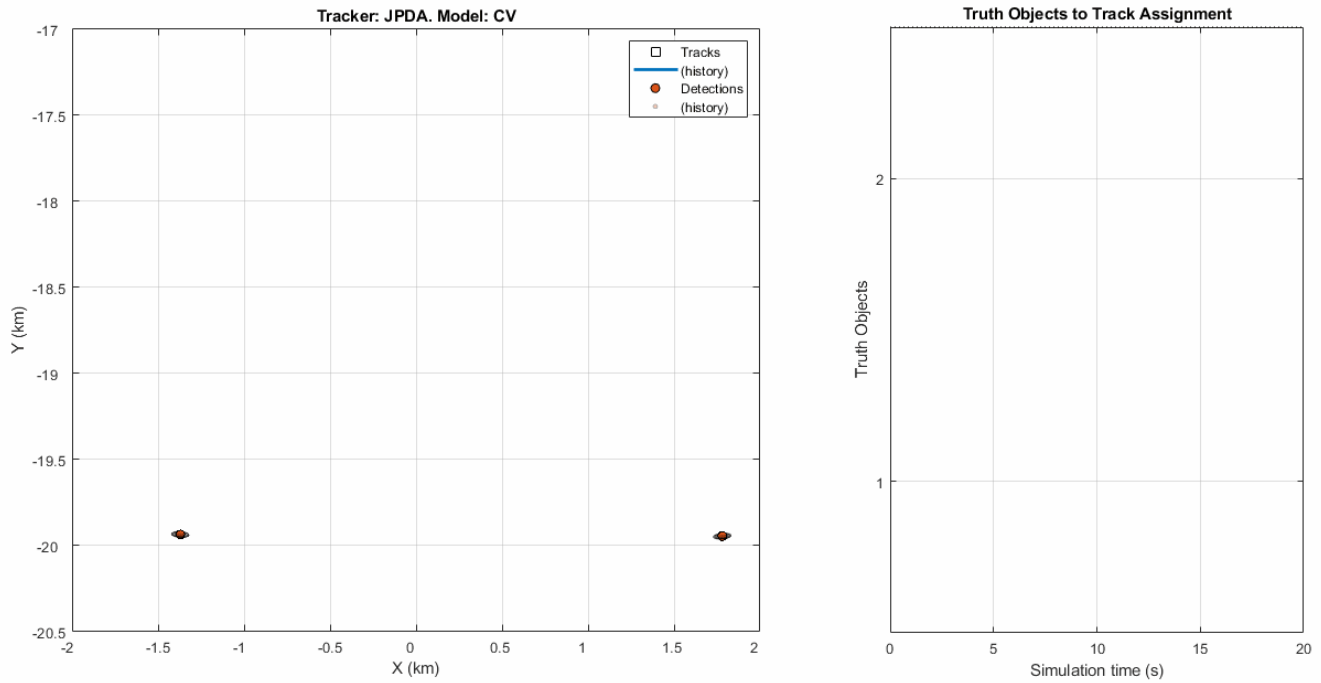


The above results are achieved from the first configuration where `Tracker = 1` in MATLAB workspace. These results show that there are two truth objects. However, the tracker generate three confirmed tracks, and one of the tracks did not survive until the end of the scenario. At the end of the scenario, 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.

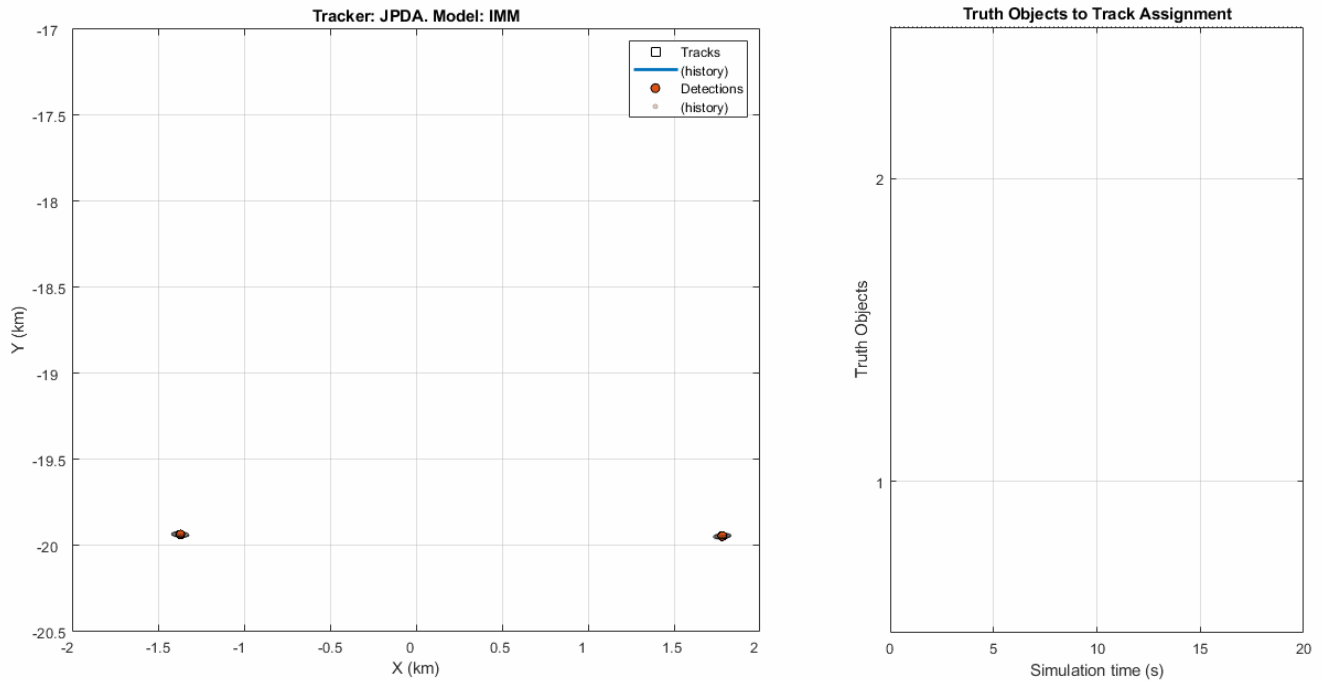


The above results are achieved from the second configuration where `Tracker = 2` in MATLAB workspace. The IMM filter enables the `trackerGNN` to track the maneuvering target correctly. Notice that truth object 1 has zero breaks due to continuous history of its associated track.

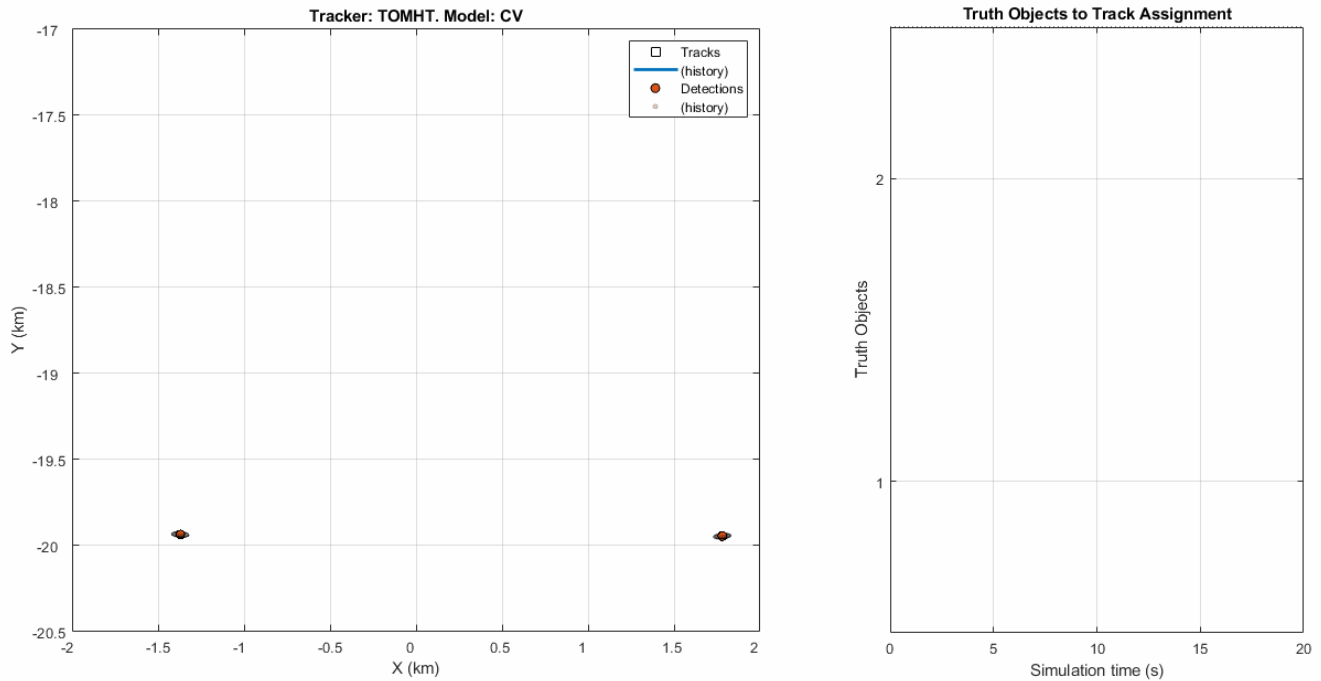
However, even with the IMM filter, one of the tracks breaks in the ambiguity region. The `trackerGNN` receives only one detection in that ambiguity region, and therefore can update only one of the tracks with it. After a few updates, the score of the coasted track falls below the deletion threshold and the tracker drops the track.



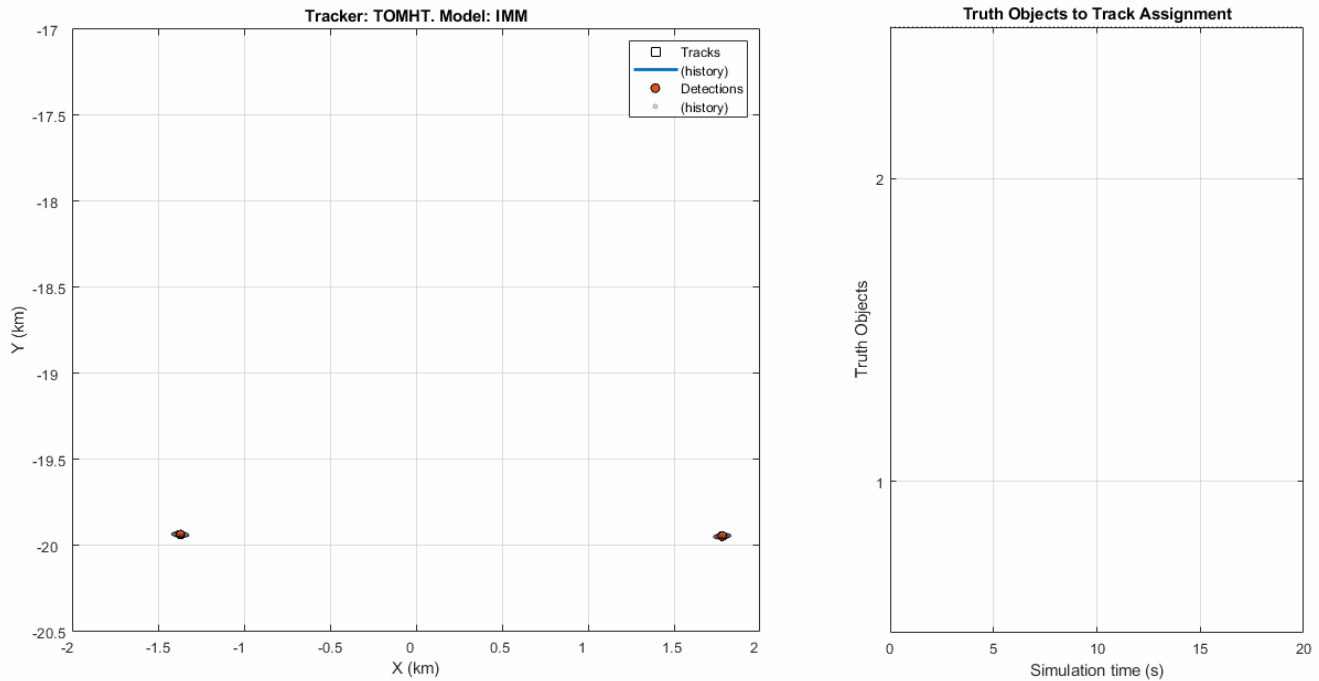
The above results are achieved from the third configuration where `Tracker = 3` in MATLAB workspace. You can observe that the `trackerJPDA` maintains both the tracks confirmed in the ambiguity region. However, there is a swap between the two tracks.



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.

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

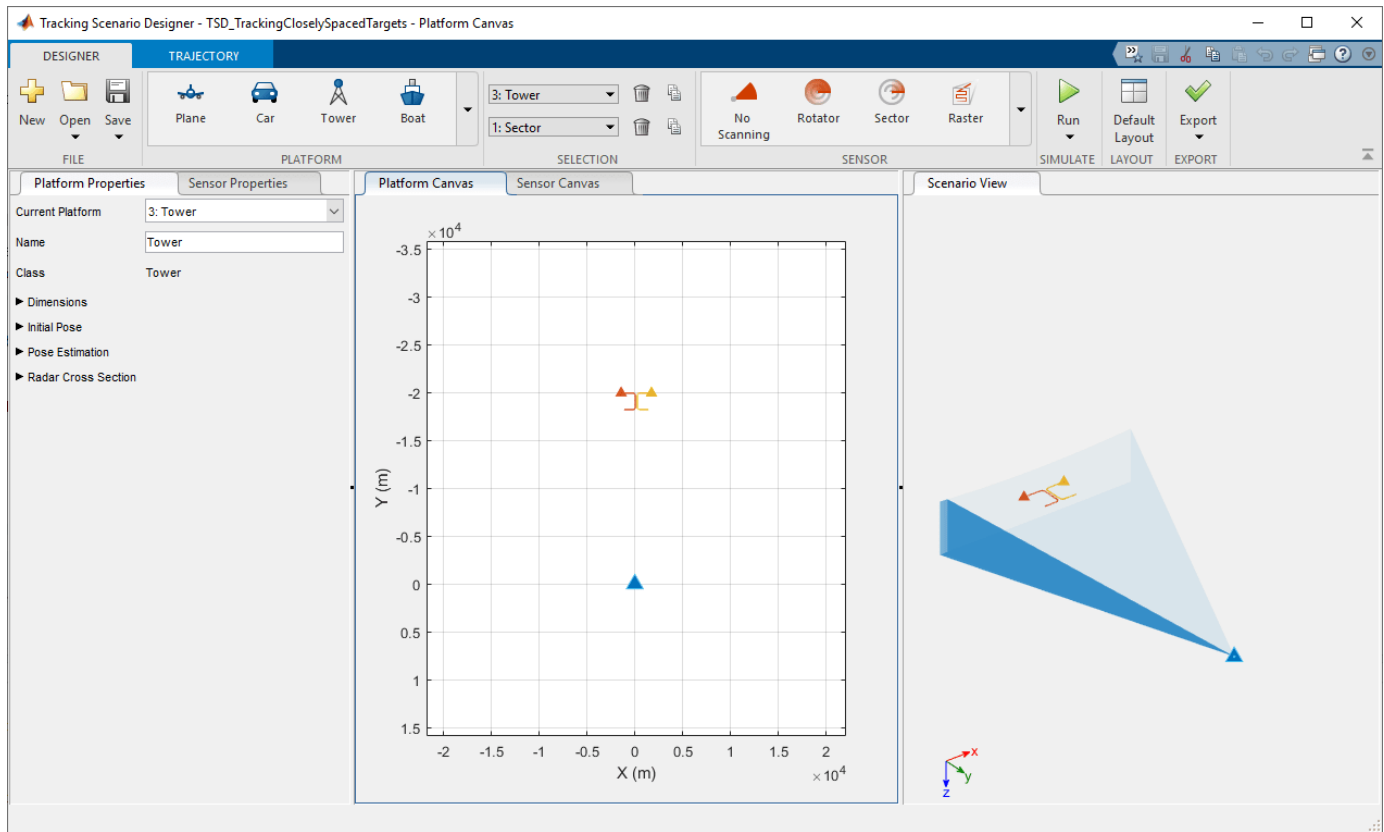
The goal of object tracking is to determine the state of an object in the presence of infrequent or uncertain measurements. In the case of multiple objects, tracking algorithms must also address the problem of data association. To test these algorithms, a *tracking scenario* can be used to synthetically 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” on page 6-168 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')
```



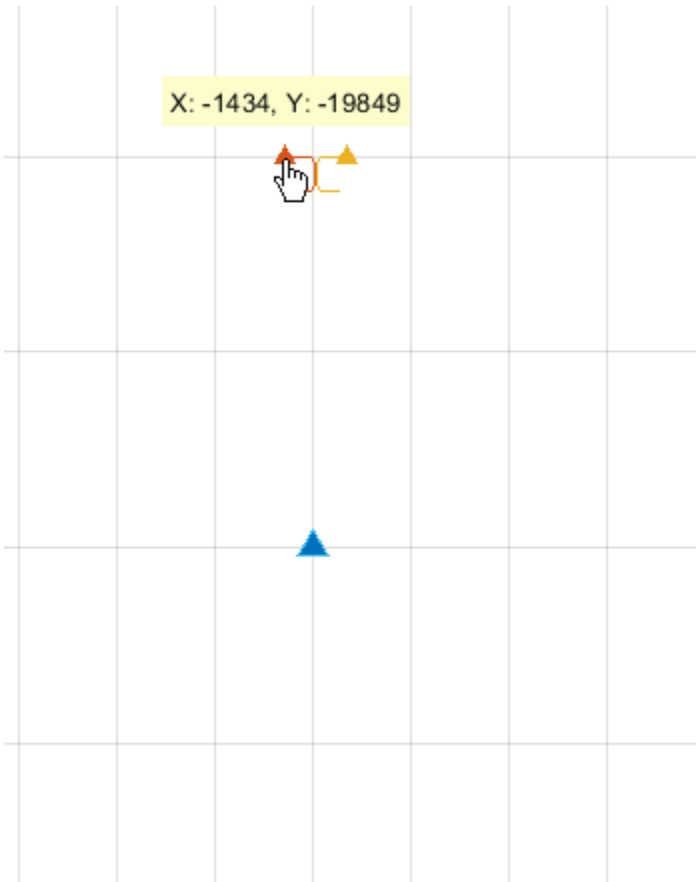
The application opens and loads the scenario. The Tracking Scenario Designer App consists of a toolbar and three document groups:

- 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

The screenshot displays the Tracking Scenario Designer - TSD_TrackingCloselySpacedTargets - Platform Properties window. The interface is divided into several sections:

- DESIGNER / TRAJECTORY:** Contains menu options like 'Waypoints', 'Delete Trajectory', and 'Trajectory Course Table'. It also features buttons for 'Trajectory Table' (highlighted in red) and 'Time-Altitude Plot' (highlighted in green).
- Platform Properties (purple border):** Shows configuration for '1 Plane'. Dimensions (Length, Width, Height) are all set to 0. Radar Cross Section is set to 10 dBsm. An Azimuth Pattern plot is visible at the bottom left of this panel.
- Platform Canvas:** A 2D plot of Y (m) vs X (m) showing a trajectory with waypoints. A Time-Altitude Plot is overlaid on this canvas, showing Altitude (m) vs Time (s) with a constant altitude of 3000 m.
- Trajectory Table:** A table at the bottom center showing waypoints for the trajectory.
- Scenario View:** A 3D perspective view of the trajectory in a blue cone.

Time (s)	X (m)	Y (m)	Altitude (m)	Course (°)	Ground Speed (m/s)	Climb Rate (m/s)	Roll (°)	Pitch (°)	Yaw (°)
0	-1.4362e+03	-19950	3000	0	83.3300	0	0	0	0
15	-186	-19950	3000	0	83.3300	0	0	0	0
19.4510	50	-19714	3000	90	83.3300	0	0	0	90
34.4510	50	-19464	3000	90	83.3300	0	0	0	90
38.8990	-186	-18228	3000	180	83.3300	0	0	0	180
50	-1.1111e+03	-18228	3000	180	83.3300	0	0	0	180

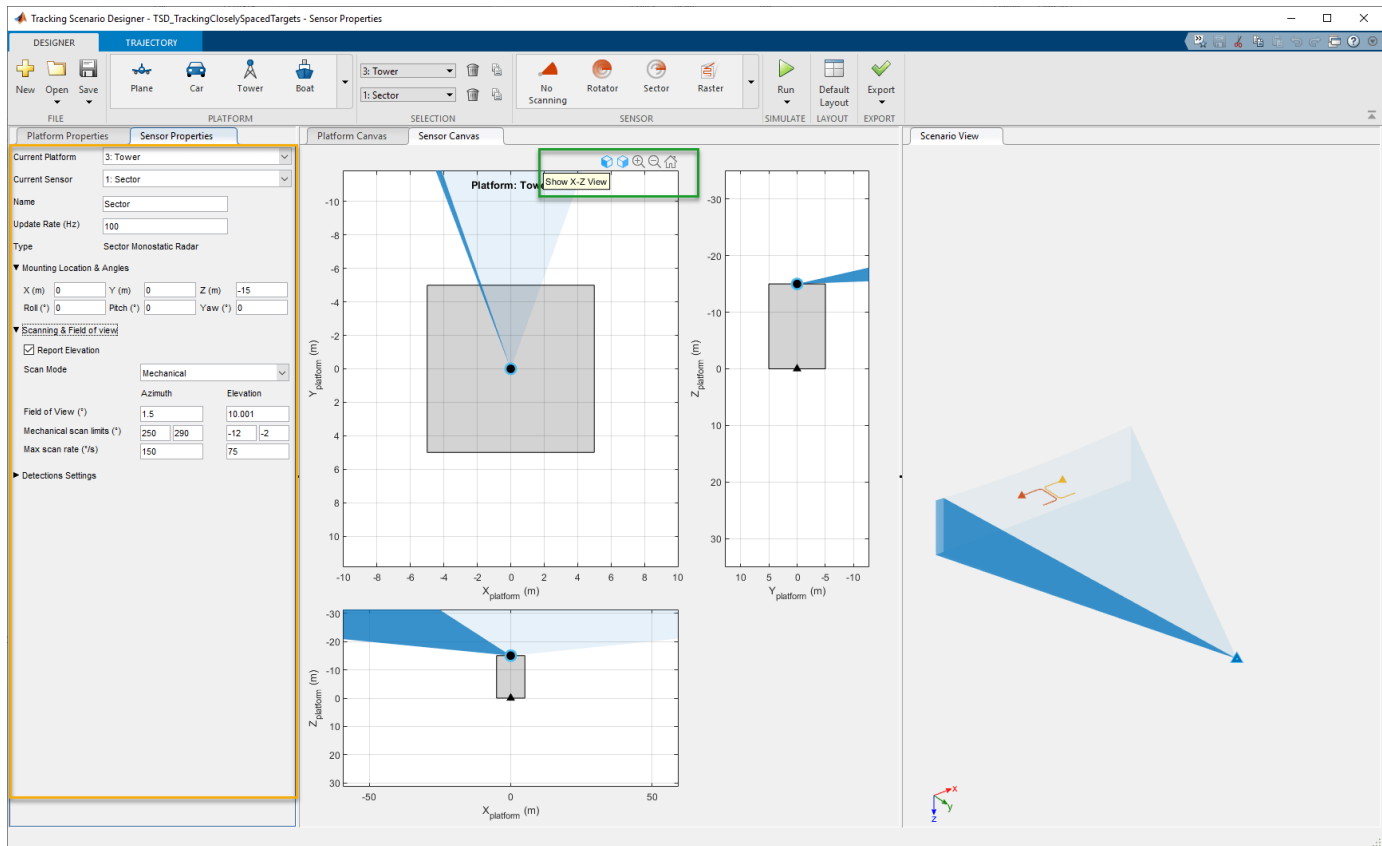
In the property panel (purple), note that the plane is modeled as a point object whose length, width, and height dimensions are zero. By default, its radar cross section is 10 dBsm.

Click on the Trajectory Table button (red) in the toolbar 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 toolbar 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.



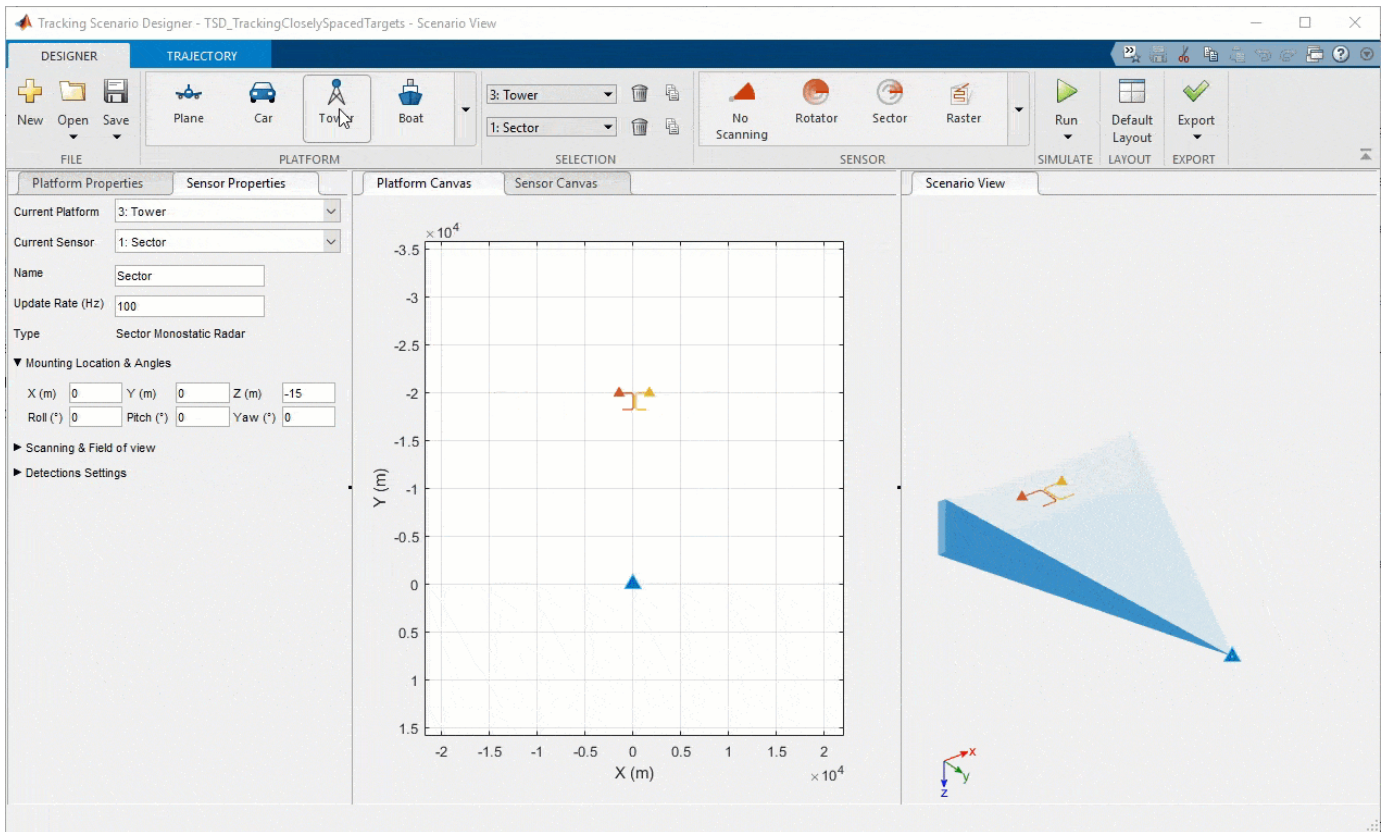
Observe the monostatic radar sensor properties in the panel (yellow). The radar has an update rate of 100 Hz, and a scan range of 40 degrees in azimuth and 10 degrees in elevation. Its field of view is 1.5 deg in azimuth and 10 deg in elevation. Note that we use 10.001 def to ensure the beam is not exactly 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.

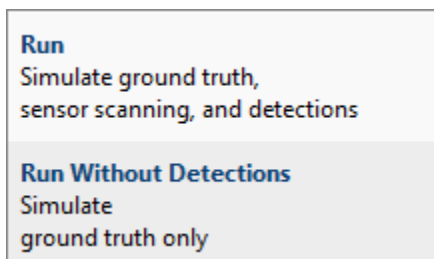
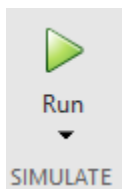
Additionally, you can select a new sensor in the toolstrip gallery and mount it on the tower, or add a new platform in the scene using the platform gallery and then mount a new sensor on it. This is illustrated in the GIF below.



Simulate the scenario

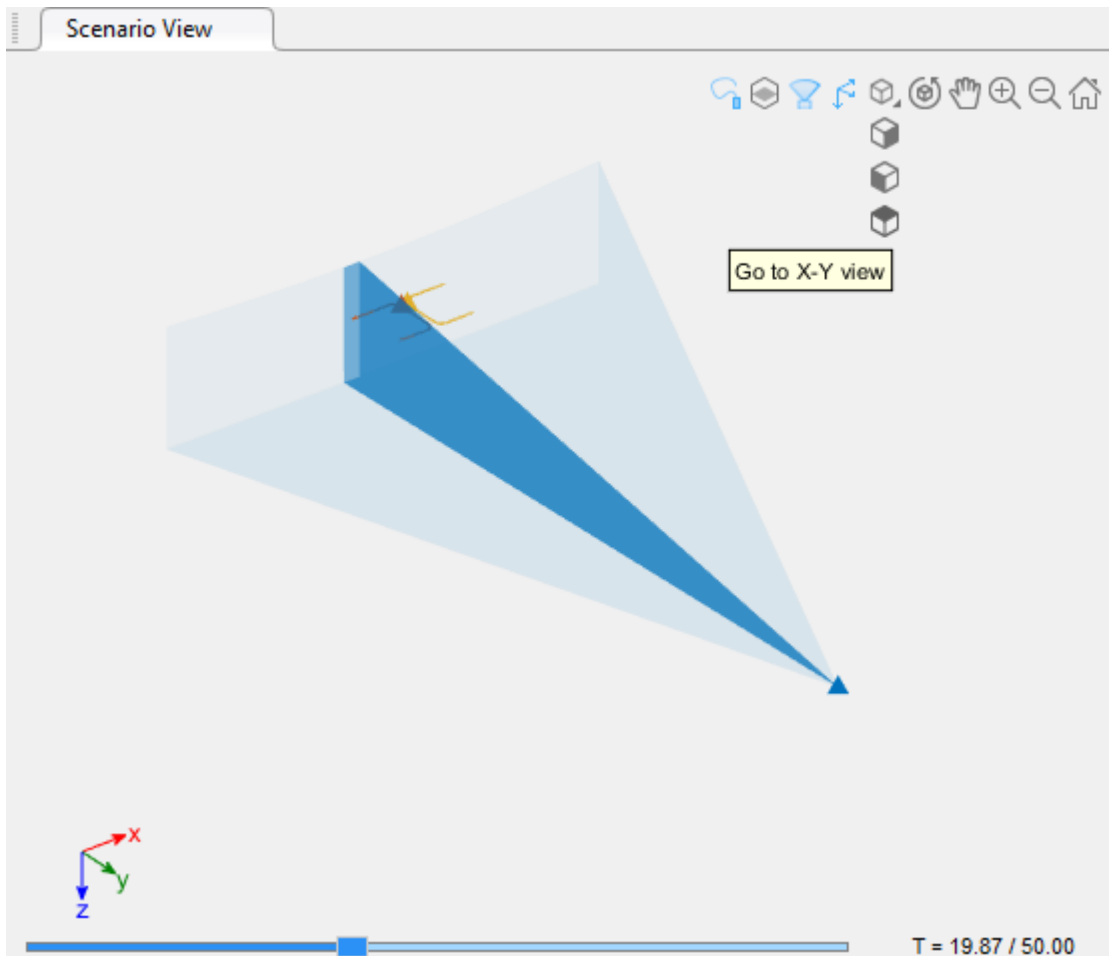
As this scenario is already set up, the next step is to run the simulation to generate the synthetic radar detections.

Click **Run** to run the simulation. Alternatively, use the drop-down to select **Run Without Detections** to simulate the ground truth only.



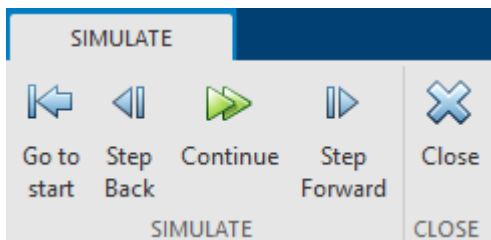
The application goes into simulation mode and the simulation begins automatically. You will see the two planes moving along their trajectories. When an aircraft is detected by the tower, you can see a dark purple marker show at the instant of detection.

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

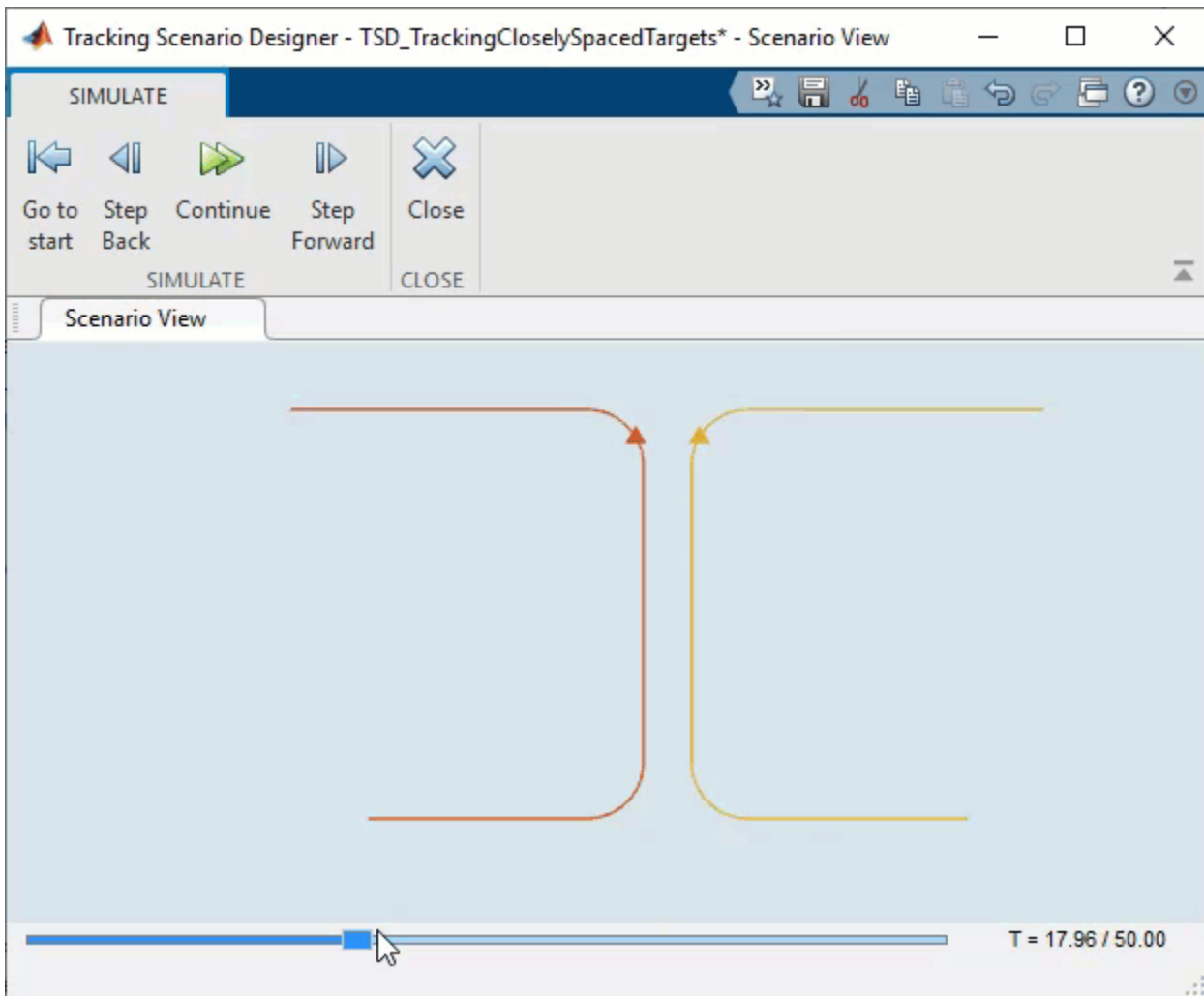
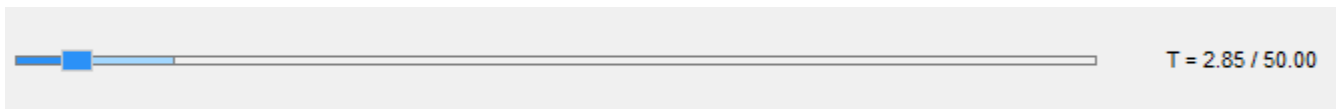
To observe the detection creation as the beam scans a target, use the playback control which allow you to pause, step back, and step forward the simulation.



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.

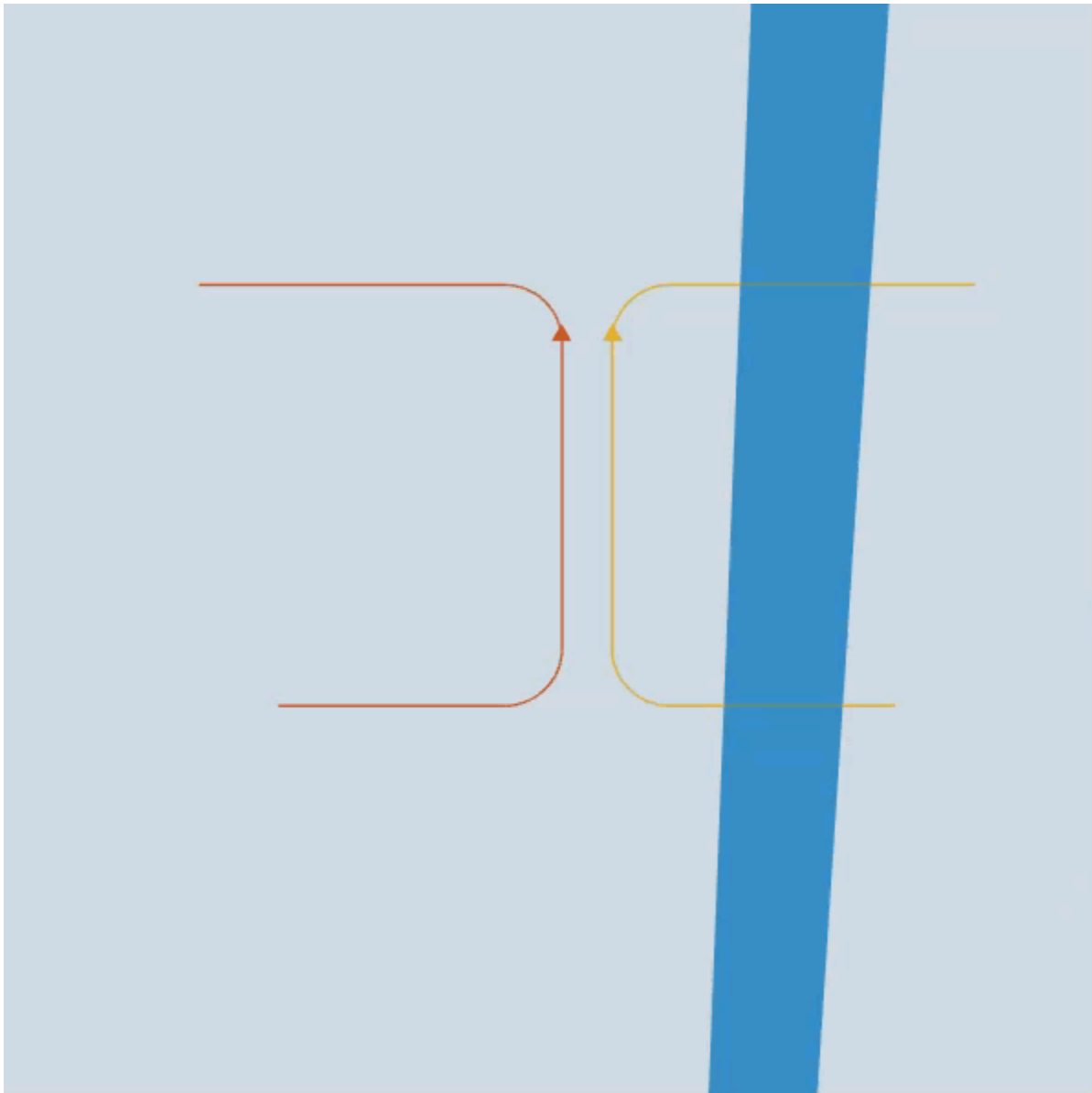
The light blue progress bar shows progress in the data simulation of the scenario. The darker blue rectangle slider shows the progress of the animation of the scenario.

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

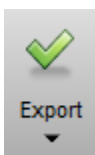
cannot be resolved by the radar.



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.

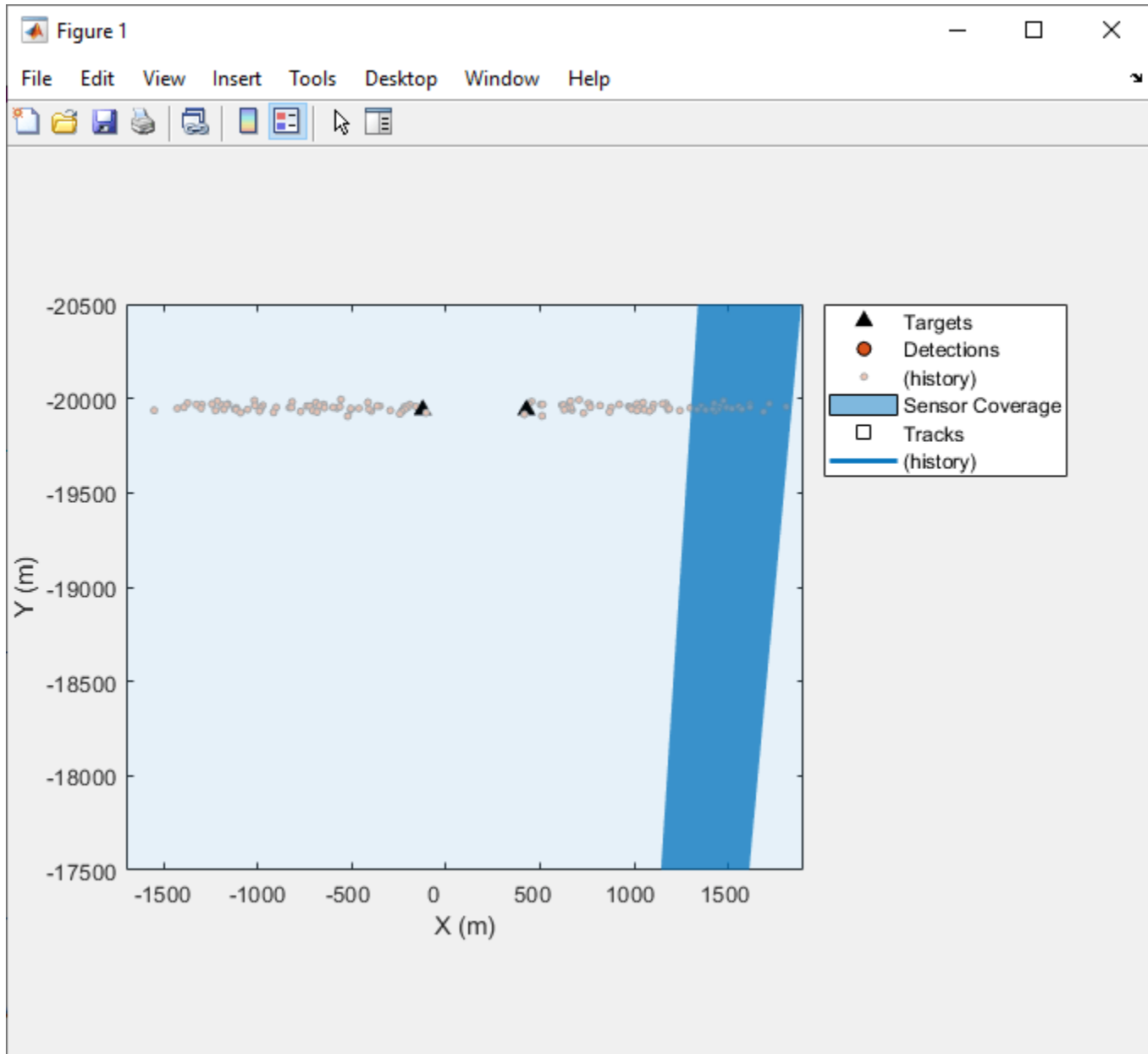


After clicking **Export**, you will see a script that can be saved to a file name of your choice. The current 3D view angles of the Scenario View are also exported to the script.

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” on page 6-168 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

```
% 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

```
% Add a trackPlotter here:
trackp = trackPlotter(tp,'DisplayName','Tracks','ConnectHistory','on','ColorizeHistory','on');
```

3) Define a buffer for detections before the loop

```
% Main simulation loop
detBuffer = {};
```

4) Update the tracker within the loop

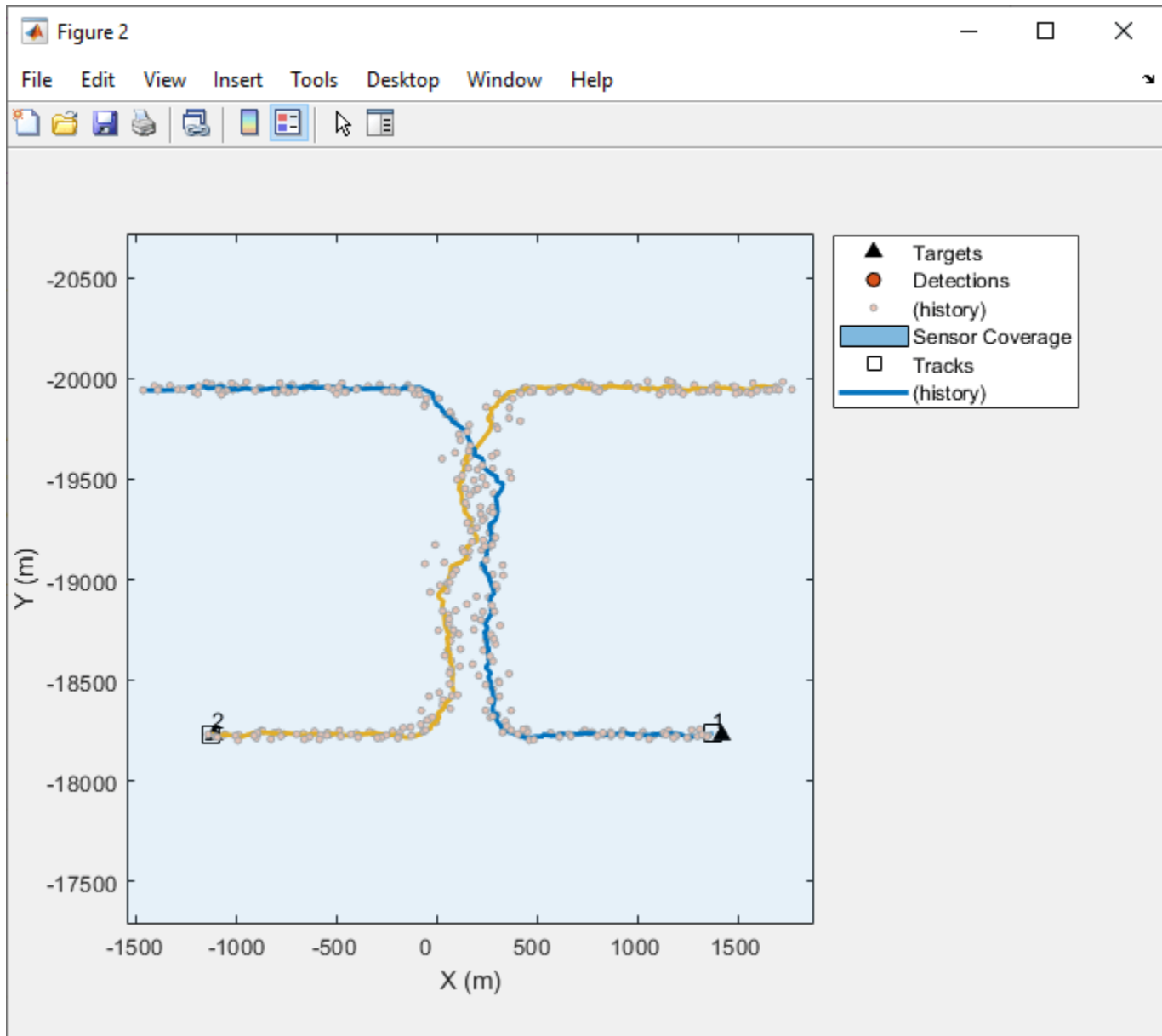
```
% Update your tracker here:
detBuffer = [detBuffer; dets]; %#ok<AGROW>
if configs.IsScanDone
    tracks = tracker(detBuffer,scenario.SimulationTime);
    pos = getTrackPositions(tracks,[1 0 0 0 0 0; 0 0 1 0 0 0 ; 0 0 0 0 1 0]);
    labels = string([tracks.TrackID]);
    detBuffer = {};
end
```

4) Update track plotter

```
% 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.

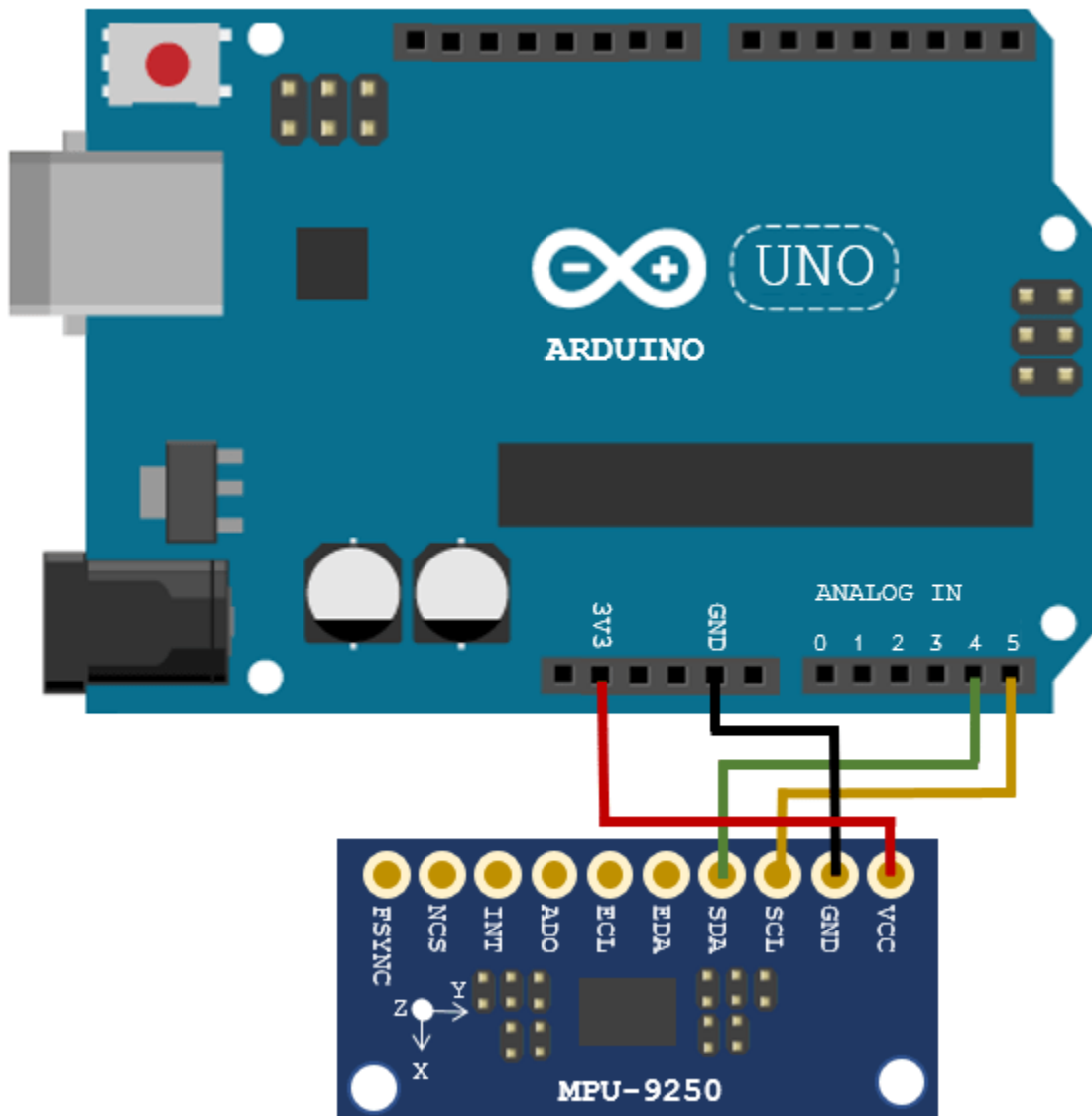
Estimate Orientation with a Complementary Filter and IMU Data

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 F_s 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

- -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);
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 6-445 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
```

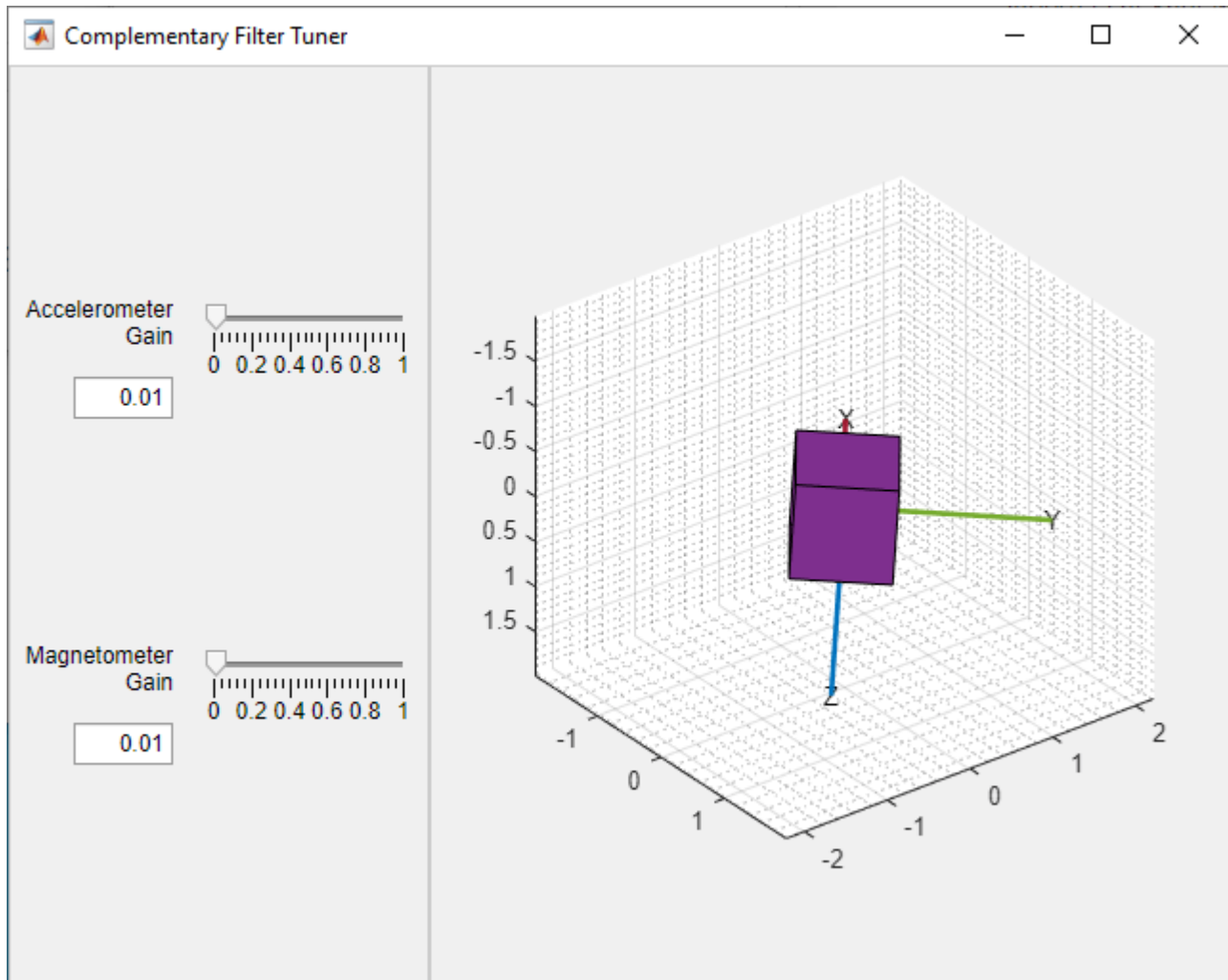
```
tic
else
    idx = 1:samplesPerRead;
    overrunIdx = 1;
end
while true
    if useHW
        [accel, gyro, mag, t, overrun] = imu();
        accel = alignAccelAxes(accel);
        gyro = alignGyroAxes(gyro);
    else
        accel = allAccel(idx,:);
        gyro = allGyro(idx,:);
        mag = allMag(idx,:);
        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
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);
```

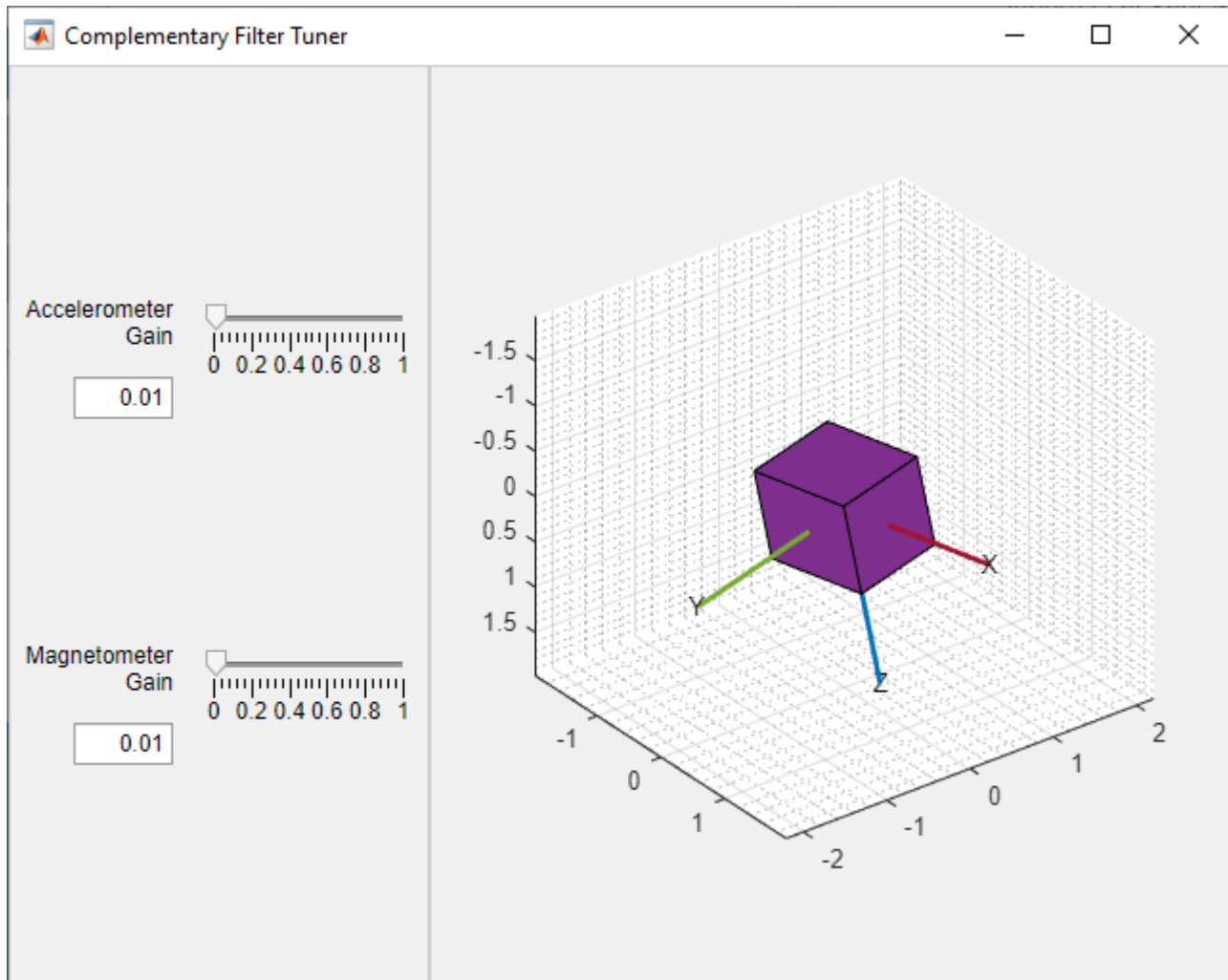
```
        mag = alignMagAxes(mag);
    else
        accel = allAccel(idx,:);
        gyro = allGyro(idx,:);
        mag = allMag(idx,:);
        t = allT(idx,:);
        overrun = allOverrun(overrunIdx,:);

        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
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.

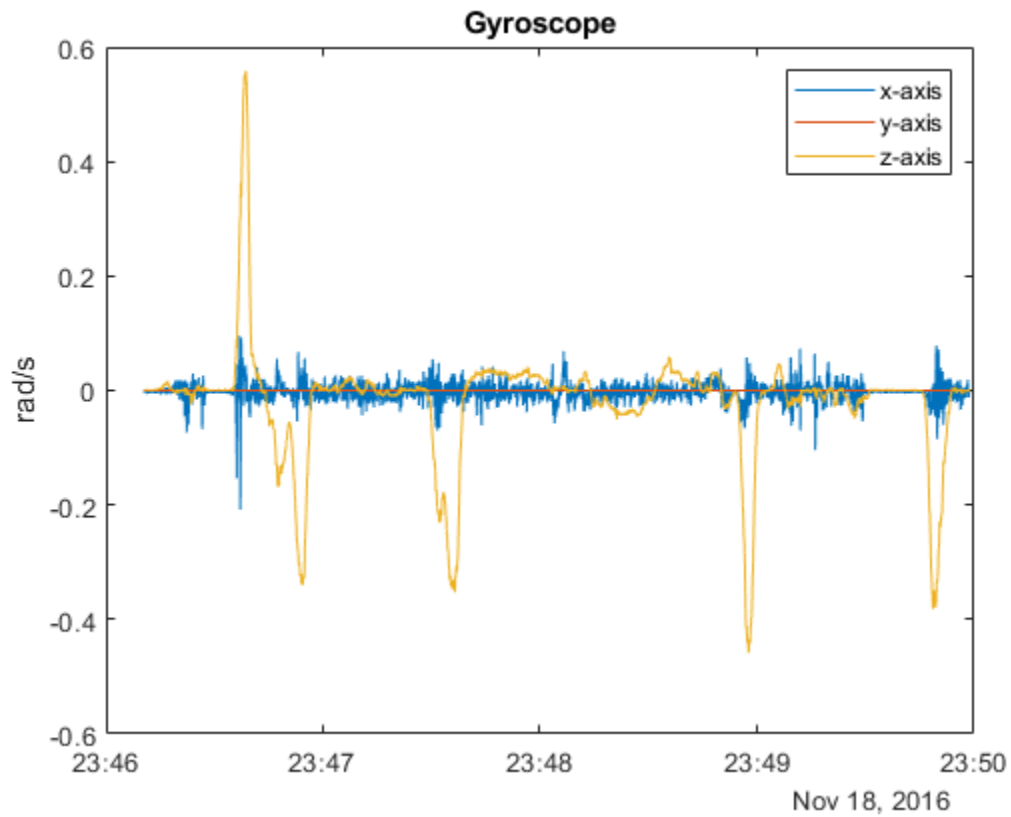
```
load('imuData', 'imuTT')

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.

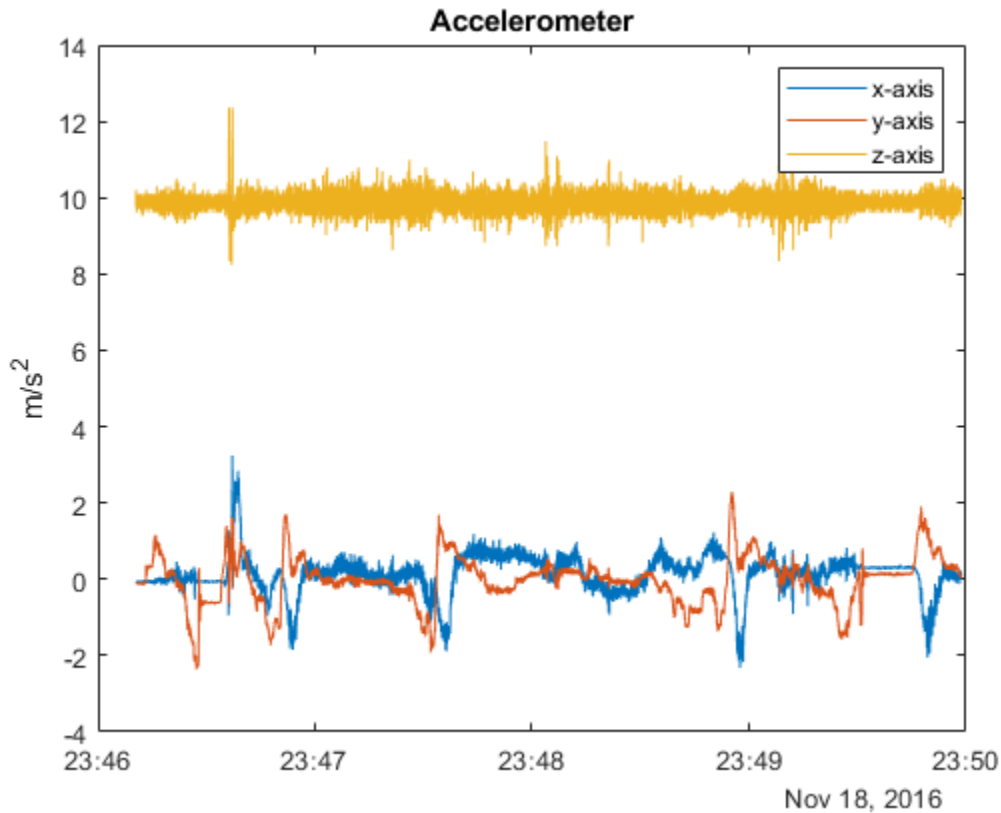
```
figure
plot(time, gyro)
title('Gyroscope')
ylabel('rad/s')
legend('x-axis', 'y-axis', 'z-axis')
```



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.

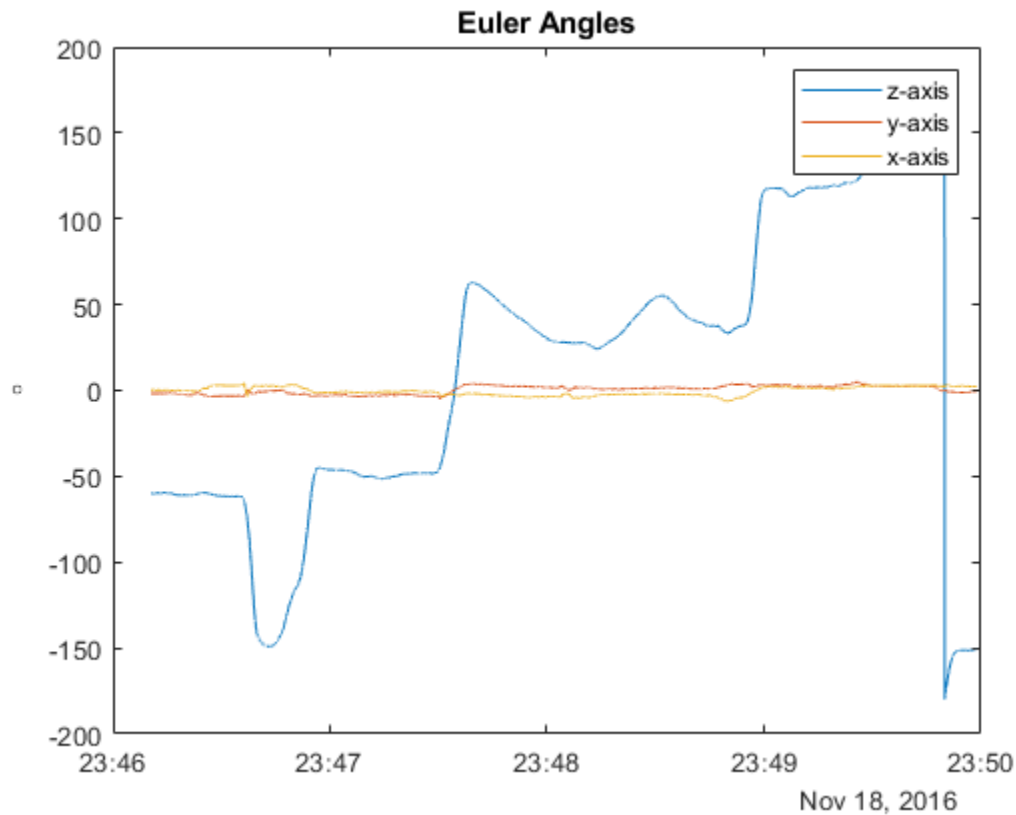
```
figure
plot(time, accel)
title('Accelerometer')
ylabel('m/s^2')
legend('x-axis', 'y-axis', 'z-axis')
```



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')
```

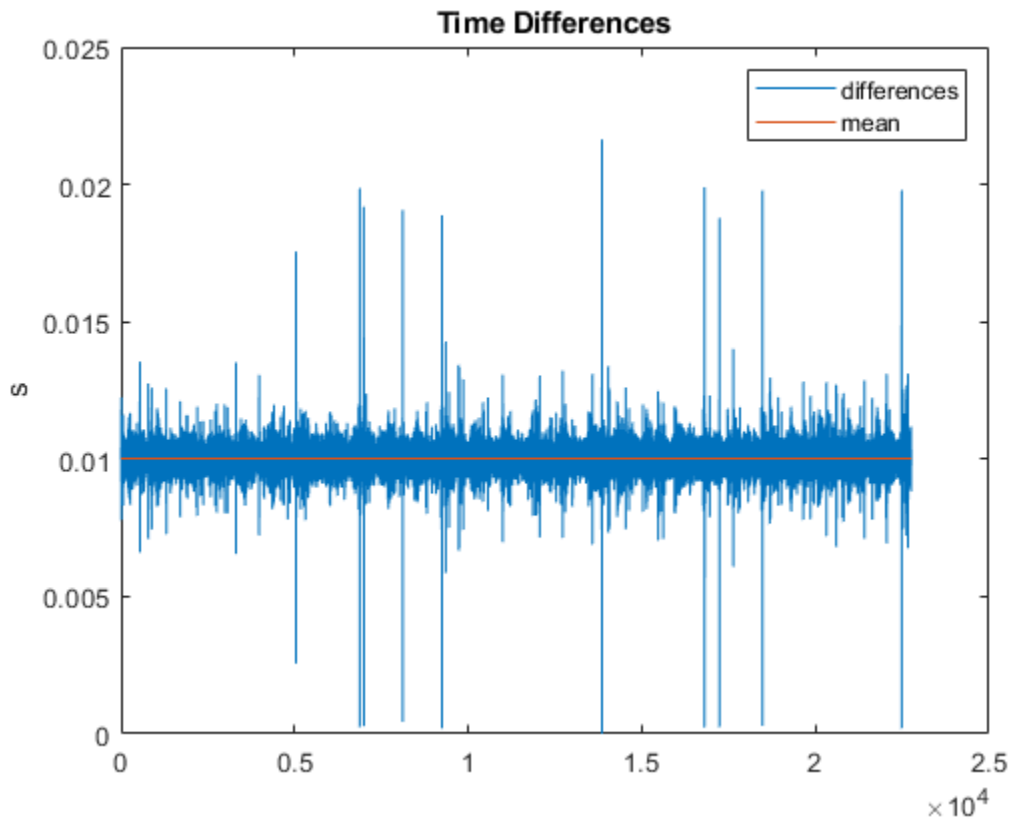


Find the Sampling Rate of the Logged Data

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')
```



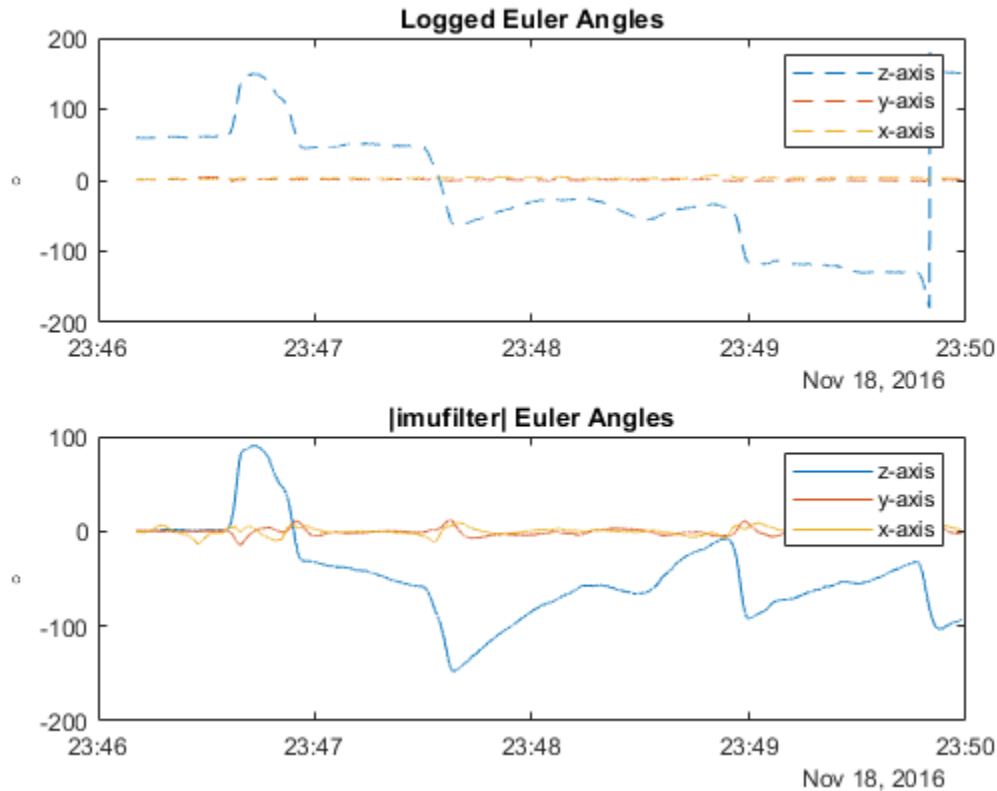
Compare the Transformed Logged Quaternion to the `imufilter` Quaternion

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')
```

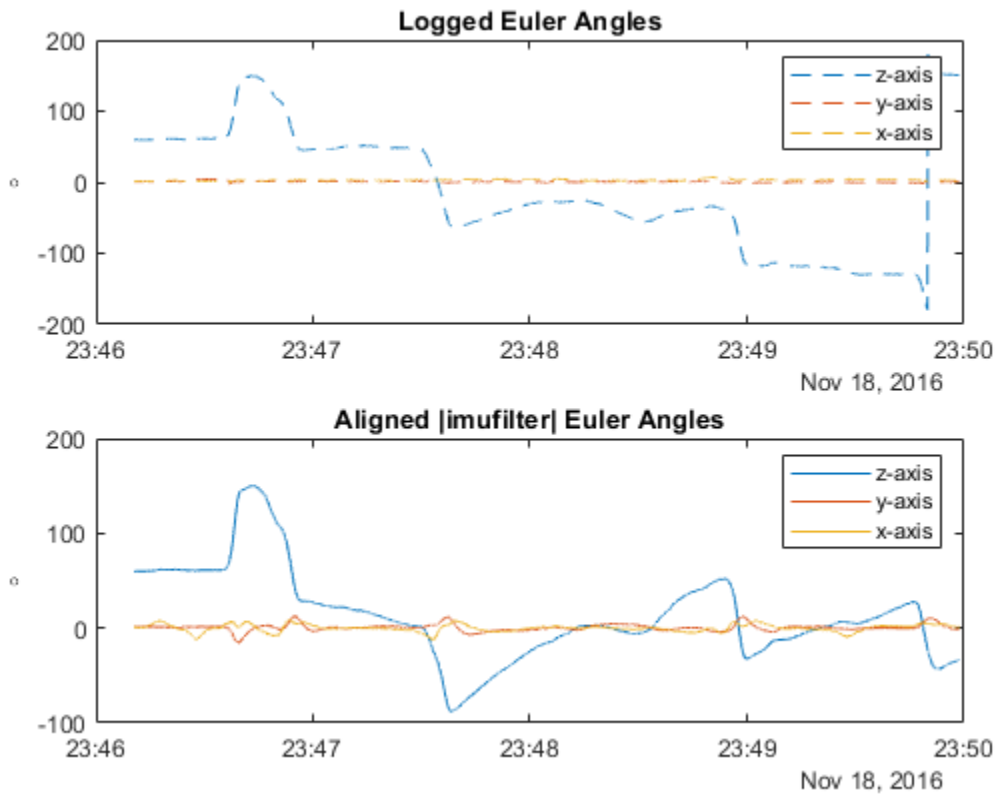


Align the Logged Orientation and imufilter Orientation

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);
```

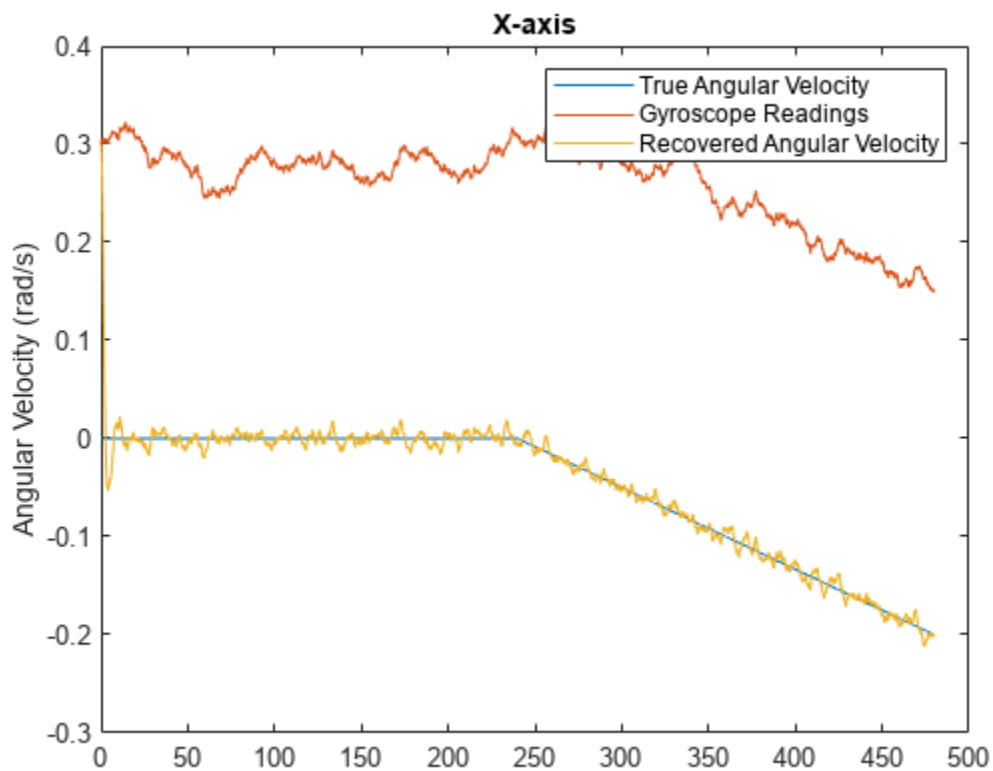
Plot the ground-truth angular velocity, the gyroscope readings, and the recovered angular velocity for each axis.

The angular velocity returned from the `imufilter` compensates for the effect of the gyroscope bias over time and converges to the true angular velocity.

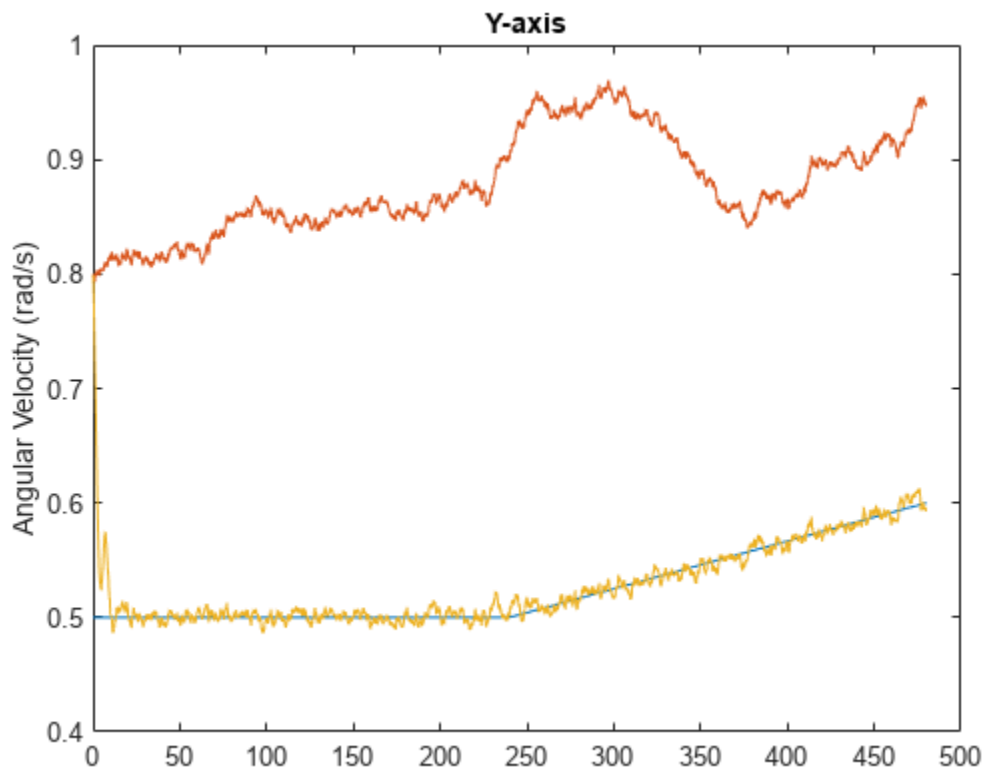
```
time = (0:numSamples-1)/fs;

figure(1)
plot(time,angVelBody(:,1), ...
    time,gyroReadingsBody(:,1), ...
    time,angVelBodyRecovered(:,1))
title('X-axis')
```

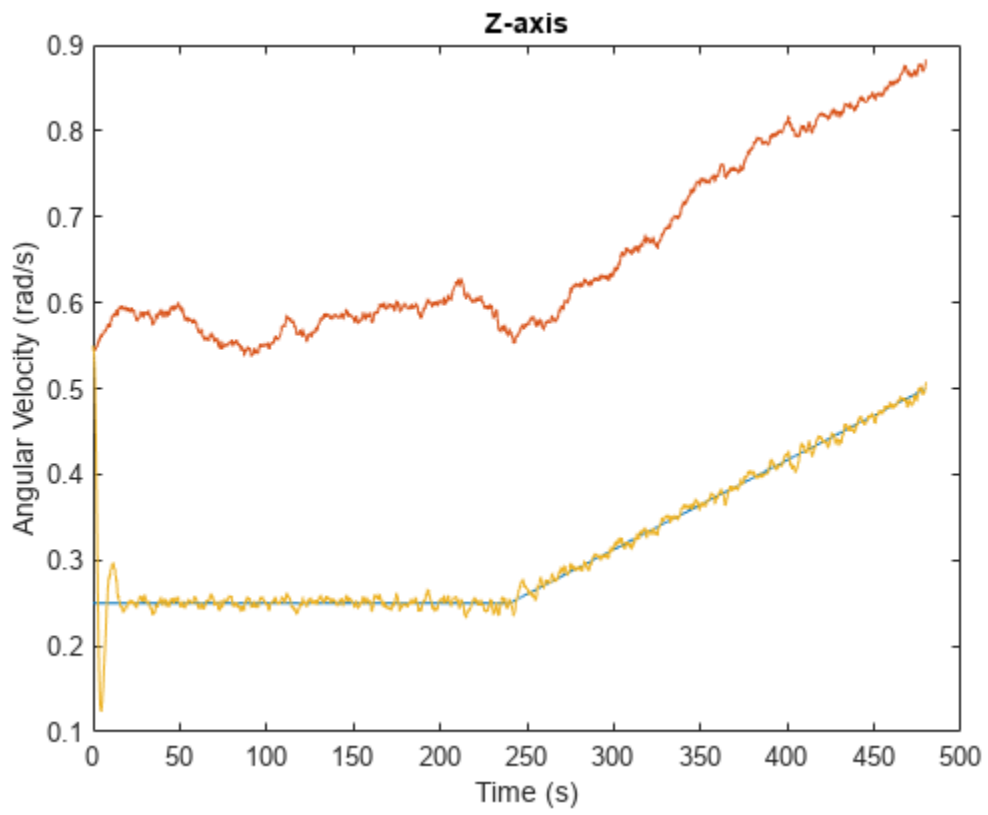
```
legend('True Angular Velocity', ...  
      'Gyroscope Readings', ...  
      'Recovered Angular Velocity')  
ylabel('Angular Velocity (rad/s)')
```



```
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

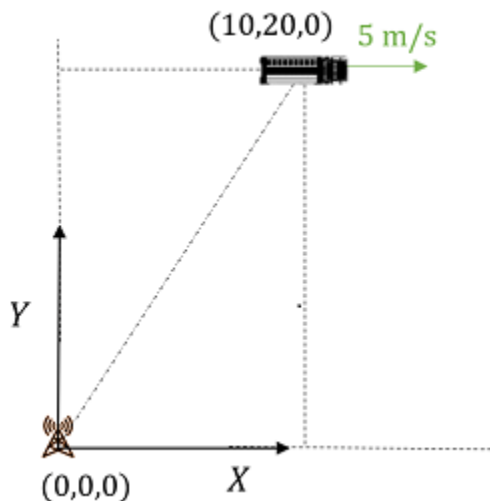
These examples show how to convert actual detections in the native format of the sensor into `objectDetection` objects. `objectDetection` is the standard input format for most filters and trackers in Sensor Fusion and Tracking toolbox. The six examples progressively show how to set up `objectDetection` with varied tracking scenarios.

- Example 1 on page 6-435 configures the detection in a stationary rectangular frame.
- Example 2 on page 6-436 configures the detection in a moving rectangular frame.
- Example 3 on page 6-438 configures the detection in a moving spherical frame.
- Example 4 on page 6-440 shows how to express detections obtained by consecutive rotations.
- Example 5 on page 6-441 shows how to configure 3-D detections.
- Example 6 on page 6-443 shows how to configure classified detections.

An `objectDetection` report must contain the basic detection information: Time and Measurement. It can also contain other key properties, including `MeasurementNoise`, `SensorIndex`, `ObjectClassID`, `ObjectClassParameters`, `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`. Examples 2 through 5 focus on how to correctly set up `MeasurementParameters`. The last example shows how to set up `ObjectClassParameters`.

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 `MeasurementNoise`, `SensorIndex`, `ObjectClassID`, and `ObjectAttributes` for the `objectDetection` object. For example, assuming the standard 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)
```

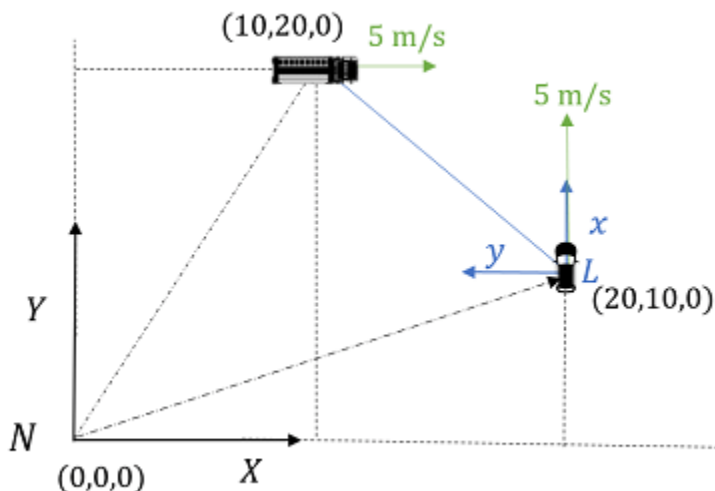
```
detect1 =
  objectDetection with properties:
```

```

      Time: 0
  Measurement: [6x1 double]
  MeasurementNoise: [6x6 double]
    SensorIndex: 1
   ObjectClassID: 0
ObjectClassParameters: []
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
  ObjectClassParameters: []
  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`.

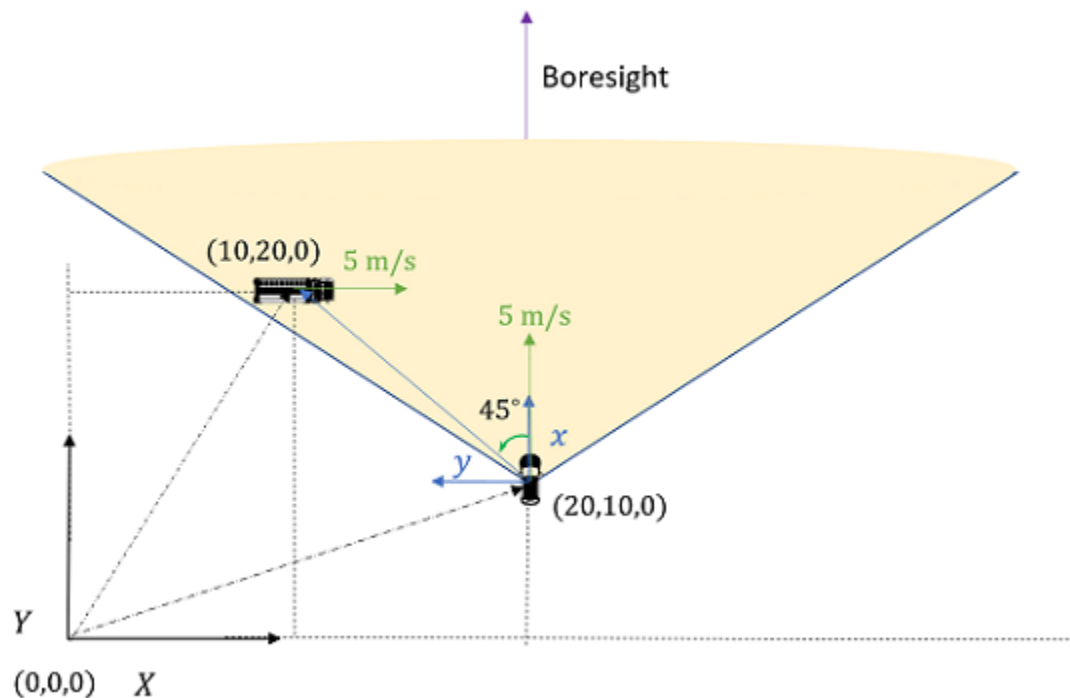
```
state2 = [10;5;20;0;0;0]; % [x;vx;y;vy;z;vz]
cvmeas2 = cvmeas(state2,MP2)% [x;y;z;vx;vy;vz]
```

```
cvmeas2 = 6x1
```

```
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 $[\text{azimuth}; \text{elevation}; \text{range}; \text{range-rate}]$, 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
  ObjectClassParameters: []
  MeasurementParameters: [1x1 struct]
      ObjectAttributes: {}
```

Verify the results using cvmeas. The results agree with measurement3.

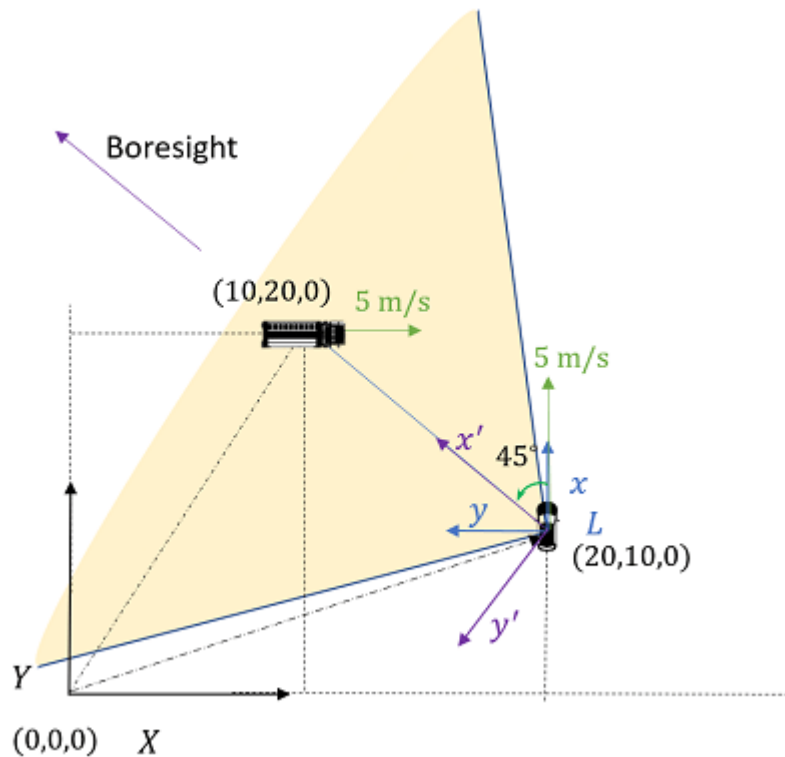
```
state3 = [10;5;20;0;0;0]; % [x;vx;y;vy;z;vz]
cvmeas3 = cvmeas(state3,MP3) % [az;el;rng;rr]
```

```
cvmeas3 = 4x1

    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 6-436. `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
  ObjectClassParameters: []
  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]

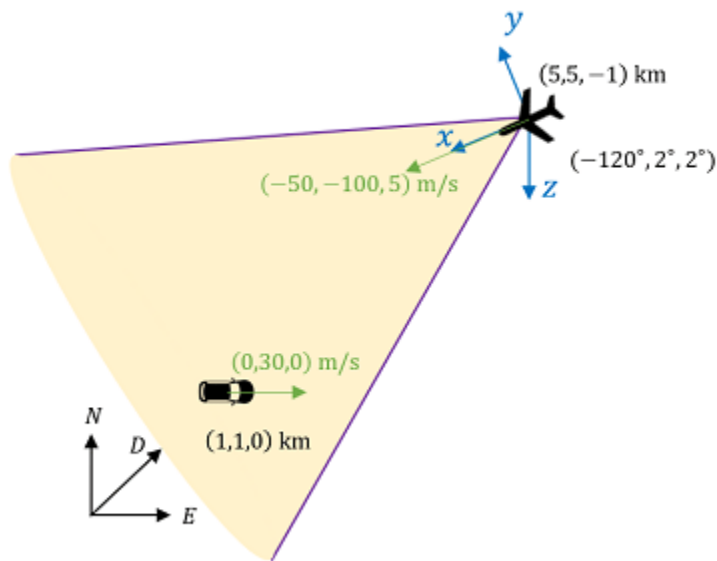
cvmeas4 = 4x1

    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 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_e15 = measurement5(2)
meas_e15 = 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
        ObjectClassParameters: []
        MeasurementParameters: [1x1 struct]
        ObjectAttributes: {}

```

Example 6: Classified Detections

Consider a vision tracking scenario where camera frames feed into an object detector. In such cases, detections often provide a classification of the object. Consider an object detector that output bounding box detections and classifies objects into the following classes {'Car', 'Pedestrian', 'Bicycle'}. The statistics of the detector are captured by its confusion matrix C. Create a detection with bounding box measurement, 'Pedestrian' classification, and confusion matrix C as defined below.

```

C = [0.9 0.05 0.05; ...
    0.05 0.9 0.05; ...
    0.05 0.05 0.9];
ClassID = 2; % Pedestrian
ClassParams = struct('ConfusionMatrix', C);

boundingbox = [250 140 300 400]; % bounding box in top left width height coordinates
detection = objectDetection(0, boundingbox, ObjectClassID=ClassID, ObjectClassParameters=struct(

```

```
detection =  
  objectDetection with properties:  
      Time: 0  
      Measurement: [250 140 300 400]  
      MeasurementNoise: [4x4 double]  
      SensorIndex: 1  
      ObjectClassID: 2  
      ObjectClassParameters: [1x1 struct]  
      MeasurementParameters: {}  
      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
- Navigation Toolbox™ or Sensor Fusion and Tracking Toolbox™

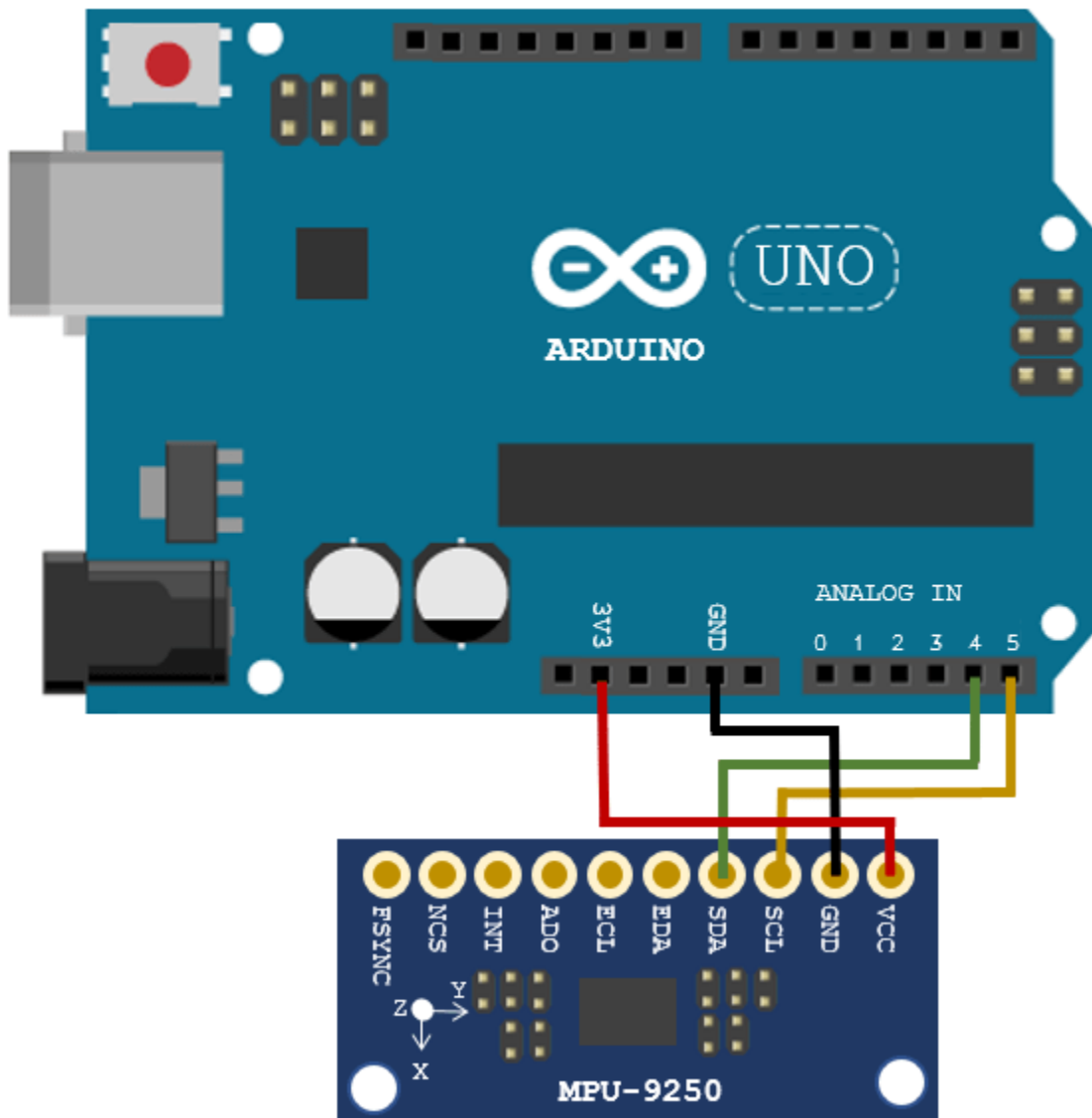
Required Hardware

- 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 Magnetometer Distortions

Fusion algorithms use magnetometer readings which need to be compensated for magnetic distortions. In general, two effects exist: Hard iron distortions and soft iron distortions. To learn more about this distortion, refer to “More About” (Navigation Toolbox). These distortions can be corrected by using the correction values that can be determined using these steps:

- 1 Rotate the sensor from 0 to 360 degree along each axis.
- 2 Use the `magcal` (Navigation Toolbox) function, as shown below, to obtain the correction coefficients.

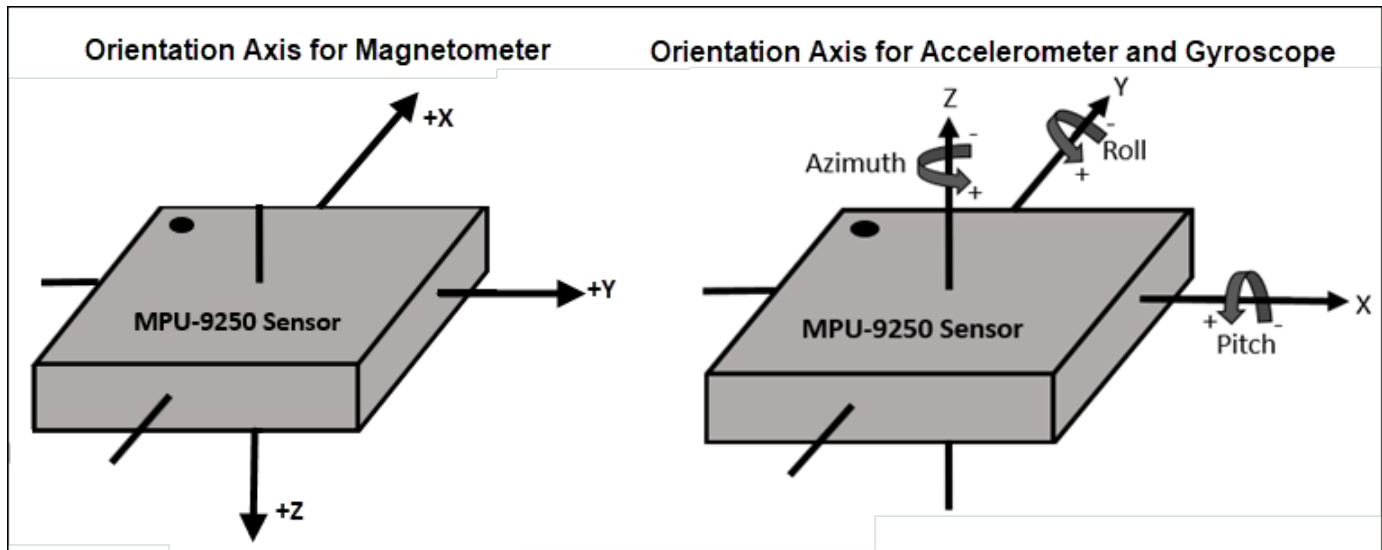
```
ts = tic;
stopTimer = 50;
magReadings=[];
while(toc(ts) < stopTimer)
    % Rotate the sensor along x axis from 0 to 360 degree.
    % Take 2-3 rotations to improve accuracy.
    % For other axes, rotate along that axes.
    [accel,gyro,mag] = read(imu);
    magReadings = [magReadings;mag];
end

[A, b] = magcal(magReadings); % A = 3x3 matrix for soft iron correction
                             % b = 3x1 vector for hard iron correction
```

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. See "Custom Tuning of Fusion Filters" on page 6-658 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.

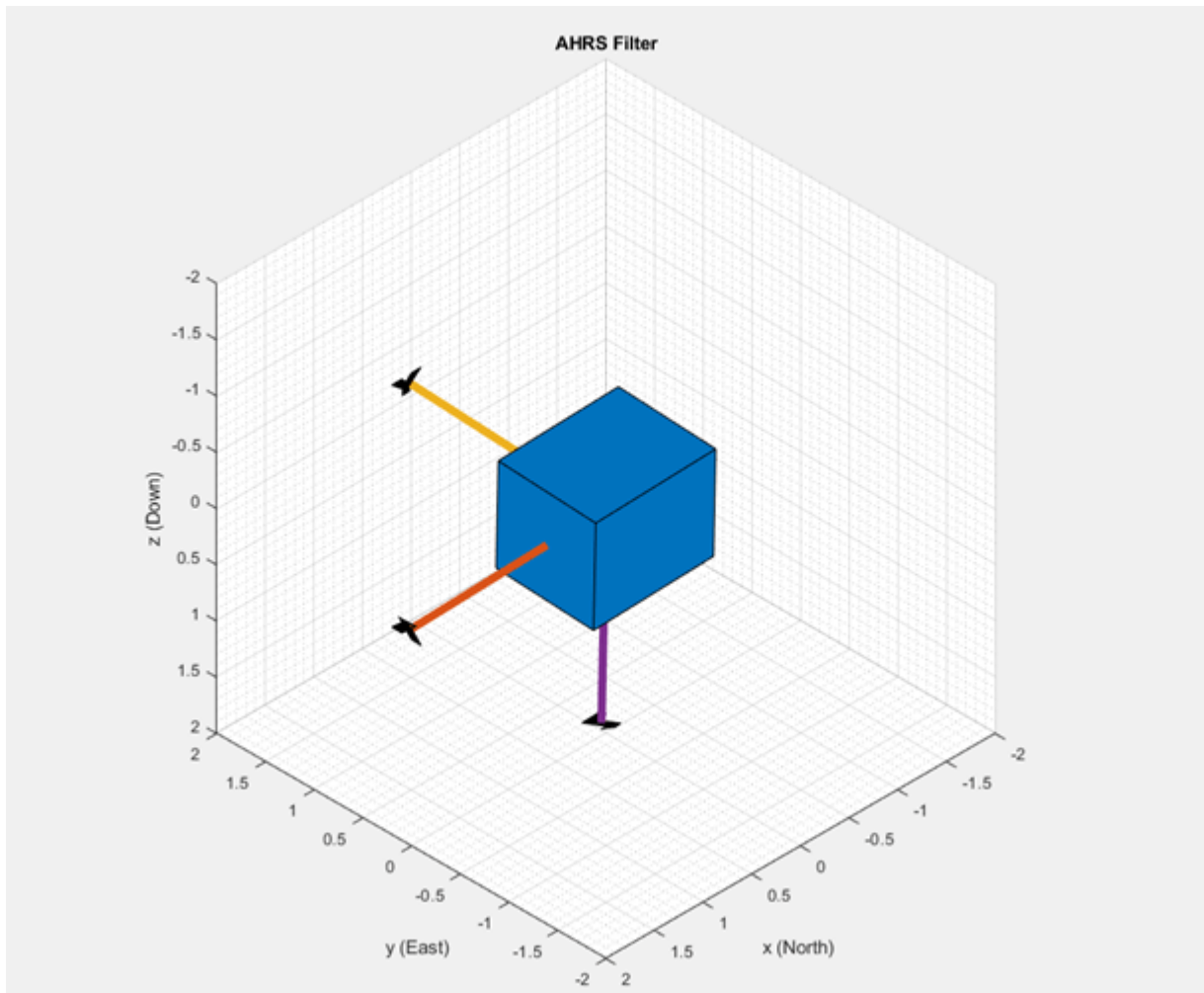
The following code snippets use `ahrsfilter` system object to determine orientation of the sensor and creates a figure which gets updated as you move the sensor. The sensor has to be stationary, before the start of this example.

```
% 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,'AccelerometerNoise',AccelerometerNoiseMPU9250);
stopTimer = 100;
```

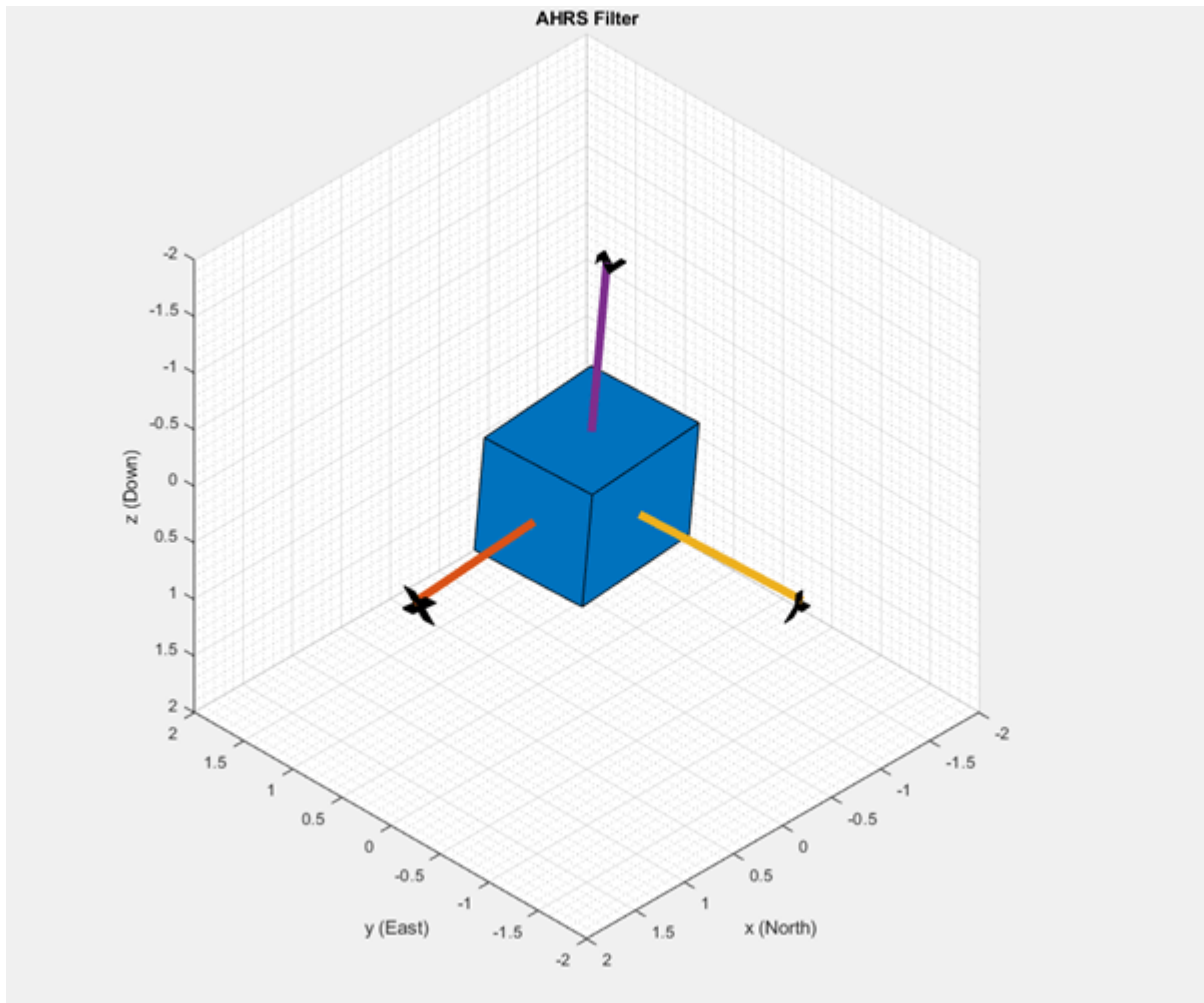
While the below code is getting executed, slowly move the sensor and check if the motion in the figure matches the motion of the sensor.

```
% The correction factors A and b are obtained using magcal function as explained in one of the
% previous sections, 'Compensating Magnetometer Distortions'.
magx_correction = b(1);
magy_correction = b(2);
magz_correction = b(3);
ts = tic;
while(toc(ts) < stopTimer)
    [accel,gyro,mag] = read(imu);
    % Align coordinates in accordance with NED convention
    accel = [-accel(:,2), -accel(:,1), accel(:,3)];
    gyro = [gyro(:,2), gyro(:,1), -gyro(:,3)];
    mag = [mag(:,1)-magx_correction, mag(:, 2)- magy_correction, mag(:,3)-magz_correction] * A;
    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'});
```

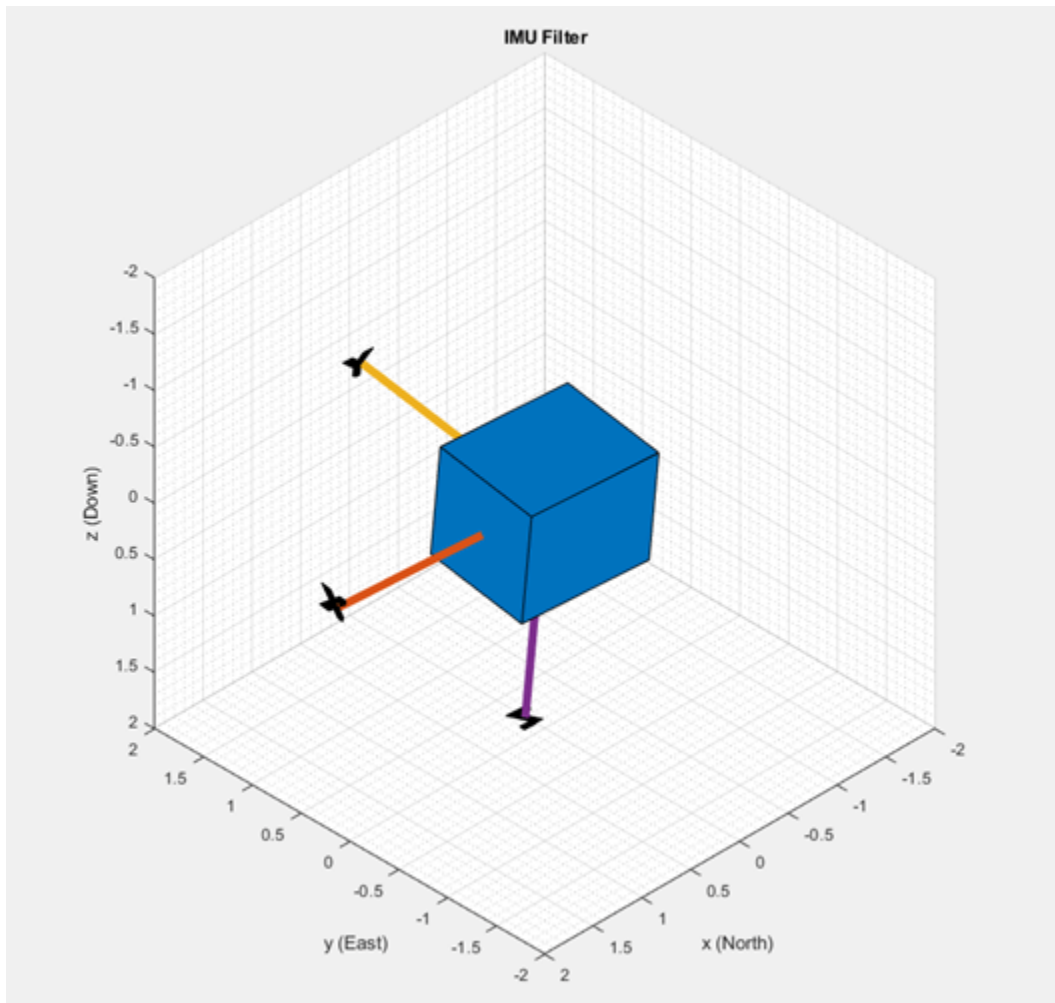
```
FUSE = imufilter('SampleRate',imu.SampleRate, 'GyroscopeNoise',GyroscopeNoiseMPU9250,'AccelerometerNoise',imu.AccelerometerNoiseMPU9250,stopTimer=100;
```

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.

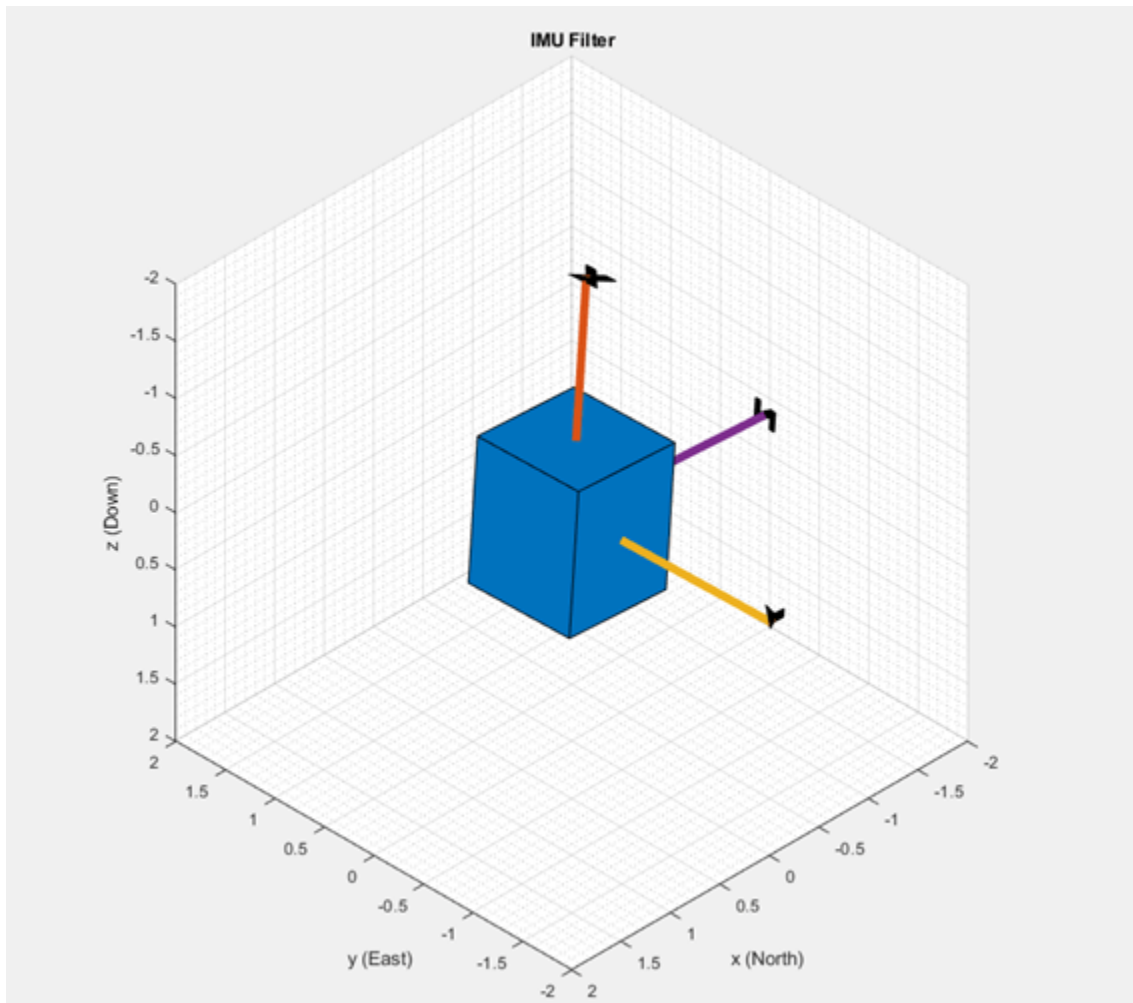
```
% Use imufilter to estimate orientation and update the viewer as the sensor moves for time specific;
tic;
while(toc < stopTimer)
    [accel,gyro] = read(imu);
    accel = [-accel(:,2), -accel(:,1), accel(:,3)];
    gyro = [gyro(:,2), gyro(:,1), -gyro(:,3)];
    rotators = FUSE(accel,gyro);
    for 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" on page 6-48 example for more details.

```
displayMessage(['This section uses \slecompass \rmfunction to determine orientation of the sensor
  \rmsystem object. Move the sensor to visualize orientation of the sensor in the figure window
  'Estimate Orientation using Ecompass algorithm.'])
viewer = HelperOrientationViewer('Title',{'Ecompass Algorithm'});
% Use ecompass algorithm to estimate orientation and update the viewer as the sensor moves for t.
% The correction factors A and b are obtained using magcal function as explained in one of the
% previous sections, 'Compensating Magnetometer Distortions'.
magx_correction = b(1);
magy_correction = b(2);
magz_correction = b(3);
stopTimer = 100;
tic;
while(toc < stopTimer)
    [accel,~,mag] = read(imu);
    accel = [-accel(:,2), -accel(:,1), accel(:,3)];
```



```
mag = [mag(:,1)-magx_correction, mag(:, 2)- magy_correction, mag(:,3)-magz_correction] * A;  
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);  
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.

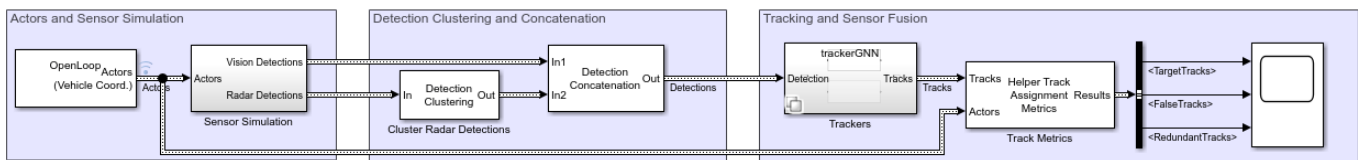
Introduction

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.

Track Simulated Vehicles Using GNN and JPDA Trackers



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`.

Global Nearest Neighbor Multi Object Tracker

The GNN Tracker block creates and manages the tracks of stationary and moving objects. The block initializes, confirms, predicts, corrects, and deletes tracks. Inputs to the tracker are detection reports from one or more sensors. Detections are assigned to tracks using a global nearest neighbor (GNN) criterion. A detection is assigned to only one track and when no assignment is possible, the tracker creates a new track.

[Source code](#)

Tracker Management	Track Logic	Port Setting
Tracker identifier:	<input type="text" value="uint32(0)"/>	
Filter initialization function:	<input type="text" value="initcvekf"/>	
Assignment algorithm name:	Munkres	
Threshold for assigning detections to tracks:	<input type="text" value="[1, Inf] * 50.0"/>	
Maximum number of tracks:	<input type="text" value="20"/>	
Maximum number of sensors:	<input type="text" value="8"/>	
Out-of-sequence measurements handling:	Terminate	
State parameters source:	Property	
Track state parameters:	<input type="text" value="struct"/>	
Simulate using:	Interpreted execution	

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`.

Joint Probabilistic Data Association Multi Object Tracker

The JPDA Tracker block creates and manages the tracks of stationary and moving objects. The block initializes, confirms, predicts, corrects, and deletes tracks. Inputs to the tracker are detection reports from one or more sensors. Detections are assigned to tracks using a Joint Probabilistic Data Association (JPDA) criterion. A detection can be assigned to multiple tracks with appropriate probabilistic weights. When a detection cannot be assigned to a track with sufficient probabilistic weight, the tracker creates a new track.

[Source code](#)

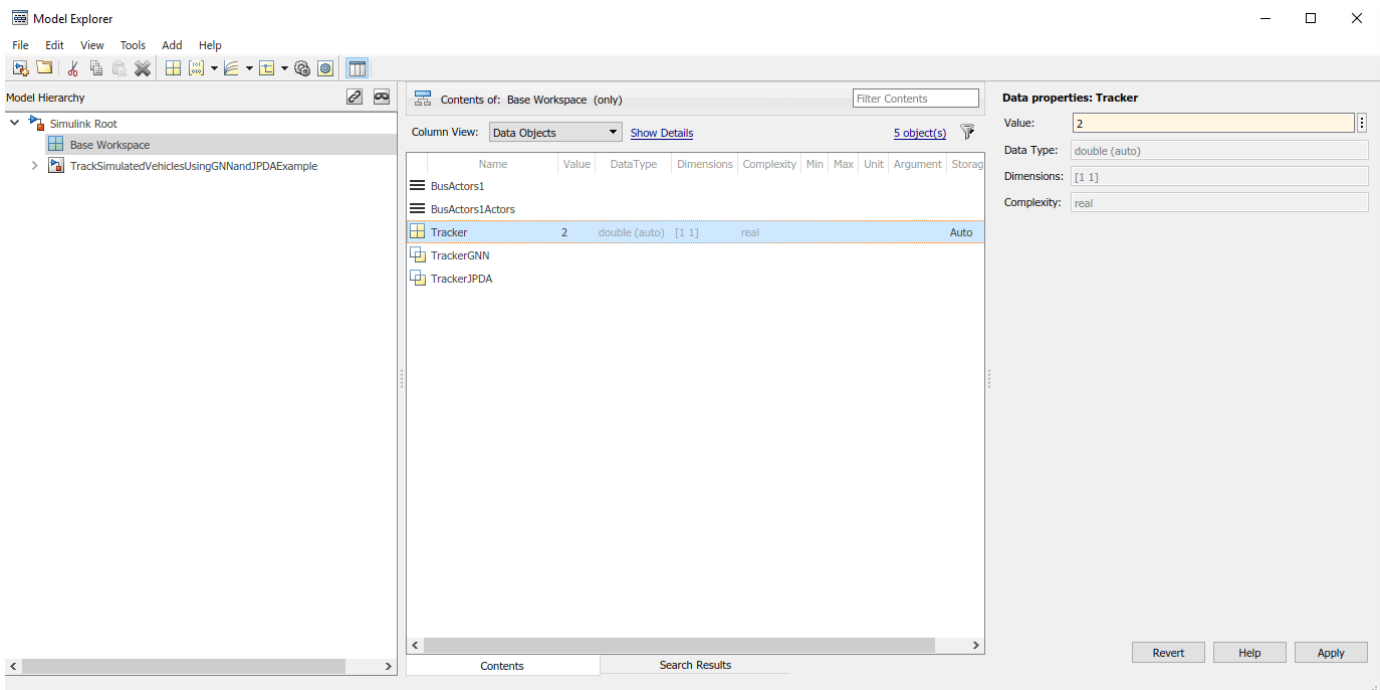
Tracker Management	Assignment	Track Logic	Port Setting
Tracker identifier:	<input type="text" value="uint32(0)"/>		
Filter initialization function:	<input type="text" value="initcvekf"/>		
Value of k for k-best JPDA:	<input type="text" value="inf"/>		
Feasible joint events generation function name:	<input type="text" value="jpdaEvents"/>		
Maximum number of tracks:	<input type="text" value="20"/>		
Maximum number of sensors:	<input type="text" value="8"/>		
Absolute tolerance between time stamps of detections:	<input type="text" value="1e-5"/>		
Out-of-sequence measurements handling:	Terminate		
State parameters source:	Property		
Track state parameters:	struct		
Simulate using:	Interpreted execution		

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

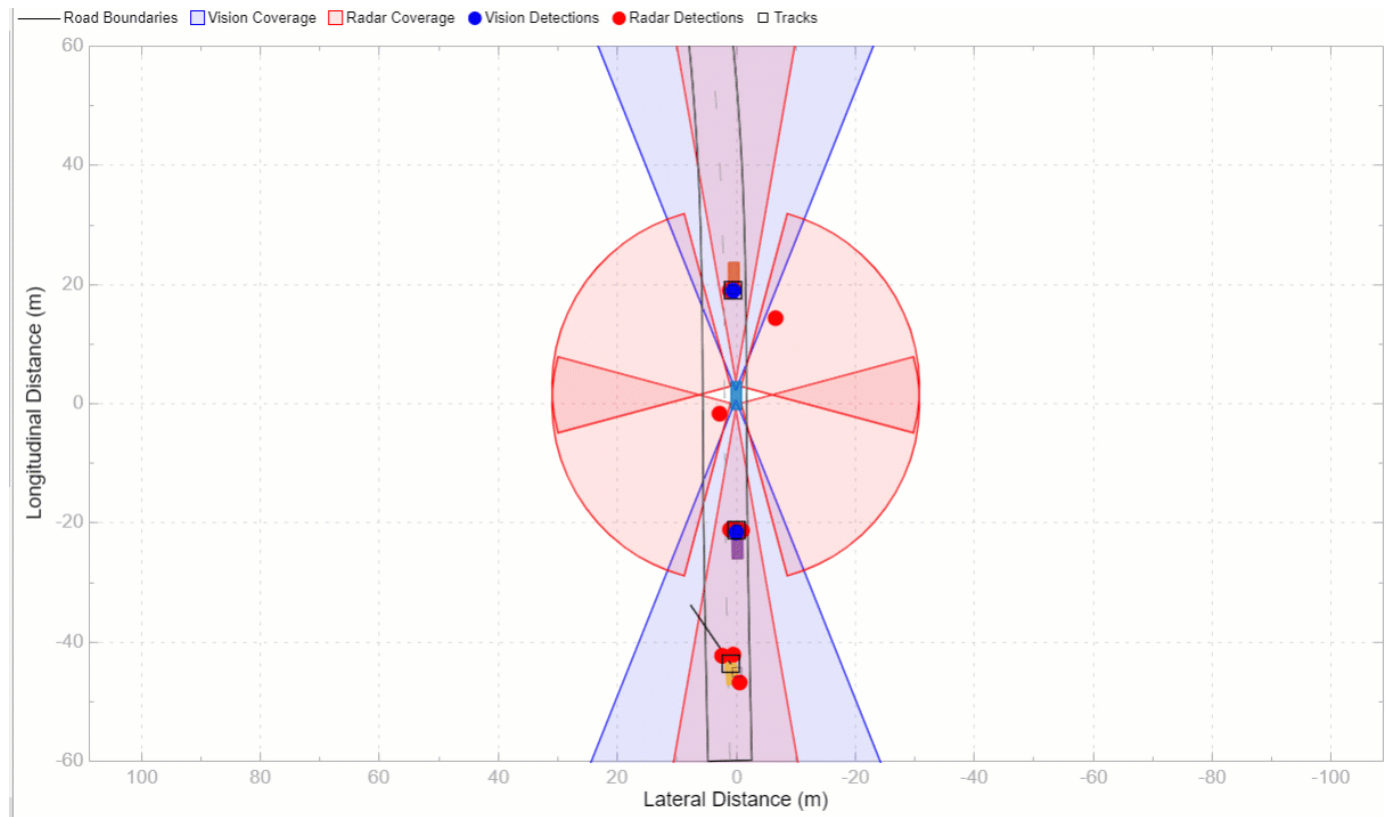
The Track Metrics is implemented using a MATLAB System (Simulink) block. The code for the block is defined by a helper class, HelperTrackMetrics.

Results

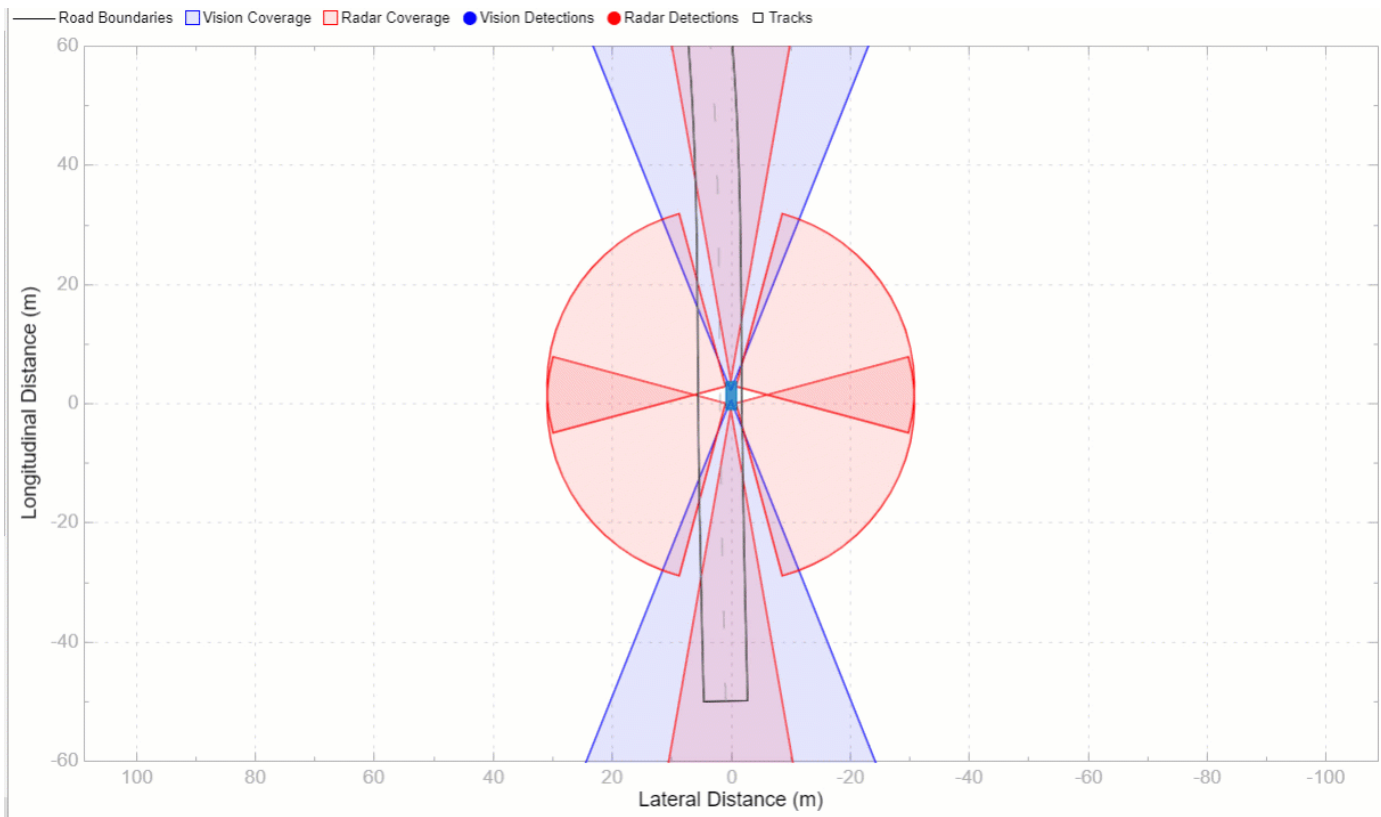
To visualize the results, use the Bird's-Eye Scope. The Bird's-Eye Scope is a model-level visualization tool via a menu provided on the Simulink model toolbar. After opening the scope, click **Find Signals** to set up the signals. Then run the simulation to display the actors, vision and radar detections, 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.



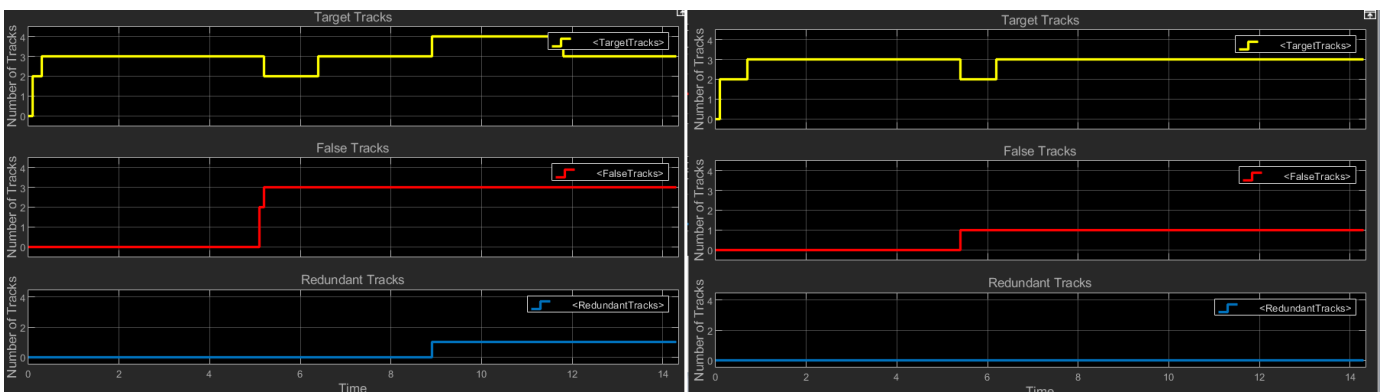
The above figure shows the tracking results using synthetic radar and vision data with `trackerGNN`. You can see that the tracker maintained tracks throughout the tracking region.



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 one redundant track was 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.

Track-to-Track Fusion for Automotive Safety Applications in Simulink

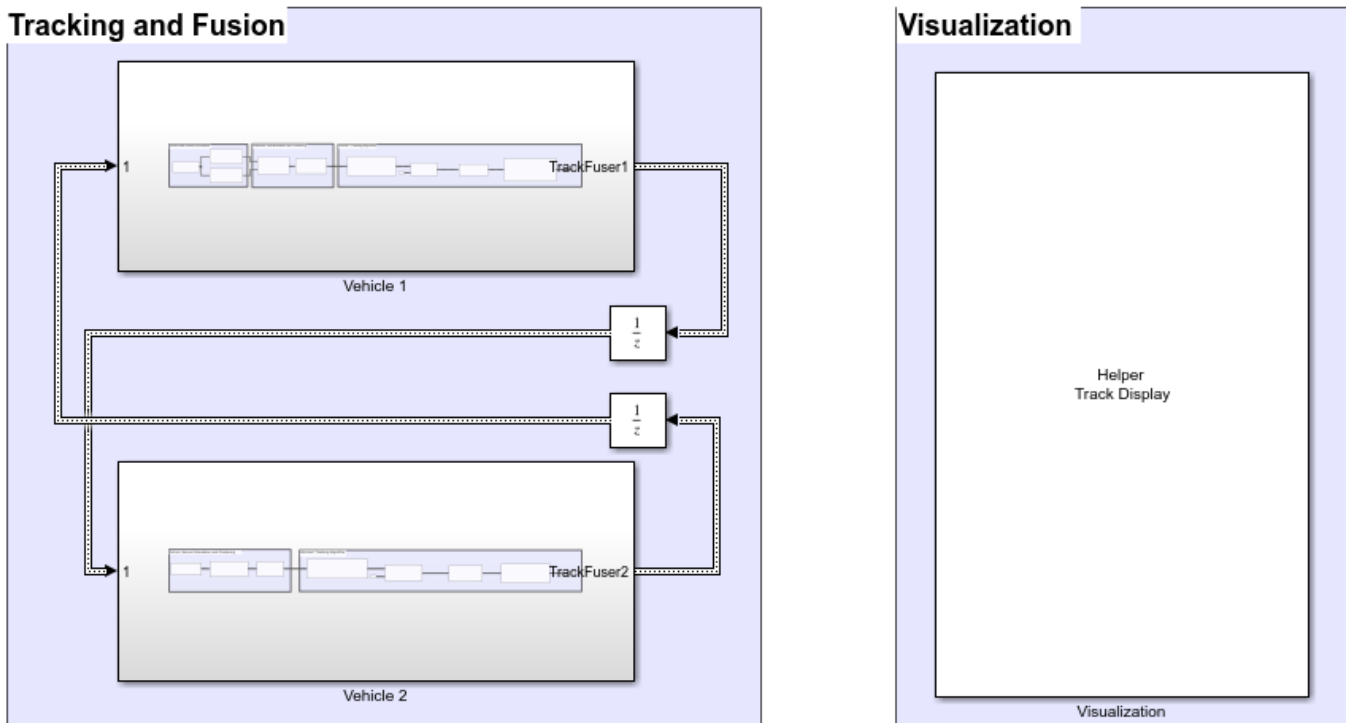
This example shows how to perform track-to-track fusion in Simulink® with Sensor Fusion and Tracking Toolbox™. In the context of autonomous driving, the example illustrates how to build a decentralized tracking architecture using a Track-To-Track Fuser block. In the example, each vehicle performs tracking independently as well as fuses tracking information received from other vehicles. This example closely follows the “Track-to-Track Fusion for Automotive Safety Applications” on page 6-693 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

Hierarchical "Track-to-Track" Fusion Example

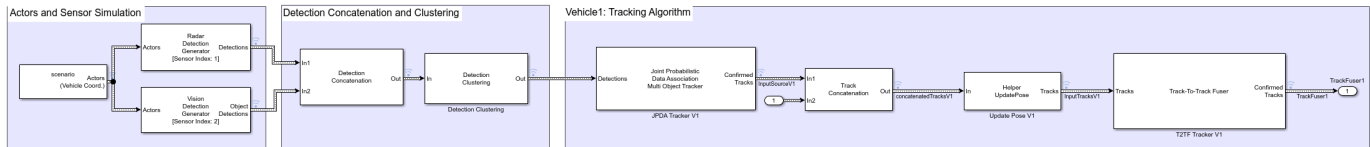


Prior to running this example, the drivingScenario object was used to create the same scenario defined in “Track-to-Track Fusion for Automotive Safety Applications” on page 6-693. The roads and actors from this scenario were then saved to the scenario object file TrackToTrackFusionScenario.mat.

Tracking and Fusion

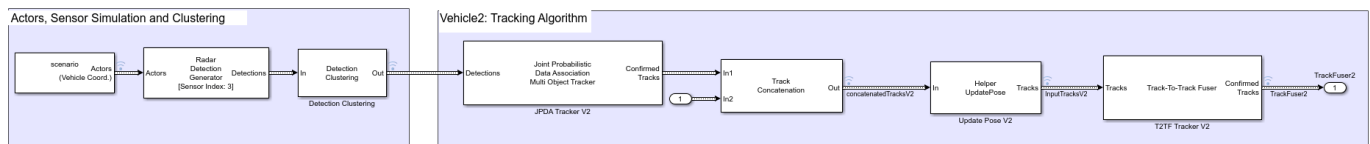
In the Tracking and Fusion section of the model there are two subsystems that implement the target tracking and fusion capabilities of Vehicle 1 and Vehicle 2 in this scenario.

Vehicle 1 Subsystem



This subsystem includes the Scenario Reader (Automated Driving Toolbox) block that reads the actor pose data from the saved file. The block converts the actor poses from the world coordinates of the scenario into ego vehicle coordinates. The actor poses are streamed on a bus generated by the block. The actor poses are used by the Sensor Simulation subsystem, which generates radar and vision detections. These detections are 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 track fuser of the other vehicle. 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 this 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 Track-To-Track Fuser T2TF Tracker V1 block fuse the local vehicle tracks with the tracks received from the track fuser of the other vehicle. After each update, the track fuser on each vehicle broadcasts its fused tracks to be fed into the update of the track fuser of the other vehicle in the next time stamp.

Vehicle 2 Subsystem



Vehicle 2 subsystem follows a similar setup as the Vehicle 1 subsystem.

Visualization

The Visualization block is implemented using the MATLAB System block and is defined using the HelperTrackDisplay block. The block uses RunTimeObject parameters Out, Confirmed Tracks, Tracks and Confirmed Tracks of Detection Clustering, JPDA Tracker V1, Update Pose V1, T2TF Tracker V1 blocks respectively for vehicle 1 and RunTimeObject parameters Out, Confirmed Tracks, Tracks and Confirmed Tracks of Detection Clustering, JPDA Tracker V2, Update Pose V2, T2TF Tracker V2 blocks respectively for vehicle 2 to display their outputs. See "Access Block Data During Simulation" (Simulink) for further information on how to access block outputs during simulation.

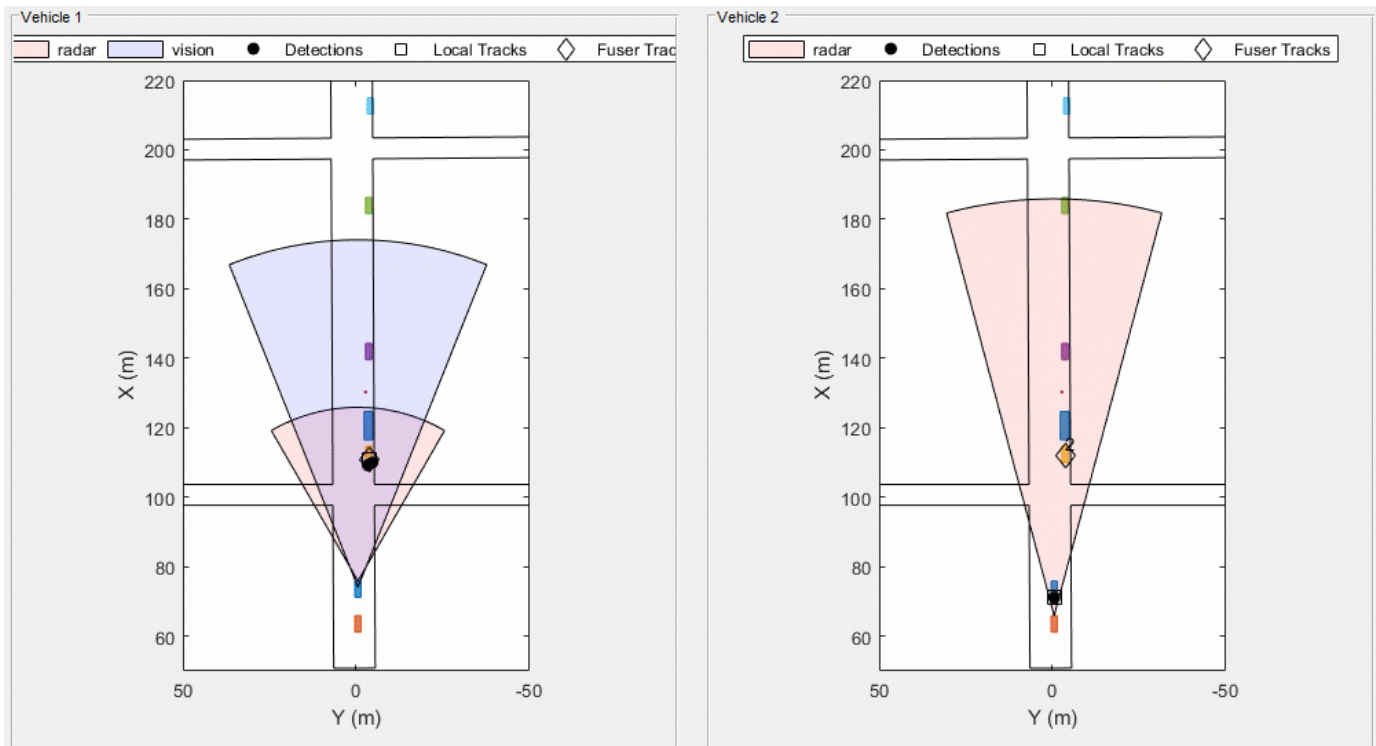
Results

After running the model, you can visualize the results. This animation shows the results for this simulation.

The visualization includes two panels. The left panel shows the detections, local tracks, and fused tracks that vehicle 1 generated during the simulation and represents the situational awareness of vehicle 1. The right panel shows the situational awareness of vehicle 2.

The recorded detections are represented by black circles. The local and fused tracks from vehicle 1 are represented by a square and a diamond, respectively. The local and fused tracks from vehicle 2 are represented by a solid black square and a diamond. At the start of simulation, vehicle 1 detects vehicles parked on the right side of the street, and tracks associated with the parked vehicles are confirmed. Currently, vehicle 2 only detects vehicle 1 which is immediately in front of it. As the simulation continues, the confirmed tracks from vehicle 1 are broadcast to the fuser on vehicle 2. After fusing the tracks, vehicle 2 becomes aware of the objects prior to detecting these objects on its own. Similarly, vehicle 2 tracks are broadcast to vehicle 1. Vehicle 1 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 cars on the right side of the street is detected and tracked by vehicle 1. Vehicle 2 first becomes aware of the pedestrian by fusing the track from Vehicle 1 at around 0.8 seconds. It takes vehicle 2 roughly 3 seconds before it starts detecting the pedestrian using its own sensor. The ability to track a pedestrian based on inputs from vehicle 1 allows vehicle 2 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 use 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 point 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.

```
% 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];

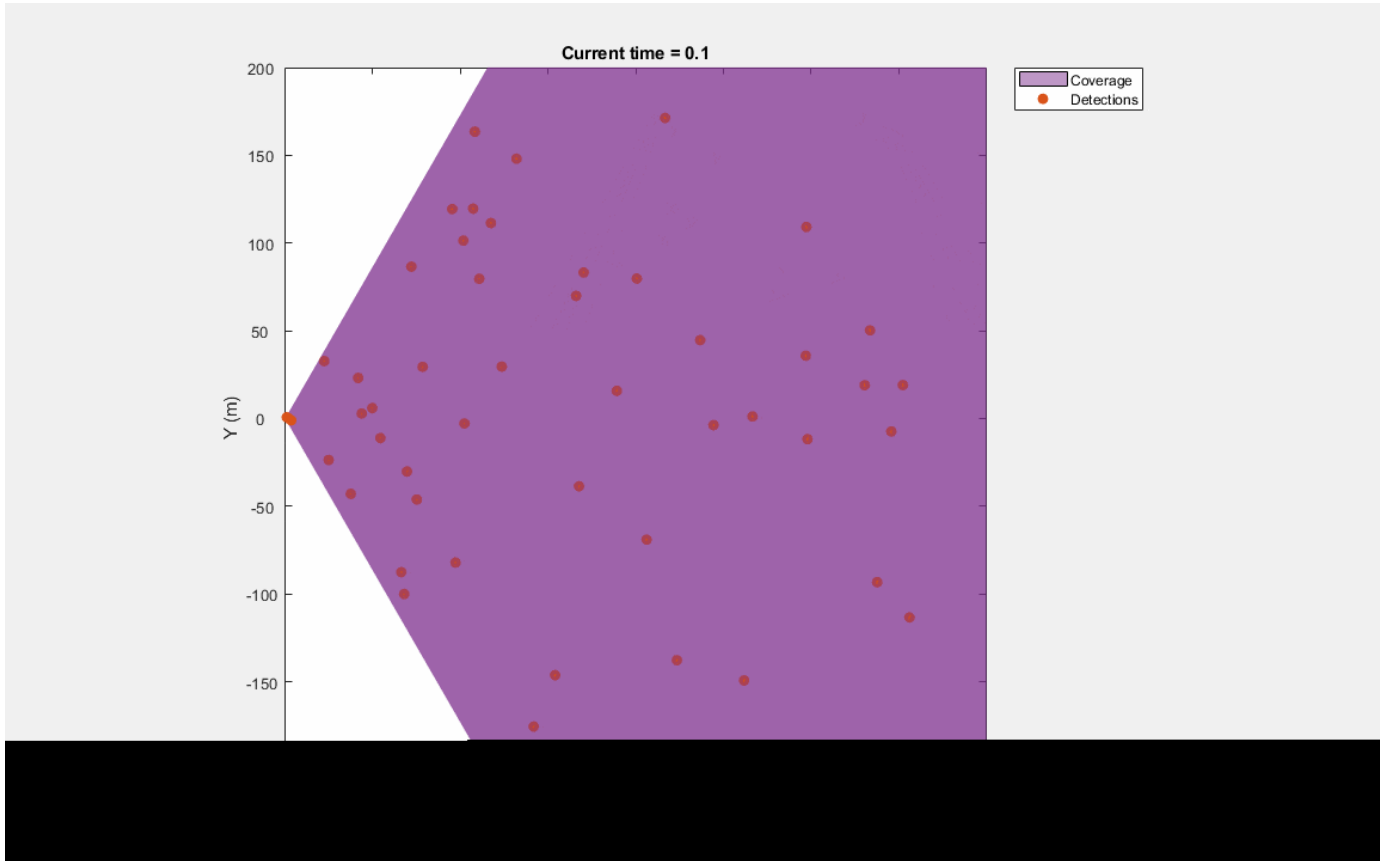
% Simulate 2-D radar using a monostaticRadarSensor.
radar = monostaticRadarSensor(1,...
    'UpdateRate',scene.UpdateRate,...
    'DetectionProbability',0.9,...% Pd
    'FalseAlarmRate',1e-3,...% Pfa
    'FieldOfView',[120 1],...
    'ScanMode','No Scanning',...
    'DetectionCoordinates','scenario',...% 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:

$$x_{\text{sensor}} = \text{SensorTransformFcn}(x_{\text{track}}, \text{SensorTransformParameters})$$

$$P_d =$$

$$\begin{cases} \text{Configuration.DetectionProbability} & \text{SensorLimits}(:, 1) \leq x_{\text{sensor}} \leq \text{SensorLimits}(:, 2) \\ \text{Configuration.MinDetectionProbability} & \text{otherwise} \end{cases}$$

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 filter. 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.AzimuthResolution;
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);

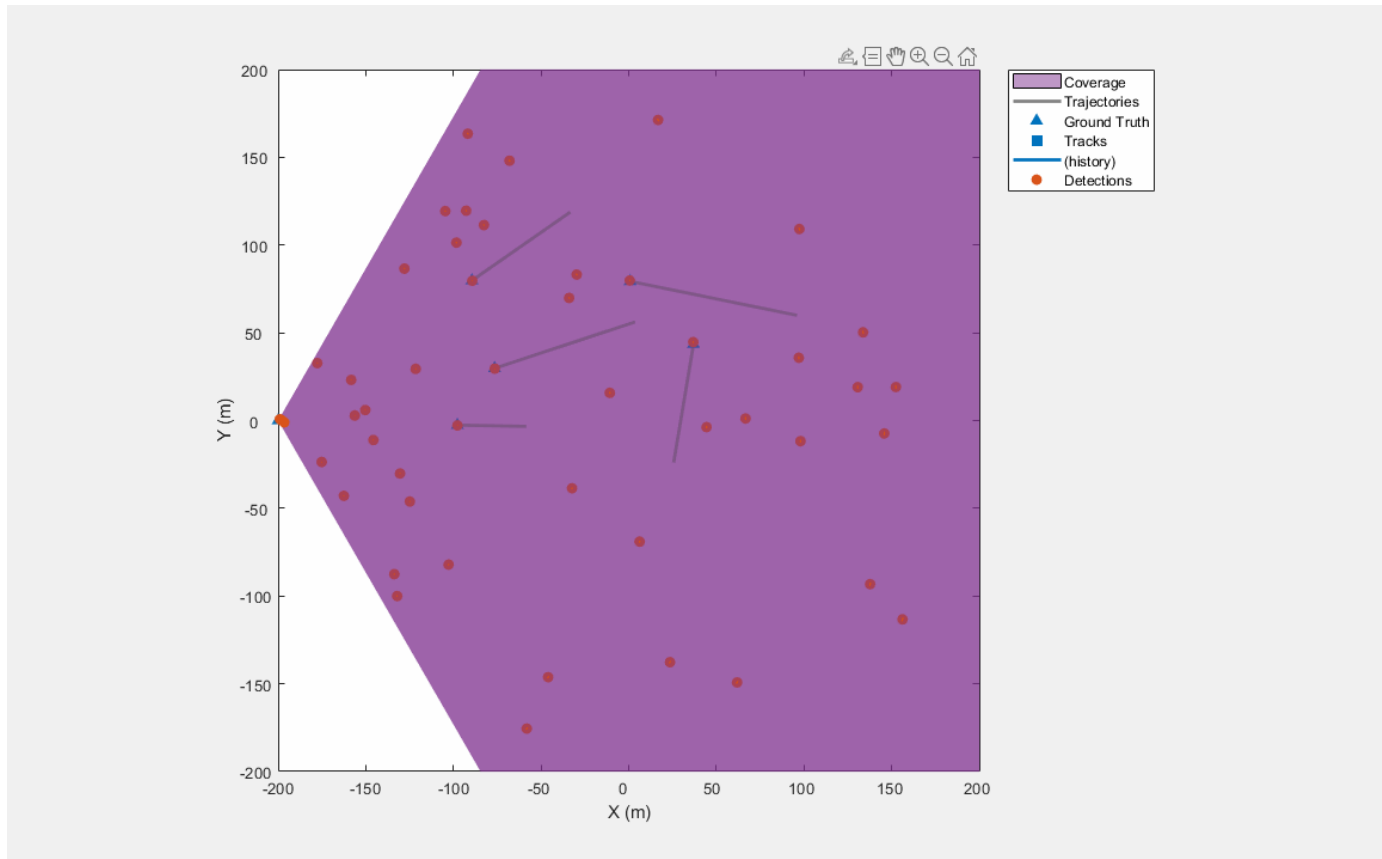
    % Tracks
    tracks = tracker(detections, time);
```

```

% 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

```

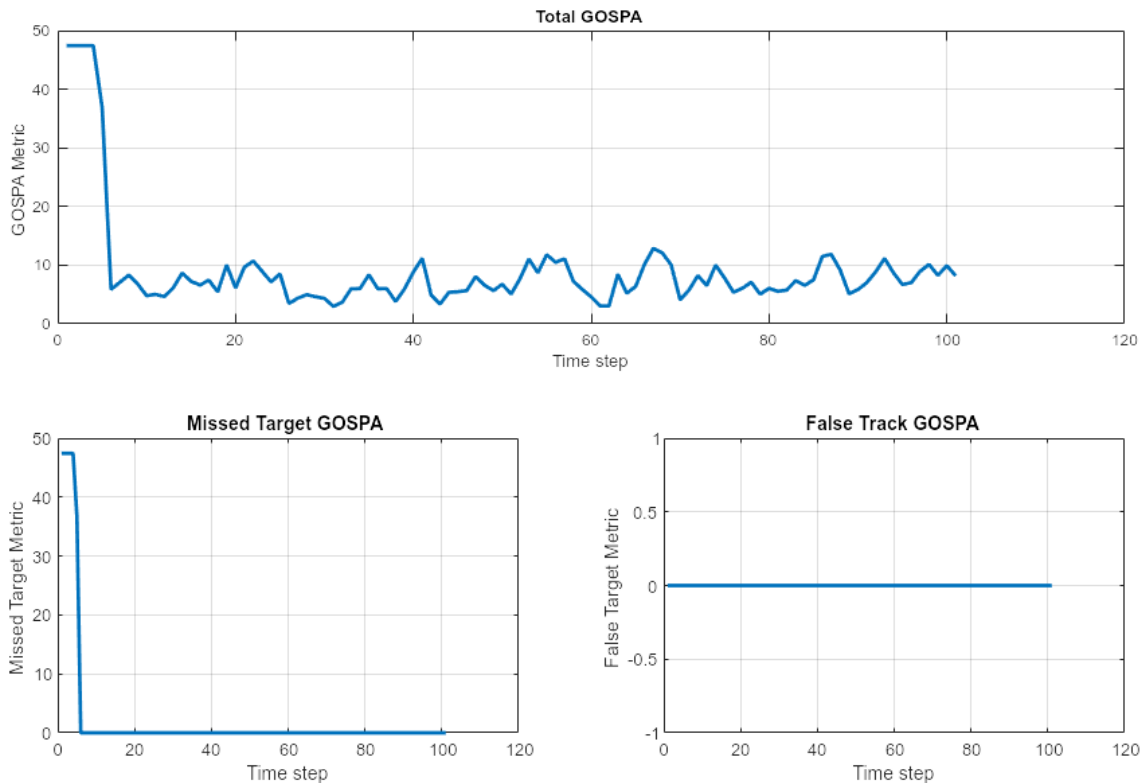


You can visualize that all targets were tracked by the PHD tracker in this scenario. This can also be quantitatively evaluated using the GOSPA metric and associated components. In the figure below, notice that GOSPA metric goes down after a few steps. The initial value of GOSPA metric is higher because 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 6-473. 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 6-474. 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.

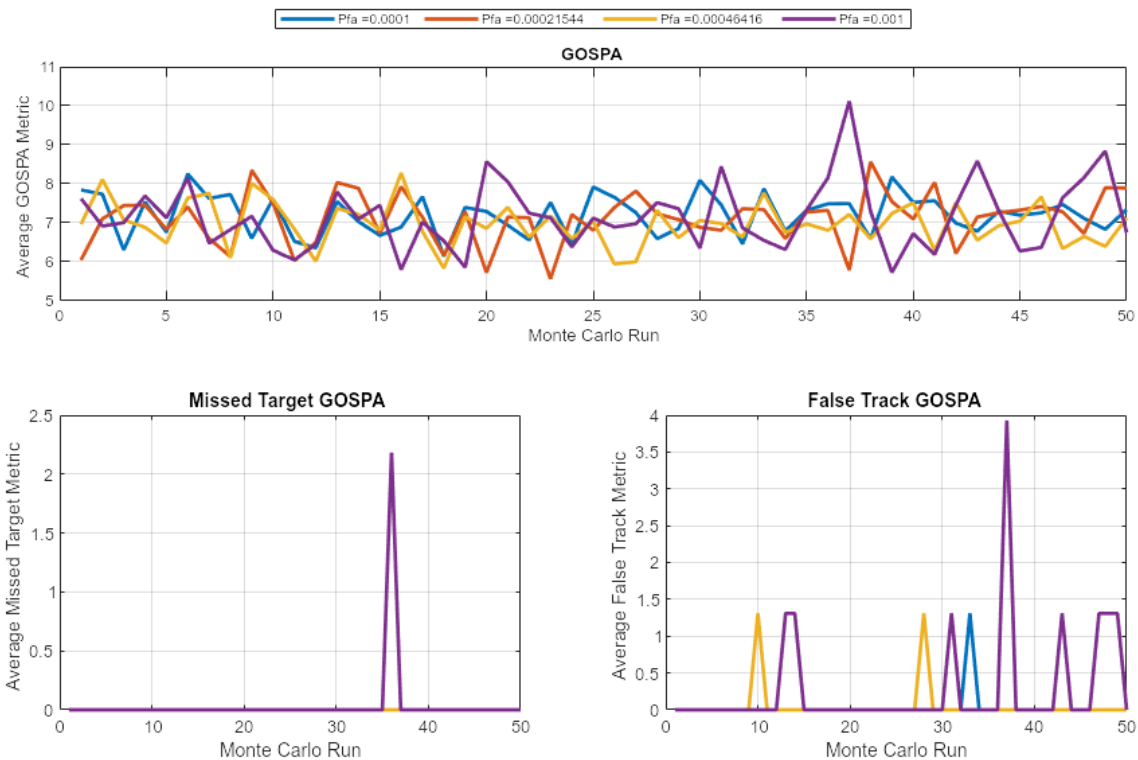
```
Pfas = 10.^linspace(-4,-3,4); % Choose your false alarm settings
helperGenerateCode(scene, Pfas); % Generate code
```

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 6-296 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
        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, t
    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

```

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 array 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;

end

```

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, tr
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

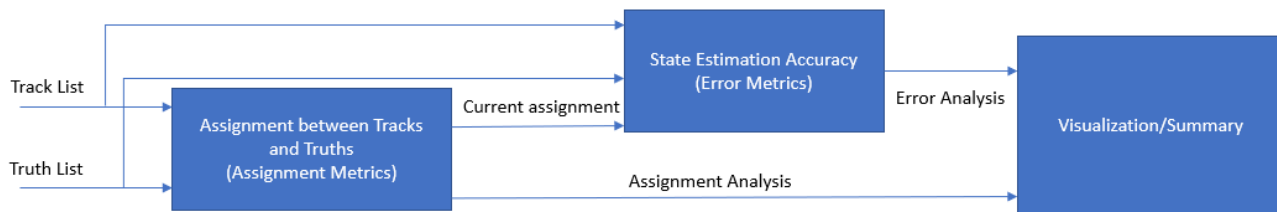
```

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

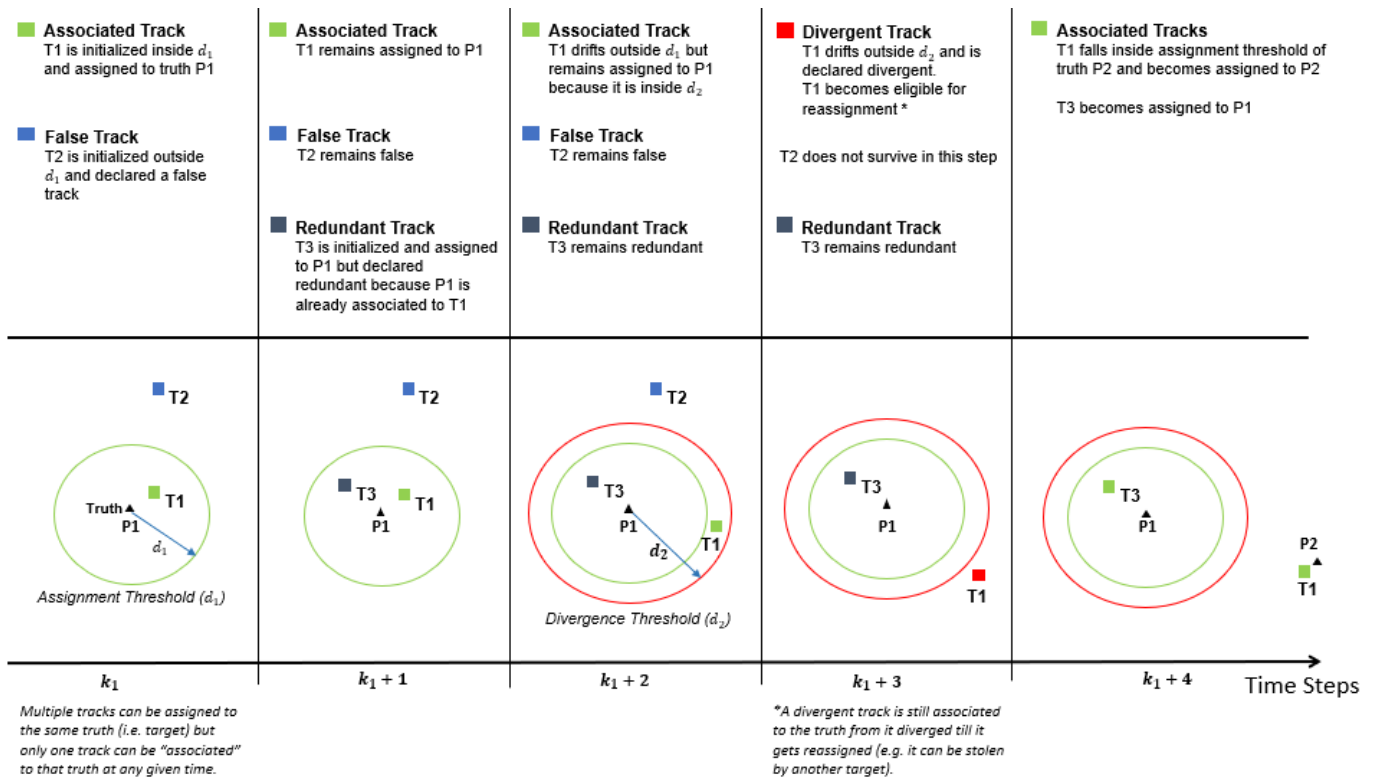
The general workflow for evaluating a multi-object tracker in the presence of ground truth can be divided into 2 main steps. First, tracks are assigned to truths using an assignment algorithm. Second, using the computed assignment as an input, state-estimation accuracy for each track is computed. In 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.



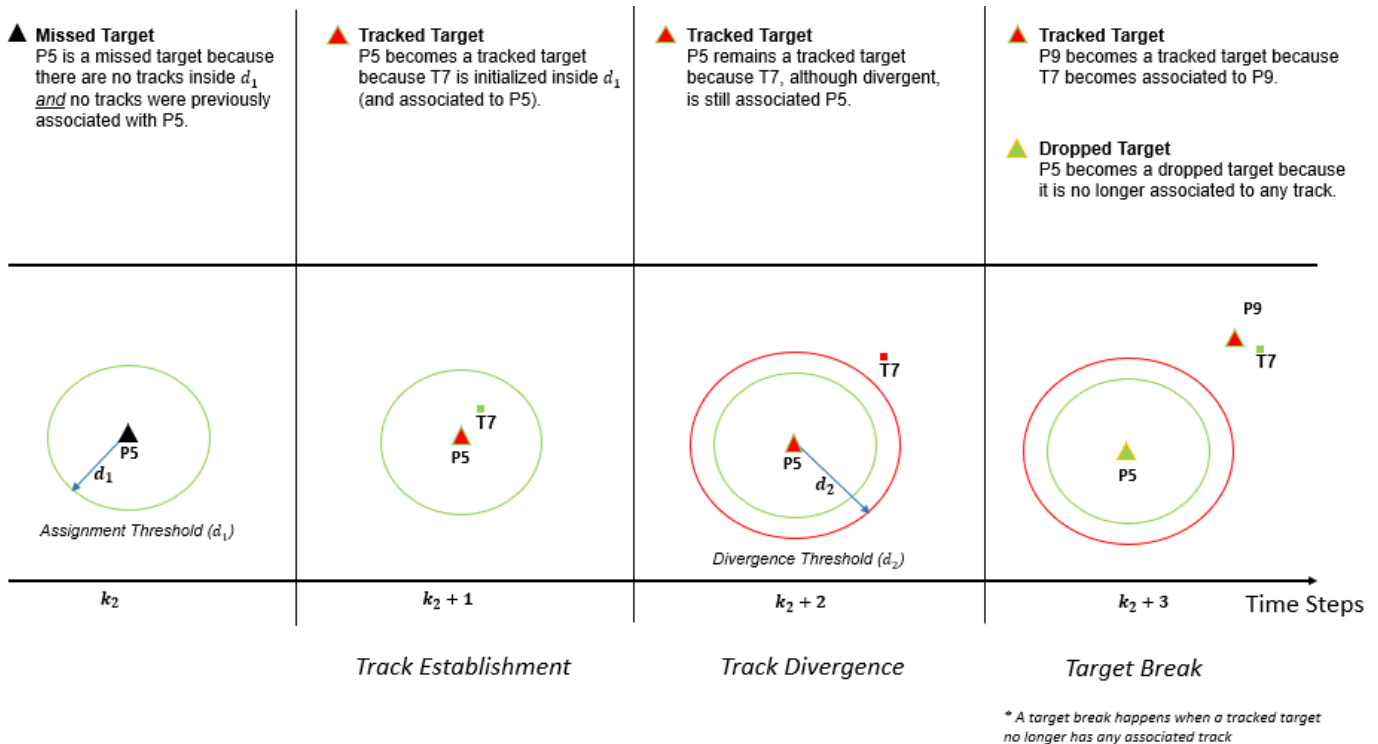
The assignment metrics assesses the characteristics of a tracker resulting due to incorrect assignment of sensor data to tracks due to ambiguity, missed targets or presence of false alarms. Incorrect assignment of sensor data can result in events like swapping of tracks between targets, creation of redundant 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.



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.



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);

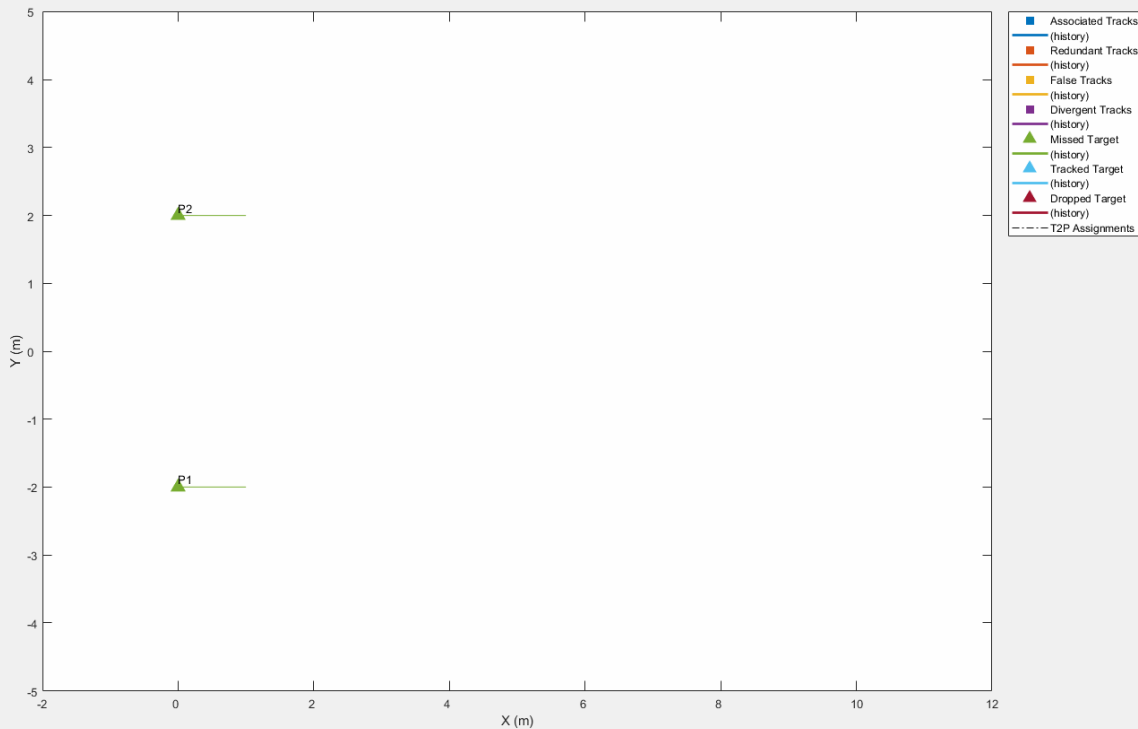
% The current assignment information as well as the list of tracks and
% truths at the current time can then be passed as an input to the
% error metrics.
[posRMSE(i), velRMSE(i), posANEES(i), velANEES(i)] = errorMetrics(tracks, assignedTrackIDs, truths, assignedTruthIDs);

% 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

```

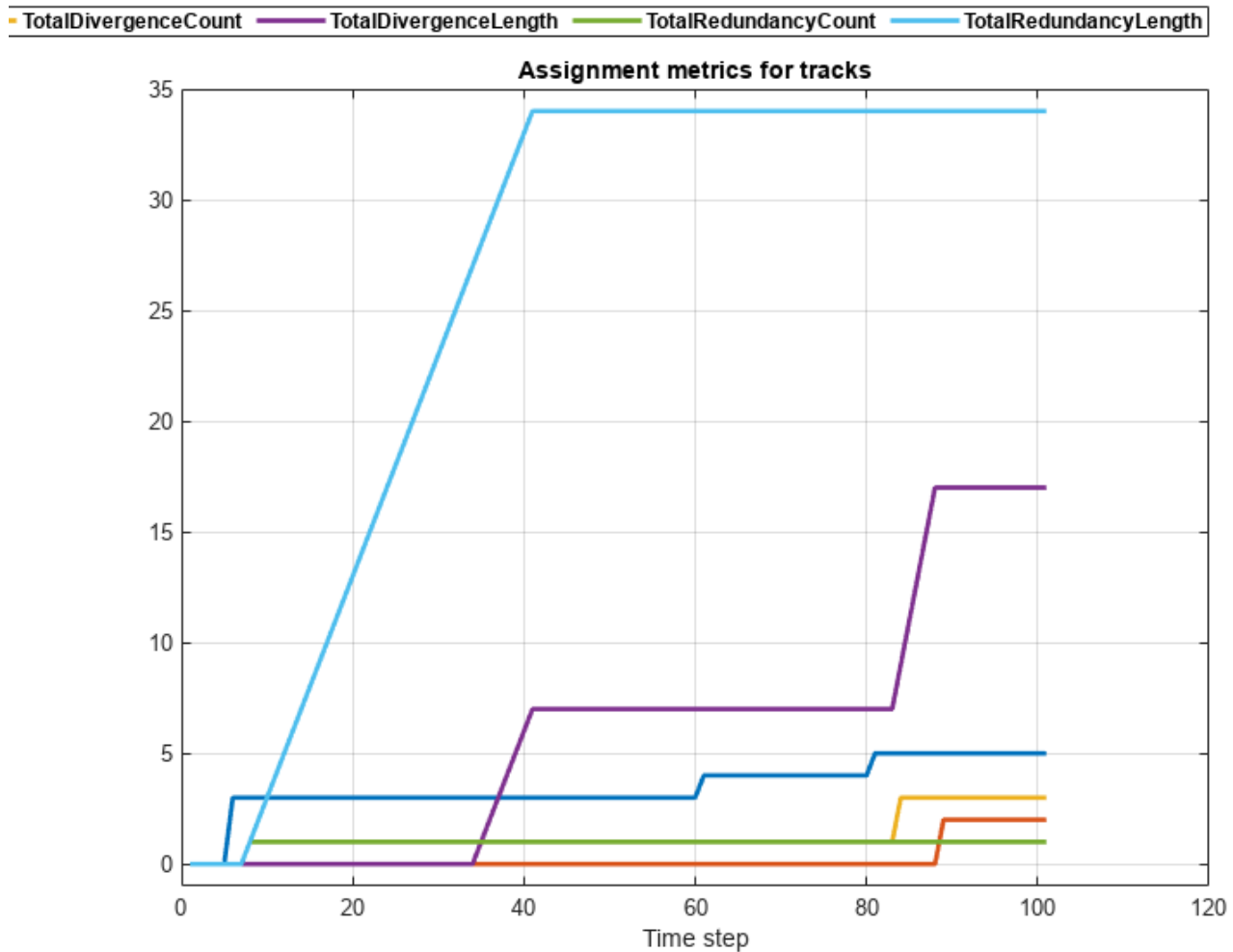
```

MaxBreakCount: 2
TotalBreakCount: 4
MaxBreakLength: 9
TotalBreakLength: 9

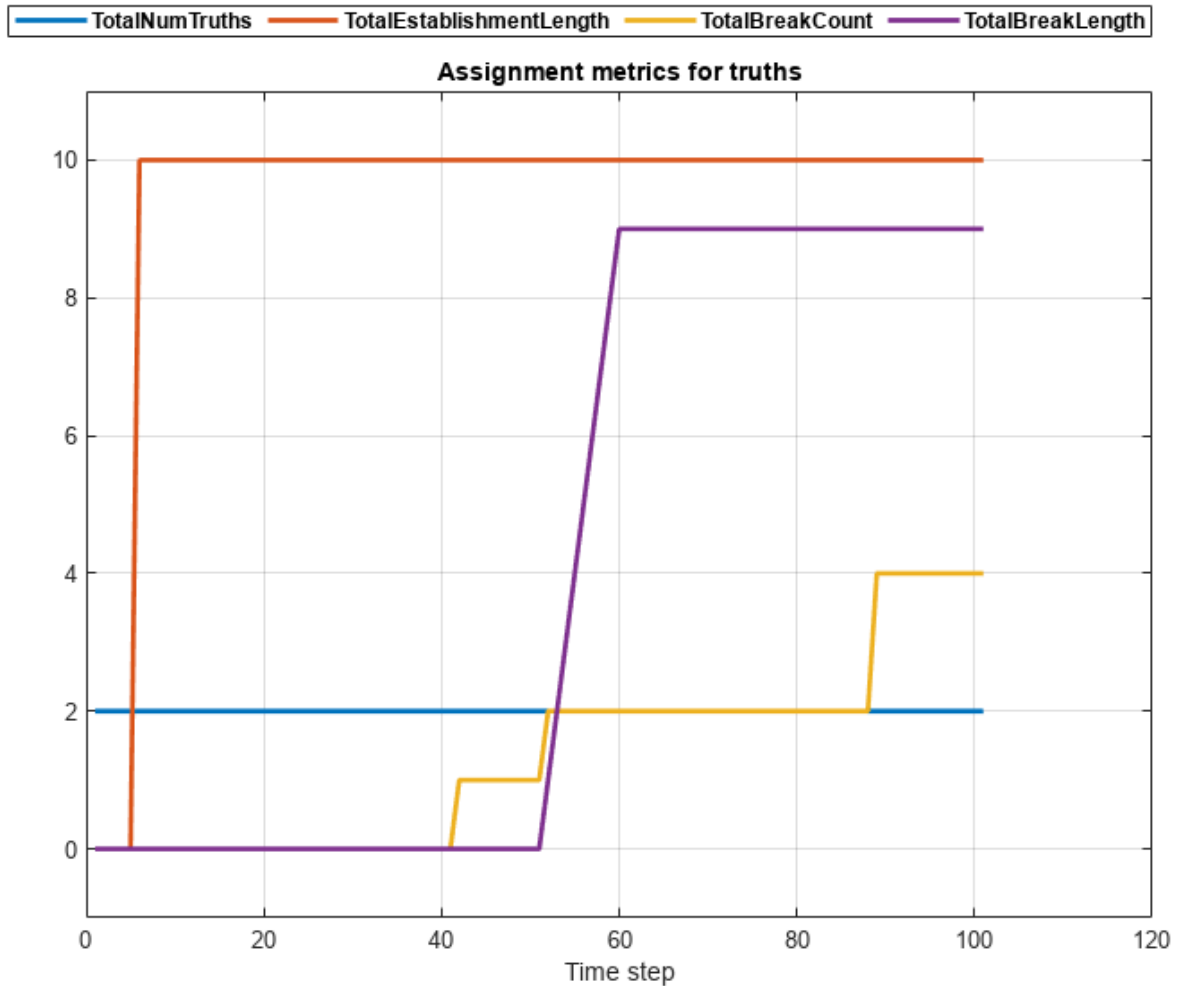
```

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

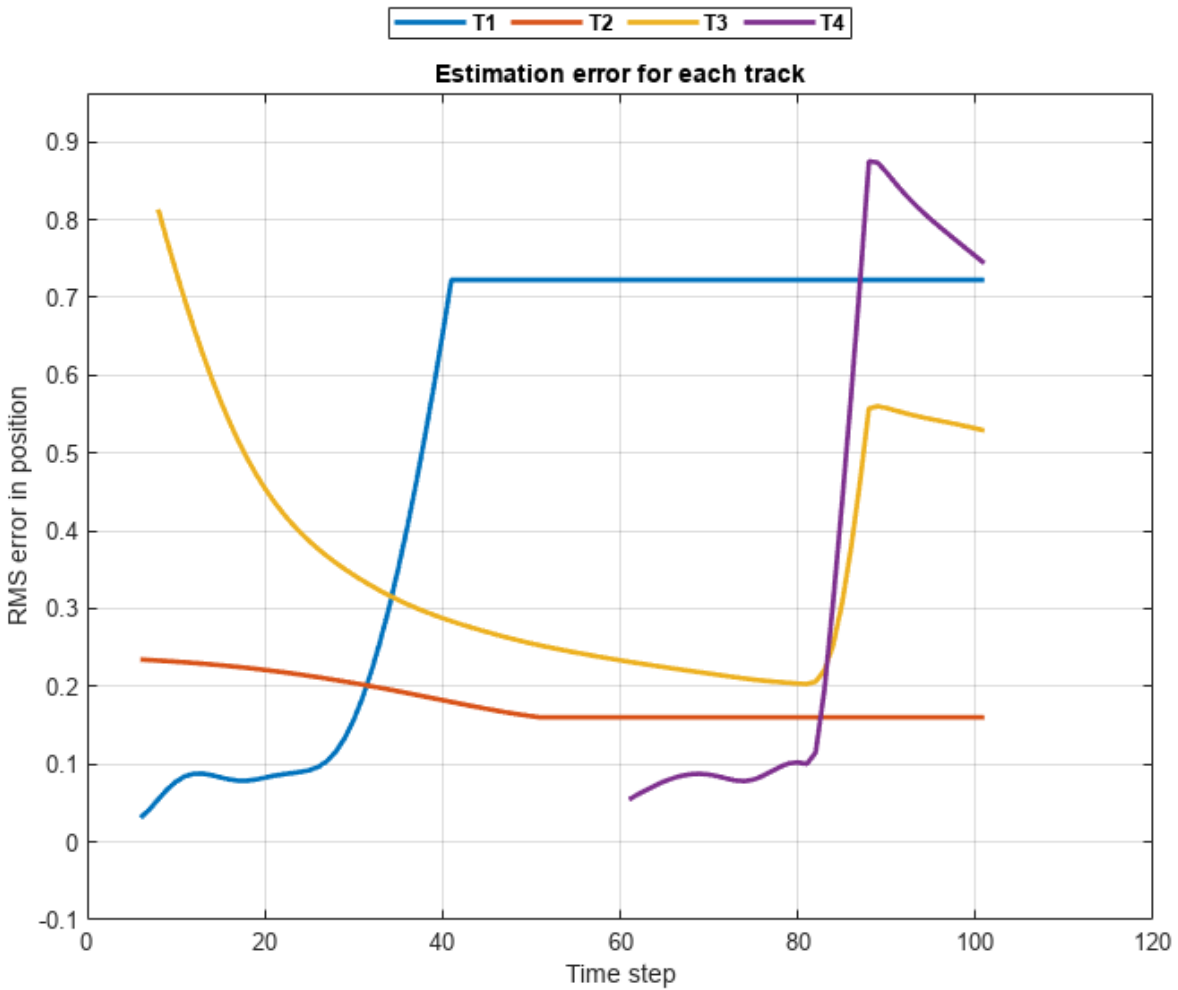
Estimation error for each truth

```
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

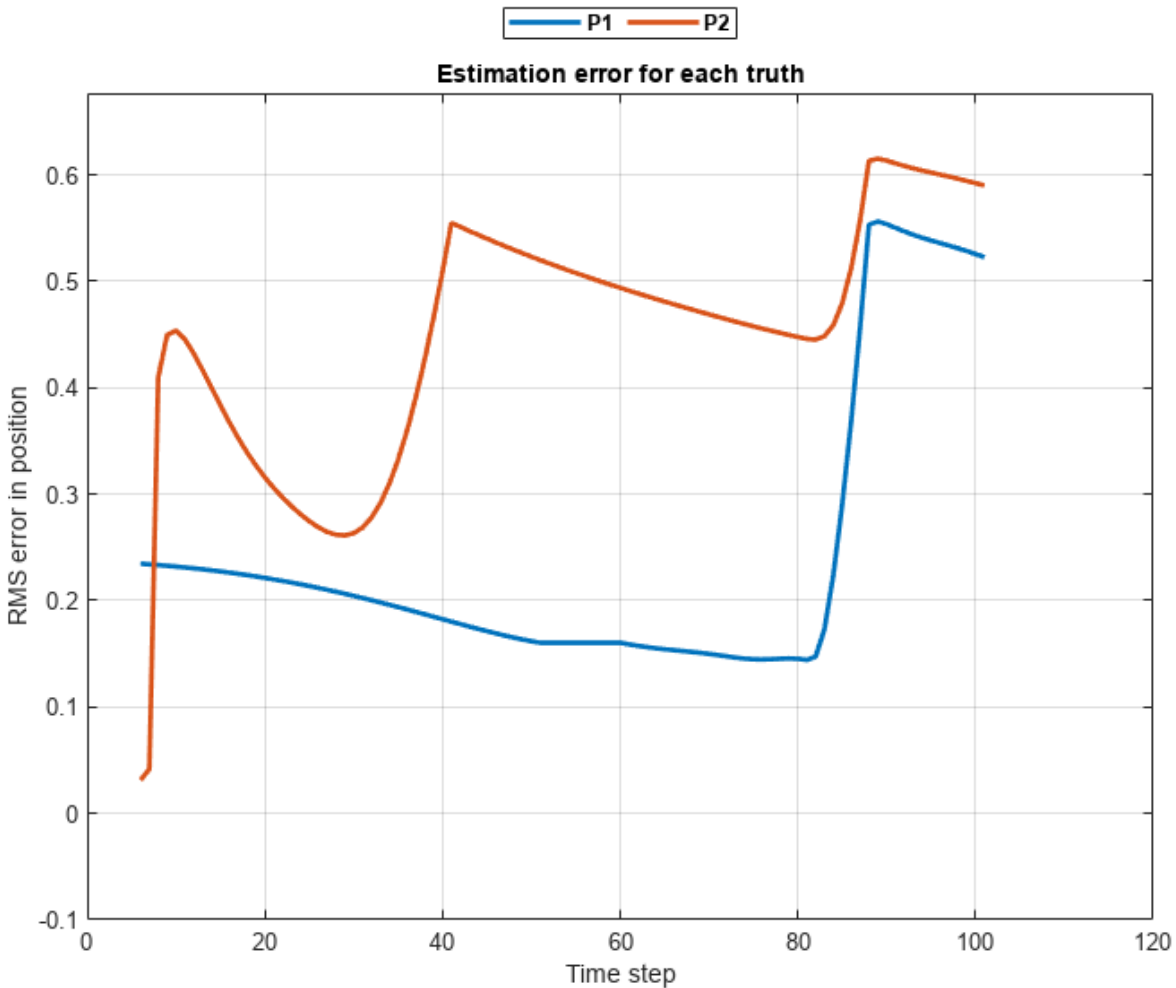
By recording the errors at each time-step, you can also plot error vs time for each track and truth. The following plots show the RMS error in position for each track and truth. After a track is deleted, the error in its estimate remains constant. Notice that in the first-half of the simulation, the 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');

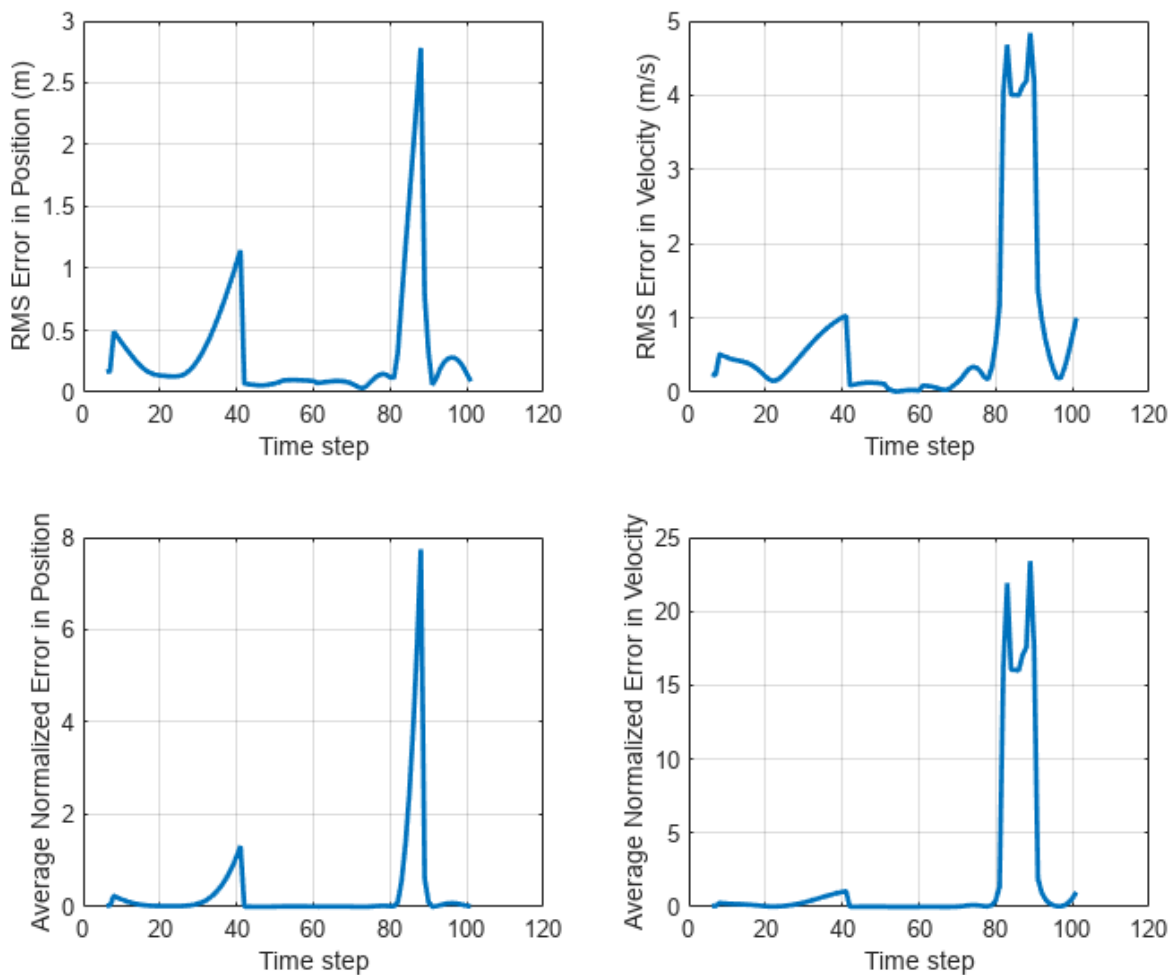
subplot(2,2,3)
plot(posANEES,'LineWidth',2);
xlabel('Time step');
ylabel('Average Normalized Error in Position');
grid('on');

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);

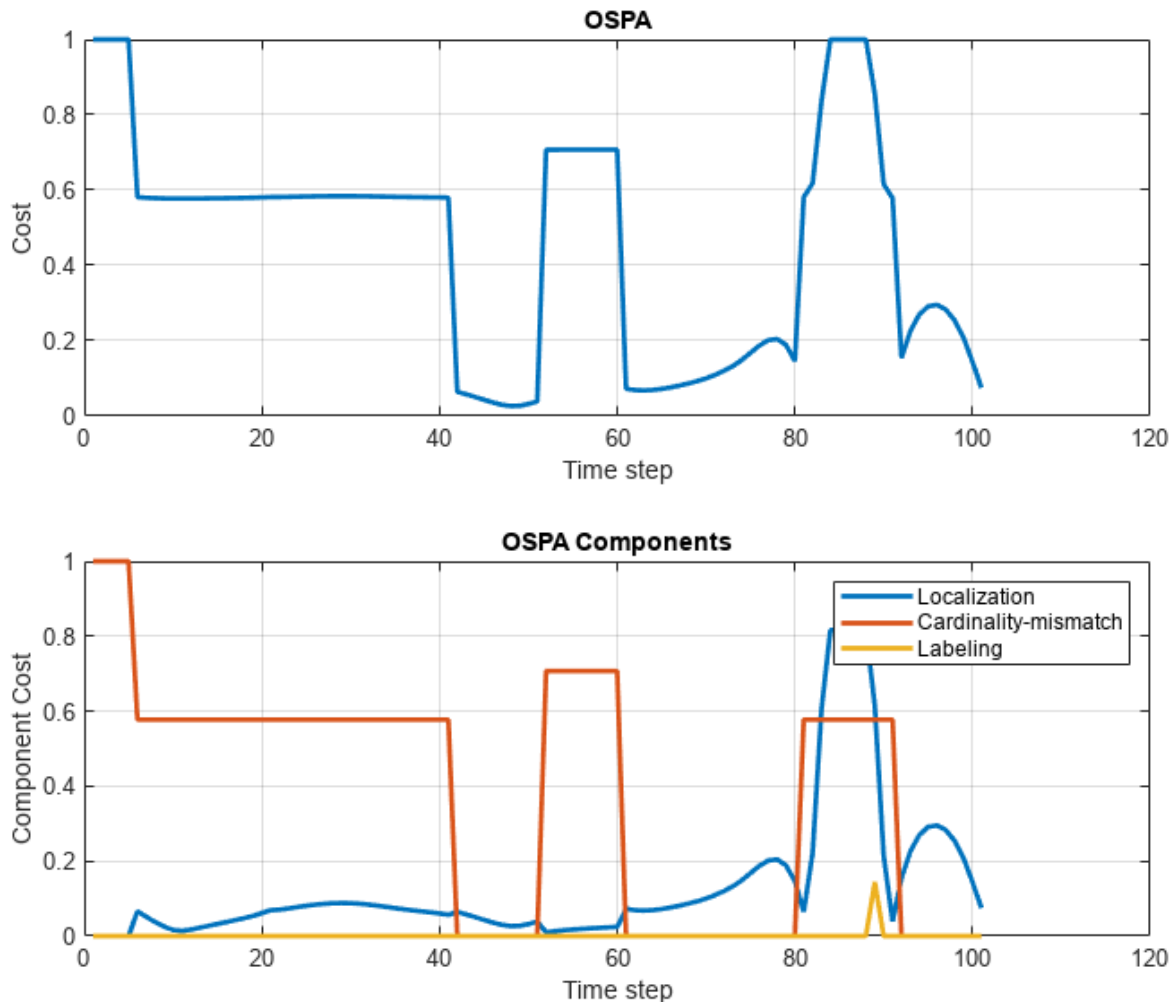
% Call the System object as a function and record OSPA and all 3 components
[ospa(i),locComponent(i),cardComponent(i),labelingComponent(i)] = ospaMetric(tracks, truths)
end
```

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);

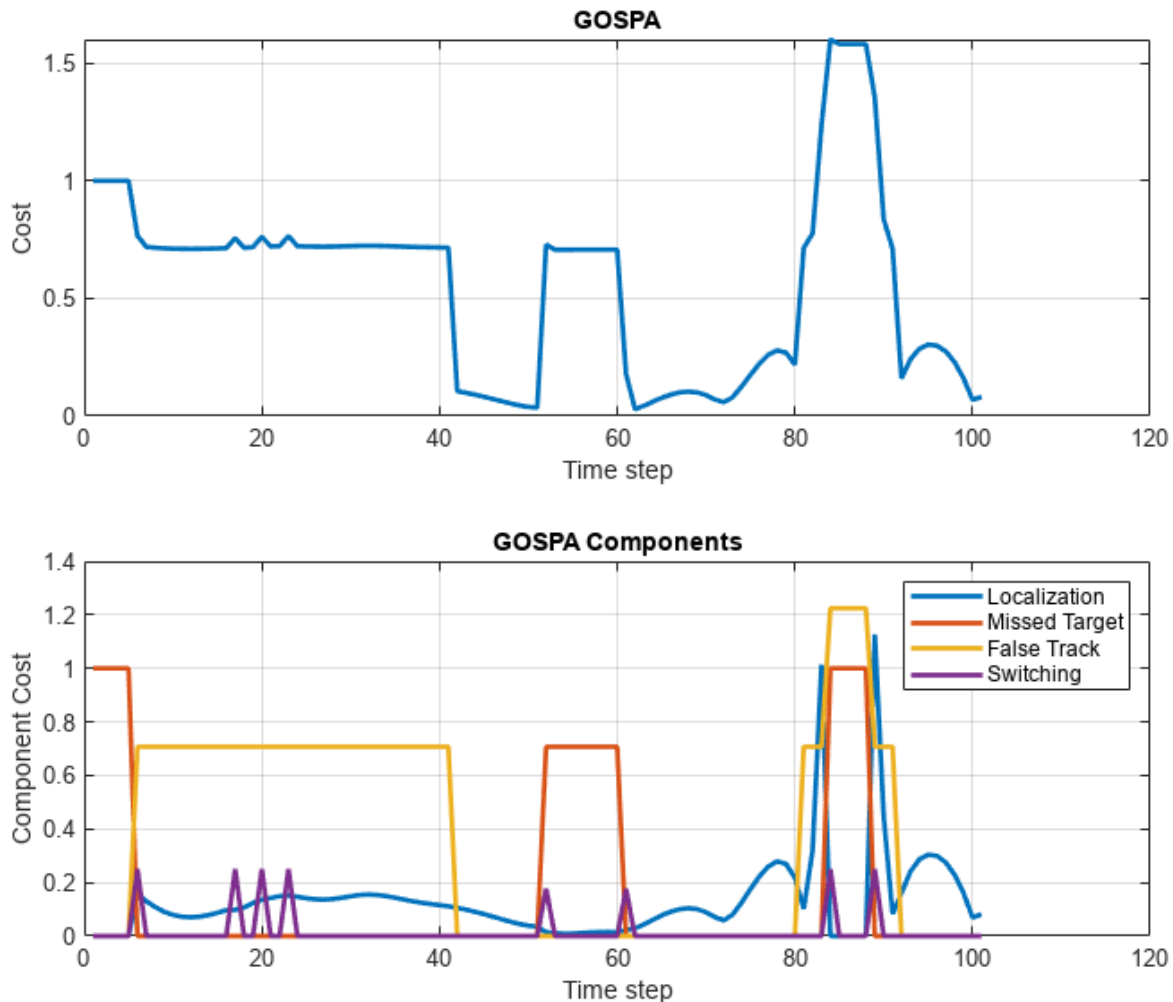
    % Call the System object as a function and get all 4 components
    [labeledGospa(i),traditionalGospa(i),switchingComponent(i),locComponent(i),missedTargetComponent(i),falseTrackComponent(i)] = gospaMetric(tracks,truths);
end
```

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');
```



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

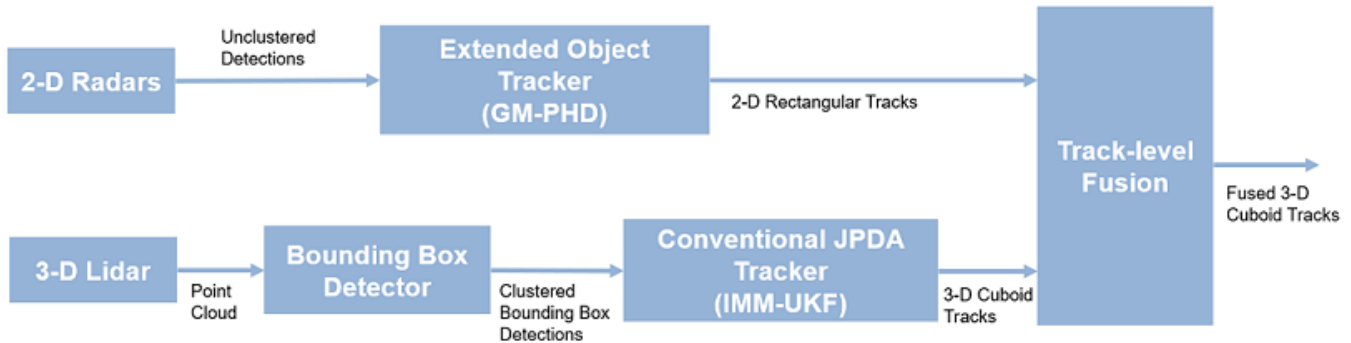
In this example you learned about three different tools to analyze the performance of a multi-object tracking system. You also learned about the workflow in MATLAB to use these metrics. You also learned about the type of information they provide and how it can be analyzed. The tools described in this 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] Garcia-Fernandez, Angel F., et al. "A Metric on the Space of Finite Sets of Trajectories for Evaluation of Multi-Target Tracking Algorithms." *IEEE Transactions on Signal Processing*, vol. 68, 2020, pp. 3917-28.

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 schematic of the workflow is shown below.



See “Fusion of Radar and Lidar Data Using ROS” (ROS Toolbox) for an example of this algorithm using recorded data on a rosbag.

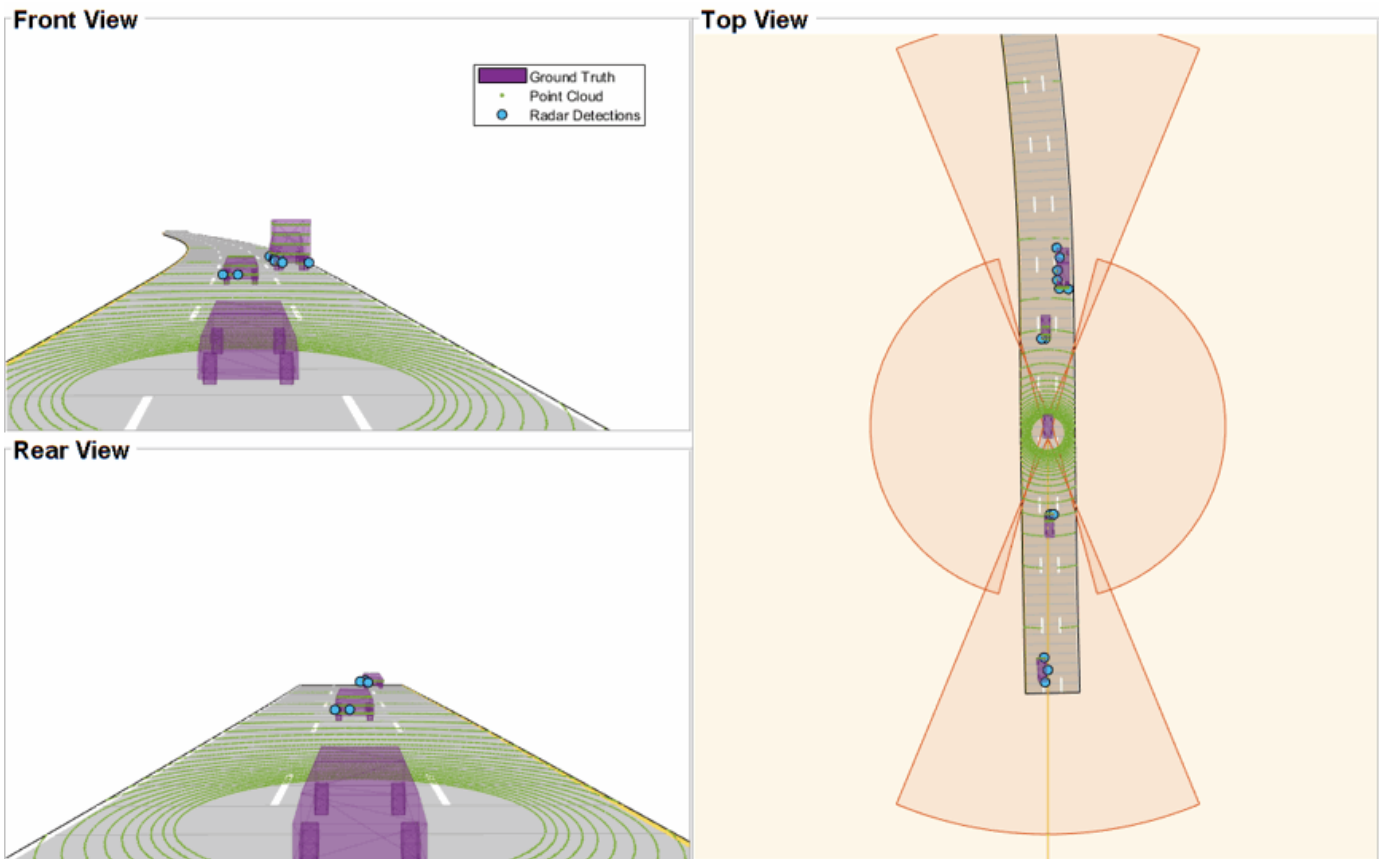
Setup Scenario for Synthetic Data Generation

The scenario used in this example is created using `drivingScenario` (Automated Driving Toolbox). The data from radar and lidar sensors is simulated using `drivingRadarDataGenerator` (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(2021);
```

```
% 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.



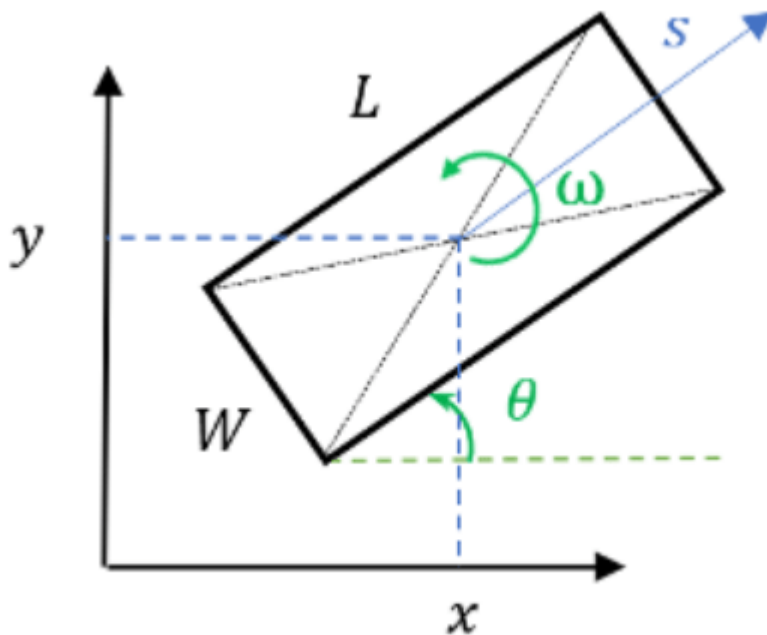
Radar Tracking Algorithm

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 6-148 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 6-148 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 returns multiple measurements per object. Further, the sensor returns a large number of points from the road, which must be removed 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 6-352 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 constrain 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 returns 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 set up three separate metrics using `trackGOSPAMetric` for each of the trackers. The 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);
```

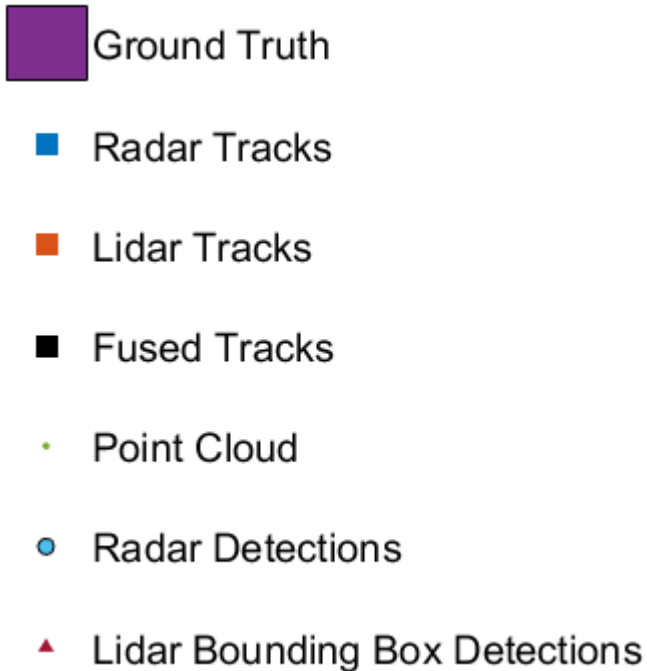
Visualization

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);
```



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 systems. You also compute the metric for each tracker using the ground truth available from the scenario.

```
% Initialize 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, groundTruth);
[gospa(2,idx),~,~,~,missTarget(2,idx),falseTracks(2,idx)] = gospaLidar(lidarTracks, groundTruth);
[gospa(3,idx),~,~,~,missTarget(3,idx),falseTracks(3,idx)] = gospaCentral(fusedTracks, groundTruth);

% 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);

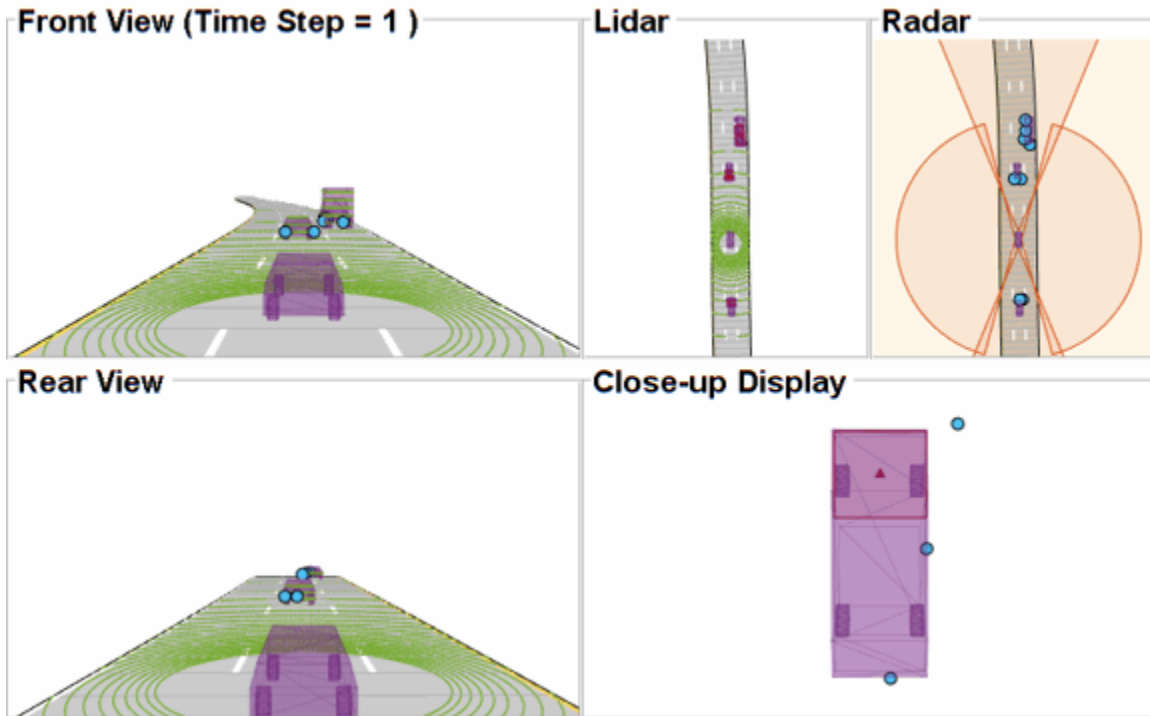
```

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

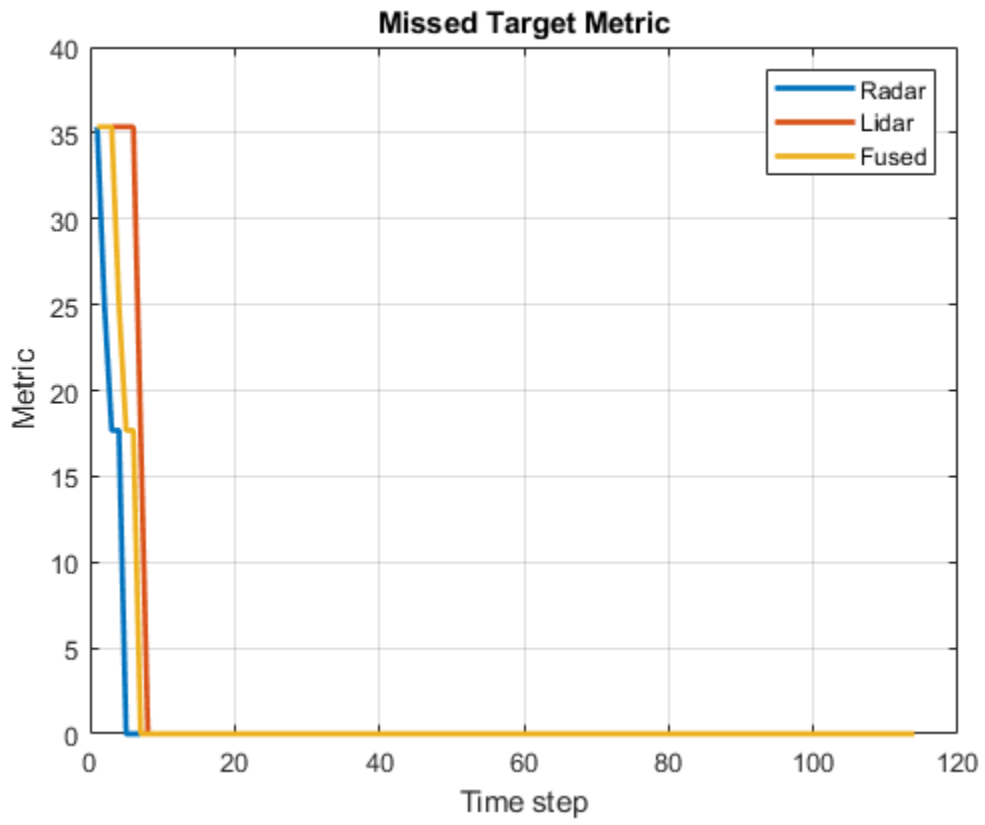
The animation below shows the entire run every three time-steps. Note that each of the three tracking systems - radar, lidar, and the track-level fusion - were able to track all four vehicles in the scenario and no false tracks were confirmed.

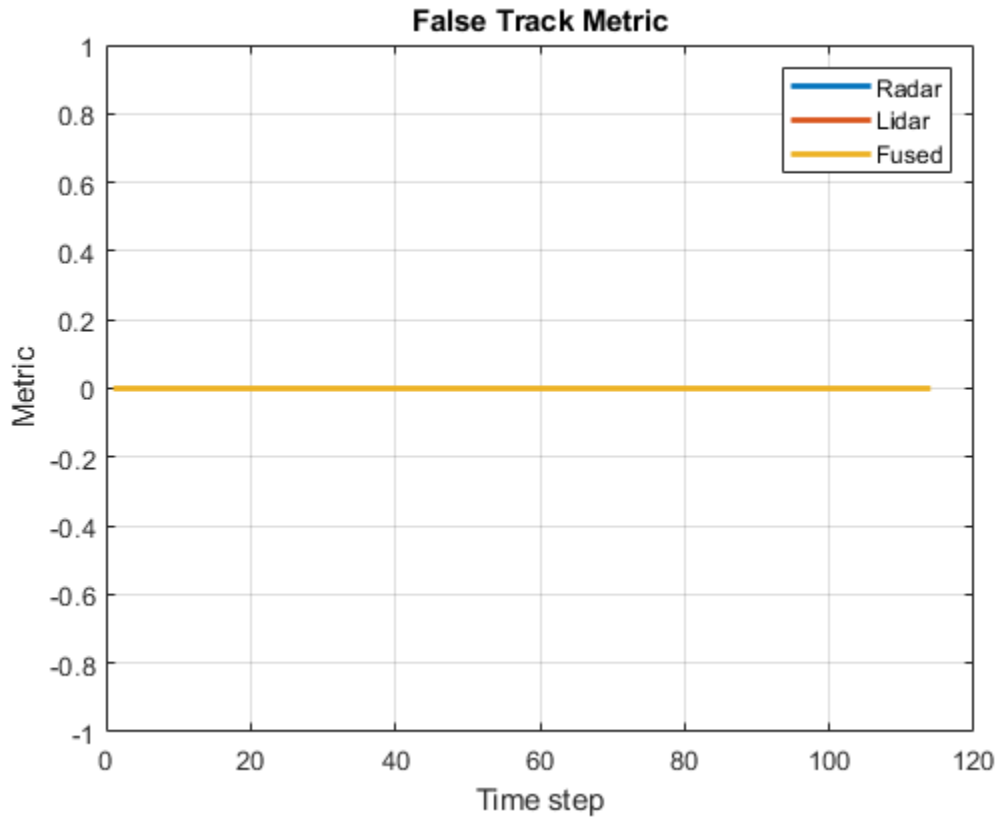


You can also quantitatively measure this aspect of the performance using "missed target" and "false track" components of the GOSPA metric. Notice in the figures below that missed target component starts from a higher value due to establishment delay and goes down to zero in about 5-10 steps for each 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;
```



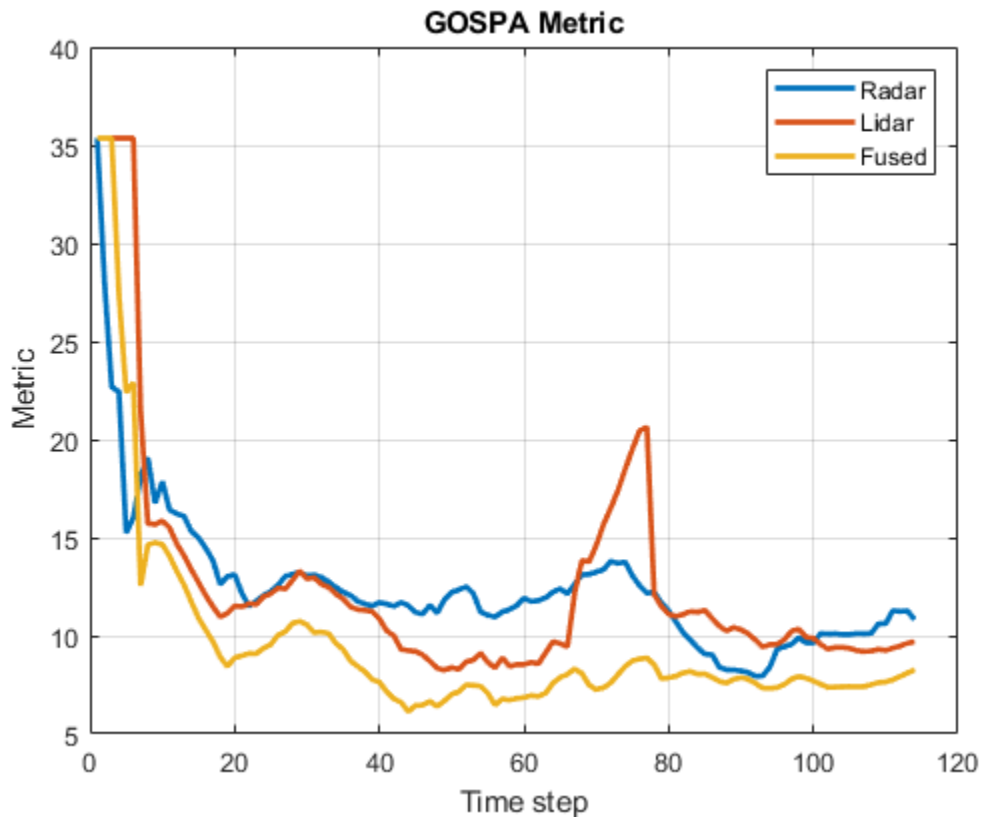


Track-level Accuracy

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.

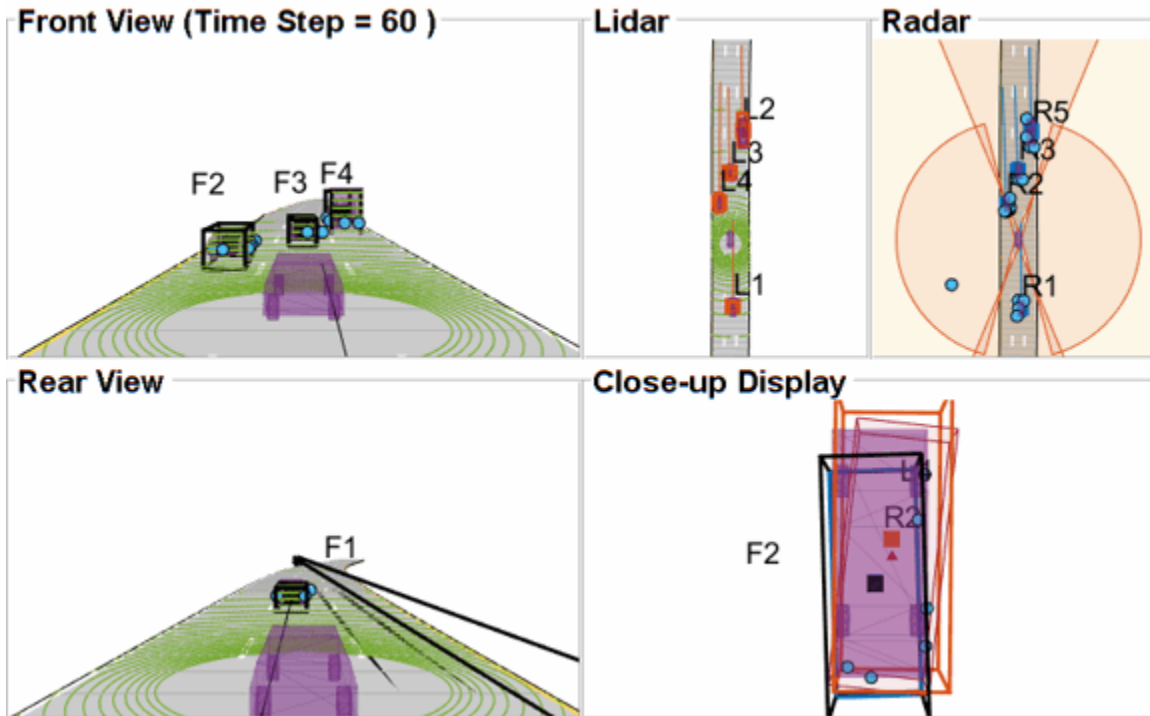
Note that the GOSPA metric for fused estimates is lower than the metric for individual sensor, which indicates that track accuracy increased after fusion of track estimates from each sensor.

```
% 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.

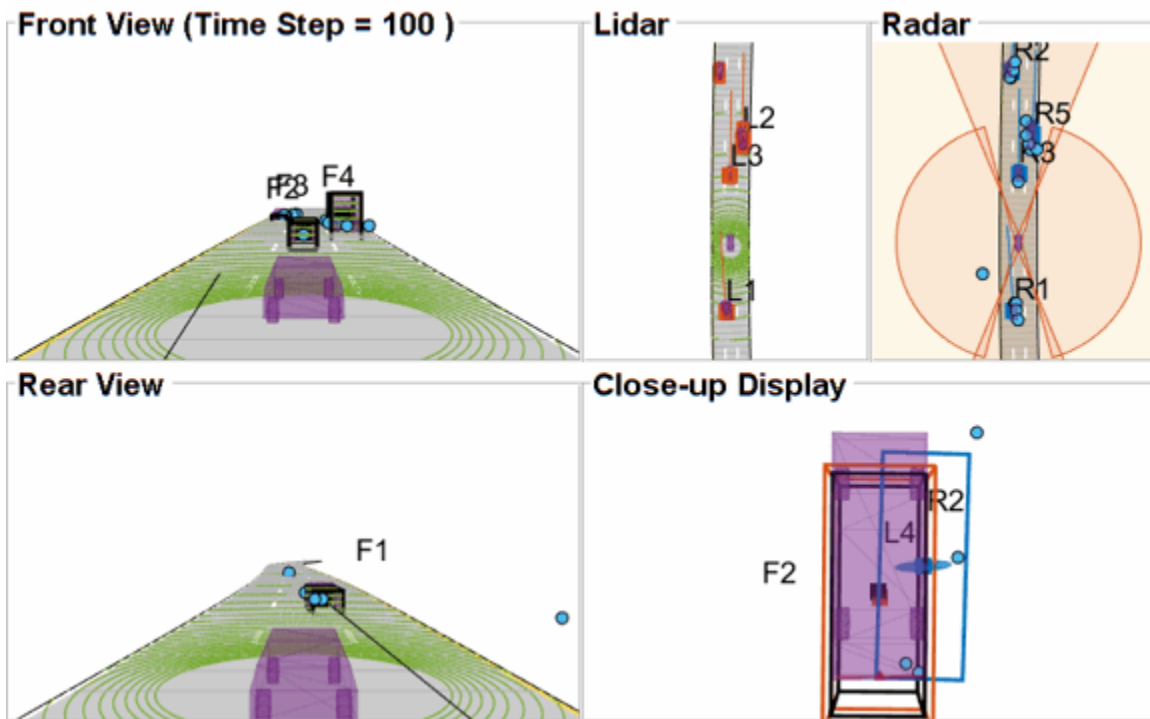


This effect is also captured in the GOSPA metric. You can notice in the GOSPA metric plot above that the lidar metric shows a peak around the 65th time step.

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 set up 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
```

radar2central

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 properties.

```
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.

```
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);
```

```

% Total distance
dist = d1 + d2 + d3 + d4 + d5;

% A constant penalty for not measuring 3-D state
dist = dist + 3;

end

helperRadarLidarFusionFcn

Function to fuse states and state covariances in central track state-space

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);

```

```

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:
%  $P^{-1} = w_1 P_1^{-1} + w_2 P_2^{-1}$ 
%  $x = P(w_1 P_1^{-1} x_1 + w_2 P_2^{-1} x_2)$ ;
% where  $w_1 + w_2 = 1$ 
% 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([20 10 0; 20 .8*v*stopTime-10 0], [0 stopTime]);
p2 = platform(scenario);
p2.Trajectory = waypointTrajectory([0 0 0; 0 v*stopTime 0], [0 stopTime]);
p3 = platform(scenario);
p3.Trajectory = waypointTrajectory([-20 -10 0; -20 1.2*v*stopTime+10 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 = fusionRadarSensor(1, 'No scanning', 'UpdateRate', 5, ...
    'MountingAngles', [0 0 0], 'AzimuthResolution', 1, ...
    'FieldOfView', [100 1], 'HasINS', true, 'DetectionCoordinates', 'Scenario');

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', [-100 100]);
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);
```

```

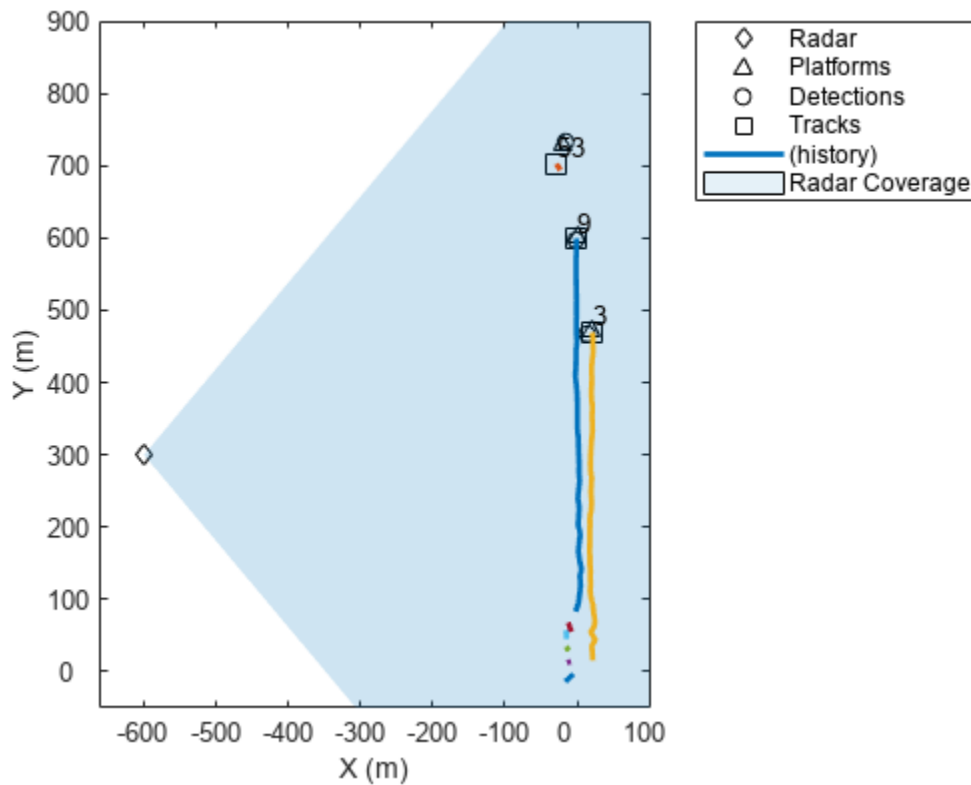
while advance(scenario)
    % Get detections
    dets = detect(scenario);

    % 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

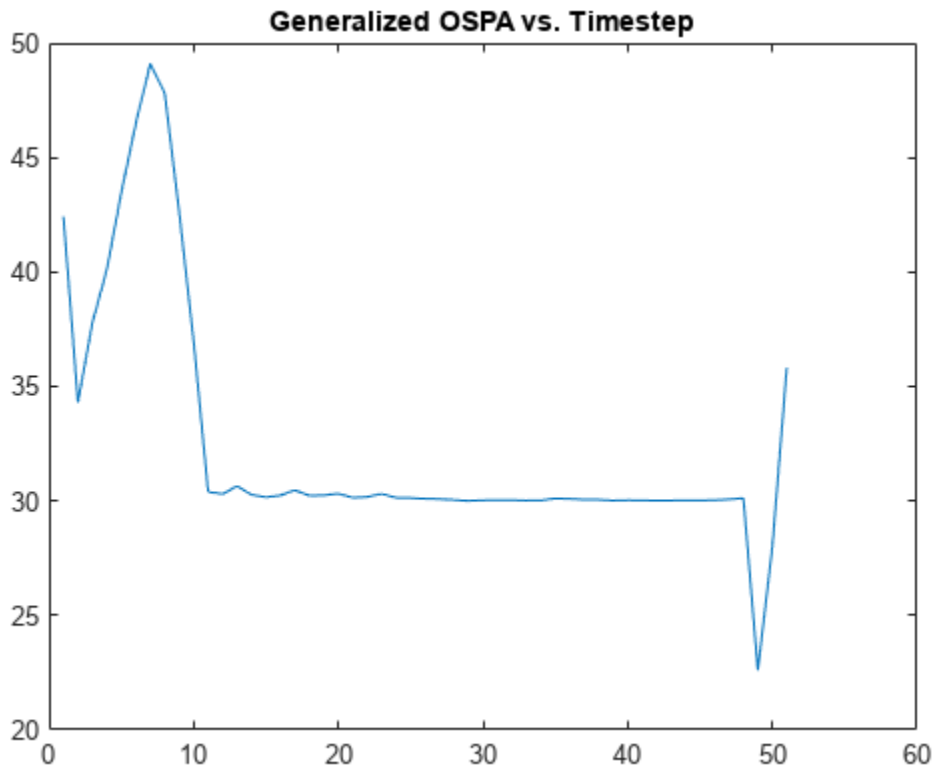
```



```

rng(s)
figure
plot(gospa)
title('Generalized OSPA vs. Timestep')

```

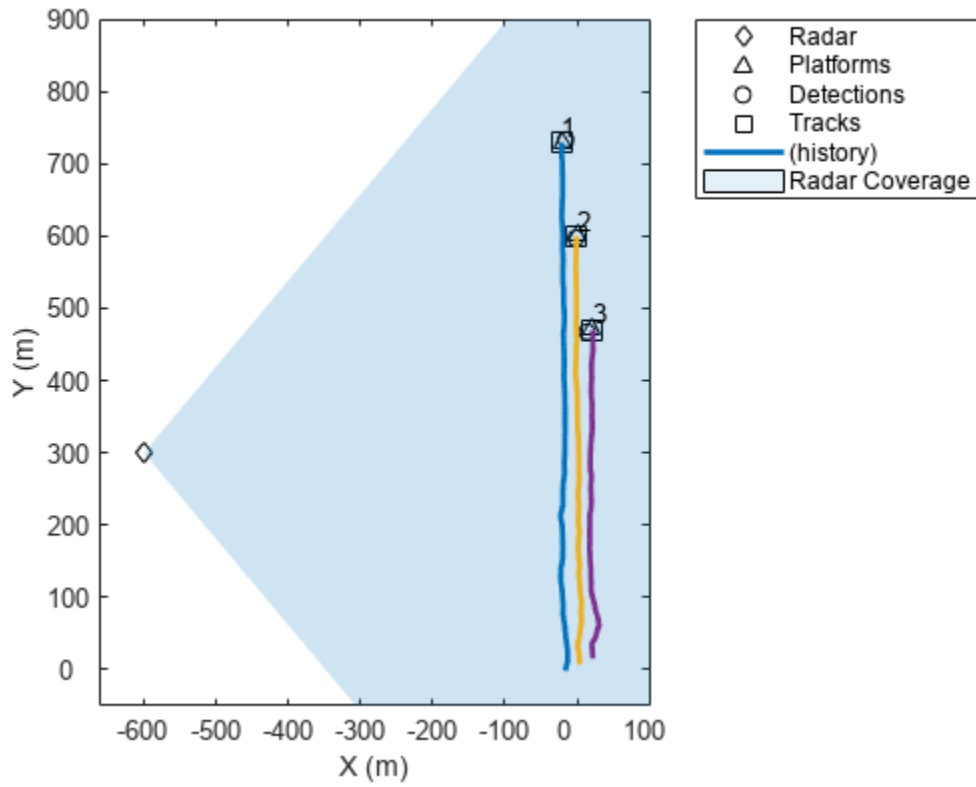


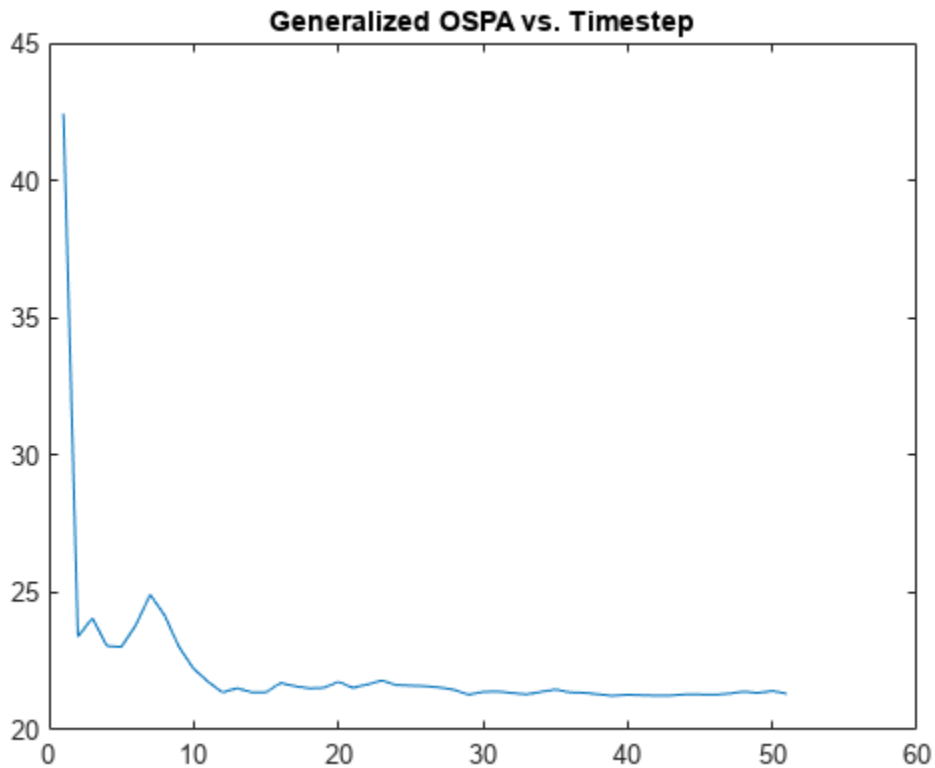
You observe that the tracker was not able to track the three objects. At some point, additional tracks are confirmed and shown in addition to the three expected tracks for the three moving objects. As a result, the value of the GOSPA metric increases. Note that lower values of the GOSPA metric indicate 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.

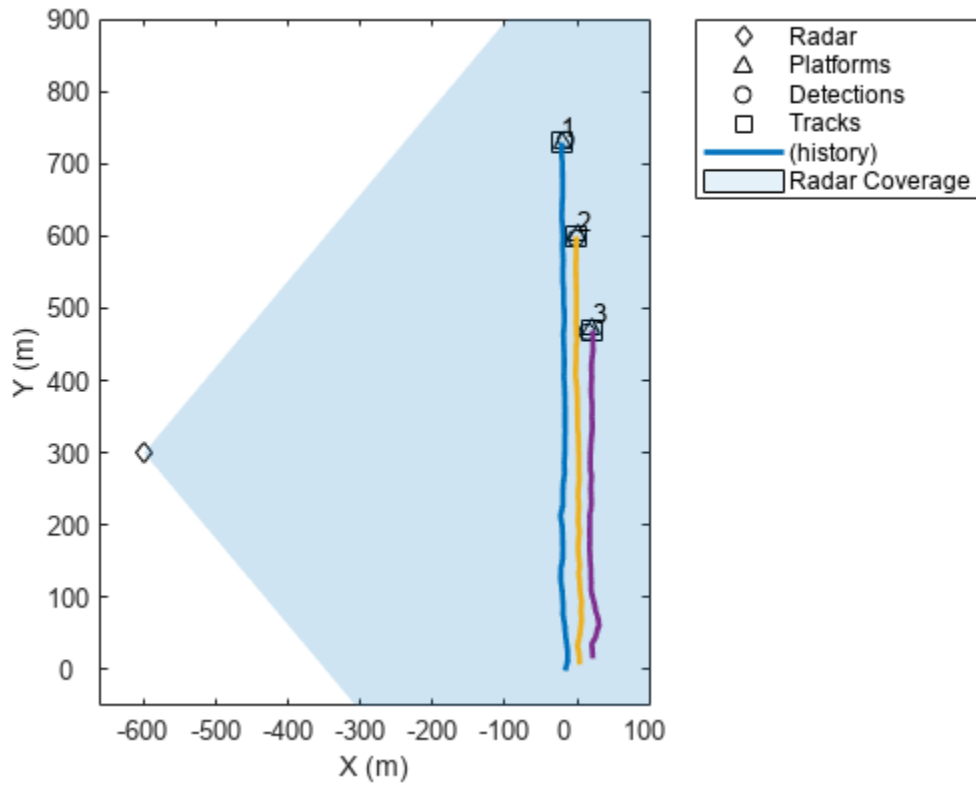
```
release(tracker);  
tracker.AssignmentThreshold = 50;  
rerunScenario(scenario, tracker, tgm, tp);
```

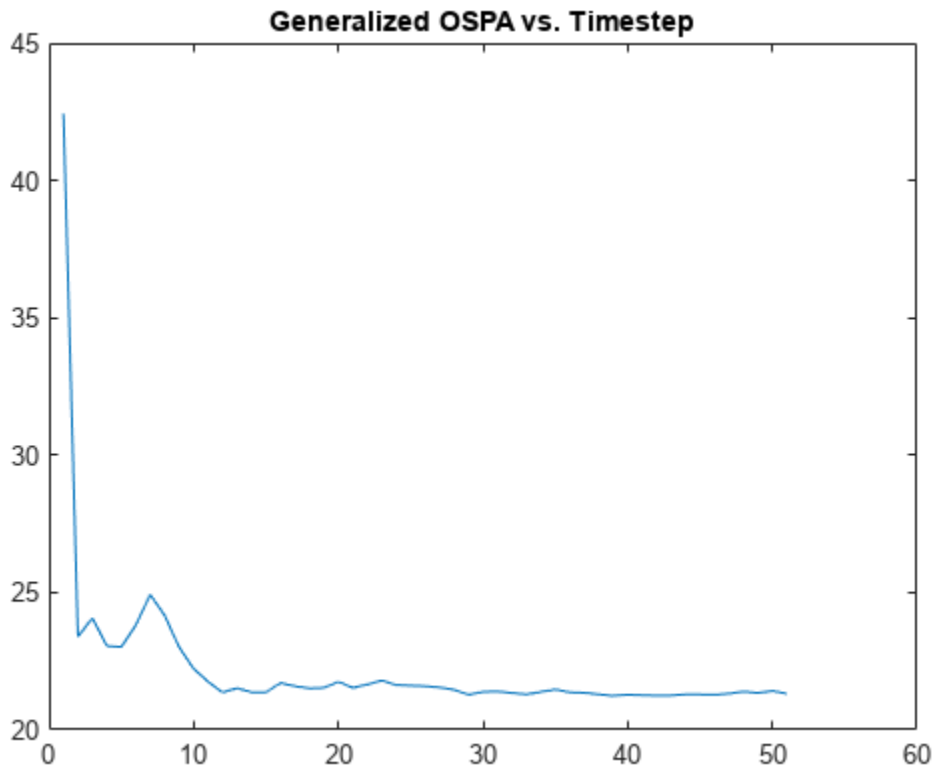





The assignment threshold property also has a maximum distance from a detection that you can set. This is the last element in the `AssignmentThreshold` value. You can use this value to speed the tracker up when handling large number of detections and tracks. Refer to the “How to Efficiently Track Large Numbers of Objects” on page 6-303 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);
```

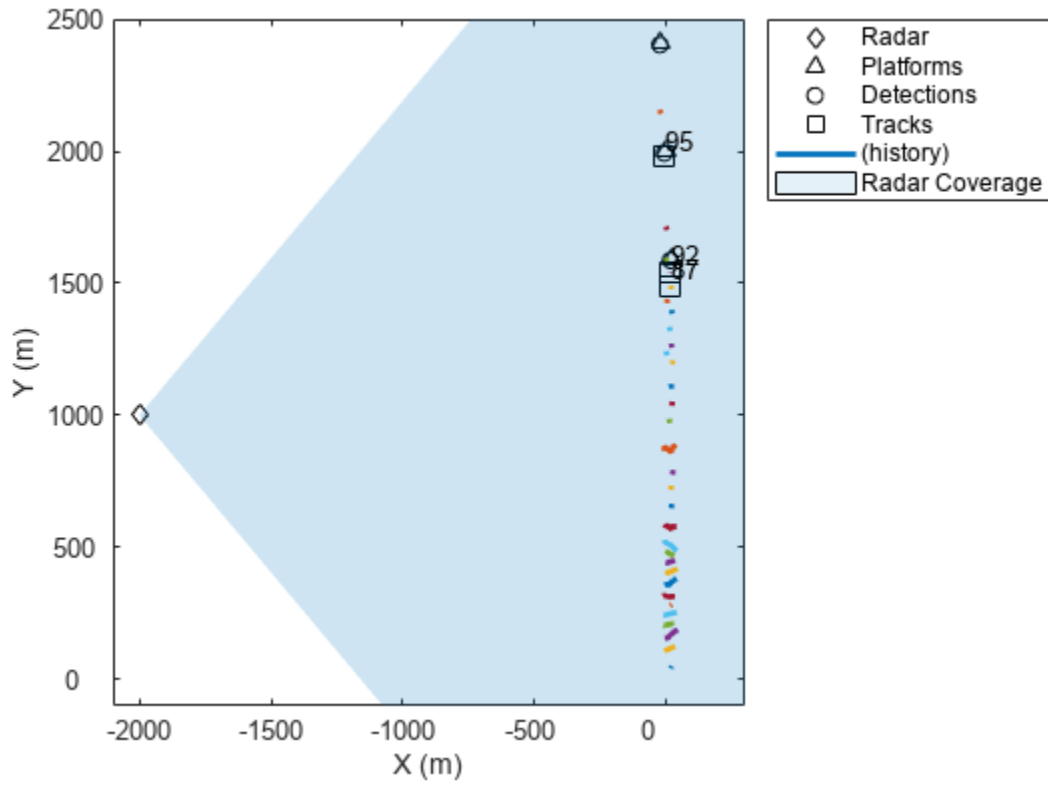


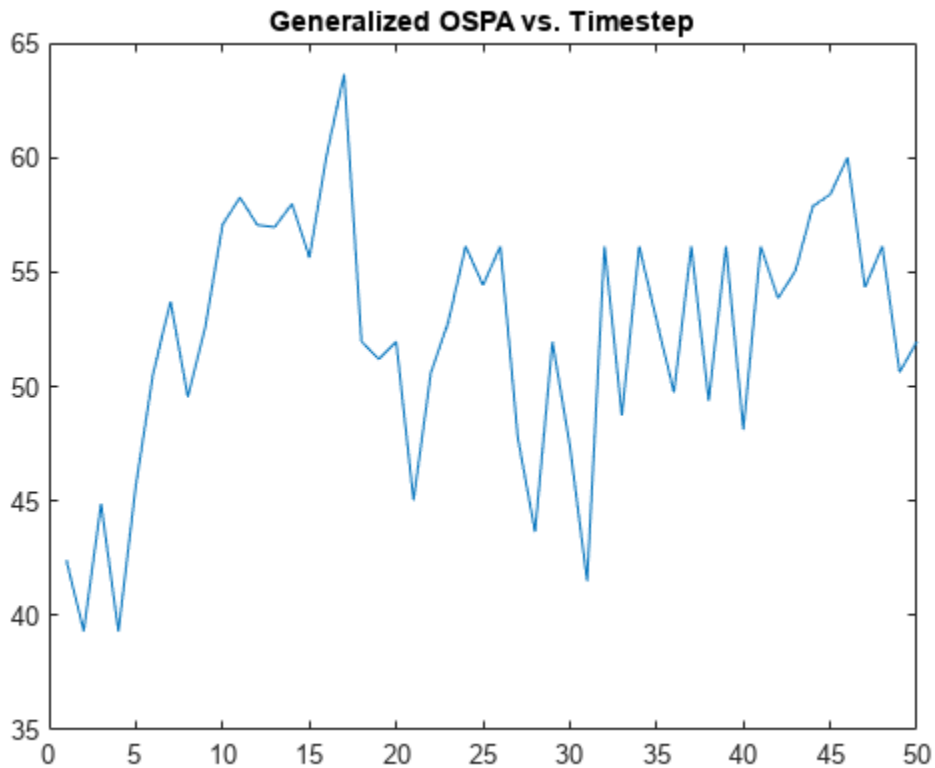


Filter Initialization Function

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([20 -10 0; 20 0.8*v*stopTime-10 0], [0 stopTime]);
p2.Trajectory = waypointTrajectory([0 0 0; 0 v*stopTime 0], [0 stopTime]);
p3.Trajectory = waypointTrajectory([-20 10 0; -20 1.2*v*stopTime+10 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.RangeLimits(2) = 3000;
rerunScenario(scenario, tracker, tgm, tp);
```

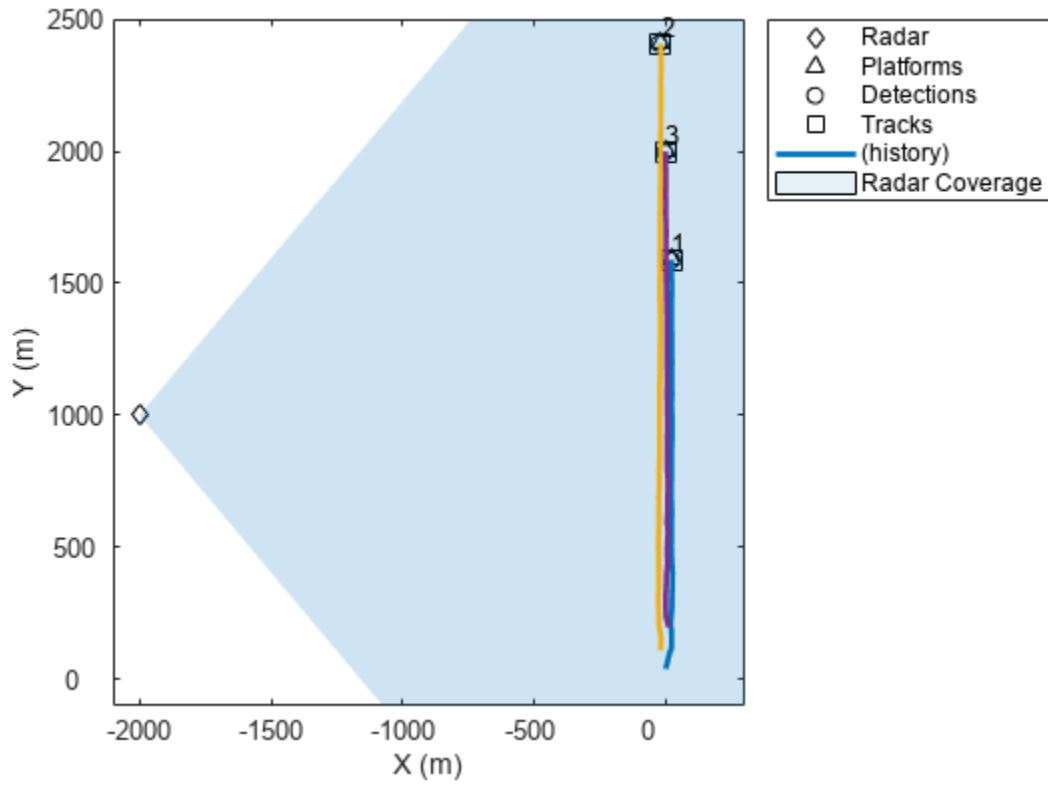


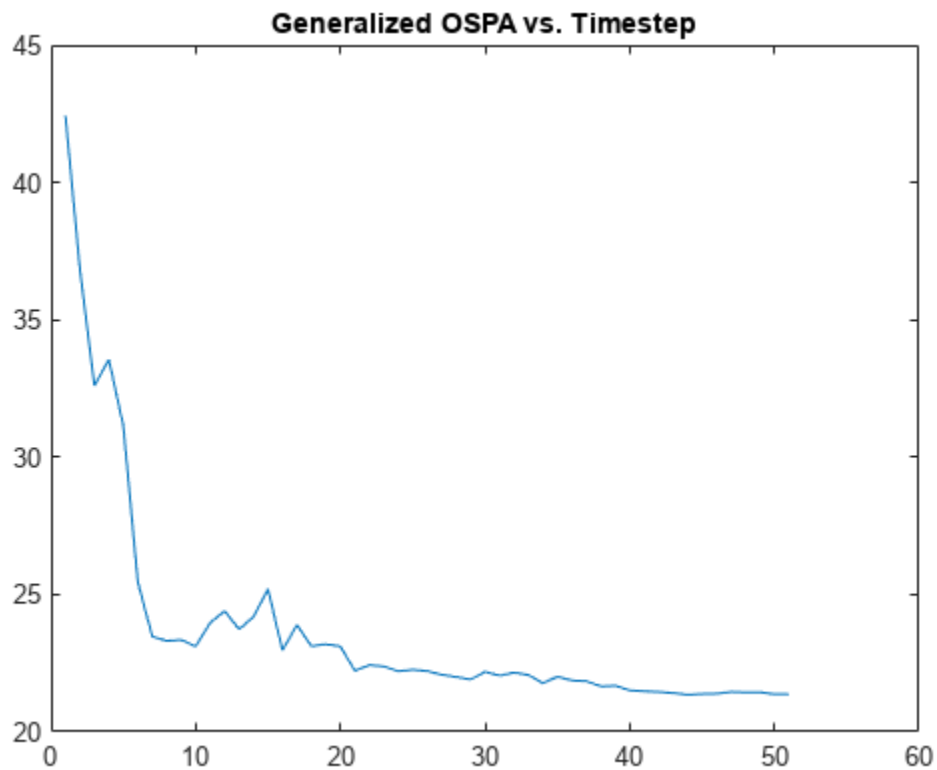


As you probably expected, the tracker could not establish stable tracks of the three objects. A possible solution is to increase the `AssignmentThreshold` even further to allow tracks to be updated. However, this option may increase the chance of random false tracks being assigned to multiple detections 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);
```





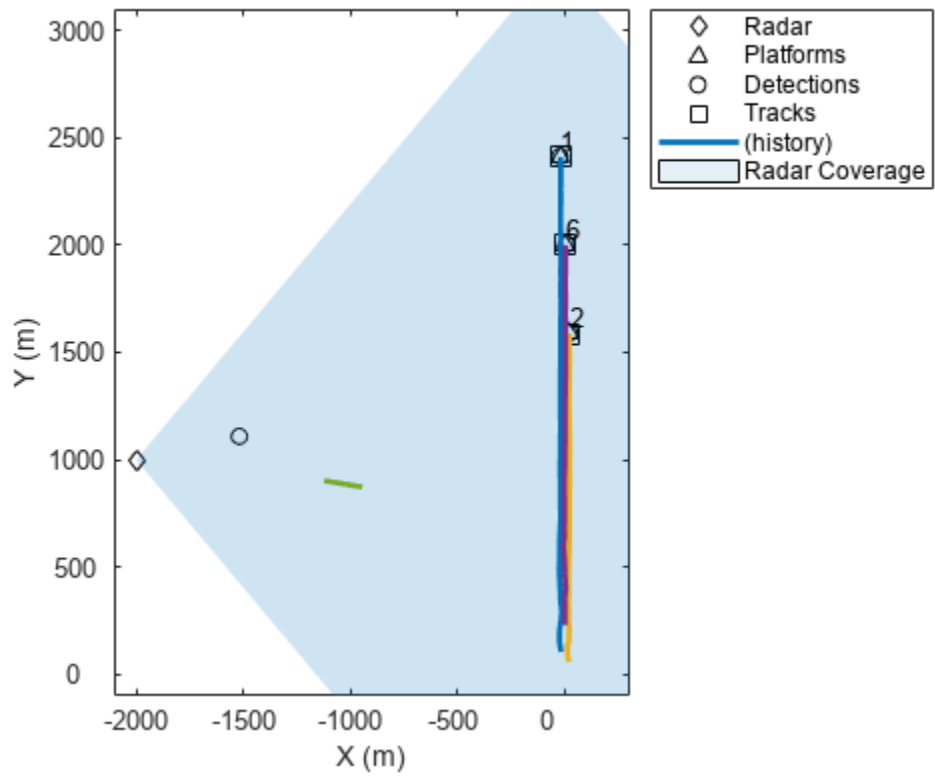
You observe that the tracker is once again able to maintain the tracks, and there are 3 tracks for the three moving objects, even though they are moving faster now. The GOSPA value is also reduced after a few steps.

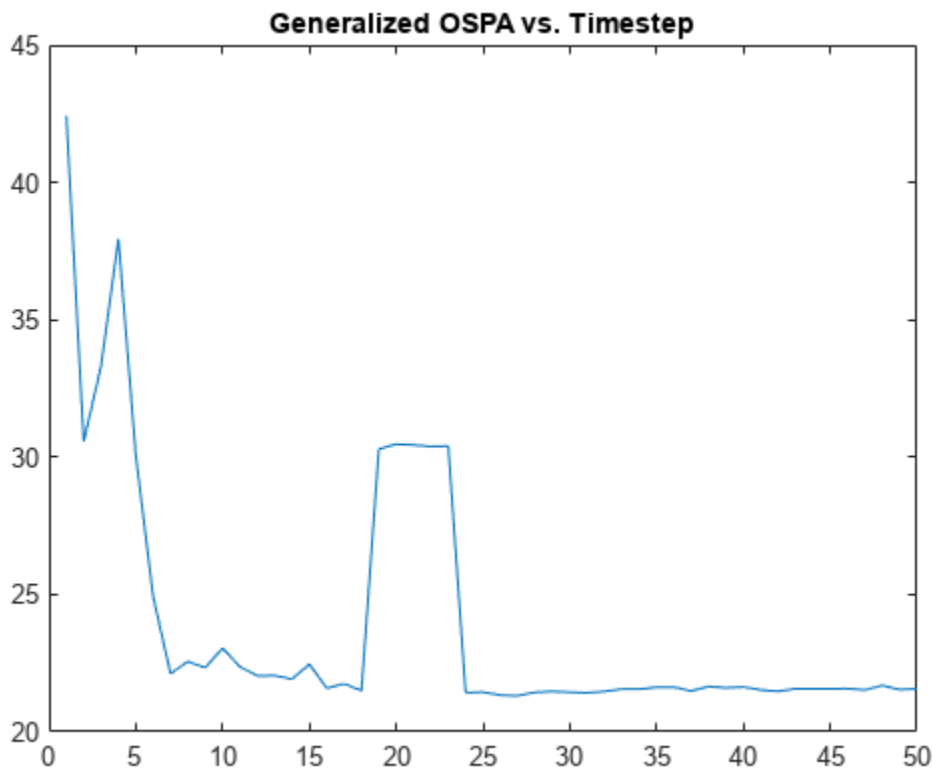
Confirmation and Deletion Thresholds

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);
```

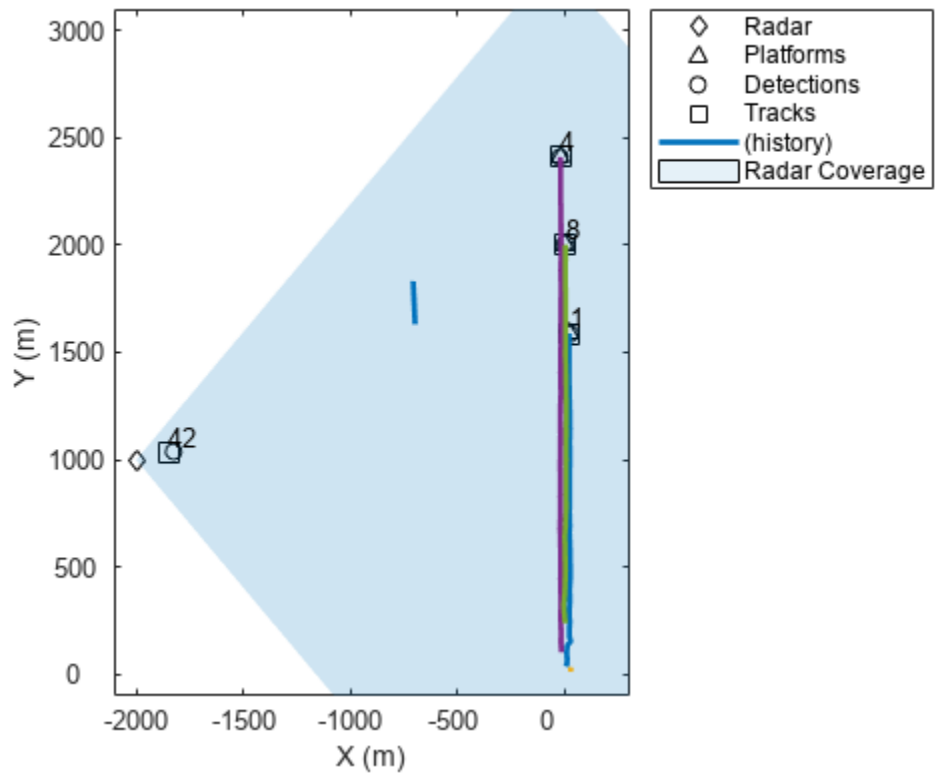



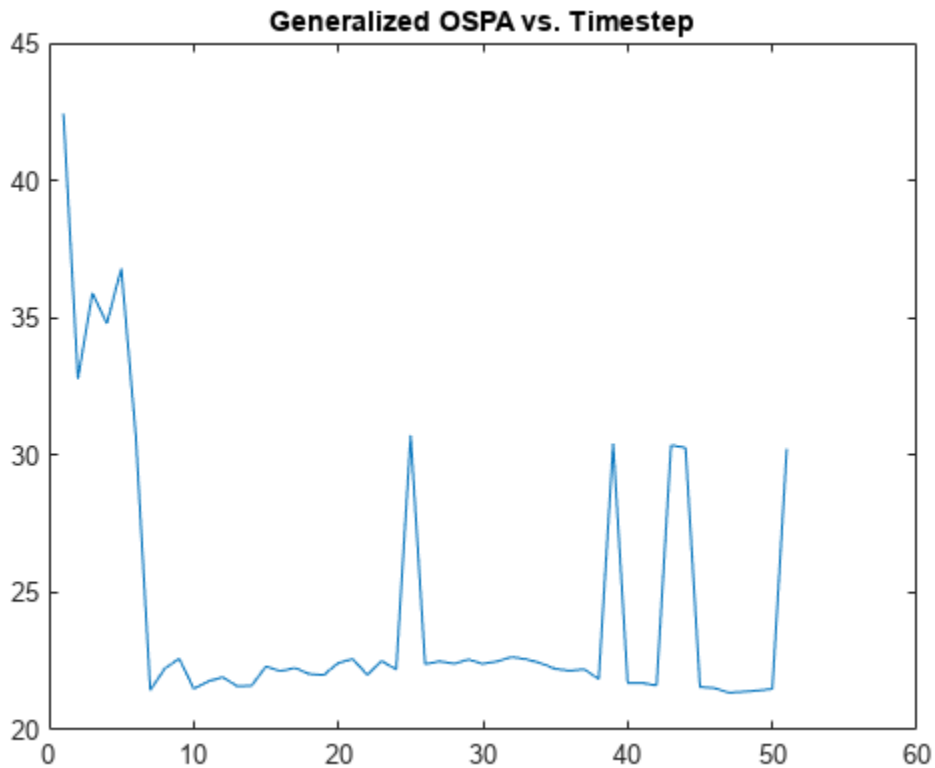


There are a few false tracks being created (not all of them shown) and they increase the GOSPA value.

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);
```





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.

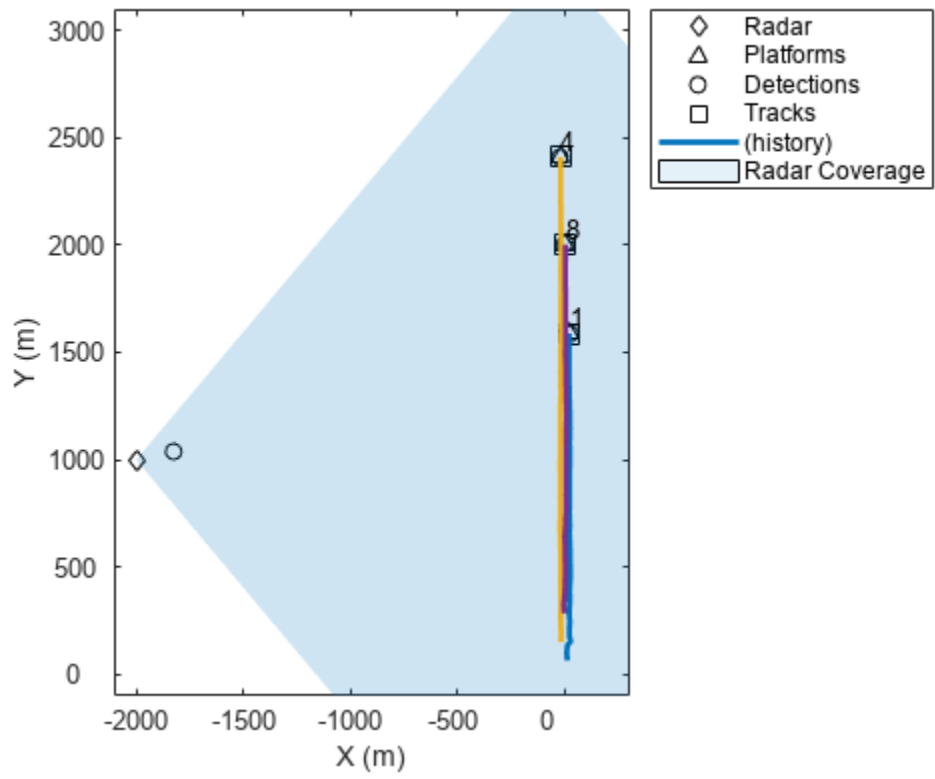
So, you want to make the confirmation of new tracks more stringent in attempt to reduce the number of false tracks. Consider the tracker `ConfirmationThreshold` property.

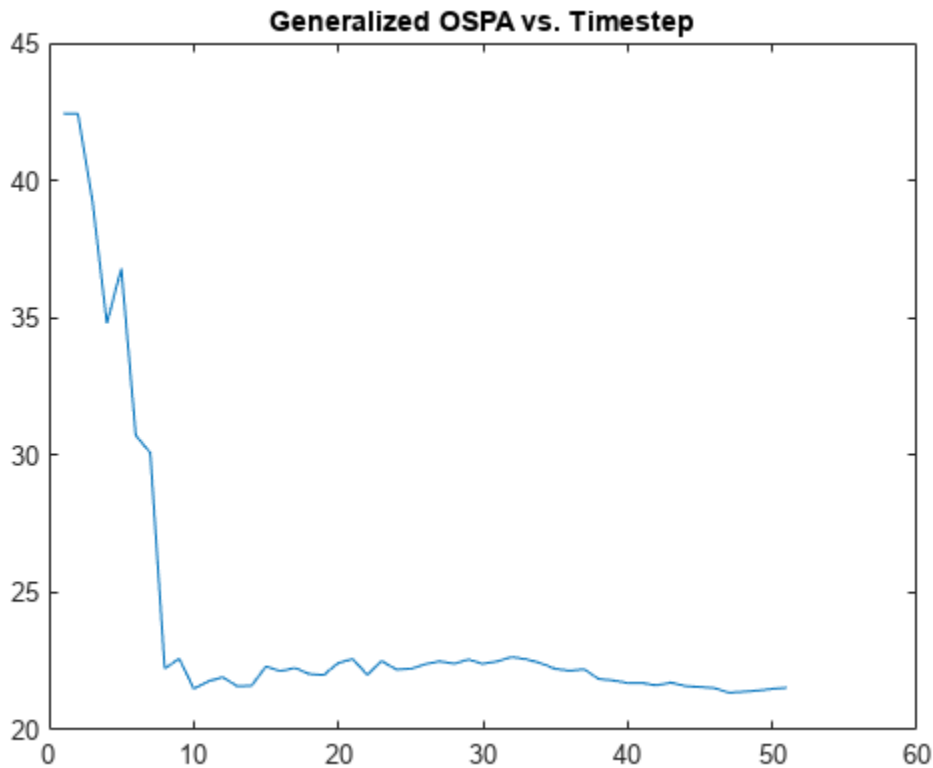
```
disp(tracker.ConfirmationThreshold);
```

```
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

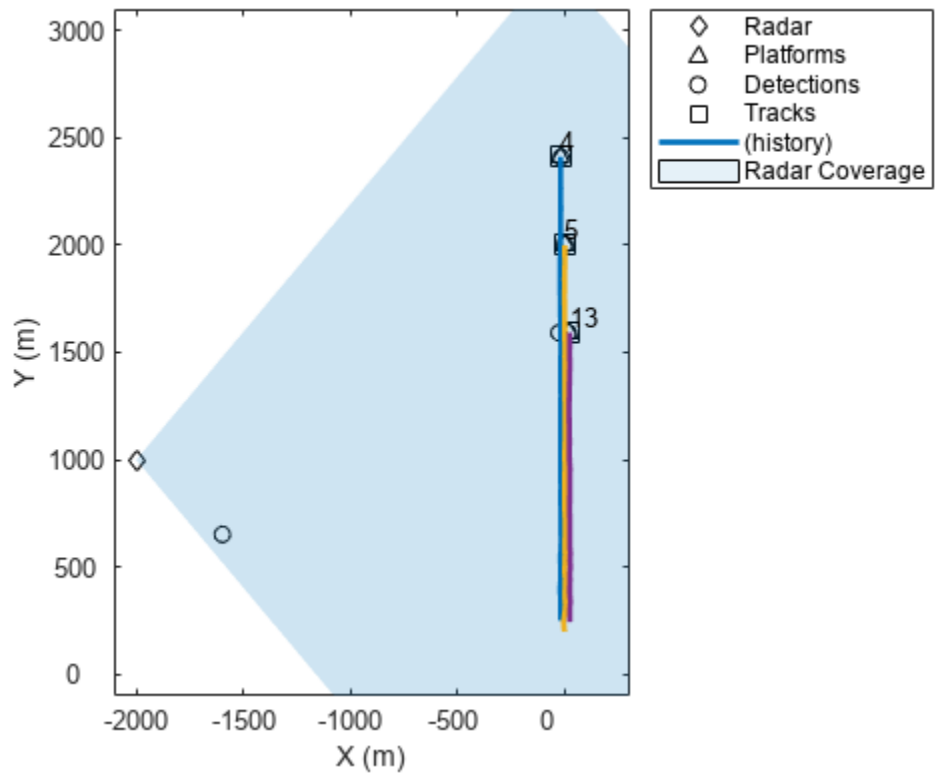
Making the confirmation more stringent allowed you to eliminate false tracks. However, the tracker still initializes a track for each false detection, which you can observe by looking at the number of tracks the tracker currently maintains.

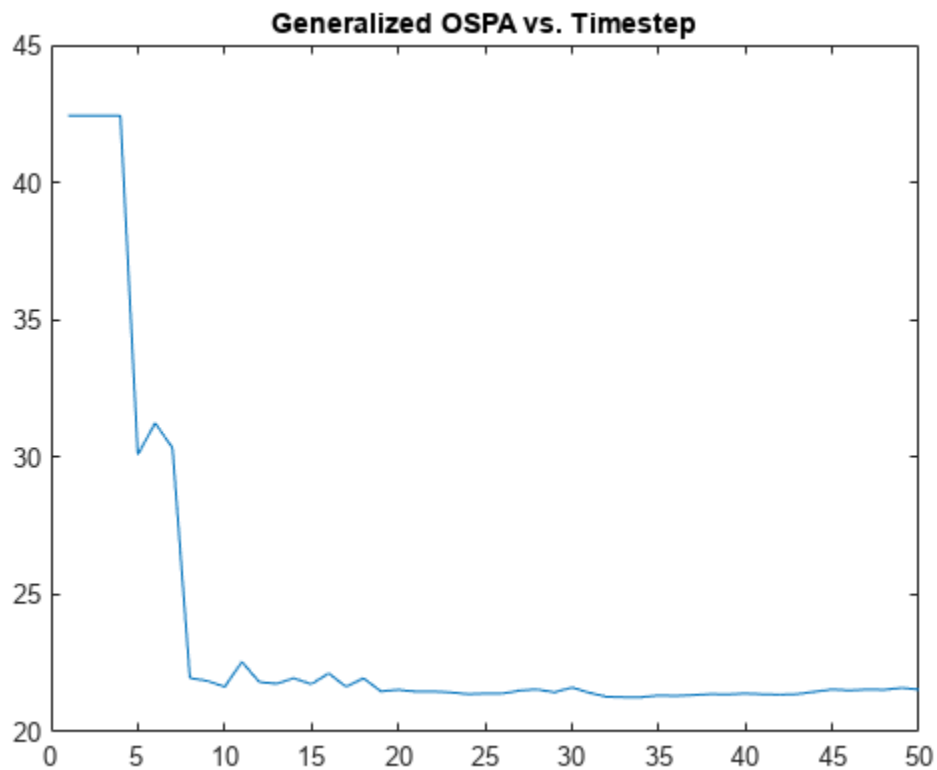
The number of tracks fluctuates through the lifetime of the tracker. However, you don't want it to exceed the maximum number of tracks that the tracker can maintain, defined by `MaxNumTracks`, which is by default 100. If the tracker exceeds this number, it issues a warning that a new track could not 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);
```



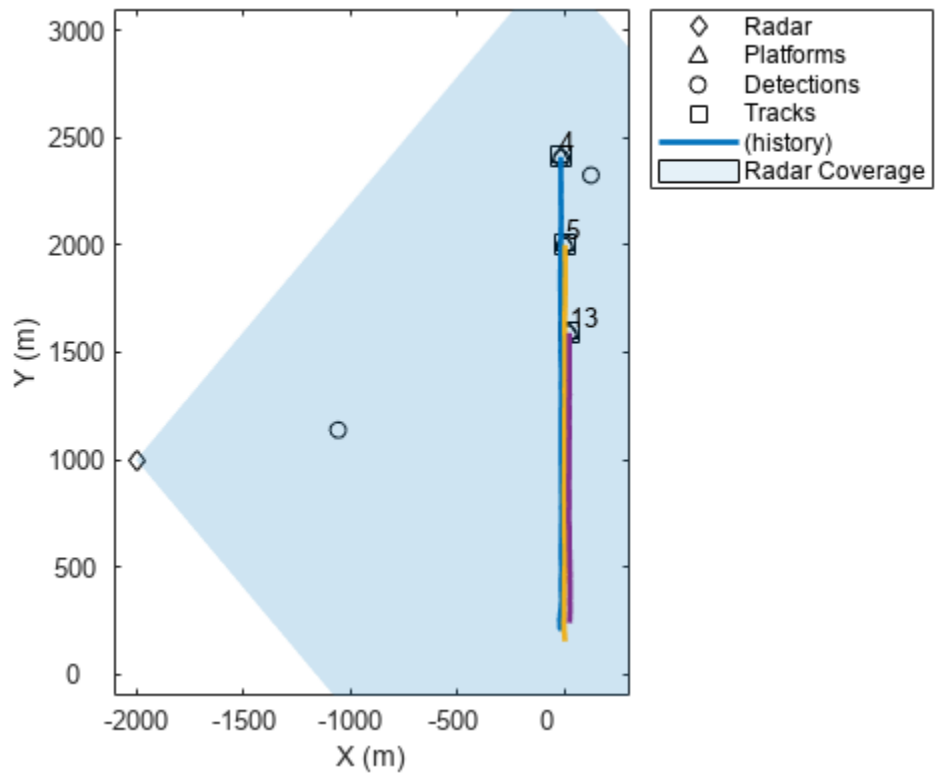


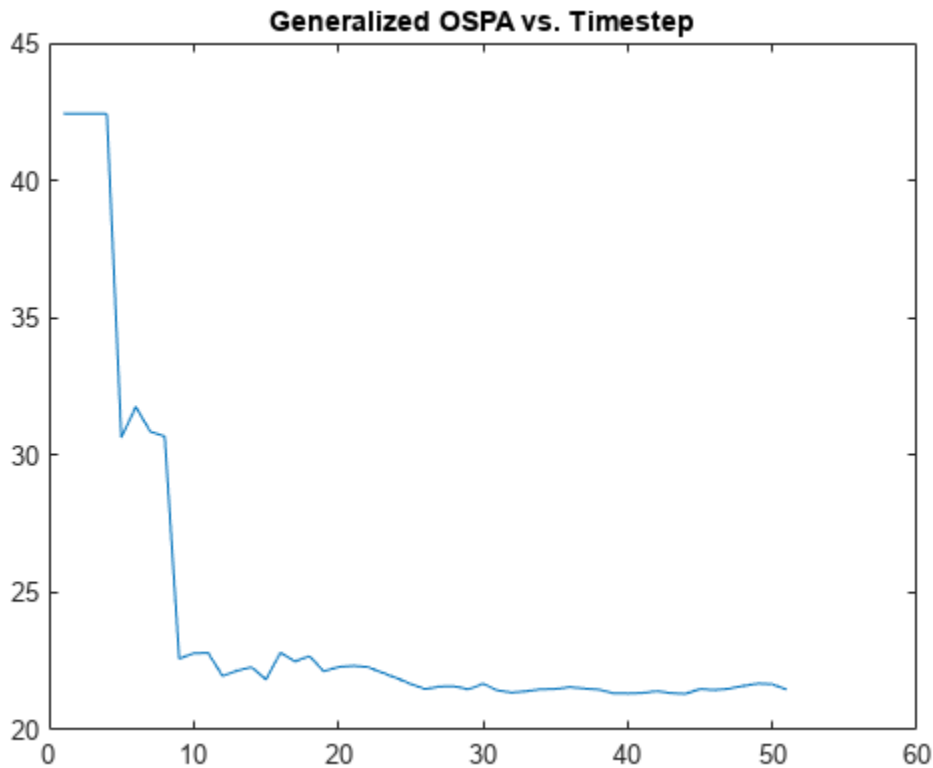
Note that making track confirmation more stringent requires more steps to confirm tracks. As a result, for the first few steps, there are no confirmed tracks and the GOSPA value remains high. You can compare that with the GOSPA graphs in the beginning of this example.

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);
```



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 6-296, “Introduction to Track Logic” on page 6-78, “Tracking Closely Spaced Targets Under Ambiguity” on page 6-168, and “Track Point Targets in Dense Clutter Using GM-PHD Tracker” on page 6-466.

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.

```
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.

```
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.

```
function jpda = helperGNN2JPDA(gnn)
    jpda = trackerJPDA(...
```

```
    '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

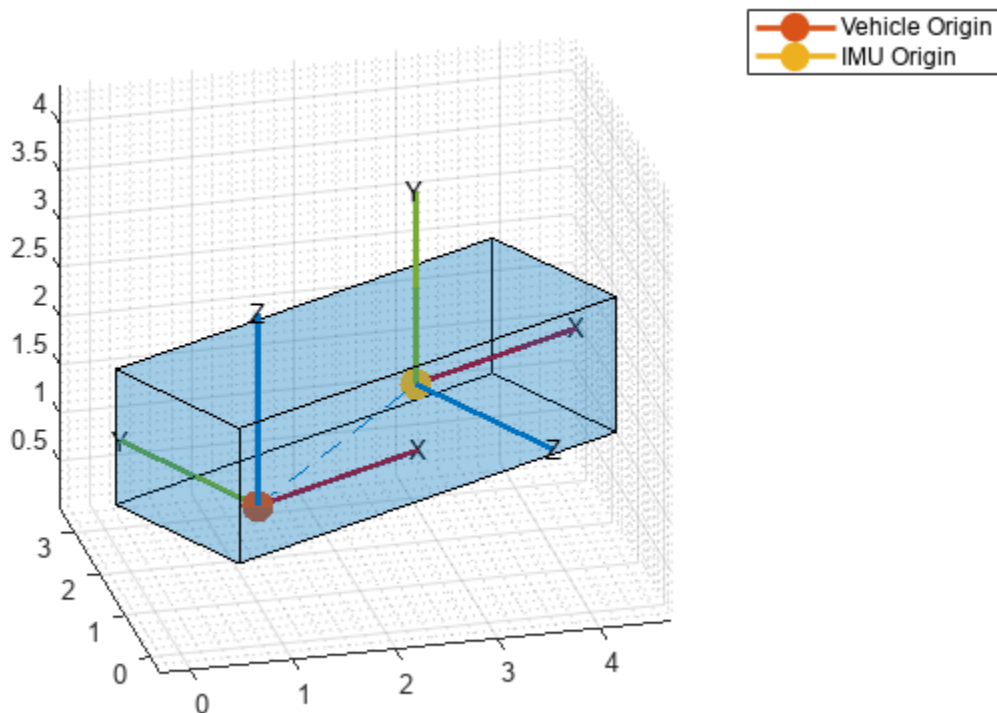
Create two 9-axis `imuSensor` objects composed of accelerometer, gyroscope, and magnetometer sensors. One `imuSensor` object generates readings of an IMU mounted at the vehicle's origin and the other one generates readings of an IMU mounted at the driver's seat. Next, specify the offset between the vehicle 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);
```

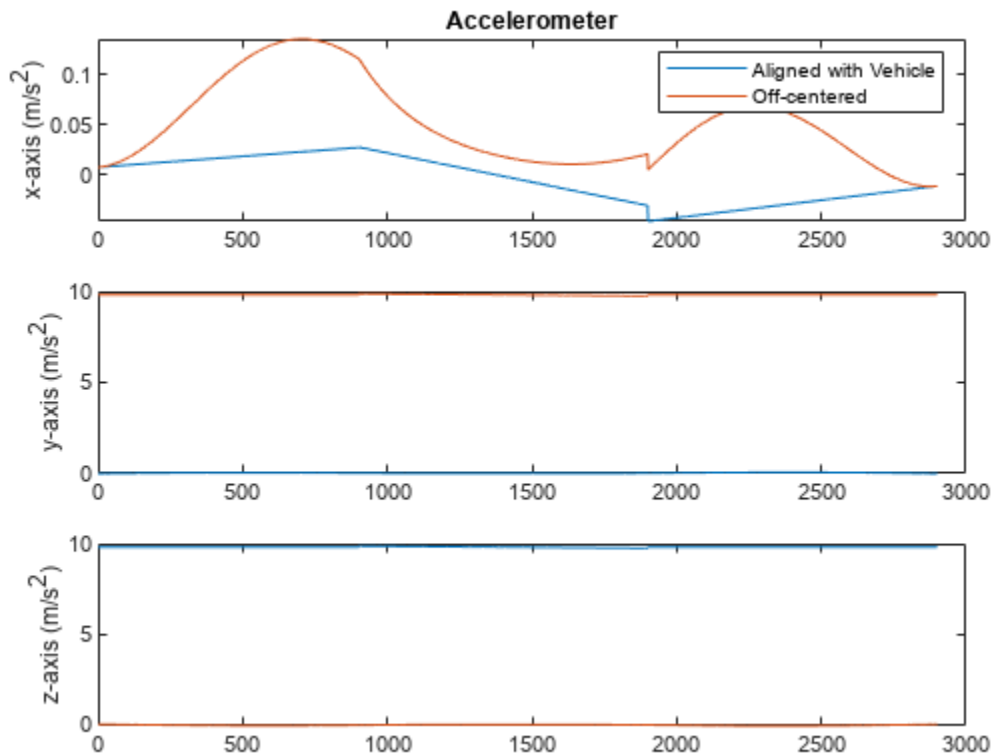
Compare Accelerometer Readings

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.

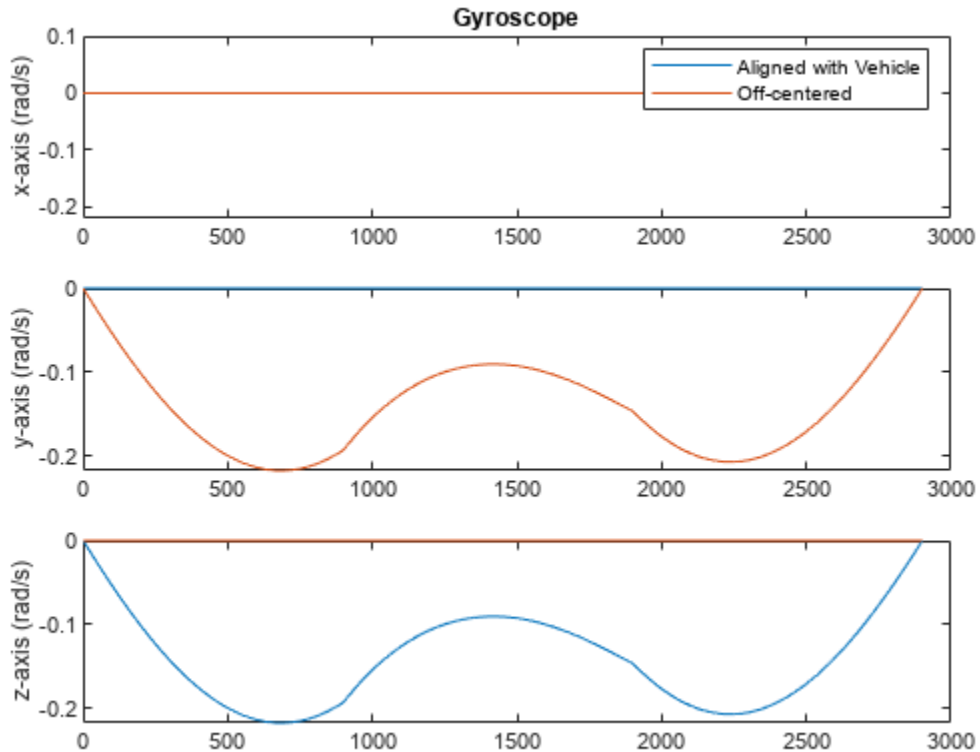
```

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)

```



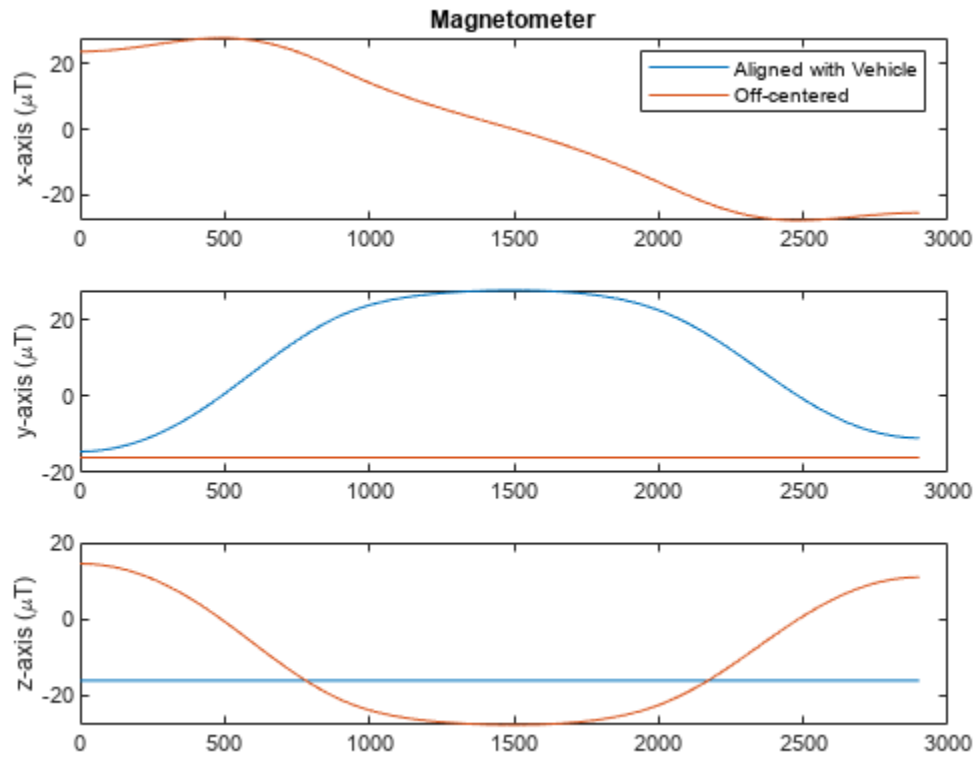
```
plot([gyro(:,3), gyroMounted(:,3)])
ylabel('z-axis (rad/s)')
```



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 Multipath GPS Reading Errors Using Residual Filtering in Inertial Sensor Fusion

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;
    end
end

```

```

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

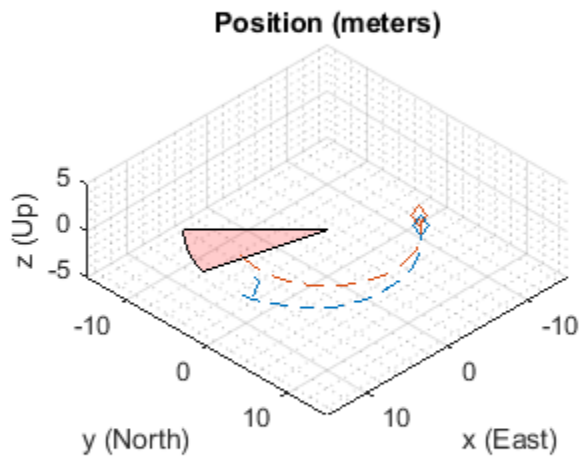
    viewerNoDetection(estPosition(idx,:), ...
        estOrientation(idx,:), truePosition(idx,:), ...
        trueOrientation(idx,:));
end
end

% This next section of code runs at the GPS sample rate.

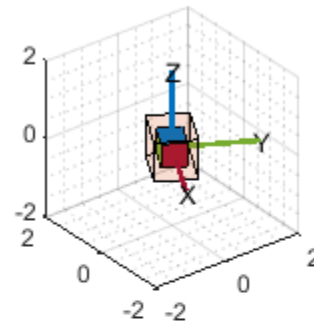
% Update the filter states based on the GPS data.
fusegps(gndFusionWithDetection, lla(sampleIdx,:), Rpos, ...
    gpsVel(sampleIdx,:), Rvel);

% 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
end

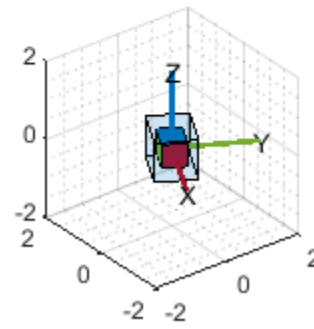
```

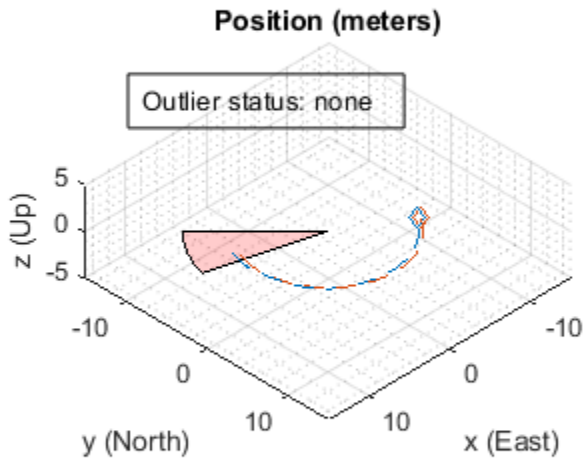
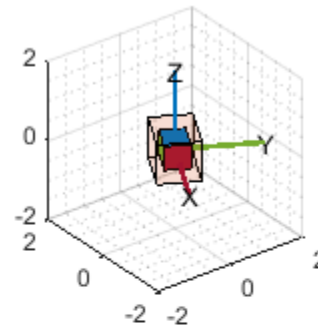
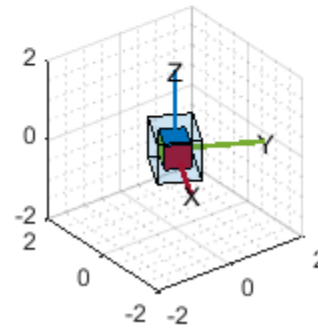


Orientation - Ground Truth



Orientation - Estimated




Orientation - Ground Truth

Orientation - Estimated


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;

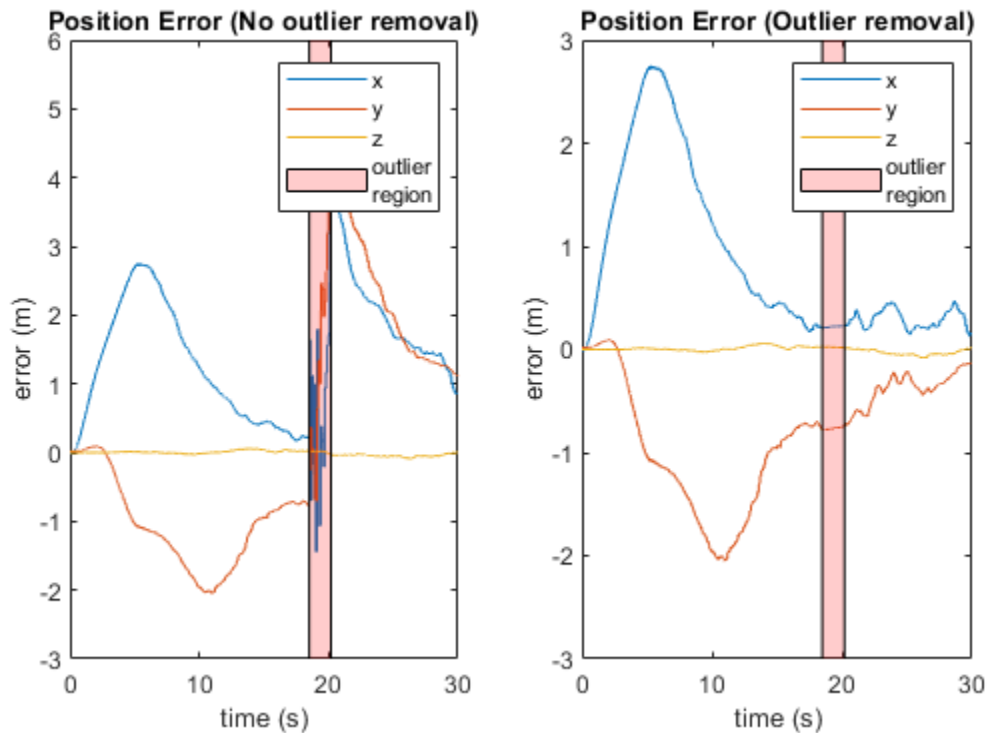
% Plot results.
t = (0:size(posd,1)-1).' ./ imuFs;
figure('Units', 'normalized', 'Position', [0.2615 0.2833 0.4552 0.3700])
subplot(1, 2, 1)
plot(t, posdNoCheck)
ax = gca;
yLims = get(ax, 'YLim');
hold on
mi = multipathIndices;
fill([t(mi(1)), t(mi(1)), t(mi(2)), t(mi(2))], [7 -5 -5 7], ...
    [1 0 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.

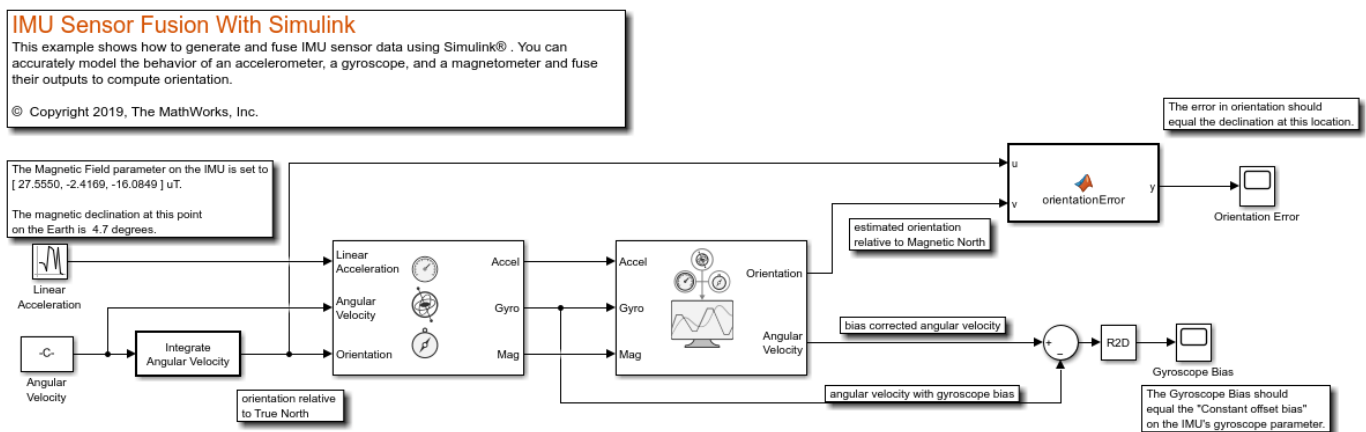
Attitude Heading and Reference System

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/s². 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 `fusionRadarSensor` in monostatic mode 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, 'InitialAdvance', 'UpdateInterval',...
    'StopTime', 3600, 'UpdateRate', 0.1);
```

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

$$\vec{a} = \frac{-\mu}{r^3} \vec{r} - 2\vec{\Omega} \times \frac{d\vec{r}}{dt} - \vec{\Omega} \times (\vec{\Omega} \times \vec{r}),$$

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 `fusionRadarSensor` 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
radar1 = fusionRadarSensor(1,...
    'UpdateRate',0.1,... 10 sec
    'ScanMode','No scanning',...
    'MountingAngles',[0 90 0],... look up
    'FieldOfView',[120;30],... degrees
    'ReferenceRange',2e6,... m
    'RangeLimits', [0 2e6],...
    '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 with trackingGlobeViewer

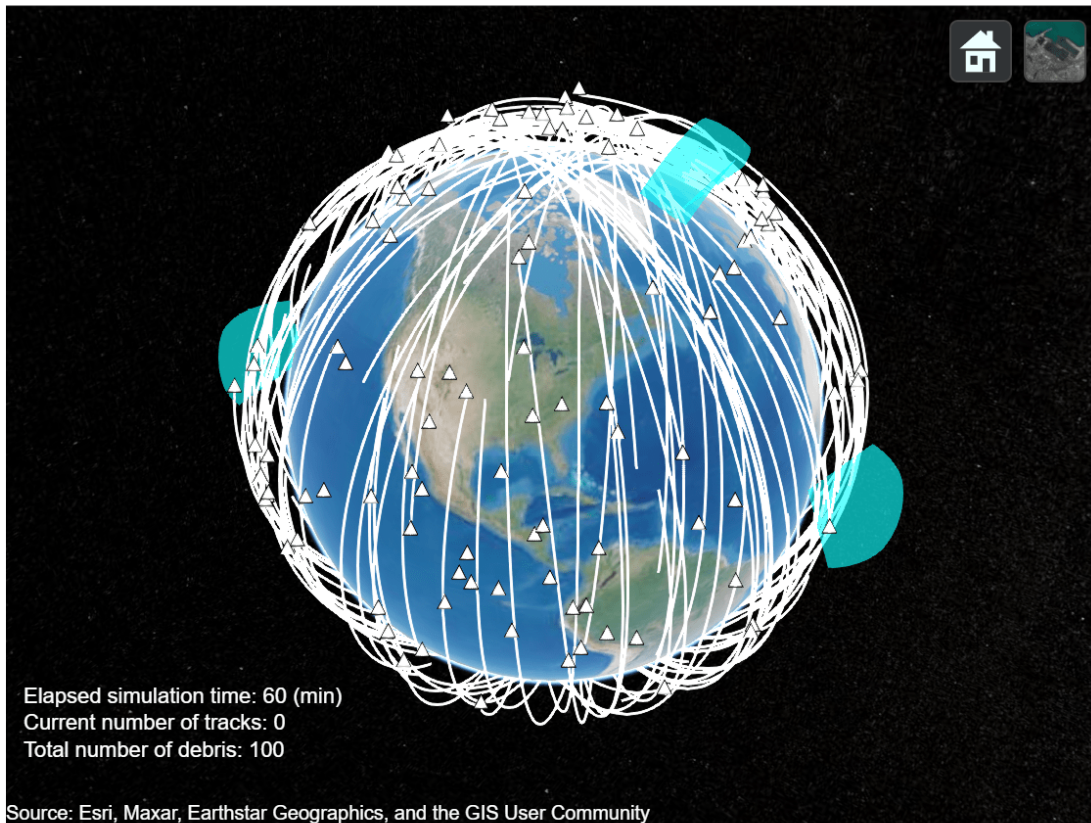
You use `trackingGlobeViewer` to visualize all the elements defined in the tracking scenario: individual debris objects and their trajectories, radar fans, radar detections, and tracks.

```
f = uifigure;
viewer = trackingGlobeViewer(f, 'Basemap','satellite','ShowDroppedTracks',false);
% Add info box on top of the globe viewer
infotext = simulationInfoText(0,0,0);
infobox = uilabel(f,'Text',infotext,'FontColor',[1 1 1],'FontSize',11,...
    'Position',[10 20 300 70],'Visible','on');

% Show radar beams on the globe
covcon = coverageConfig(scene);
plotCoverage(viewer,covcon, 'ECEF');

while advance(scene)
    infobox.Text = simulationInfoText(scene.SimulationTime, 0, numDebris);
    plotPlatform(viewer, debris, 'ECEF');
end

%Take a snapshot of the scenario
snapshot(viewer);
```



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.

```

% Define Tracker
tracker = trackerJPDA('FilterInitializationFcn',@initKeplerUKF,...
    'HasDetectableTrackIDsInput',true,...
    'ClutterDensity',1e-20,...
    'AssignmentThreshold',1e3,...
    'DeletionThreshold',[7 10]);

% Reset scenario, seed, and globe display
restart(scene);
scene.StopTime = 1800; % 30 min
clear(viewer);
% Reduce history depth for debris
viewer.PlatformHistoryDepth = 2;
% Show 10-sigma covariance ellipses for better visibility
viewer.NumCovarianceSigma = 10;
plotCoverage(viewer,covcon, 'ECEF');

% Initialize tracks
confTracks = objectTrack.empty(0,1);
while advance(scene)
    plotPlatform(viewer, debris);
    time = scene.SimulationTime;

    % Generate detections
    detections = detect(scene);
    plotDetection(viewer, detections, 'ECEF');

    % Generate and update tracks
    detectableInput = isDetectable(tracker,time, covcon);
    if isLocked(tracker) || ~isempty(detections)
        [confTracks, ~, allTracks,info] = tracker(detections,time,detectableInput);
        confTracks = deleteBadTracks(tracker,confTracks);
        plotTrack(viewer, confTracks, 'ECEF');
    end

    infobox.Text = simulationInfoText(time, numel(confTracks), numDebris);
% Move camera during simulation and take snapshots
switch time
    case 100
        campos(viewer,[90 150 5e6]);
        camorient(viewer,[0 -65 345]);
        drawnow
        im1 = snapshot(viewer);
    case 270
        campos(viewer,[60 -120 2.6e6]);
        camorient(viewer, [20 -45 20]);
        drawnow
    case 380
        campos(viewer,[60 -120 2.6e6]);
        camorient(viewer, [20 -45 20]);
        drawnow
        im2 = snapshot(viewer);
    case 400
        % reset
        campos(viewer,[17.3 -67.2 2.400e7]);
        camorient(viewer, [360 -90 0]);
        drawnow
    case 1500

```

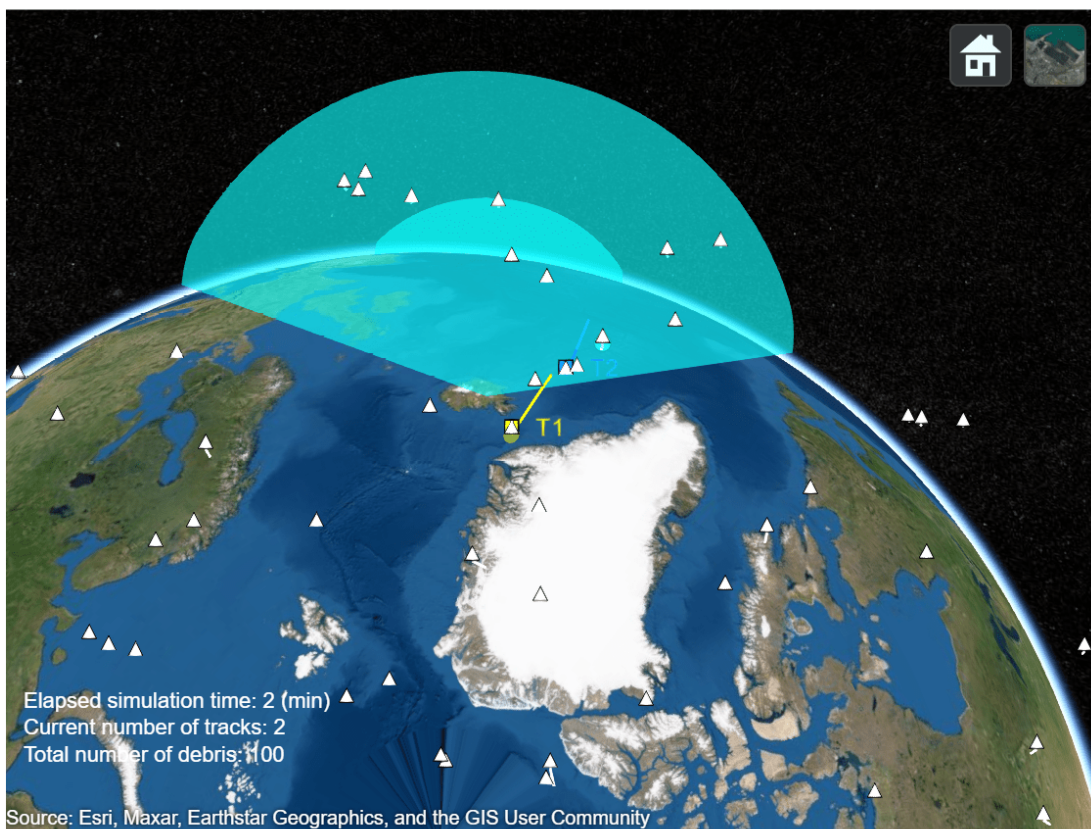
```

        campos(viewer,[54 2.3 6.09e6]);
        camorient(viewer, [0 -73 348]);
        drawnow
    case 1560
        im3 = snapshot(viewer);
        drawnow
    end
end

% Restore random seed
rng(s);

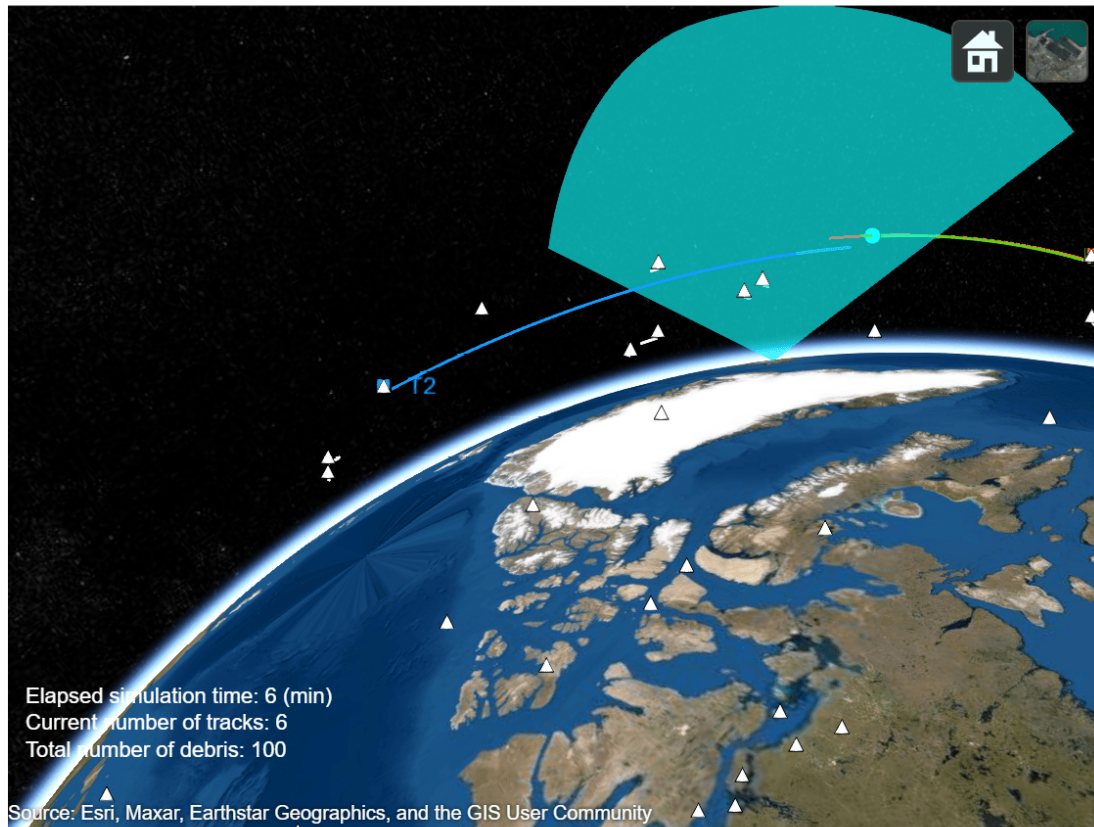
imshow(im1);

```



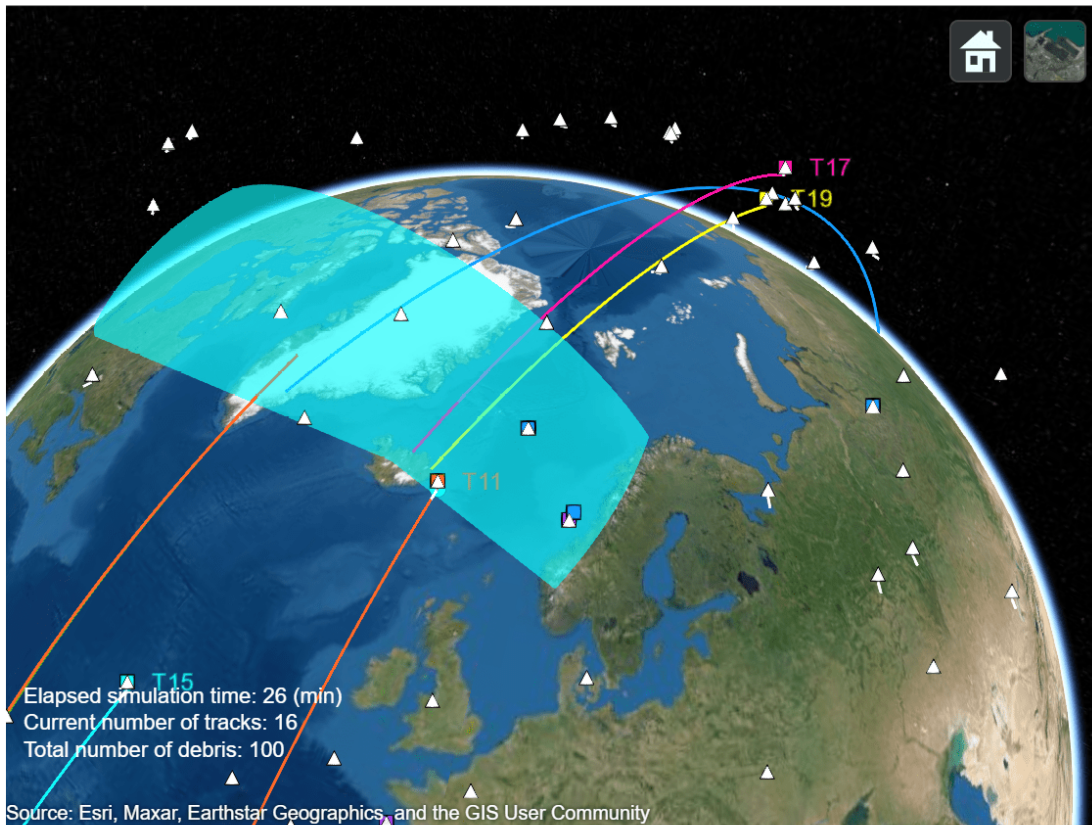
On the first snapshot, you can see an object being tracked as track T1 in yellow. This object was only detected twice, which was not enough to reduce the uncertainty of the track. Therefore, the size of its covariance ellipse is relatively large. You can also observe another track T2 in blue, which is 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 T15 (in light blue) is about to enter the radar surveillance area. Track T11 (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, 18 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 $[x; vx; y; vy; z; vz]$.

```

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`.

```

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.

```

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
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
```

simulationInfoText is used to display the simulation time, the number of debris, and the current number of tracks in the scenario.

```
function text = simulationInfoText(time,numTracks, numDebris)
    text = vertcat(string(['Elapsed simulation time: ' num2str(round(time/60)) ' (min)']
        string(['Current number of tracks: ' num2str(numTracks)]),...
        string(['Total number of debris: ' num2str(numDebris)]));
    drawnow limitrate
end
```

References

[1] https://www.esa.int/Space_Safety/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 times, 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 control centers.

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);
```

Define the airplane model and 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 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, respectively. 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 a `gpsSensor` model with a position accuracy of 50 m and a velocity accuracy of 10 m/s to configure the `adsbTransponder` model. You also use a more realistic RCS signature for the airplane, inspired by that of a Boeing 737.

```

posAccuracy = 50; % meters
velAccuracy = 10; % m/s

gps = gpsSensor('PositionInputFormat','Geodetic','HorizontalPositionAccuracy',posAccuracy,...
    'VerticalPositionAccuracy', posAccuracy,'VelocityAccuracy',10);

adsbTx = adsbTransponder('MW2020','UpdateRate', 1,'GPS', gps,'Category',adsbCategory.Large);

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 a 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 North East Down 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 Earth-Centered Earth-Fixed (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 = fusionRadarSensor(1,'UpdateRate',updaterate,...
    'FieldOfView',[360 ; 30.001],...
    'HasElevation',true,...
    'ScanMode','Mechanical',...
    'MechanicalAzimuthLimits',[0 360],...
    'MechanicalElevationLimits',[-30 0],...
    'HasINS',true,...
    'HasRangeRate',true,...
    'HasFalseAlarms',false,...
    'ReferenceRange',463000,...

```

```

    'ReferenceRCS',0,...
    'RangeLimits',[0 463000],...
    '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

```

You use `adsbReceiver` to model the reception of ADS-B messages. An ADS-B message contains the position measured by the airplane own GPS instrument. The message is usually encoded and broadcasted on 1090 MHz channel for nearby ADS-B receivers to pick up. You define a reception range of 200 km around each surveillance station. In this example you assume that surveillance stations have perfect communication between each other. Therefore, a central receiver picks up broadcasted ADS-B messages within range of at least one station.

```

% Define central adsbReceiver
adsbRx = adsbReceiver('ReceiverIndex',2);
adsbRange = 2e5;

```

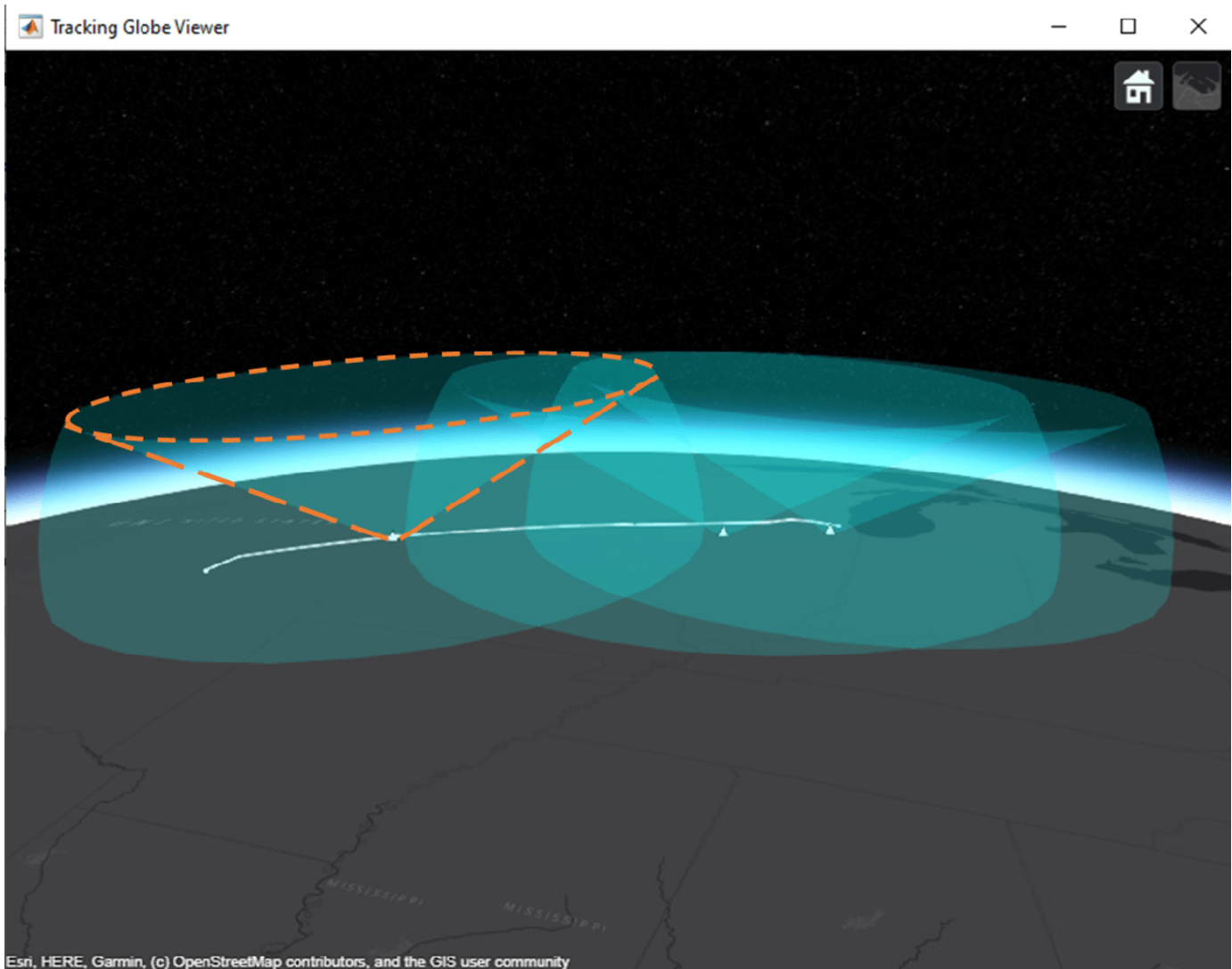
Visualize the scene

You use the `trackingGlobeViewer` to display platforms, trajectories, detections, and tracks on Earth.

```

viewer = trackingGlobeViewer('Basemap','streets-dark',...
    'TrackLabelScale',1.3,'TrackHistoryDepth',4000,...
    'CoverageMode','Coverage');
% Show radar sites
plotPlatform(viewer,scene.Platforms(2:end));
% Show radar coverage
covcon = coverageConfig(scene);
plotCoverage(viewer,covcon,'ECEP','Alpha',0.1);
% Show flight route
plotTrajectory(viewer,flightRoute);

```

Surveillance radars have an antenna blind cone, sometimes referred to as "cone of silence". It is a volume of space, directly above the radar, that cannot be surveilled due to antenna scanning limitations. Overlapping coverages in networks of radars is a common mitigation strategy for this blind cone region. With an overlapping strategy, however, there can still be areas not fully covered by the network. In this example, the southernmost radar's cone of silence (shown in orange in the picture above) is only partially covered by the adjacent radar in the network. This creates a blind spot where the airplane will not be detected by any radar.

Define a central radar tracker and a track fuser.

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.

```
radarGNN = trackerGNN('TrackerIndex',1,...
    'MaxNumTracks',50,...
    'FilterInitializationFcn',@initfilter,...
```

```

'ConfirmationThreshold',[3 5],...
'DeletionThreshold',[5 5],...
'AssignmentThreshold',[250 2000]);

```

You also fuse radar tracks with ADS-B tracks obtained from the ADS-B receiver. To do this, you configure a central `trackFuser` object. You set the `ConfirmationThreshold` and `DeletionThreshold` to account for the update rate difference between the ADS-B receiver rate at 1 Hz, and the radar tracker rate at 1/12 Hz. To allow for at least two radar tracks to be assigned to a central track, the number of hits must then be at least 2×12 .

```

fuser = trackFuser('FuserIndex',3,'MaxNumSources',2,...
'AssignmentThreshold',[250 500],...
'StateFusion','Intersection',...
'StateFusionParameters','trace',...
'ConfirmationThreshold',[2 3*12],...
'DeletionThreshold',[4 4]*12);

```

Track the flight using radars and ADS-B

In this section, you simulate the scenario and step the radar tracker, track fuser, ADS-B transponder, and receiver.

```

% Declare some variables
detBuffer = {};
radarTrackLog = {};
fusedTrackLog = {};
adsbTrackLog = {};
images = {};
snapTimes = [84,564,1128, 2083];

% Track labels and colors
adsblabel = "      ADS-B";
radarlabel = "  Radar";
fusedlabel = string(sprintf('%s\n',"","Fused"));
adsbclr = [183 70 255]/255;
radarclr = [255 255 17]/255;
fusedclr = [255 105 41]/255;

while advance(scene)
    time = scene.SimulationTime;

    % Update position of airplane on the globe
    plotPlatform(viewer,airplane,'TrajectoryMode','None');

    % Generate and plot synthetic radar detections
    [dets, configs] = detect(scene);
    dets = postProcessDetection(dets);
    detBuffer = [detBuffer; dets]; %#ok<AGROW>
    plotDetection(viewer,detBuffer,'ECEF');

    % tracks
    adsbTracks = objectTrack.empty;
    radarTracks = objectTrack.empty;
    fusedTracks = objectTrack.empty;

    % Generate ADS-B messages
    truePose = pose(airplane, 'true','CoordinateSystem','Geodetic');
    adsbMessage = adsbTx(truePose.Position, truePose.Velocity);

```

```

% Messages can be received within range of each surveillance station
if isWithinRange(radarsitesLLA, truePose.Position, adsbRange)
    adsbTracks = adsbRx(adsbMessage, time);
end

% Update radar tracker at end of scan
if all([configs.IsValidTime]) && (~isempty(detBuffer) || isLocked(radarGNN))
    radarTracks = radarGNN(detBuffer,time);
    detBuffer = {};
end

% Fuse tracks
if isLocked(fuser) || ~isempty([adsbTracks;radarTracks])
    fusedTracks = fuser([adsbTracks;radarTracks],time);
end

% plot tracks only once every radar scan
if all([configs.IsValidTime])

    labels = [repmat(adsblabel,1,numel(adsbTracks)),...
              repmat(radarlabel,1,numel(radarTracks)),...
              repmat(fusedlabel,1,numel(fusedTracks))];

    colors = [repmat(adsbclr, numel(adsbTracks), 1) ;...
              repmat(radarclr, numel(radarTracks), 1);...
              repmat(fusedclr, numel(fusedTracks), 1)];

    plotTrack(viewer,[adsbTracks; radarTracks; fusedTracks],'ECEF',...
              'LineStyle','Custom',"CustomLabel",labels,'Color',colors,...
              'LineWidth',3);
end

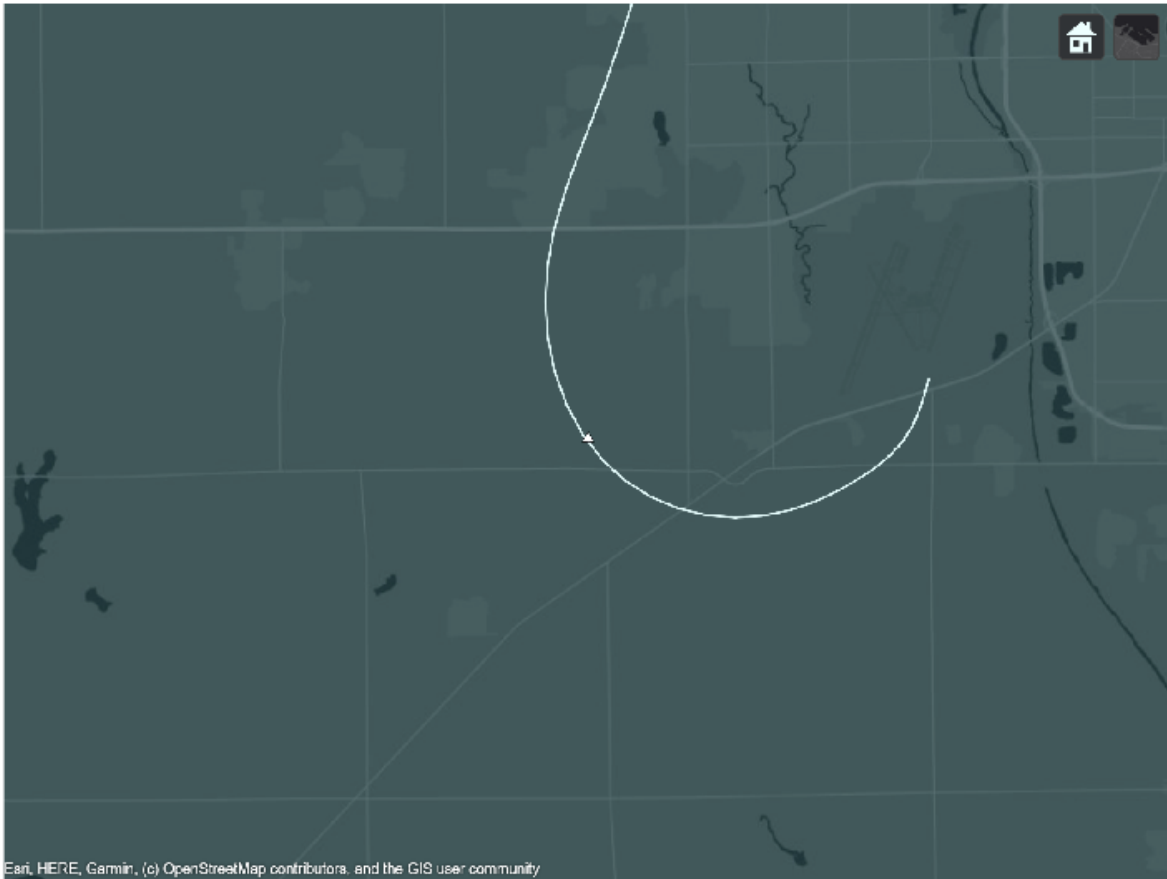
% Record the estimated airplane data for metrics
fusedTrackLog = [fusedTrackLog, {fusedTracks}]; %#ok<AGROW>
radarTrackLog = [radarTrackLog, {radarTracks}]; %#ok<AGROW>
adsbTrackLog = [adsbTrackLog, {adsbTracks}]; %#ok<AGROW>

% Move camera and take snapshots
images = moveCamera(viewer,airplane,time,snapTimes,images);

end

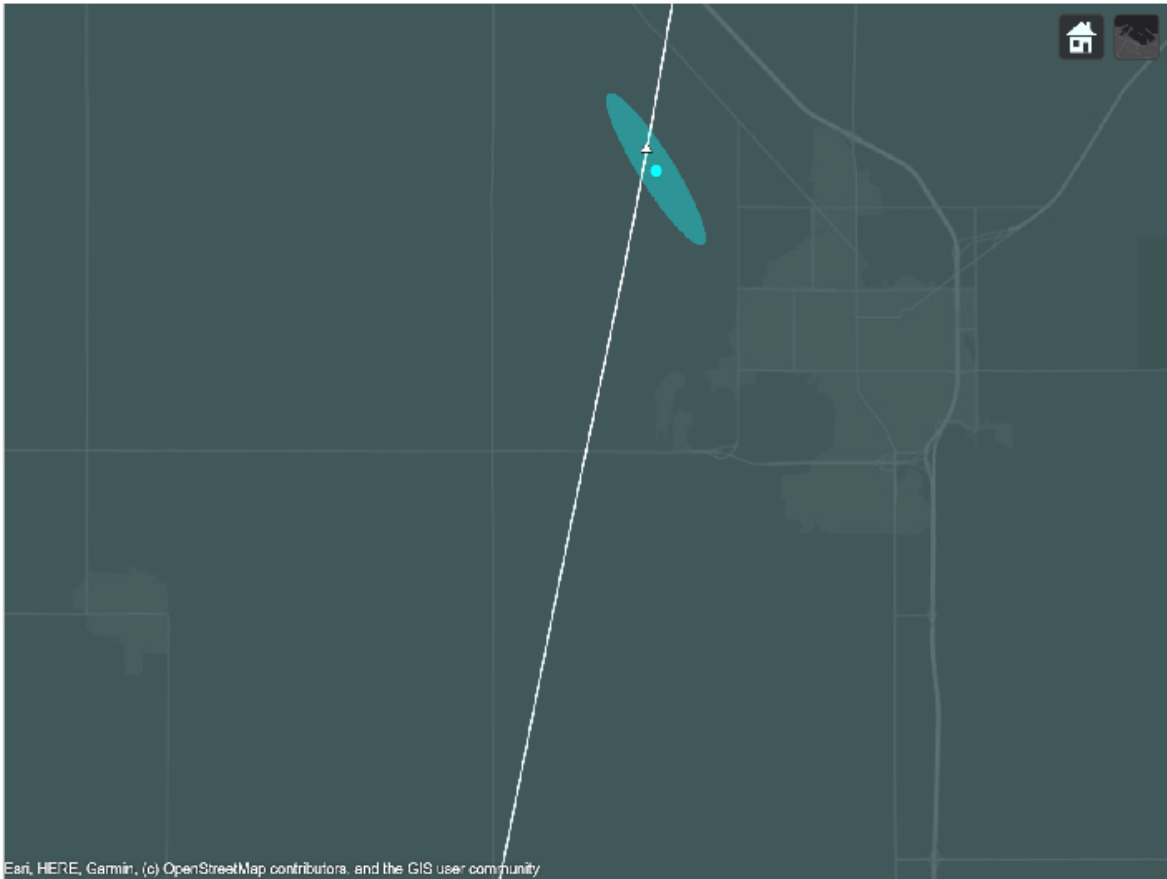
figure
imshow(images{1});

```



At the beginning of the scenario, the airplane is far from the southernmost surveillance station and no ADS-B message is transmitted. As a result, the airplane is tracked by radar only. 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 represents the true trajectory and the yellow line represents the trajectory estimated by the radar tracker. The first leg of the flight is far from any radar and the corresponding detections have high measurement noise. Additionally, the constant velocity motion model does not model the motion well during the initial flight turn after taking off. The radar track is passed to the fuser which outputs a fused track, shown in orange, following closely the radar track. The fused track is different from the radar track because the fuser adds its own process noise to the track state.

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



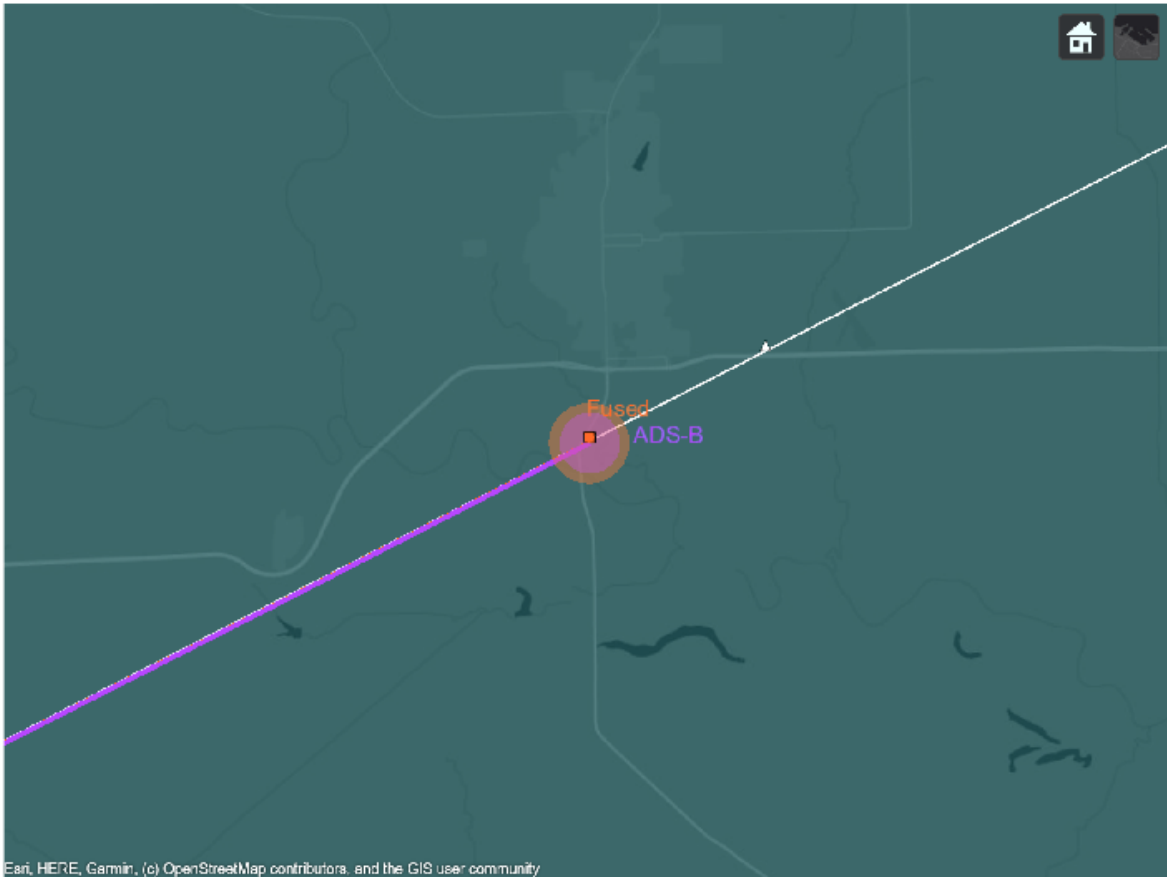
In the above snapshot, the airplane is within the ADS-B communication range and a new ADS-B track is established. The fuser processed this new track and improved the accuracy of the fused track.

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



In the above snapshot, the airplane enters the cone of silence. The radar tracker deletes the track after several updates without any new detections. At this point the fuser relies only on the ADS-B track to estimate the position of the airplane.

```
figure  
imshow(images{4});
```



As the airplane enters the area covered by the second and third surveillance radar stations, a new radar track is established. Detections from the two radar stations are fused by the radar tracker and the track fuser fuses the new radar track with the ADS-B track.

Analyze results

You compare the recorded track data for radar and ADS-B. The true position and velocity are available from the `geoTrajectory`. You use the OSPA metric to compare the quality of tracking from radar only, ADS-B only, and from radar fused with ADS-B. You use the default settings of the `trackOSPAMetric` object to compare NEES (normalized estimation error squared) for track position and detection assignment.

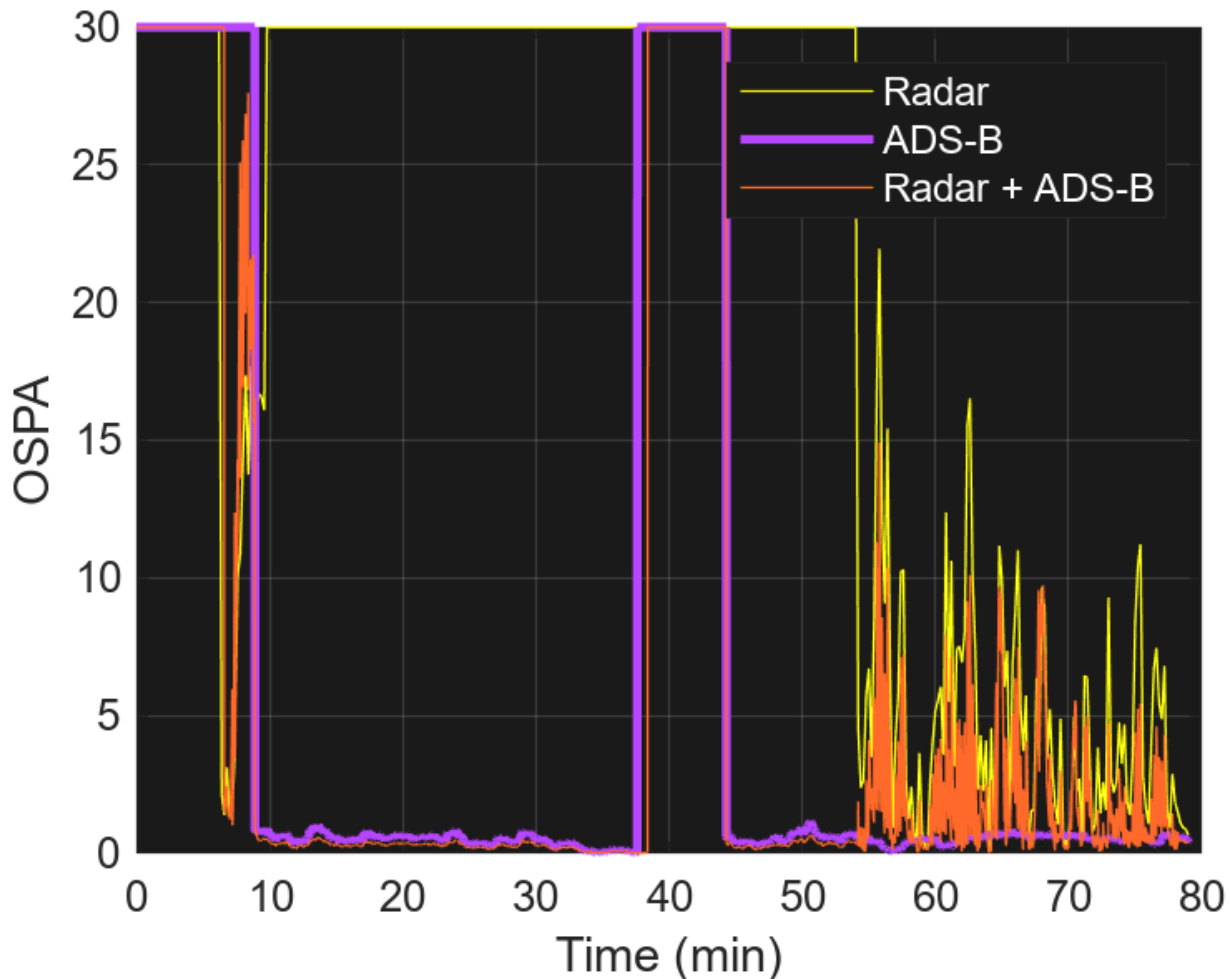
```
ospa = trackOSPAMetric;
radarospa = zeros(ceil(numel(radarTrackLog)/12),1);

% Compute radar tracker OSPA
for i=1:12:numel(radarTrackLog)
    tracks = radarTrackLog{i};
    [truths.Position,~,truths.Velocity] = lookupPose(flightRoute, i-1,'ECEF');
    radarospa(i) = ospa(tracks, truths);
end
```

```
% Compute ADS-B OSPA
adsbospa = zeros(numel(adsbTrackLog),1);
reset(ospa);
for i=1:numel(adsbTrackLog)
    tracks = adsbTrackLog{i};
    [truths.Position, ~,truths.Velocity] = lookupPose(flightRoute, i-1,'ECEF');
    adsbospa(i) = ospa(tracks, truths);
end

% Compute fused OSPA
fusedospa = zeros(numel(fusedTrackLog),1);
reset(ospa);
for i=1:numel(fusedTrackLog)
    tracks = fusedTrackLog{i};
    [truths.Position, ~,truths.Velocity] = lookupPose(flightRoute, i-1,'ECEF');
    fusedospa(i) = ospa(tracks, truths);
end

% Plot OSPA
figure
hold on
plot((0:12:(numel(radarTrackLog)-1))/60, radarospa(1:12:end), "Color",radarclr);
plot((0:(numel(adsbTrackLog)-1))/60,adsbospa, "Color", adsbclr, 'LineWidth',2);
plot((0:(numel(fusedTrackLog)-1))/60,fusedospa, "Color", fusedclr);
l=legend('Radar', 'ADS-B', 'Radar + ADS-B');
l.Color = [0.1 0.1 0.1];
l.TextColor = [ 1 1 1];
xlabel('Time (min)')
ylabel('OSPA')
ax = gca;
grid on;
box on;
ax.Color = [0.1 0.1 0.1];
ax.GridColor = [1 1 1];
```

The OSPA metric shows improvements obtained from fusing ADS-B tracks with radar tracks. From simulation time 19 min to 25 min, the radar only OSPA is high because the airplane is flying over the blind spot of the radar network. The availability of ADS-B in this area significantly increases the tracking performance, as indicated by the fused OSPA. Additionally, the metric shows two peaks at the beginning, which can be attributed to the poor performance of a constant-velocity filter during the initial turns of the trajectory and the unavailability of ADS-B. Around time 40 min, the ADS-B only OSPA is degraded by the loss of ADS-B availability in the region. In later segments of the simulation, both radar and ADS-B are available. Radar only OSPA is overall worse than ADS-B only. This is because the radars have poor vertical accuracy in comparison to a GPS.

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 modeled ADS-B reports and integrated them to the tracking solution. In this example, only a single flight was modeled. The benefit of ADS-B can be further

manifested when modeling multiple flights in scenarios where safe separation distances must be maintained by air traffic controllers.

Supporting functions

initfilter defines the extended Kalman filter used by `trackerGNN`. Airplane motion is well approximated by a constant velocity motion model. Therefore, use a relatively small process noise to allocate 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 = 100*filter.StateCovariance; % initcvekf uses measurement noise as the d
filter.ProcessNoise = 0.1;
end
```

postProcessDetection post-processes the detections by two operations:

- 1 It removes radar detections that lie below the surface of the Earth as these cannot be created by a real radar.
- 2 It increases the detection noise level for the velocity measurement component normal to the radial component.

```
function detsout = postProcessDetection(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);

velCovTransversal = 100^2;
for i=1:numel(detsout)
    velCov = detsout{i}.MeasurementNoise(4:6,4:6);
    [P,D] = eig(velCov);
    [m,ind] = min(diag(D));
    D = velCovTransversal * eye(3);
    D(ind,ind) = m;
    correctedCov = P*D*P';
    detsout{i}.MeasurementNoise(4:6,4:6) = correctedCov;
end
end
```

isWithinRange returns true if the airplane position is within ADS-B receiver range of any of the surveillance stations.

```
function flag = isWithinRange(stationsLLA, pos, range)
cartDistance = fusion.internal.frames.lla2ecef(stationsLLA) - fusion.internal.frames.lla2ecef(pos);
flag = any(vecnorm(cartDistance,2,2) < range);
end
```

moveCamera specifies new camera positions to follow the airplane and take snapshots.

```
function images = moveCamera(viewer, airplane, time, snapTimes,images)
if mod(time,12) == 0
    pos = airplane.Position;
    if time == 0
        campos(viewer, pos(1),pos(2), 2e4);
    elseif time == 1100 % Zoom out
        campos(viewer,pos(1),pos(2), 1e5);
    elseif time == 2000 % Zoom in
        campos(viewer,pos(1),pos(2), 2.6e4);
    else % Keep previous height but follow airplane
        campos(viewer,pos(1),pos(2));
    end
end

% Snapshots
if any(time == snapTimes)
    drawnow
    img = snapshot(viewer);
    images = [ images, {img}];
end
end
```

Reference

[1] Automatic Dependent Surveillance - Broadcast (ADS-B): https://www.faa.gov/about/office_org/headquarters_offices/avs/offices/afx/afs/afs400/afs410/ads-b

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 `helperPertScenarioGlobeViewer` (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]);

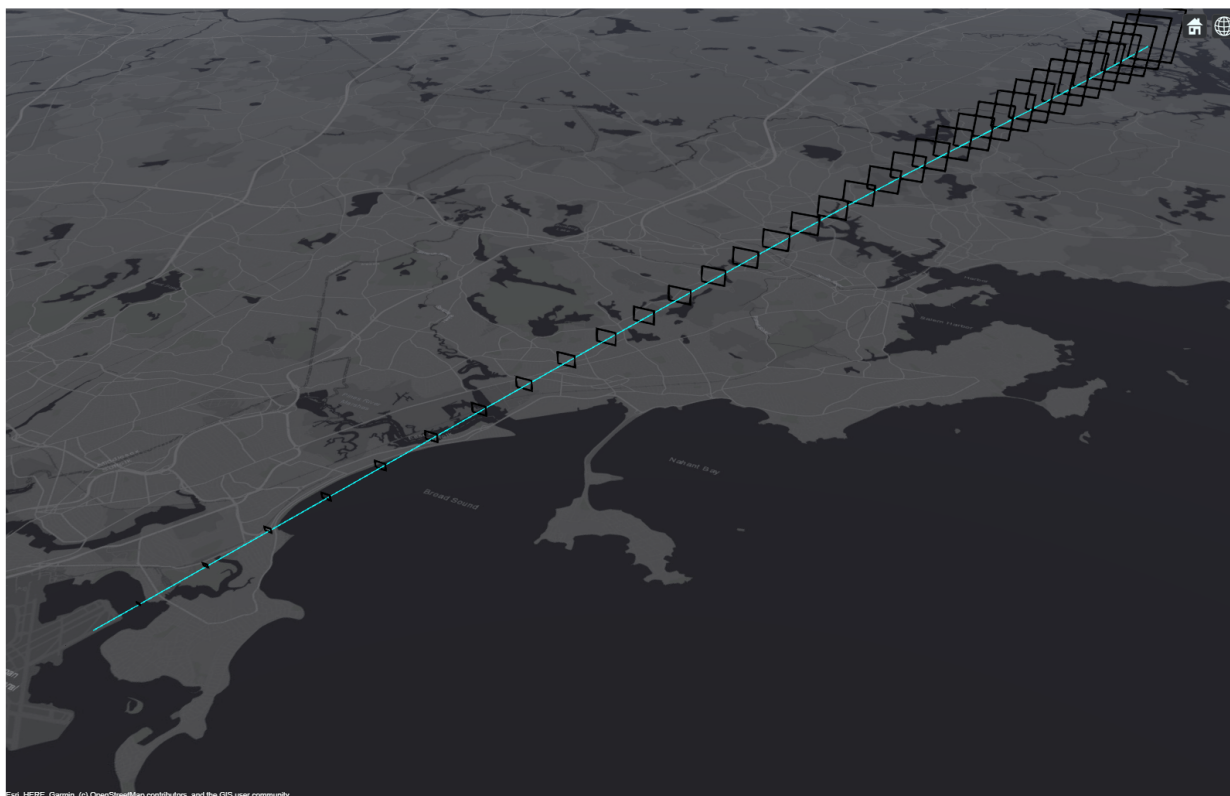
viewer = helperPertScenarioGlobeViewer;
viewer.TargetHistoryLength = 0;
viewer.TrackHistoryLength = 0;
viewer.TrackLabelScale = 0.75;
positionCamera(viewer, [42.3072 -70.8463 12455], [0 -34 335]);
plotTrajectory(viewer, baselineApproachTrajectory, 'Color', [15 255 255]/255, "Width", 1);
```

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-3 150; 0 1e-4 0])
```

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.

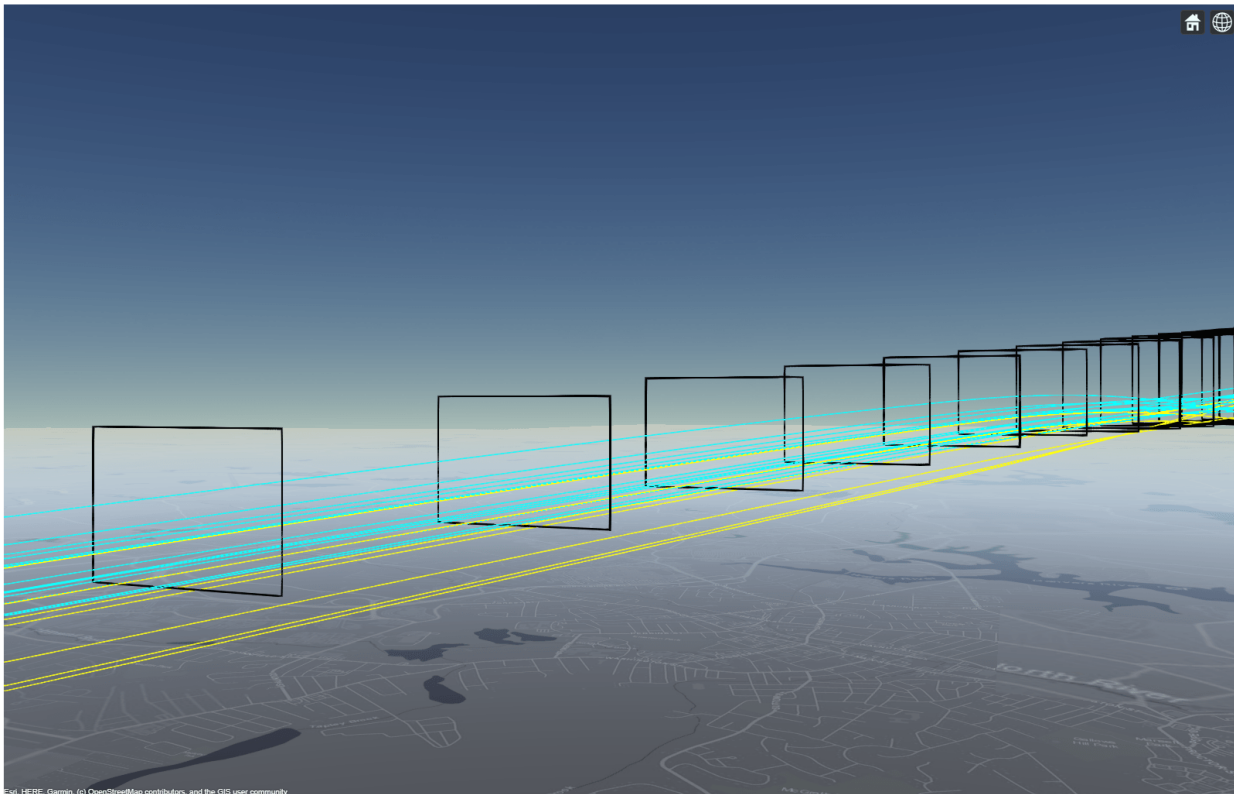
```
[truthAnomalyFlags, truthPercentAnomalousSteps] = validateTrajectory(trajectories, trajRules);
```

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, fusionRadarSensor, 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.

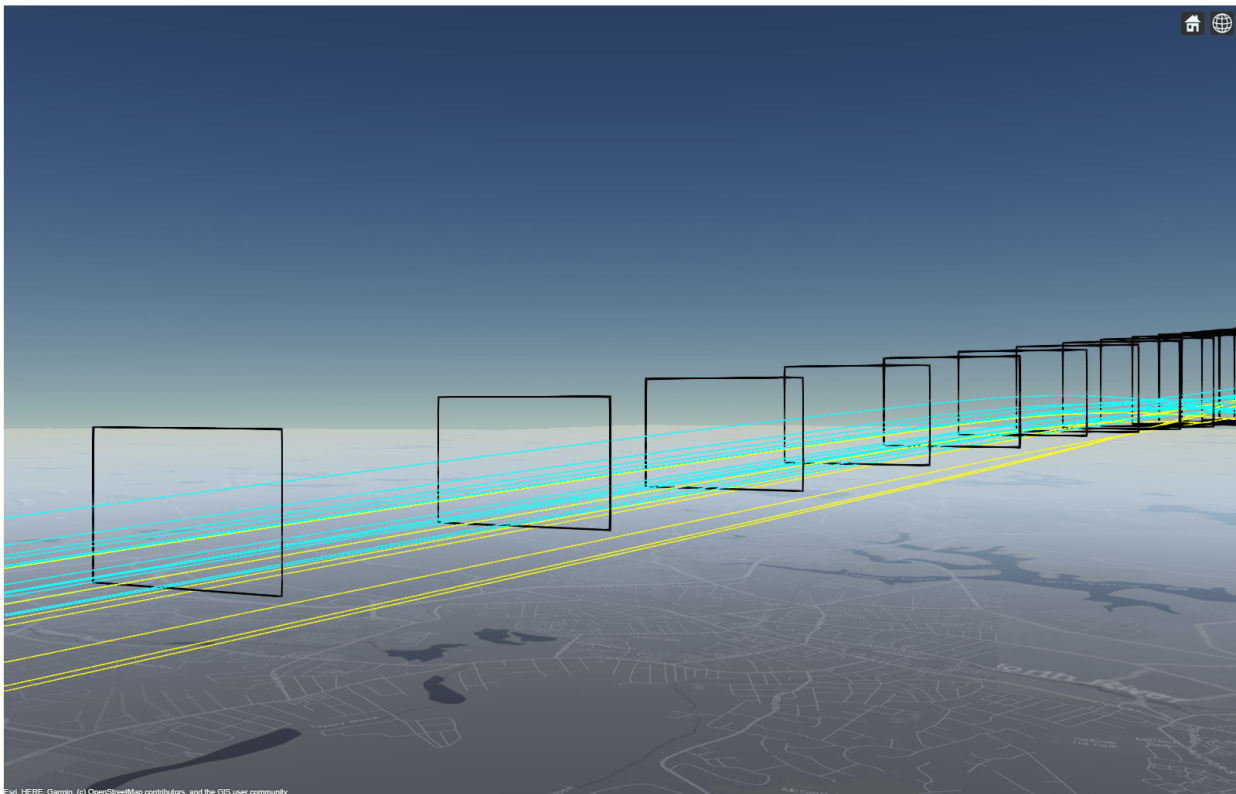
```
asdex = fusionRadarSensor(1, ...
    'ScanMode', 'No Scanning', ...
    'MountingAngles', [0 0 0], ...
    'FieldOfView', [360;20], ...
    'UpdateRate', 1, ...
    'ReferenceRange', 40000, ...
    'RangeLimits', [0 50000], ...
    'RangeResolution', 100, ...
    'HasElevation', true, ...
    'HasINS', 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)
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

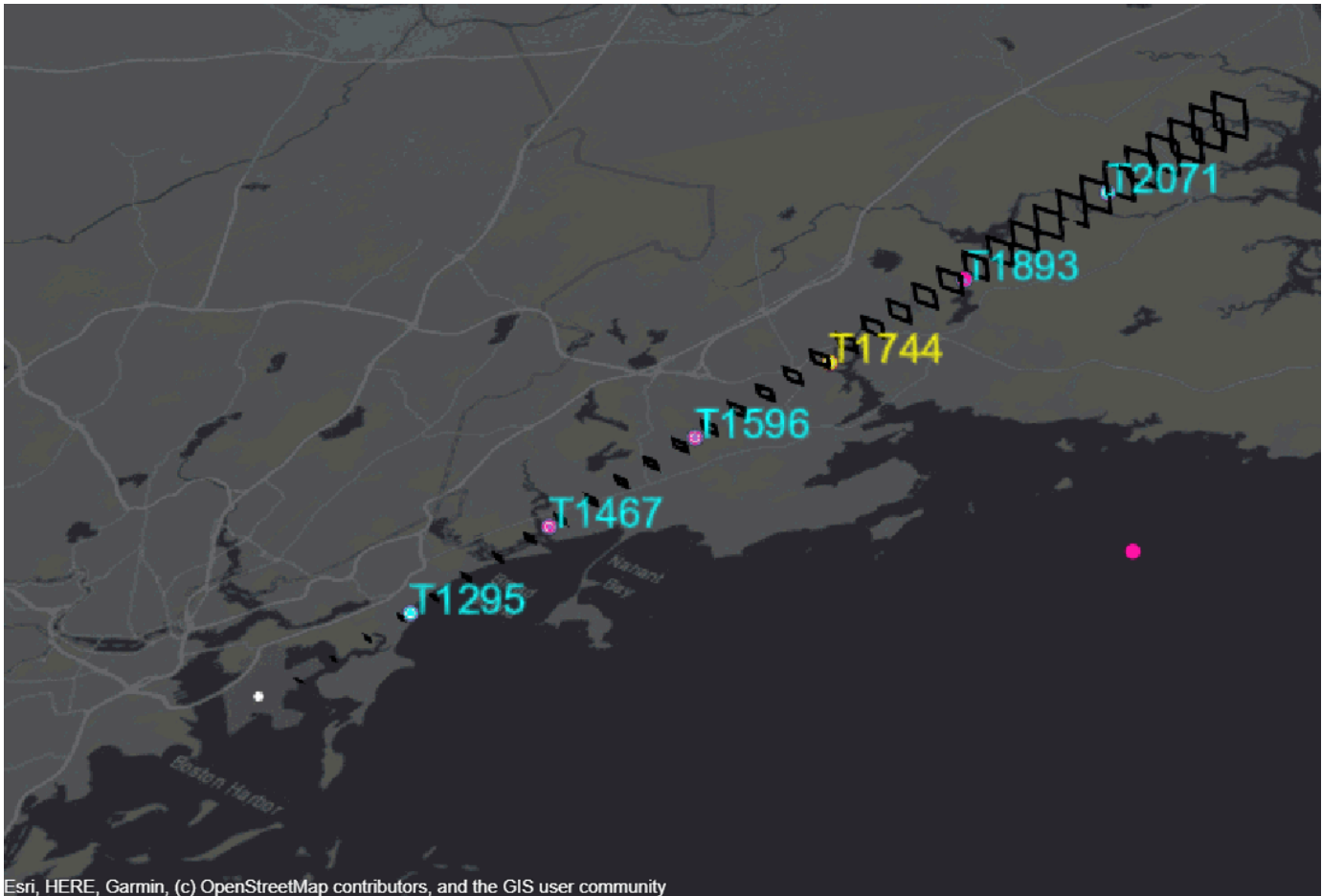
    % Get platform poses and assignment between tracks and truths.
    poses = platformPoses(scenario,"Quaternion","CoordinateSystem","Cartesian");
    tam(tracks,poses);
    [assignedTrackIDs, assignedTruthIDs] = currentAssignment(tam);

    % Validate the tracks with rules to find anomalous tracks.
    [tracks, trackAnomalyHistory] = validateTracks(tracks, trajRules, assignedTrackIDs, assignedTruthIDs);

    % Visualize
    updateDisplay(viewer,scenario.SimulationTime,[scenario.Platforms{:}],dets,[],tracks);
end

```

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 1893.



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 `fusionRadarSensor` 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)". 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, -92.73]);

% 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.
trajRules = [longitudeRule;altitudeRule1;altitudeRule2;speedRule];
end

```

validateTrajectory Validate each trajectory

```

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(i,3)));
        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

```

function [tracks, history] = validateTracks(tracks, rules, assignedTrackIDs, assignedTruthIDs)
persistent trackAnomalyHistory

```

```

if isempty(trackAnomalyHistory)
    trackAnomalyHistory = repmat(struct('TrackID', 0, 'AssignedTruthID', 0, 'NumSteps', 0, 'NumAnomalousSteps', 0), numTracks, 1);
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,pos(tr,1)));
        isSpeedValid = validate(rules(4),distance,norm(vel(tr,:)));
        isValid = isLongitudeValid && isAltitudeValid && isSpeedValid;
    else
        isValid = true;
    end

    tracks(tr).ObjectClassID = uint8(~isValid) + uint8(isValid)*6; % To get the right color for track

    % Update anomaly history
    inHistory = (tracks(tr).TrackID == trackIDs);
    if any(inHistory)
        trackAnomalyHistory(inHistory).NumSteps = trackAnomalyHistory(inHistory).NumSteps + 1;
        trackAnomalyHistory(inHistory).NumAnomalousSteps = trackAnomalyHistory(inHistory).NumAnomalousSteps + 1;
    else
        ind = find(trackIDs == 0, 1, 'first');
        trackAnomalyHistory(ind).AssignedTruthID = assignedTruthIDs(tracks(tr).TrackID == assignedTruthIDs);
        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 6-496 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',@helperpctcuboid,...
        'StateTransitionJacobianFcn',@helperpctcuboidjac,...
        'ProcessNoise',diag([1 3 1]),...
        'HasAdditiveProcessNoise',false,...
        'AssignmentThreshold',[250 inf],...
        'ConfirmationThreshold',[3 5],...
    );
end

tracks = fuser(localTracks,time);
```

```

        '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 6-496 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.

```
% 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);
```



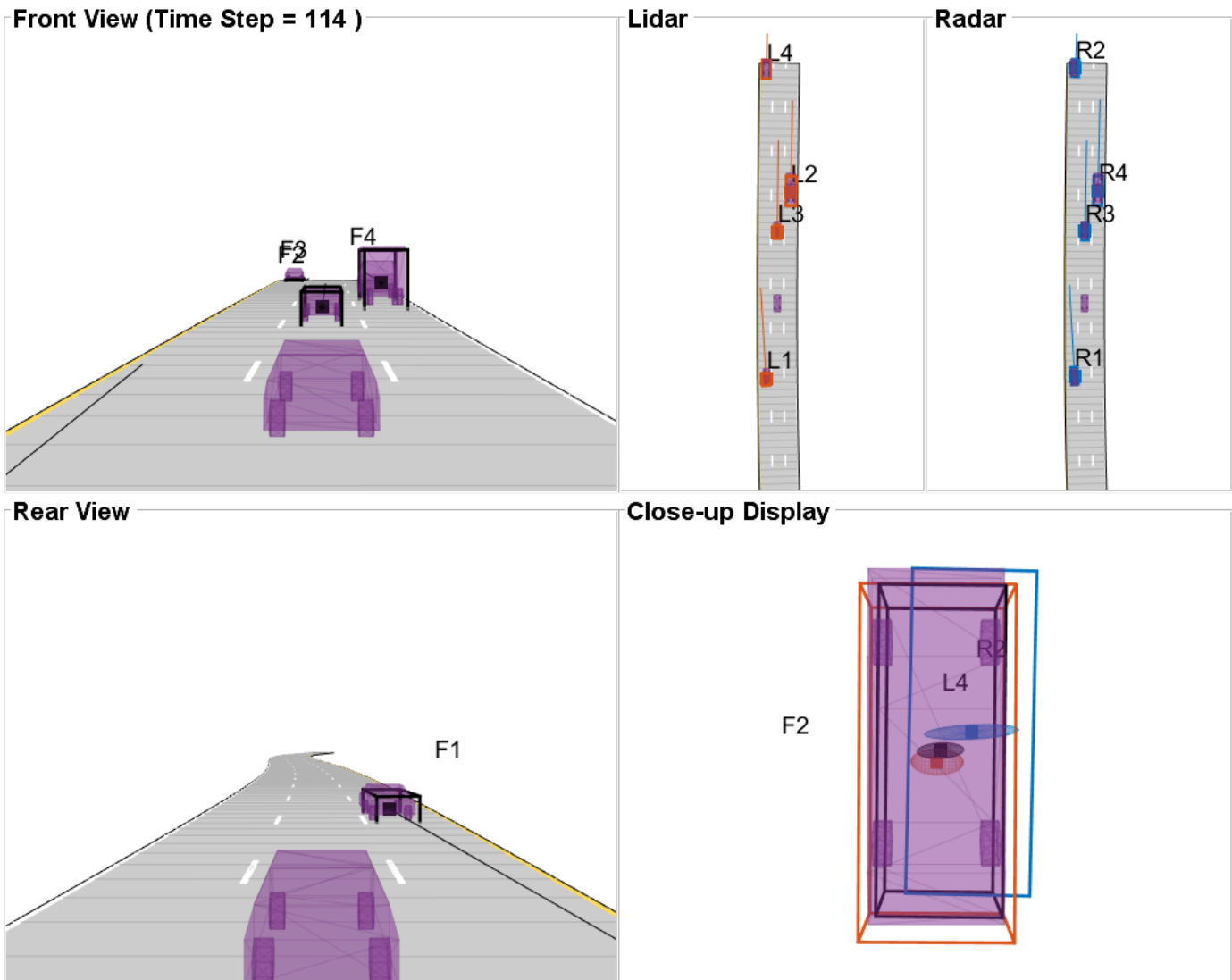
```

% Capture GOSPA and its components for all trackers
[gospa(1,idx),~,~,~,missedTargets(1,idx),falseTracks(1,idx)] = gospaRadar(radarTracks, groundTruth);
[gospa(2,idx),~,~,~,missedTargets(2,idx),falseTracks(2,idx)] = gospaLidar(lidarTracks, groundTruth);
[gospa(3,idx),~,~,~,missedTargets(3,idx),falseTracks(3,idx)] = gospaCentral(fusedTracks, groundTruth);

% 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), ...
    'ObjectClassProbabilities', double(1),...
    '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;
```

Code generation successful.

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`

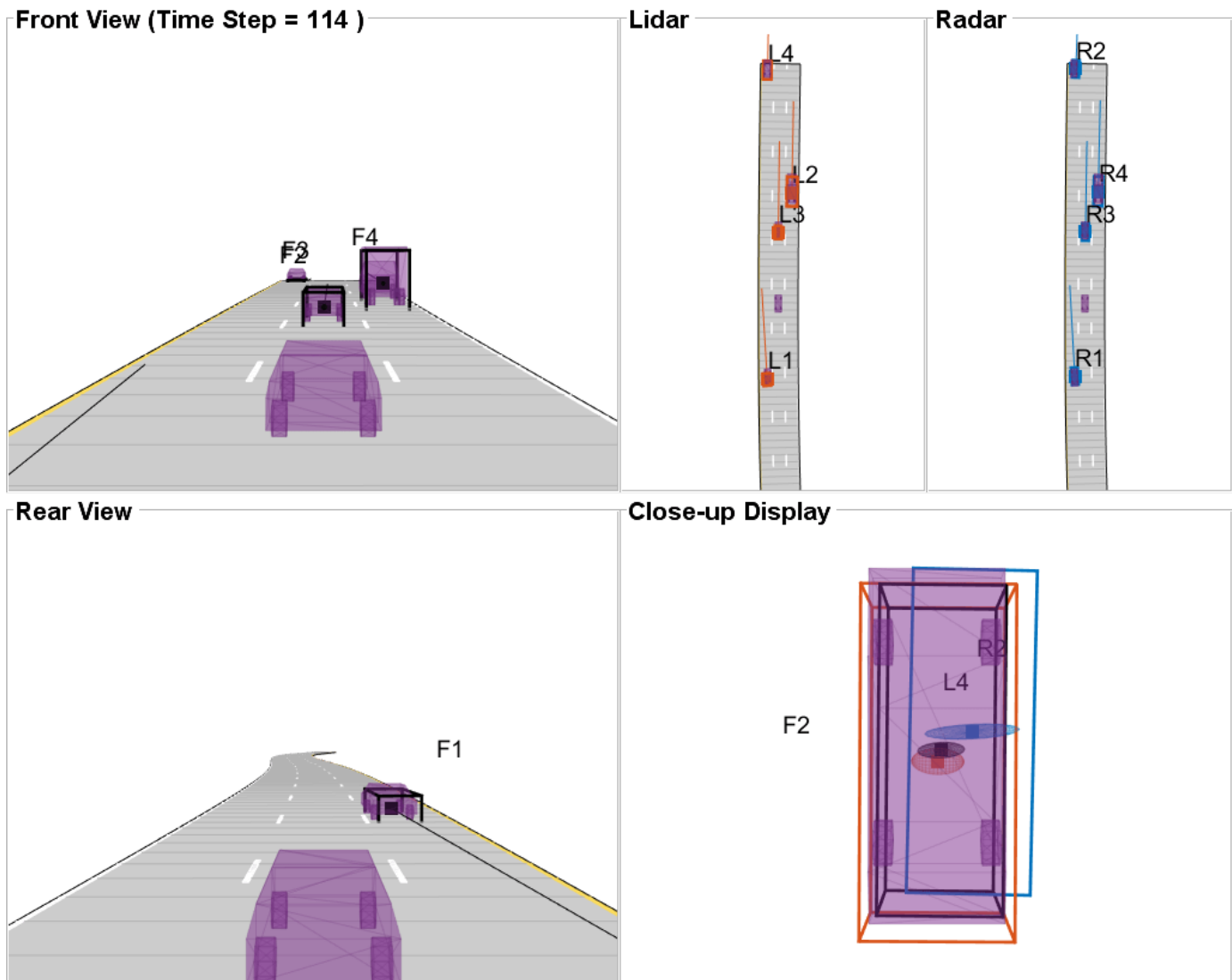
```
[gospaCG(1,idx),~,~,~,missedTargetsCG(1,idx),falseTracksCG(1,idx)] = gospaRadar(radarTracks,
[gospaCG(2,idx),~,~,~,missedTargetsCG(2,idx),falseTracksCG(2,idx)] = gospaLidar(lidarTracks,
[gospaCG(3,idx),~,~,~,missedTargetsCG(3,idx),falseTracksCG(3,idx)] = gospaCentral(fusedTracks,
```

`% 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;
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.

```

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);

% 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.

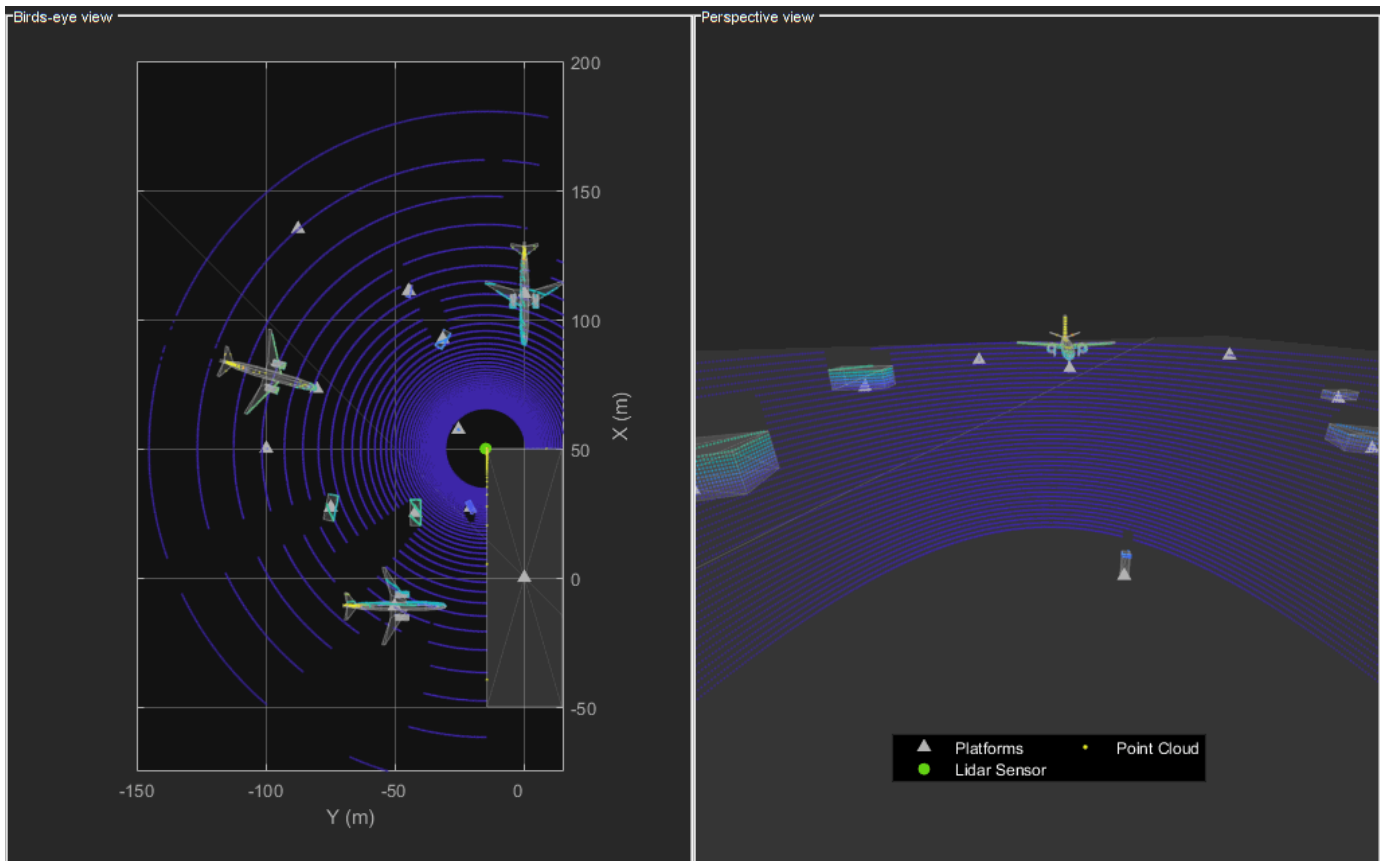
```
% Create a lidar sensor
lidar = monostaticLidarSensor( ...
    'SensorIndex',1,...
    'UpdateRate',5,...
    'HasINS',true,...
    'AzimuthLimits',[-180 180],...
    'ElevationLimits',[-10 27],...
    'AzimuthResolution',0.32,...
    'ElevationResolution',0.5781,...
    'DetectionCoordinates','scenario',...
    '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 6-352 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 6-609, 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 measurement 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 6-610, included in the Supporting Functions on page 6-605 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

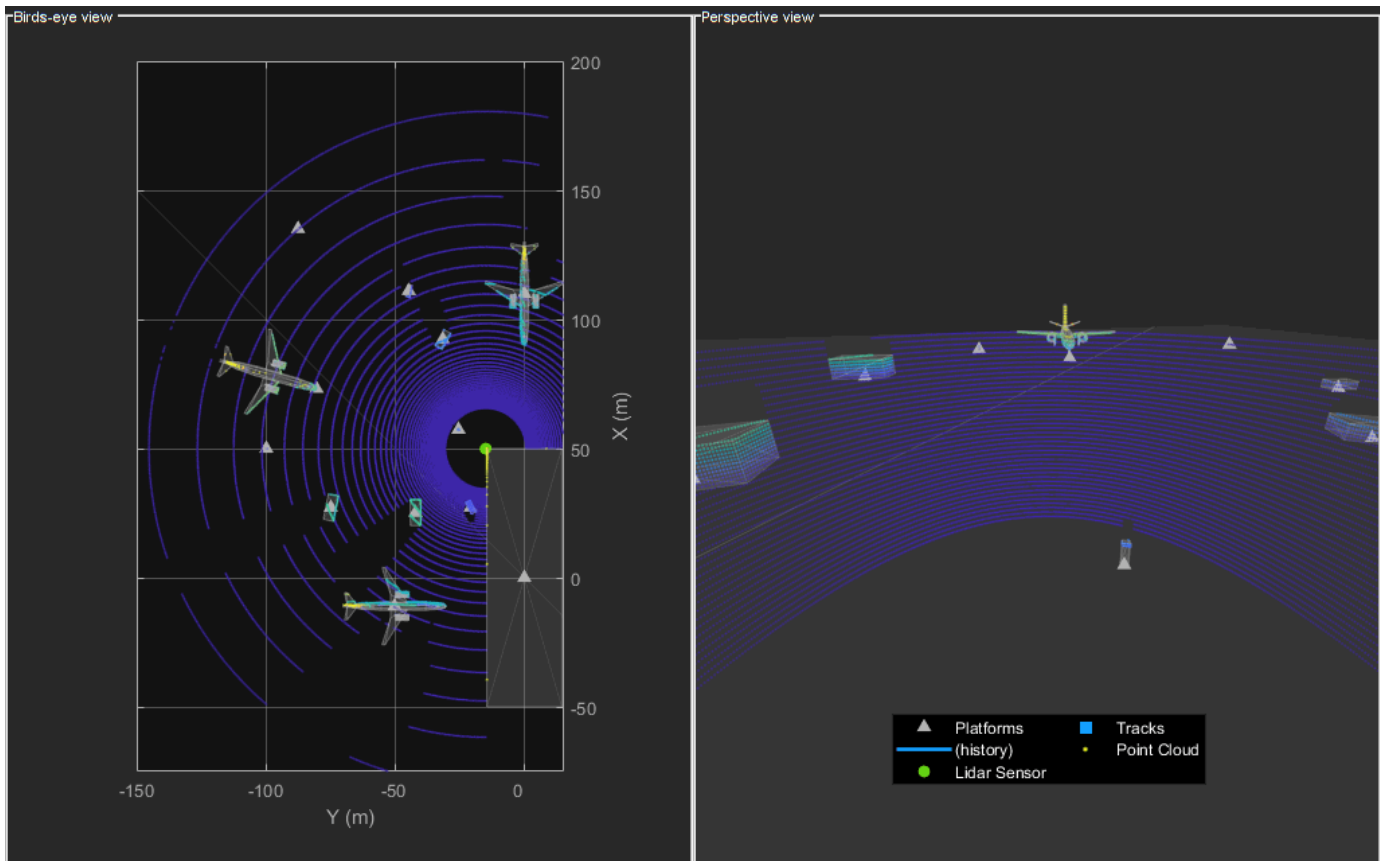
    % Call tracker step using detections and time
    confTracks = tracker(detections, time);

    % 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.

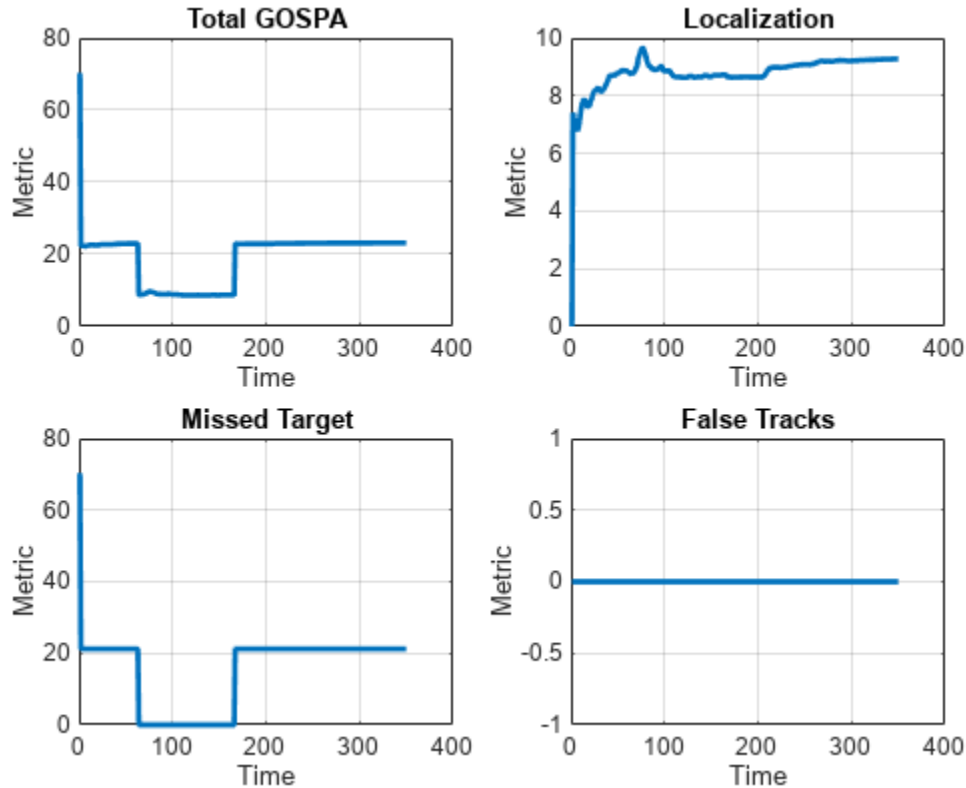
You can also quantitatively analyze this performance of the tracker using the GOSPA metric and its components. Notice that the false track component of the metric remains zero, which indicates that no false tracks were confirmed. The non-zero missed target component represents the undetected marshaller. 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 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 0.5...
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 0.5...
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]);

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]);

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([-27.95
    'GroundSpeed', [0;5.8574;0], ...
    'ClimbRate', [0;0;0]);

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]);

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([-169.2
    'GroundSpeed', [0;5;0], ...
    'ClimbRate', [0;0;0]);

delayTrajectory(Car, 20, 70);

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

```

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 adja
```

```

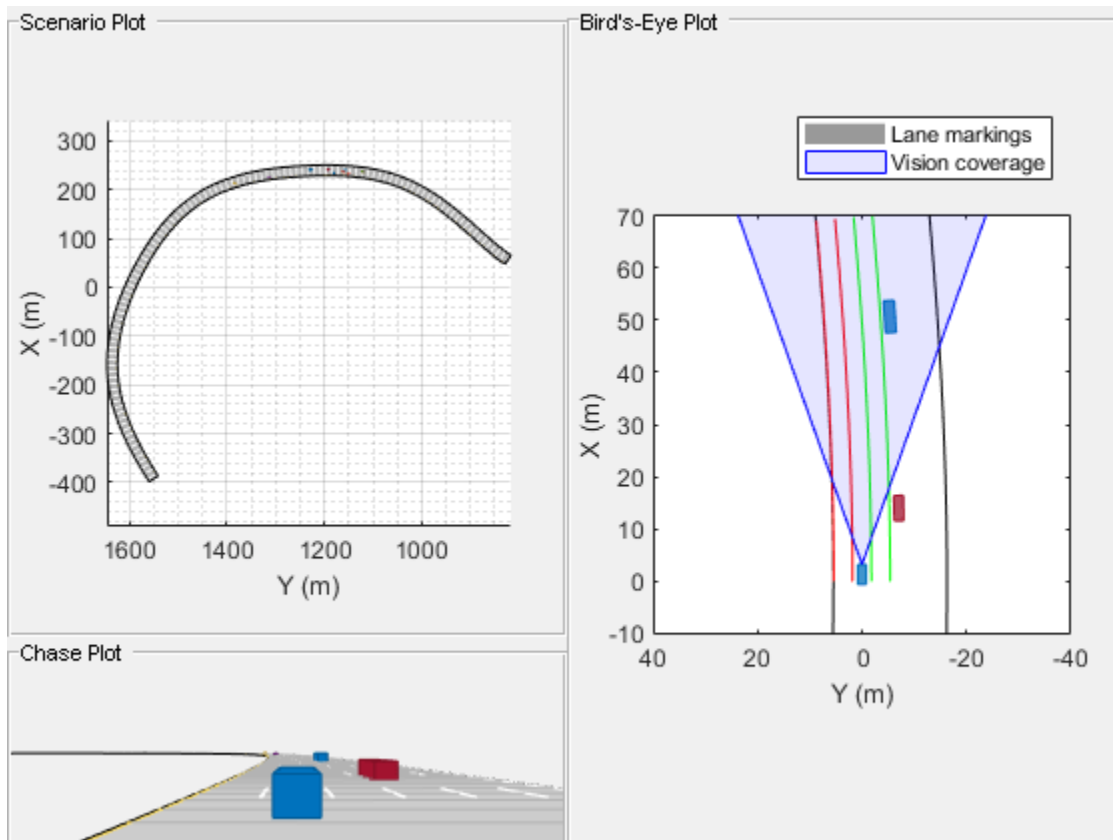
laneBoundaryDetections = laneDetector(egoBoundaries(egoBoundaryExist),simTime); %% ego lane a
laneObjectDetections = packLanesAsObjectDetections(laneBoundaryDetections); %% convert lane l

% 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, laneDetector);

% Advance to the next step
timeStep = timeStep + 1;
running = advance(scenario);
end

```

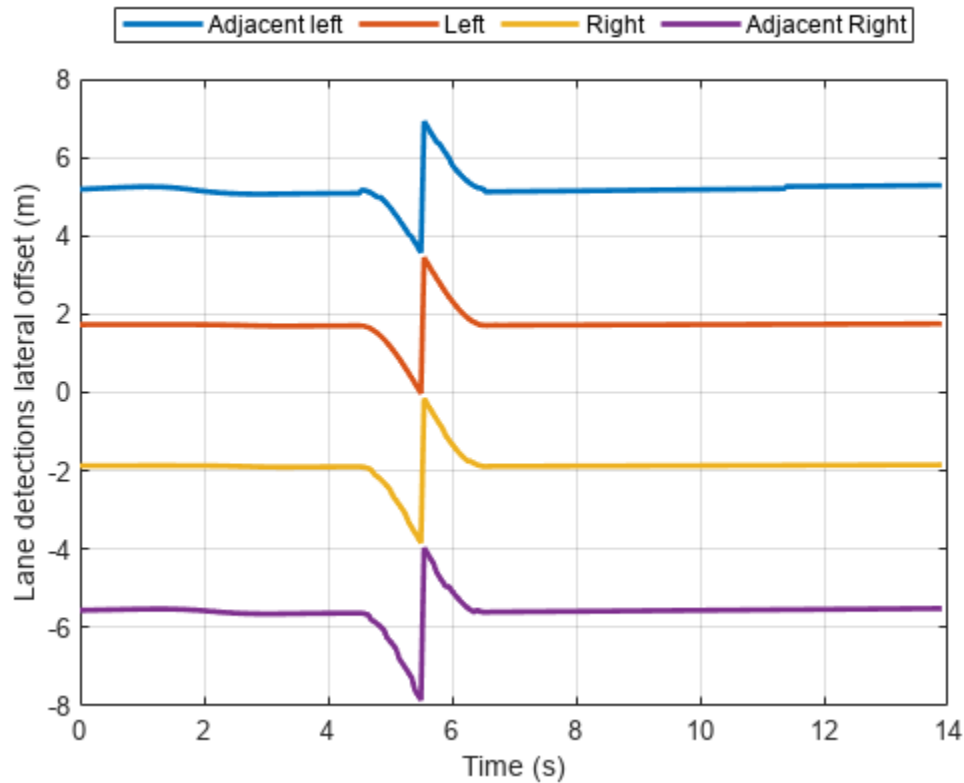


Since the lane detector always reports the two lane markings on each side of the ego vehicle, the lane change causes it to report discontinuous lane markings. You can observe it in the graph below.

```

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 6-617 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 adjacent
    laneBoundaryDetections = laneDetector(egoBoundaries(egoBoundaryExist),simTime); %% ego lane and adjacent
    laneObjectDetections = packLanesAsObjectDetections(laneBoundaryDetections); %% convert lane boundaries to objects

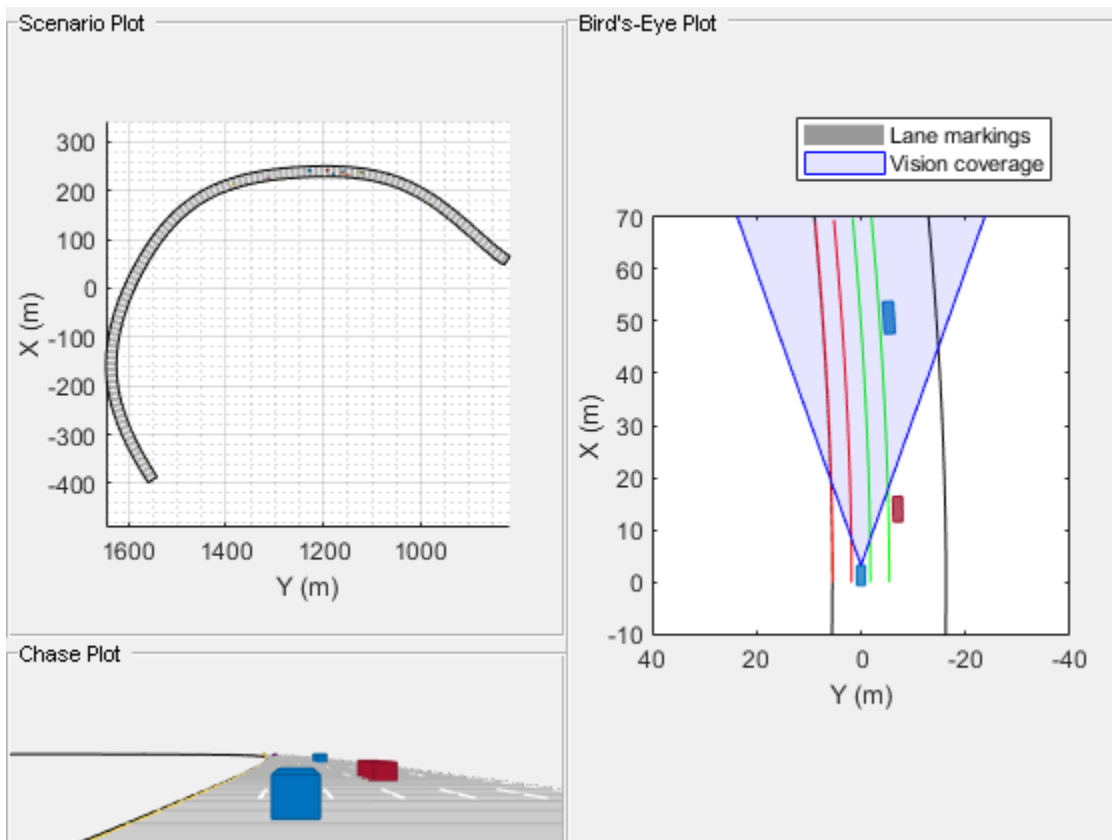
    % Track the lanes
    laneTracks = laneTracker(laneObjectDetections,simTime);

    % 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,laneDetector);

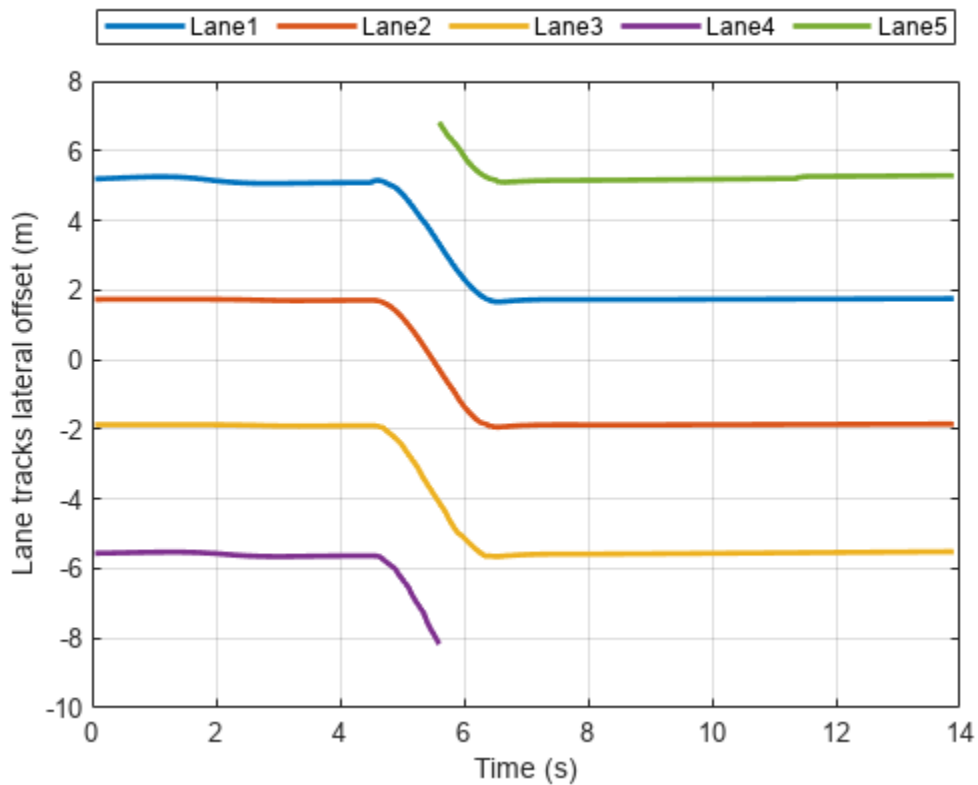
    % Advance to the next step
    timeStep = timeStep + 1;
    running = advance(scenario);
end

```



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', 'north')
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
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','E
end

```

updateDisplay Update the display for this example

```

function updateDisplay(egoCarBEP,egoVehicle,LaneBdryIn,laneDetector)
[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');
plotLaneBoundary(roadPlotter,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});
caPlotter = findPlotter(egoCarBEP,'Tag','Vision');
plotCoverageArea(caPlotter, laneDetector.SensorLocation, laneDetector.MaxRange, laneDetector.Yaw
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)));
end

```

```

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).

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 6-598 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.

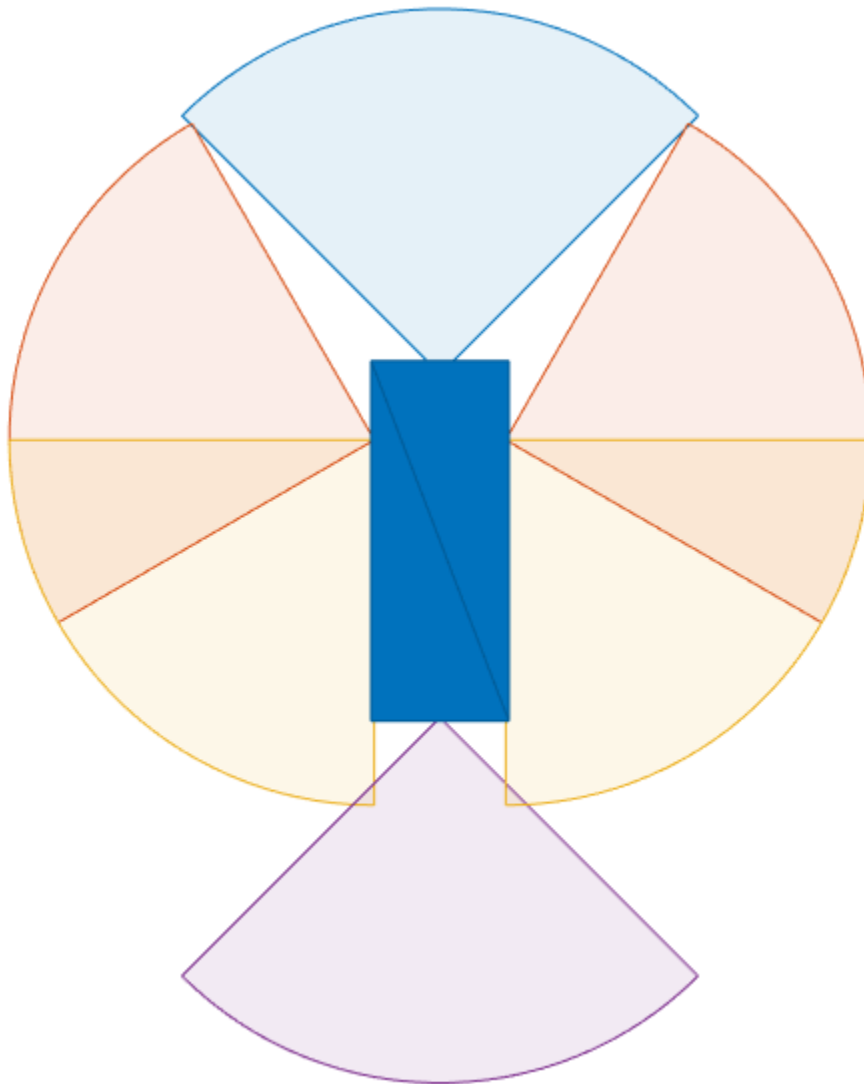
In this example, you use the `trackerGridRFS` System object™ to configure the grid-based tracker. This tracker uses the Random Finite Set (RFS) formulation with Dempster-Shafer approximation [1] to estimate the dynamic map. Further, it uses a nearest neighbor cell-to-track association [2] scheme to track 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 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 6-628 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;

```

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  $P_s = 0.9999$  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 it is 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 t
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 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 outputs 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);

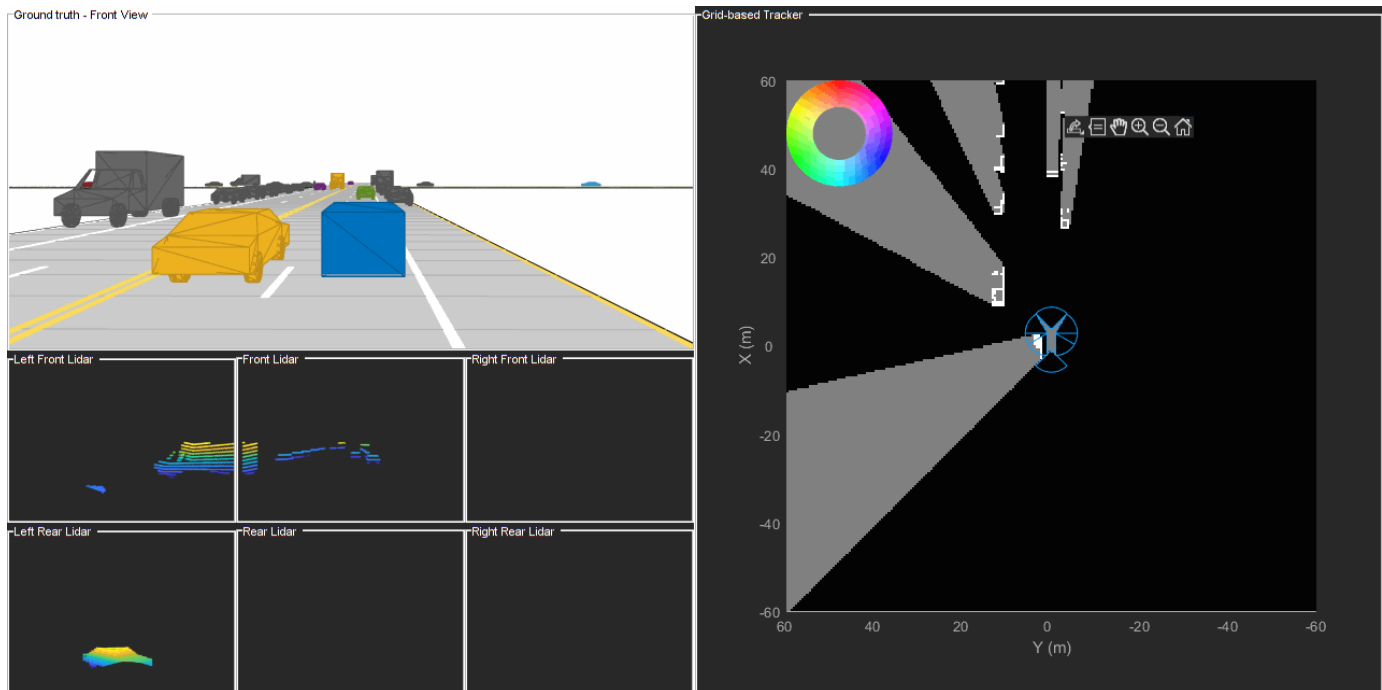
% Update the display
display(scenario, egoVehicle, lidars, ptClouds, tracker, tracks);
drawnow;
end
```

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.

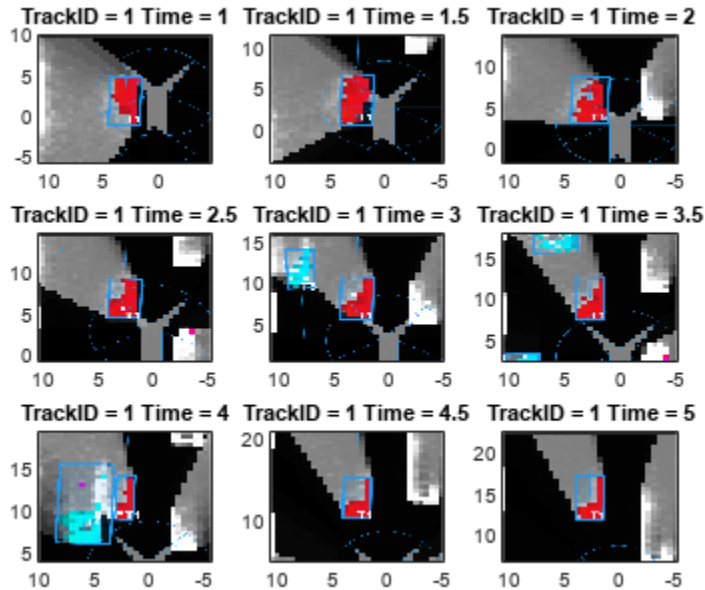


Next, you look at the history of a few tracks to understand how the state of a track gets affected by the estimation of the dynamic grid.

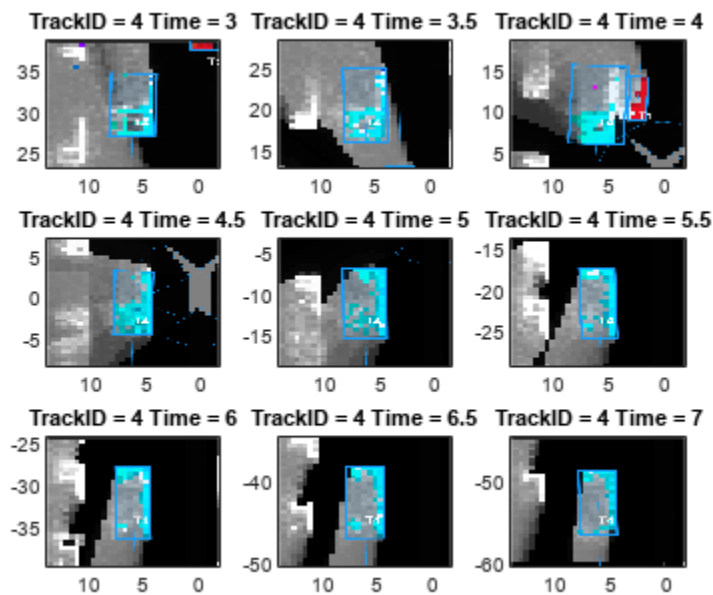
Longitudinally Moving Tracks

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);
```



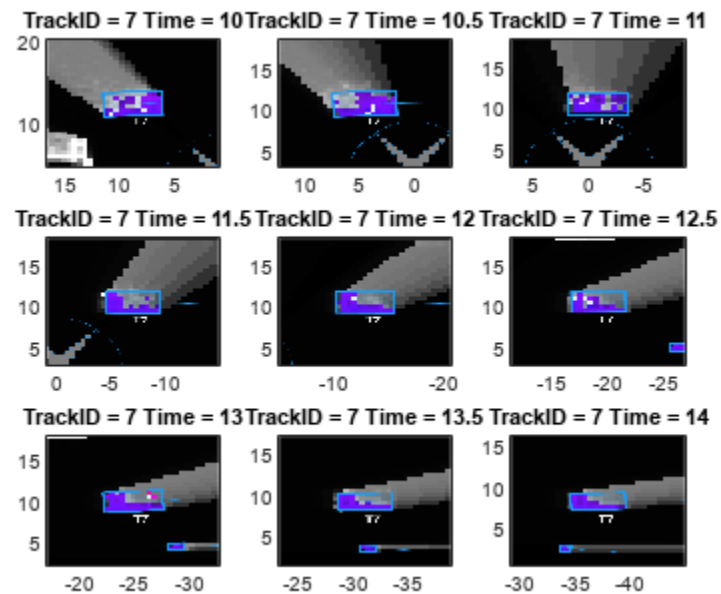
Next, take a closer look at the history of T4. The T4 track represents the truck moving in the opposite direction of the ego vehicle. Notice that the grid cells representing this track are colored in blue, representing the estimated motion direction of the grid cell. Also, notice that there are grid 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 properly.

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.

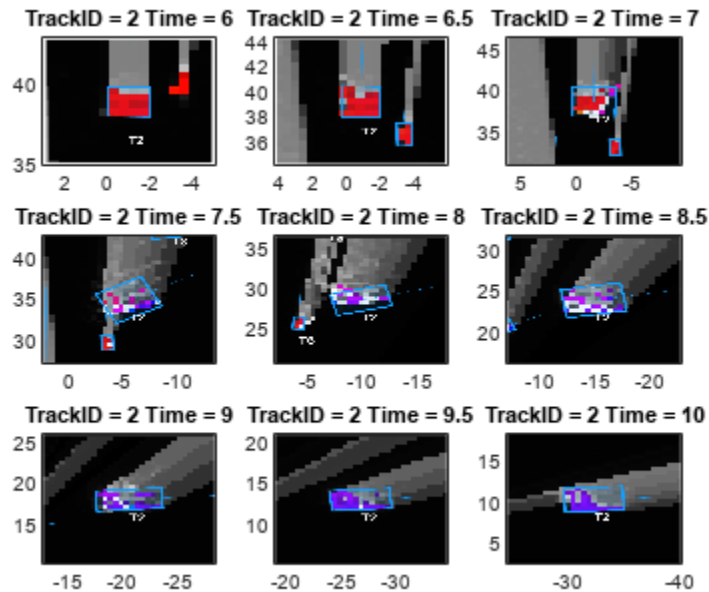
```
showSnapshots(display.GridView,7);
```



Tracks Changing Direction

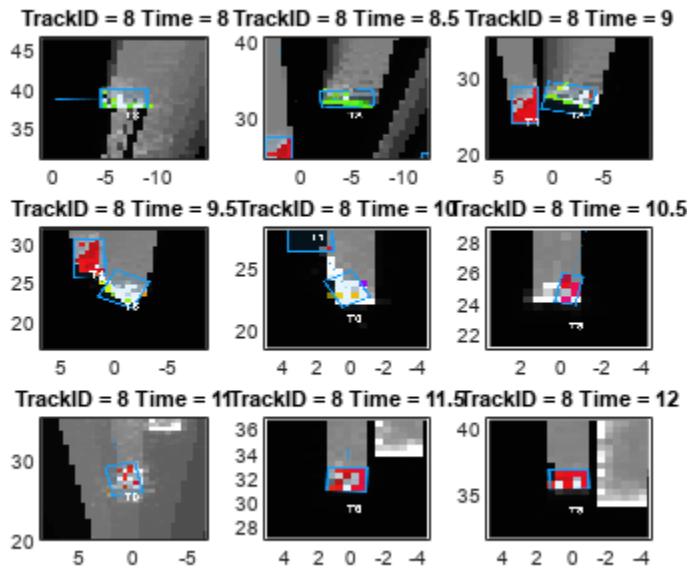
In this example, you used a "constant-velocity" model with the tracker. This motion model assumes that the targets move at a constant velocity, meaning constant speed and direction. However, in urban scenes, this assumption is usually not accurate. To compensate for the unknown acceleration of the objects, 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 a 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 an 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 defining 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;
sensorData(i).MeasurementParameters(1).Frame = fusionCoordinateFrameType(1);
end

end

function config = helperGetLidarConfig(lidar, egoVehicle)
platPose = struct(Position = egoVehicle.Position(:), Yaw = egoVehicle.Yaw, Pitch = ...
    egoVehicle.Pitch, Roll = egoVehicle.Roll, Velocity = egoVehicle.Velocity);
config = trackingSensorConfiguration(lidar,platPose, IsValidTime = true);
config.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.

```
% 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


```
filtUntuned = insfilterAsync;
```

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	4.5898
1	GyroscopeNoise	4.5694
1	MagnetometerNoise	4.5481
1	GPSPositionNoise	4.4737
1	GPSVelocityNoise	4.2984
1	QuaternionNoise	4.2984
1	AngularVelocityNoise	3.9668
1	PositionNoise	3.9668
1	VelocityNoise	3.9668
1	AccelerationNoise	3.9556
1	GyroscopeBiasNoise	3.9556
1	AccelerometerBiasNoise	3.9511
1	GeomagneticVectorNoise	3.9511
1	MagnetometerBiasNoise	3.9360
2	AccelerometerNoise	3.9360
2	GyroscopeNoise	3.9360

2	MagnetometerNoise	3.9064
2	GPSPositionNoise	3.9064
2	GPSVelocityNoise	3.7129
2	QuaternionNoise	3.7122
2	AngularVelocityNoise	3.0116
2	PositionNoise	3.0116
2	VelocityNoise	3.0116
2	AccelerationNoise	2.9850
2	GyroscopeBiasNoise	2.9850
2	AccelerometerBiasNoise	2.9824
2	GeomagneticVectorNoise	2.9824
2	MagnetometerBiasNoise	2.9821
3	AccelerometerNoise	2.9821
3	GyroscopeNoise	2.9613
3	MagnetometerNoise	2.9432
3	GPSPositionNoise	2.9432
3	GPSVelocityNoise	2.8373
3	QuaternionNoise	2.8369
3	AngularVelocityNoise	2.6993
3	PositionNoise	2.6993
3	VelocityNoise	2.6993
3	AccelerationNoise	2.6993
3	GyroscopeBiasNoise	2.6993
3	AccelerometerBiasNoise	2.6971
3	GeomagneticVectorNoise	2.6971
3	MagnetometerBiasNoise	2.6955
4	AccelerometerNoise	2.6941
4	GyroscopeNoise	2.6797
4	MagnetometerNoise	2.6676
4	GPSPositionNoise	2.6626
4	GPSVelocityNoise	2.5530
4	QuaternionNoise	2.5530
4	AngularVelocityNoise	2.4285
4	PositionNoise	2.4285
4	VelocityNoise	2.4285
4	AccelerationNoise	2.4232
4	GyroscopeBiasNoise	2.4232
4	AccelerometerBiasNoise	2.4217
4	GeomagneticVectorNoise	2.4217
4	MagnetometerBiasNoise	2.4023
5	AccelerometerNoise	2.4019
5	GyroscopeNoise	2.3900
5	MagnetometerNoise	2.3900
5	GPSPositionNoise	2.3887
5	GPSVelocityNoise	2.2986
5	QuaternionNoise	2.2986
5	AngularVelocityNoise	2.1945
5	PositionNoise	2.1945
5	VelocityNoise	2.1945
5	AccelerationNoise	2.1863
5	GyroscopeBiasNoise	2.1863
5	AccelerometerBiasNoise	2.1856
5	GeomagneticVectorNoise	2.1856
5	MagnetometerBiasNoise	2.1615
6	AccelerometerNoise	2.1613
6	GyroscopeNoise	2.1393
6	MagnetometerNoise	2.1199
6	GPSPositionNoise	2.1112

6	GPSVelocityNoise	2.0140
6	QuaternionNoise	2.0140
6	AngularVelocityNoise	1.9288
6	PositionNoise	1.9288
6	VelocityNoise	1.9288
6	AccelerationNoise	1.9242
6	GyroscopeBiasNoise	1.9242
6	AccelerometerBiasNoise	1.9216
6	GeomagneticVectorNoise	1.9216
6	MagnetometerBiasNoise	1.9054
7	AccelerometerNoise	1.9047
7	GyroscopeNoise	1.8887
7	MagnetometerNoise	1.8820
7	GPSPositionNoise	1.8803
7	GPSVelocityNoise	1.7713
7	QuaternionNoise	1.7712
7	AngularVelocityNoise	1.7145
7	PositionNoise	1.7145
7	VelocityNoise	1.7145
7	AccelerationNoise	1.7122
7	GyroscopeBiasNoise	1.7122
7	AccelerometerBiasNoise	1.7112
7	GeomagneticVectorNoise	1.7112
7	MagnetometerBiasNoise	1.6873
8	AccelerometerNoise	1.6853
8	GyroscopeNoise	1.6790
8	MagnetometerNoise	1.6694
8	GPSPositionNoise	1.6565
8	GPSVelocityNoise	1.5700
8	QuaternionNoise	1.5676
8	AngularVelocityNoise	1.5366
8	PositionNoise	1.5366
8	VelocityNoise	1.5366
8	AccelerationNoise	1.5366
8	GyroscopeBiasNoise	1.5366
8	AccelerometerBiasNoise	1.5353
8	GeomagneticVectorNoise	1.5337
8	MagnetometerBiasNoise	1.5166
9	AccelerometerNoise	1.5129
9	GyroscopeNoise	1.5117
9	MagnetometerNoise	1.5114
9	GPSPositionNoise	1.4953
9	GPSVelocityNoise	1.4106
9	QuaternionNoise	1.4106
9	AngularVelocityNoise	1.3742
9	PositionNoise	1.3742
9	VelocityNoise	1.3742
9	AccelerationNoise	1.3731
9	GyroscopeBiasNoise	1.3731
9	AccelerometerBiasNoise	1.3715
9	GeomagneticVectorNoise	1.3715
9	MagnetometerBiasNoise	1.3576
10	AccelerometerNoise	1.3528
10	GyroscopeNoise	1.3528
10	MagnetometerNoise	1.3510
10	GPSPositionNoise	1.3322
10	GPSVelocityNoise	1.2512
10	QuaternionNoise	1.2512

10	AngularVelocityNoise	1.2507
10	PositionNoise	1.2507
10	VelocityNoise	1.2507
10	AccelerationNoise	1.2497
10	GyroscopeBiasNoise	1.2497
10	AccelerometerBiasNoise	1.2480
10	GeomagneticVectorNoise	1.2480
10	MagnetometerBiasNoise	1.2301
11	AccelerometerNoise	1.2233
11	GyroscopeNoise	1.2222
11	MagnetometerNoise	1.2220
11	GPSPositionNoise	1.2008
11	GPSVelocityNoise	1.1043
11	QuaternionNoise	1.1043
11	AngularVelocityNoise	1.1038
11	PositionNoise	1.1038
11	VelocityNoise	1.1038
11	AccelerationNoise	1.1028
11	GyroscopeBiasNoise	1.1028
11	AccelerometerBiasNoise	1.1013
11	GeomagneticVectorNoise	1.1013
11	MagnetometerBiasNoise	1.0867
12	AccelerometerNoise	1.0782
12	GyroscopeNoise	1.0767
12	MagnetometerNoise	1.0733
12	GPSPositionNoise	1.0505
12	GPSVelocityNoise	0.9564
12	QuaternionNoise	0.9563
12	AngularVelocityNoise	0.9563
12	PositionNoise	0.9563
12	VelocityNoise	0.9563
12	AccelerationNoise	0.9550
12	GyroscopeBiasNoise	0.9550
12	AccelerometerBiasNoise	0.9534
12	GeomagneticVectorNoise	0.9534
12	MagnetometerBiasNoise	0.9402
13	AccelerometerNoise	0.9303
13	GyroscopeNoise	0.9291
13	MagnetometerNoise	0.9269
13	GPSPositionNoise	0.9036
13	GPSVelocityNoise	0.8072
13	QuaternionNoise	0.8071
13	AngularVelocityNoise	0.8071
13	PositionNoise	0.8071
13	VelocityNoise	0.8071
13	AccelerationNoise	0.8065
13	GyroscopeBiasNoise	0.8065
13	AccelerometerBiasNoise	0.8052
13	GeomagneticVectorNoise	0.8052
13	MagnetometerBiasNoise	0.7901
14	AccelerometerNoise	0.7806
14	GyroscopeNoise	0.7806
14	MagnetometerNoise	0.7768
14	GPSPositionNoise	0.7547
14	GPSVelocityNoise	0.6932
14	QuaternionNoise	0.6930
14	AngularVelocityNoise	0.6923
14	PositionNoise	0.6923

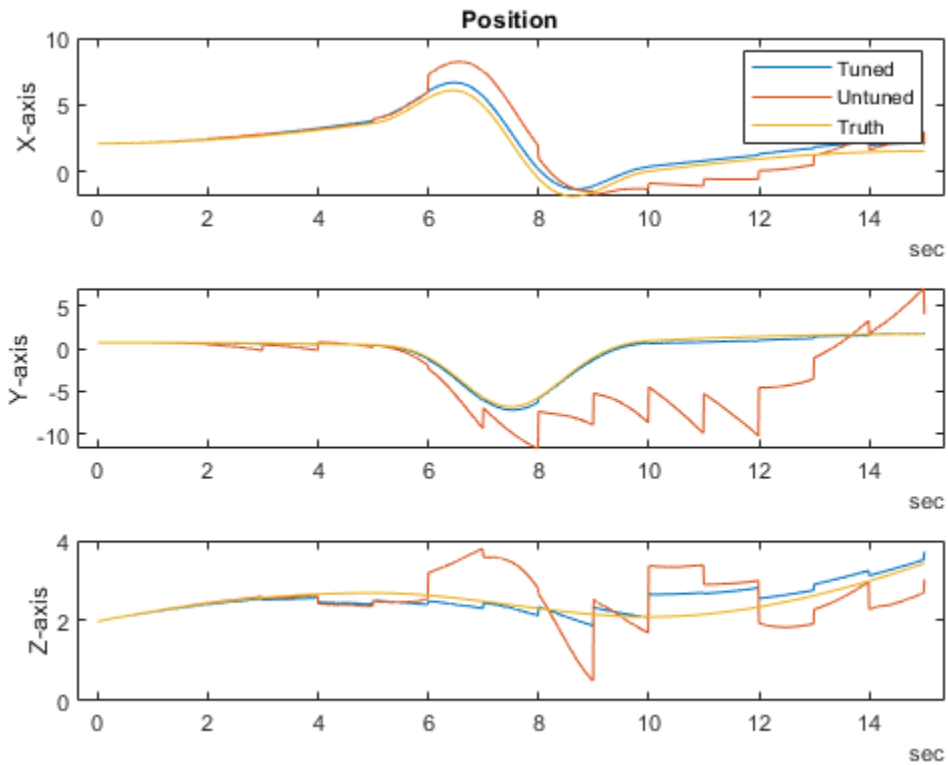
14	VelocityNoise	0.6923
14	AccelerationNoise	0.6906
14	GyroscopeBiasNoise	0.6906
14	AccelerometerBiasNoise	0.6896
14	GeomagneticVectorNoise	0.6896
14	MagnetometerBiasNoise	0.6801
15	AccelerometerNoise	0.6704
15	GyroscopeNoise	0.6676
15	MagnetometerNoise	0.6653
15	GPSPositionNoise	0.6469
15	GPSVelocityNoise	0.6047
15	QuaternionNoise	0.6047
15	AngularVelocityNoise	0.6037
15	PositionNoise	0.6037
15	VelocityNoise	0.6037
15	AccelerationNoise	0.6019
15	GyroscopeBiasNoise	0.6019
15	AccelerometerBiasNoise	0.6015
15	GeomagneticVectorNoise	0.6015
15	MagnetometerBiasNoise	0.5913

```
[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.

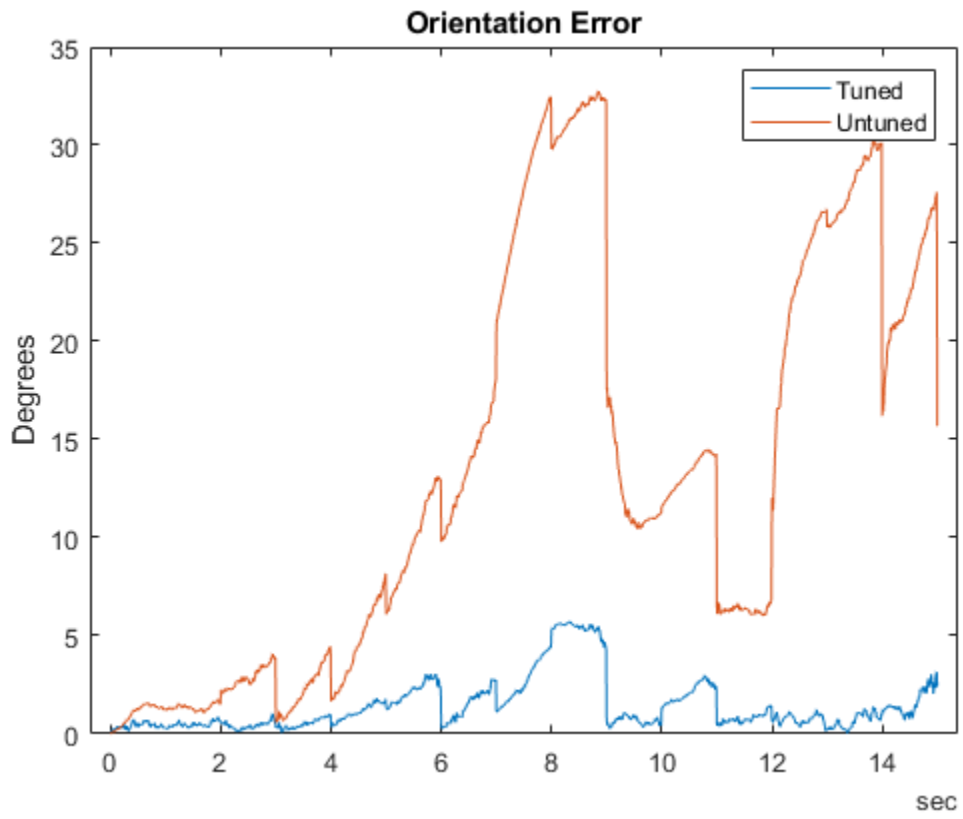
```
% Position error
figure;
t = sensorData.Time;
subplot(3,1,1);
plot(t, [posTunedEst(:,1) posUntunedEst(:,1) Position(:,1)]);
title('Position');
ylabel('X-axis');
legend('Tuned', 'Untuned', 'Truth');
subplot(3,1,2);
plot(t, [posTunedEst(:,2) posUntunedEst(:,2) Position(:,2)]);
ylabel('Y-axis');
subplot(3,1,3);
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.

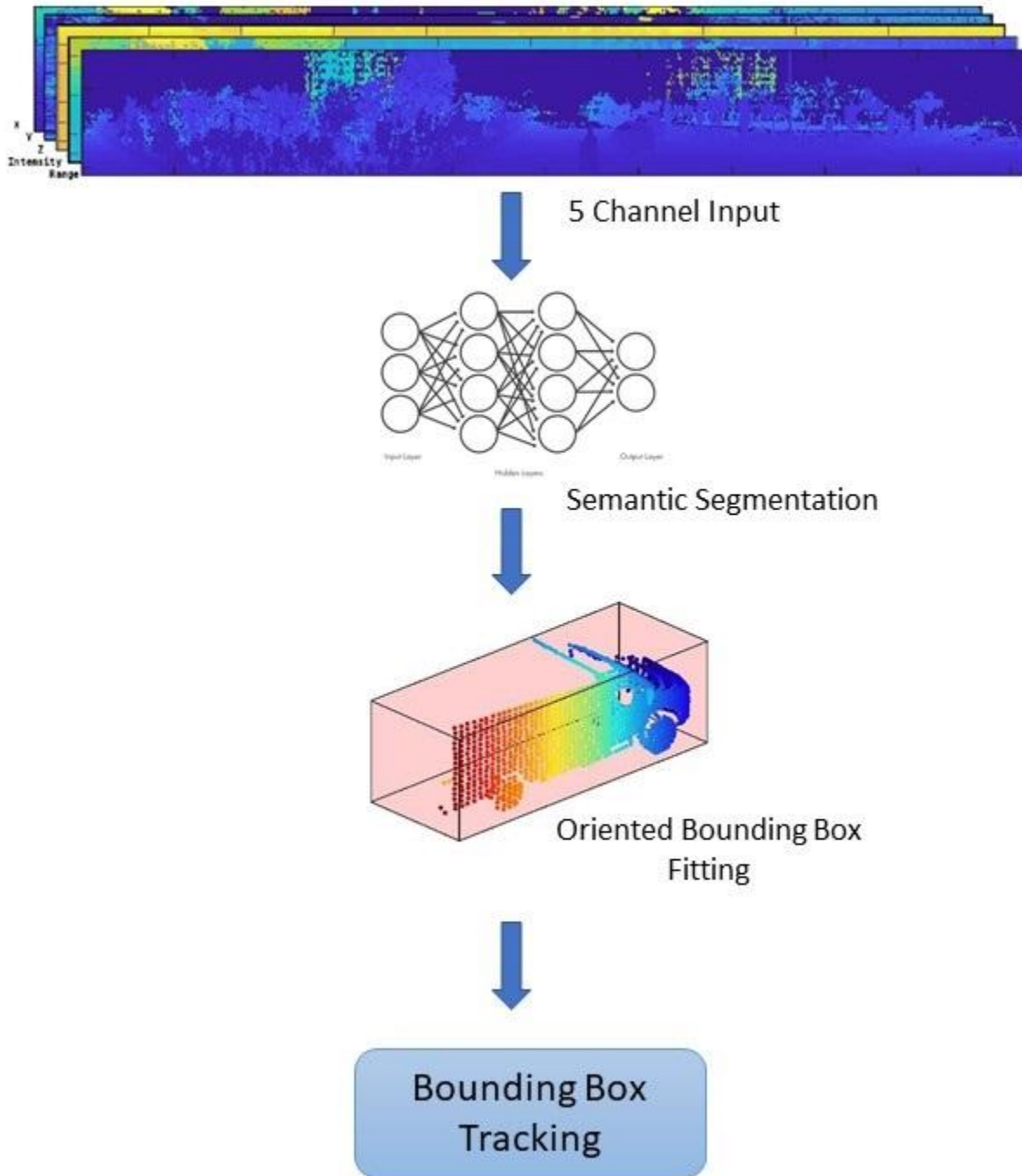
Overview

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.



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. Use the helper function `helperDownloadData` to download the data and load them into the MATLAB® workspace.

Note: This download can take a few minutes.

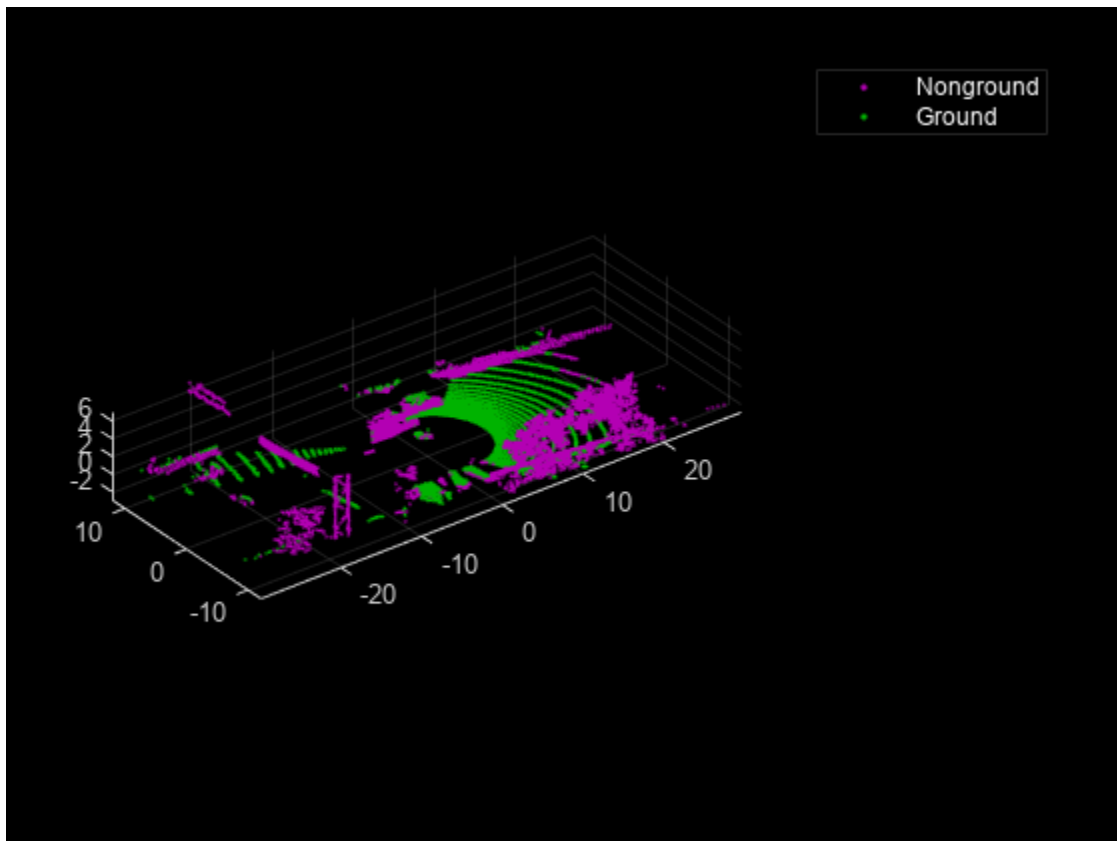
```
[ptClouds,pretrainedModel] = helperDownloadData;
```

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
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);
axis on;
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” (Lidar Toolbox) example.

Prepare Input Data

The `helperPrepareData` function generates five-channel data from the loaded point cloud data.

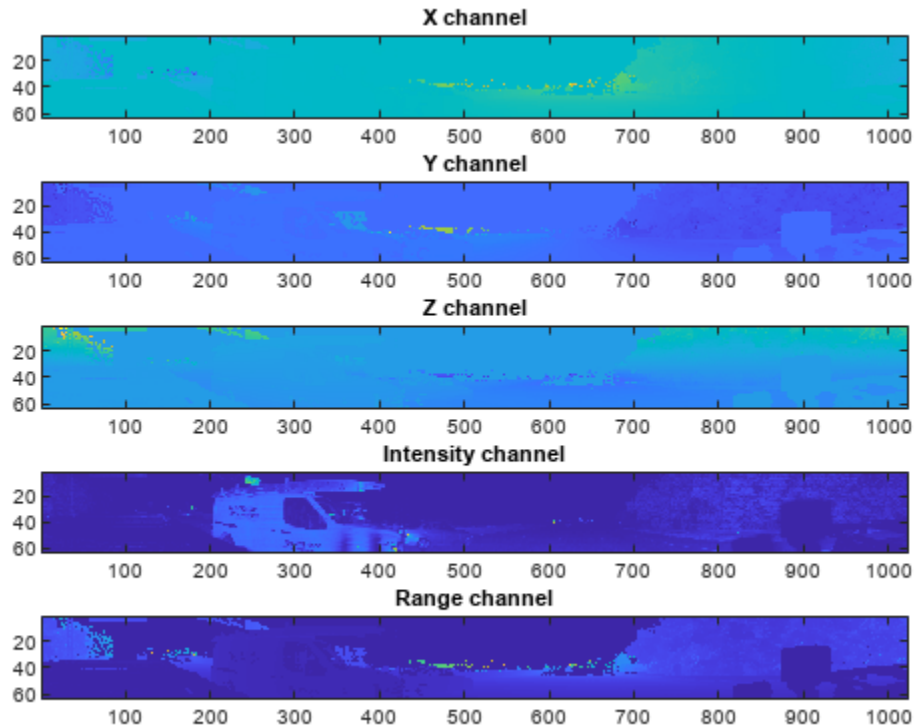
```
% Load and visualize a sample frame
frame = helperPrepareData(ptCloud);
figure;
subplot(5,1,1);
imagesc(frame(:,:,1));
title('X channel');

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');
```



Run forward inference on one frame from the loaded pre-trained network.

```

if ~exist('net','var')
    net = pretrainedModel.net;
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,'Transparent');

% Display results
figure;
imshow(segmentedImage);
helperPixelLabelColorbar(lidarColorMap,classes);

```

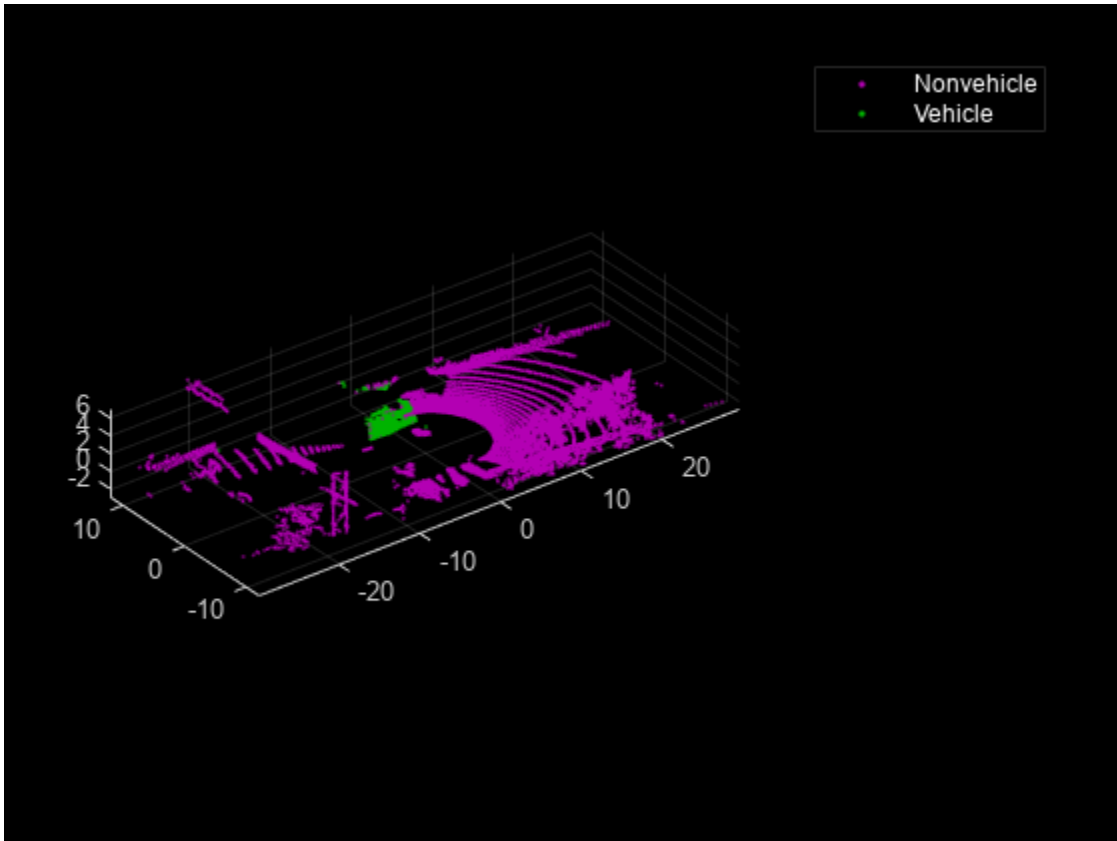


Use the generated semantic mask to filter point clouds containing trucks. Similarly, filter point clouds for other classes.

```
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);
axis on;
legend({'\color{white} Nonvehicle', '\color{white} Vehicle'}, 'Location', 'northeastoutside');
```

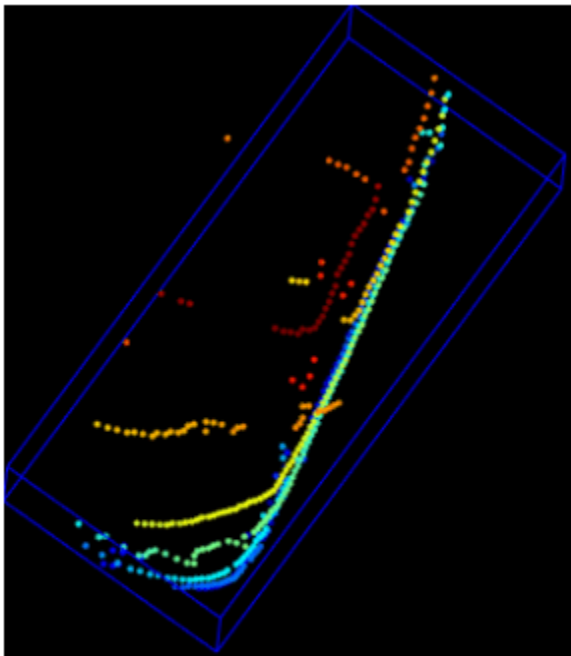


Clustering and Bounding Box Fitting

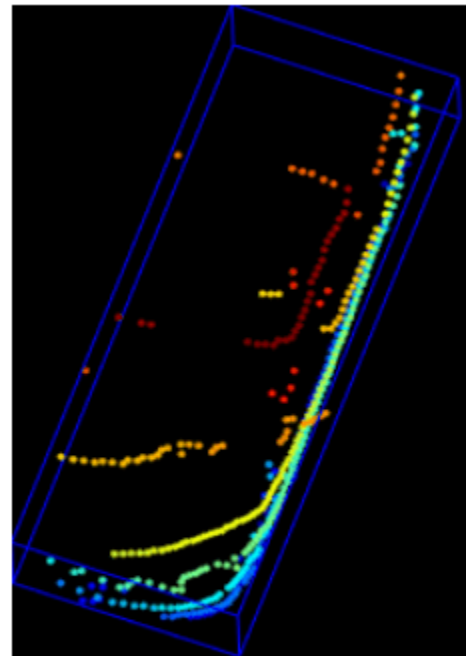
After extracting point clouds of different object classes, the objects are clustered by applying Euclidean clustering using the `pcsegdist` (Computer Vision Toolbox) function. To group all the

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.

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
```

```

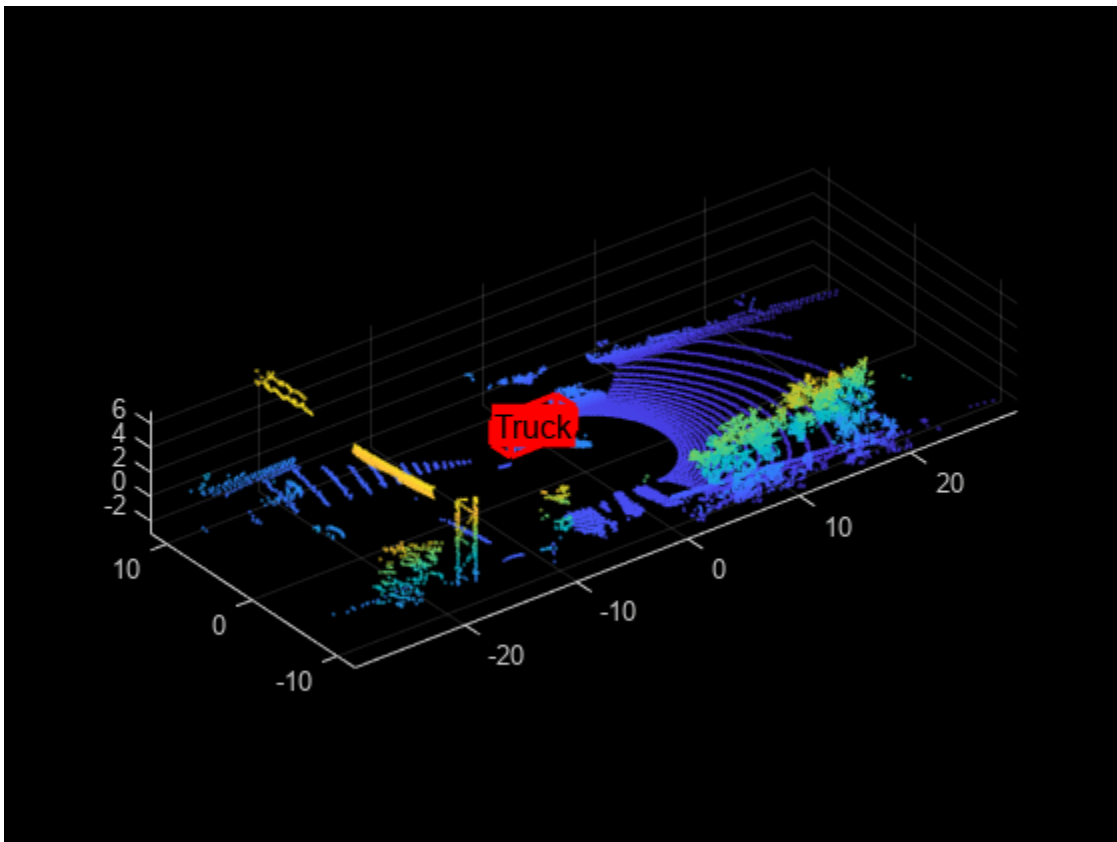
indices = zeros(0,1);
objectPtCloud = pointCloud(location);
for i = 1:size(location,1)
    seedPoint = location(i,:);
    indices(end+1) = findNearestNeighbors(nonGround,seedPoint,1);
end

% 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");
axis on;

```



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('Model',net,'XLimits',xLimit,...
    'YLimit',yLimit,'ZLimit',zLimit);

% Prepare 5-D lidar data
inputData = helperPrepareData(ptClouds);

% 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);

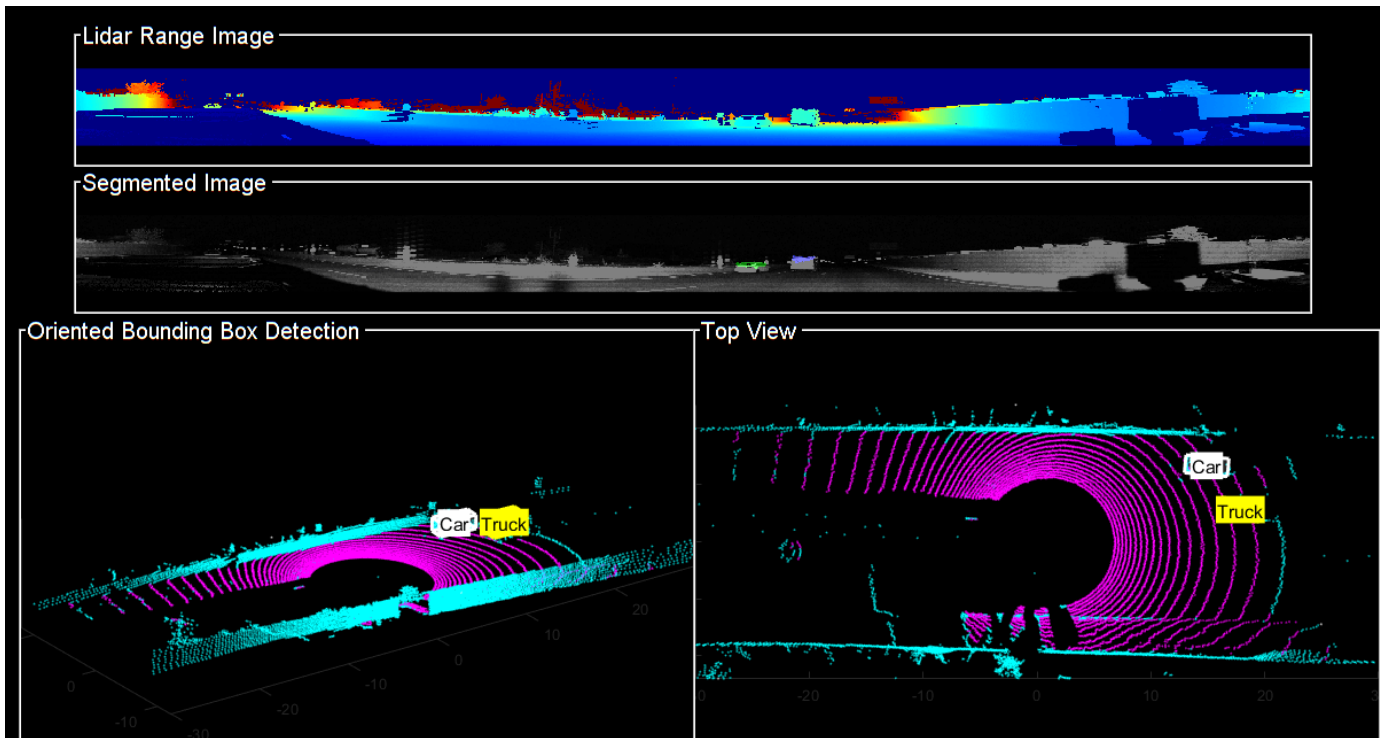
    % Extact bounding boxes from lidar data
    [boundingBox,coloredPtCloud,pointLabels] = detectBbox(lidarDetector,input);

    % 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 the “Track Vehicles Using Lidar: From Point Cloud to Track List” on page 6-352 example. 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 logic
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);

    % Extract bounding boxes from lidar data
    [boundingBox,coloredPtCloud,pointLabels] = detectBbox(lidarDetector,input);

    % Assemble bounding boxes into objectDetections
    detections = helperAssembleDetections(boundingBox,measNoise,time);

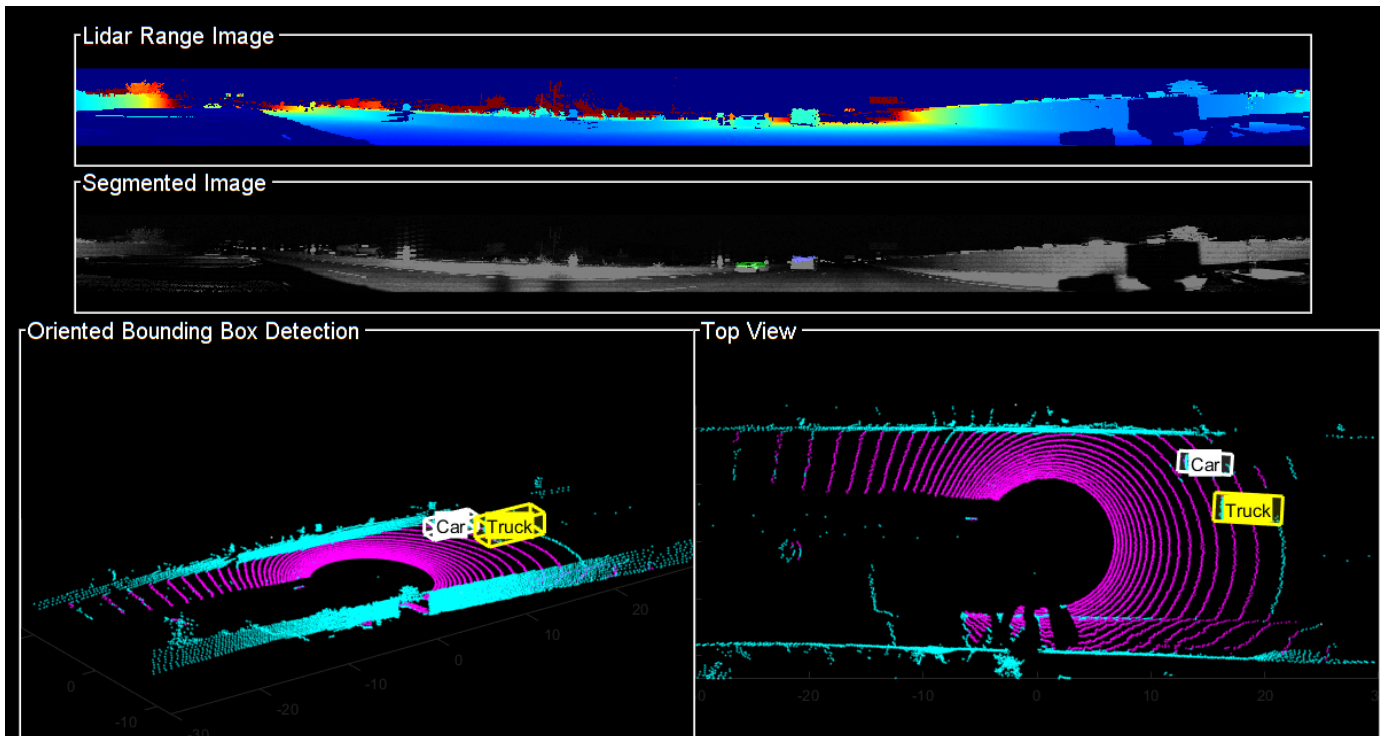
    % Pass detections to tracker
    if ~isempty(detections)
        % Update the tracker
        [confirmedTracks,tentativeTracks,allTracks,info] = tracker(detections,time);
        numTracks(count,1) = numel(confirmedTracks);
    end

    % 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

    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.

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

```
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

```
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
[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

```

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

```

helperDownloadData

```

function [lidarData, pretrainedModel] = helperDownloadData
outputFolder = fullfile(tempdir,'WPI');
url = 'https://ssd.mathworks.com/supportfiles/lidar/data/lidarSegmentationAndTrackingData.tar.gz';
lidarDataTarFile = fullfile(outputFolder,'lidarSegmentationAndTrackingData.tar.gz');
if ~exist(lidarDataTarFile,'file')
    mkdir(outputFolder);
    websave(lidarDataTarFile,url);
    untar(lidarDataTarFile,outputFolder);
end
% Check if tar.gz file is downloaded, but not uncompressed
if ~exist(fullfile(outputFolder,'highwayData.mat'),'file')

```

```
        untar(lidarDataTarFile,outputFolder);
end
% Load lidar data
data = load(fullfile(outputFolder,'highwayData.mat'));
lidarData = data.ptCloudData;

% Download pretrained model
url = 'https://ssd.mathworks.com/supportfiles/lidar/data/pretrainedPointSegModel.mat';
modelFile = fullfile(outputFolder,'pretrainedPointSegModel.mat');
if ~exist(modelFile,'file')
    websave(modelFile,url);
end
pretrainedModel = load(fullfile(outputFolder,'pretrainedPointSegModel.mat'));
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.

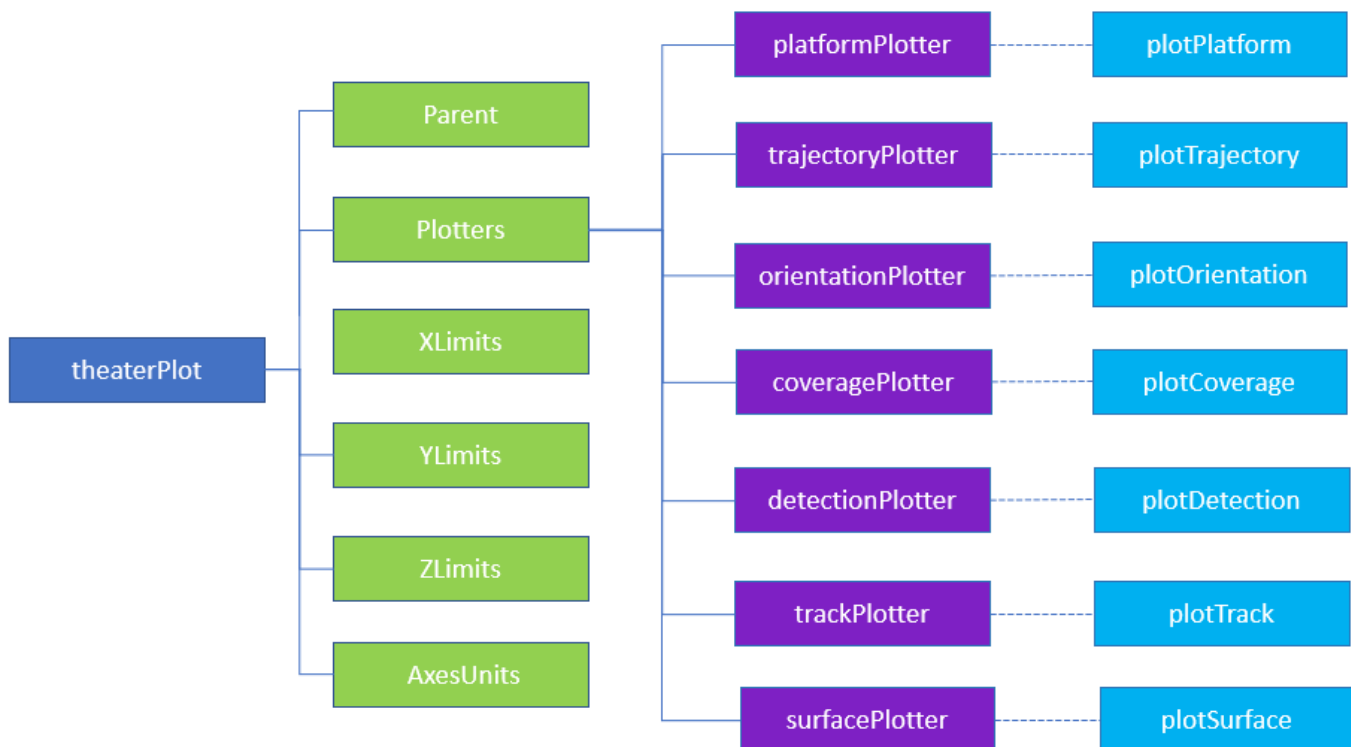
Use theaterPlot to Visualize Tracking Scenario

This example shows how to use the `theaterPlot` object to visualize various aspects of a tracking scenario.

Introduction

`theaterPlot` is an efficient tool for visualizing various aspects of a tracking scenario. It is composed of a primary object, which hosts the plotting environment based on a parent axes, and plotters to plot the desired aspects of features from the tracking scenario.

This figure shows a structural representation of a `theaterPlot` object.



The `Parent` property specifies the axes on which the theater plot is enabled. You can specify the parent axes of a theater plot during object creation. If you do not specify a parent axes, `theaterPlot` creates a new figure and uses the current axes of the created figure as its `Parent` property. You can also set the axes limits of the parent axes using the `XLimits`, `YLimits`, and `ZLimits` properties by using name-value pair arguments during object creation. Set the units of measurement for each axes using the `AxesUnits` property.

The `Plotters` property holds plotters that you added to the `theaterPlot` object.

- `platformPlotter` — Plot platforms in a tracking scenario
- `trajectoryPlotter` — Plot trajectories in a tracking scenario
- `orientationPlotter` — Plot orientation of platforms in a tracking scenario

- `coveragePlotter` — Plot sensor coverage and sensor beams in a tracking scenario
- `detectionPlotter` — Plot sensor detections in a tracking scenario
- `trackPlotter` — Plot tracks in a tracking scenario
- `surfacePlotter` — Plot surfaces in a tracking scenario

You can specify visual elements and effects for each plotter during the creation of the plotter. Each plotter is also paired with a `theaterPlot` object function, which you need to call to plot the results. For example, a `coveragePlotter` is paired with a `plotCoverage` object function that shows the sensor coverage.

This example showcases a few plotters for visualizing a tracking scenario. `theaterPlot` can work efficiently with a `trackingScenario` object even though you do not necessarily need a `trackingScenario` object to use the `theaterPlot` object.

Create theaterPlot and trackingScenario Objects

Create a `trackingScenario` object and a `theaterPlot` object.

```
simulationDuration = 100;
scene = trackingScenario('StopTime',simulationDuration);
tp = theaterPlot('XLimits',[-250 250],'YLimits',[-250 250],'ZLimits',[0 120]);
view(3);grid on;
```

Create Trajectory Plotter and Platform Plotter for Target

Create a waypoint trajectory for a target platform.

```
timeOfArrival = [0 simulationDuration];
waypoints = [100 -100 10; 100 100 80];
trajectory = waypointTrajectory(waypoints,timeOfArrival);
```

Add a cuboid target platform that follows the specified trajectory. First add a target platform to the tracking scenario.

```
target = platform(scene,'Trajectory',trajectory,'Dimensions', ...
    struct('Length',35,'Width',15,'Height',5.5,'OriginOffset',[0 0 0]));
```

Then add a `trajectoryPlotter` object to the `theaterPlot` object, and use the `plotTrajectory` function to plot the waypoint trajectory.

```
trajPlotter = trajectoryPlotter(tp,'DisplayName','Trajectory','Color','k','LineWidth',1.2);
plotTrajectory(trajPlotter,{trajectory.Waypoints})
```

Tip You can plot multiple same-type features (platforms, trajectories, orientations, coverages, detections, or tracks) together using one plotter. For example, you can plot multiple trajectories together by specifying a cell array of waypoints as the second argument of the `plotTrajectory` function. See the syntax description of `plotTrajectory` for more details.

Define a plotter for the target platform.

```
targetPlotter = platformPlotter(tp,'DisplayName','Target', ...
    'Marker','s','MarkerEdgeColor','g','MarkerSize',2);
plotPlatform(targetPlotter,target.Position, ...
    target.Dimensions,quaternion(target.Orientation,'rotvecd'))
```


You can add graphical objects other than the plotter objects on the `theaterPlot` by directly plotting on the parent axes of the `theaterPlot` object. Put a circle marker at the origin.

```
hold on
plot3(tp.Parent,0,0,0,'Color','k','Marker','o','MarkerSize',4)
```

Create Platform with Mounted Radar Sensor

Add a tower platform to the scenario.

```
tower = platform(scene,'Position',[-100,0,0],'Dimensions', ...
    struct('Length',5,'Width',5,'Height',30,'OriginOffset',[0 0 -15]));
```

Display the tower using a platform plotter.

```
towerPlotter = platformPlotter(tp,'DisplayName','Tower','Marker','s','MarkerSize',2);
plotPlatform(towerPlotter,tower.Position,tower.Dimensions,quaternion(tower.Orientation,'rotvecd'))
```

Mount a monostatic radar to the top of the tower.

```
radar = fusionRadarSensor(1,'DetectionMode','Monostatic', ...
    'UpdateRate',5, ...
    'MountingLocation',[0, 0, 30], ...
    'FieldOfView',[4, 30], ...
    'MechanicalAzimuthLimits',[-60 60], ...
    'MechanicalElevationLimits',[0 0], ...
    'HasElevation',true, ...
    'RangeResolution',200, ...
    'AzimuthResolution',20, ...
    'ElevationResolution',20);
tower.Sensors = radar;
```

Add a `coveragePlotter` and plot the coverage and initial beam for the monostatic radar. When plotting the coverage, the `plotCoverage` object function requires a second argument that specifies the configuration of the sensor coverage. Obtain the configuration by using the `coverageConfig` function on the tracking scenario scene.

```
radarPlotter = coveragePlotter(tp,'Color','b','DisplayName','Radar beam');
plotCoverage(radarPlotter,coverageConfig(scene))
```

Create a detection plotter to plot the detections that the radar generates.

```
detPlotter = detectionPlotter(tp,'DisplayName','Detection','MarkerFaceColor','r','MarkerSize',4)
```

Run Scenario and Update Theater Plot

Iterate through the tracking scenario and generate radar detections. Plot the platform, radar coverage, and detections.

```
rng(2019) % for repeatable results
while advance(scene)
    % Plot target.
    plotPlatform(targetPlotter,target.Position, ...
        target.Dimensions,quaternion(target.Orientation,'rotvecd'))

    % Plot sensor coverage.
    plotCoverage(radarPlotter,coverageConfig(scene))
```

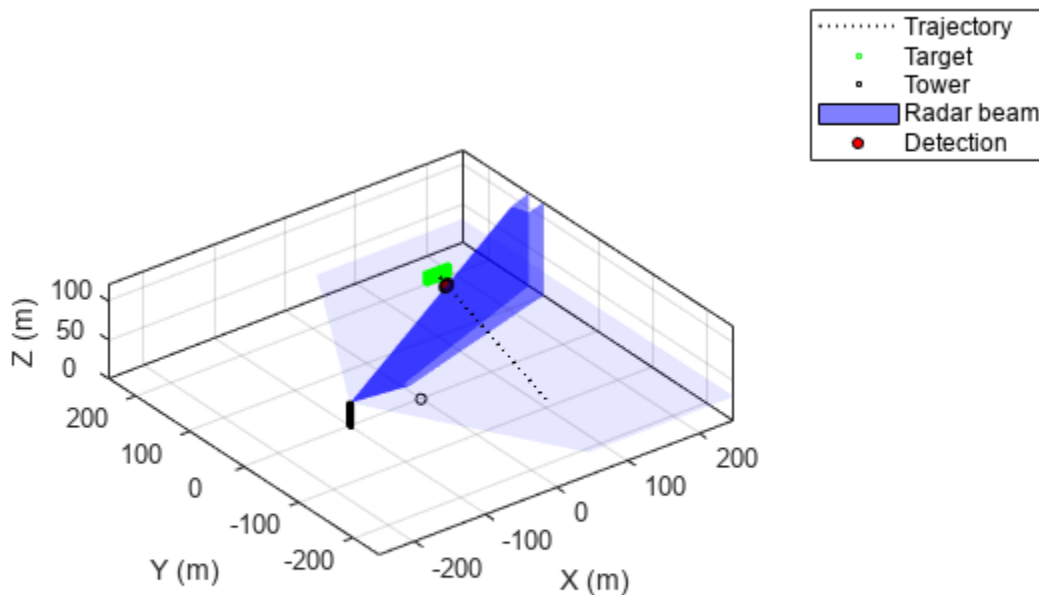
```

% Extract target pose from the view of the tower and use the extracted
% pose to generate detections.
poseInTower = targetPoses(tower);
[detections, numDets] = radar(poseInTower, scene.SimulationTime);
detPos = zeros(numDets,3);
detNoise = zeros(3,3,numDets);

% Obtain detection pose relative to the scenario frame. Also, obtain
% the covariance of the detection.
for i=1:numDets
    a = detections;
    detPos(i,:) = tower.Trajectory.Position + detections{i}.Measurement';
    detNoise(:,:,i) = detections{i}.MeasurementNoise;
end

% Plot any generated detections with the covariance ellipses.
if ~isempty(detPos)
    plotDetection(detPlotter, detPos, detNoise)
end
end

```



You can zoom in on the detection in the figure to visualize the plotted covariance ellipses of the generated detections.

Summary

In this example, you learned about the organization of a `theaterPlot` object. You also learned how to visualize a simple tracking scenario using the `theaterPlot` object.

Custom Tuning of Fusion Filters

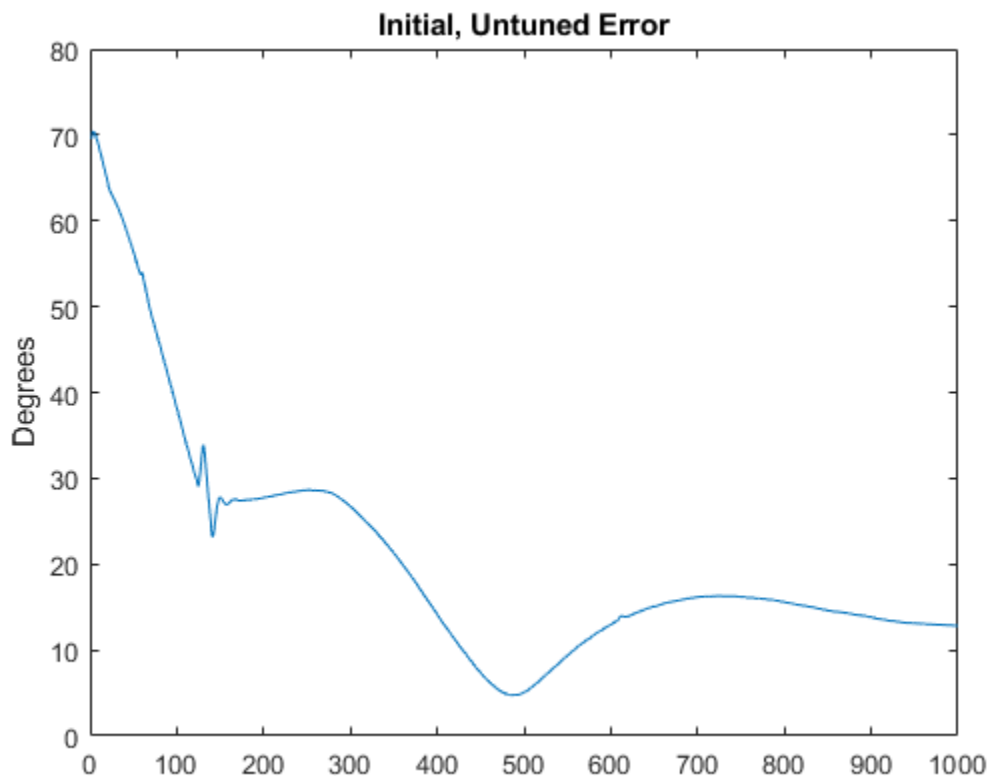
Use the `tune` function to optimize the noise parameters of several fusion filters, including the `ahrsfilter` object. This example shows how to custom a cost function for various optimization goals.

Load Sensor Data and Ground Truth

The sensor data contains sensor recordings of a UAV executing some small maneuvers. Create an `ahrsfilter` object to fuse the sensor data and estimate those maneuvers.

```
load AHRSCustomTune.mat

% Create a filter to process the data, decimating by 10.
filt = ahrsfilter('SampleRate',Fs,'DecimationFactor',10);
% Filter the sensor data and show the estimation error.
oEstInit = filt(sensorData.Accelerometer, sensorData.Gyroscope,sensorData.Magnetometer);
plotPerformance(oEstInit,groundTruth.Orientation, "Initial, Untuned Error");
```



Tune the Filter to Improve Estimation

The performance of the `ahrsfilter` without tuning noise parameters is not ideal. Use the `tune` function to improve the filter performance.

```

reset(filt);
cfg1 = tunerconfig("ahrsfilter", "MaxIterations", 20, "ObjectiveLimit", 0.0001);
tune(filt, sensorData, groundTruth(1:10:end, :), cfg1);

```

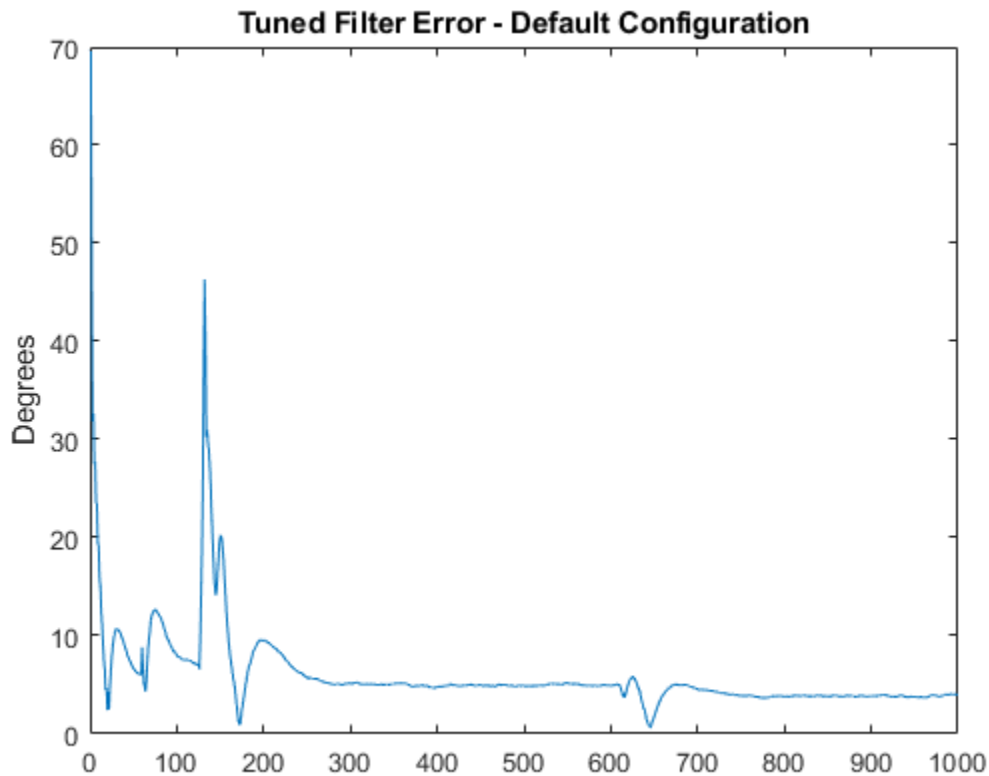
Iteration	Parameter	Metric
1	AccelerometerNoise	0.4382
1	GyroscopeNoise	0.4371
1	MagnetometerNoise	0.4370
1	GyroscopeDriftNoise	0.4370
1	LinearAccelerationNoise	0.4202
1	MagneticDisturbanceNoise	0.4188
1	LinearAccelerationDecayFactor	0.4087
1	MagneticDisturbanceDecayFactor	0.4087
2	AccelerometerNoise	0.4086
2	GyroscopeNoise	0.4066
2	MagnetometerNoise	0.4066
2	GyroscopeDriftNoise	0.4066
2	LinearAccelerationNoise	0.3937
2	MagneticDisturbanceNoise	0.3932
2	LinearAccelerationDecayFactor	0.3856
2	MagneticDisturbanceDecayFactor	0.3854
3	AccelerometerNoise	0.3853
3	GyroscopeNoise	0.3826
3	MagnetometerNoise	0.3825
3	GyroscopeDriftNoise	0.3825
3	LinearAccelerationNoise	0.3690
3	MagneticDisturbanceNoise	0.3676
3	LinearAccelerationDecayFactor	0.3613
3	MagneticDisturbanceDecayFactor	0.3611
4	AccelerometerNoise	0.3610
4	GyroscopeNoise	0.3577
4	MagnetometerNoise	0.3576
4	GyroscopeDriftNoise	0.3576
4	LinearAccelerationNoise	0.3431
4	MagneticDisturbanceNoise	0.3414
4	LinearAccelerationDecayFactor	0.3364
4	MagneticDisturbanceDecayFactor	0.3363
5	AccelerometerNoise	0.3362
5	GyroscopeNoise	0.3328
5	MagnetometerNoise	0.3326
5	GyroscopeDriftNoise	0.3326
5	LinearAccelerationNoise	0.3190
5	MagneticDisturbanceNoise	0.3183
5	LinearAccelerationDecayFactor	0.3152
5	MagneticDisturbanceDecayFactor	0.3150
6	AccelerometerNoise	0.3149
6	GyroscopeNoise	0.3121
6	MagnetometerNoise	0.3119
6	GyroscopeDriftNoise	0.3119
6	LinearAccelerationNoise	0.3040
6	MagneticDisturbanceNoise	0.3035
6	LinearAccelerationDecayFactor	0.3024
6	MagneticDisturbanceDecayFactor	0.3022
7	AccelerometerNoise	0.3022
7	GyroscopeNoise	0.2990
7	MagnetometerNoise	0.2989
7	GyroscopeDriftNoise	0.2989

7	LinearAccelerationNoise	0.2970
7	MagneticDisturbanceNoise	0.2955
7	LinearAccelerationDecayFactor	0.2952
7	MagneticDisturbanceDecayFactor	0.2948
8	AccelerometerNoise	0.2948
8	GyroscopeNoise	0.2903
8	MagnetometerNoise	0.2902
8	GyroscopeDriftNoise	0.2902
8	LinearAccelerationNoise	0.2883
8	MagneticDisturbanceNoise	0.2860
8	LinearAccelerationDecayFactor	0.2856
8	MagneticDisturbanceDecayFactor	0.2851
9	AccelerometerNoise	0.2851
9	GyroscopeNoise	0.2778
9	MagnetometerNoise	0.2777
9	GyroscopeDriftNoise	0.2777
9	LinearAccelerationNoise	0.2709
9	MagneticDisturbanceNoise	0.2698
9	LinearAccelerationDecayFactor	0.2690
9	MagneticDisturbanceDecayFactor	0.2689
10	AccelerometerNoise	0.2689
10	GyroscopeNoise	0.2593
10	MagnetometerNoise	0.2593
10	GyroscopeDriftNoise	0.2593
10	LinearAccelerationNoise	0.2492
10	MagneticDisturbanceNoise	0.2490
10	LinearAccelerationDecayFactor	0.2482
10	MagneticDisturbanceDecayFactor	0.2482
11	AccelerometerNoise	0.2481
11	GyroscopeNoise	0.2370
11	MagnetometerNoise	0.2369
11	GyroscopeDriftNoise	0.2369
11	LinearAccelerationNoise	0.2240
11	MagneticDisturbanceNoise	0.2237
11	LinearAccelerationDecayFactor	0.2230
11	MagneticDisturbanceDecayFactor	0.2228
12	AccelerometerNoise	0.2227
12	GyroscopeNoise	0.2117
12	MagnetometerNoise	0.2117
12	GyroscopeDriftNoise	0.2117
12	LinearAccelerationNoise	0.1984
12	MagneticDisturbanceNoise	0.1979
12	LinearAccelerationDecayFactor	0.1974
12	MagneticDisturbanceDecayFactor	0.1974
13	AccelerometerNoise	0.1973
13	GyroscopeNoise	0.1878
13	MagnetometerNoise	0.1878
13	GyroscopeDriftNoise	0.1878
13	LinearAccelerationNoise	0.1766
13	MagneticDisturbanceNoise	0.1763
13	LinearAccelerationDecayFactor	0.1761
13	MagneticDisturbanceDecayFactor	0.1761
14	AccelerometerNoise	0.1760
14	GyroscopeNoise	0.1686
14	MagnetometerNoise	0.1685
14	GyroscopeDriftNoise	0.1685
14	LinearAccelerationNoise	0.1601
14	MagneticDisturbanceNoise	0.1599

14	LinearAccelerationDecayFactor	0.1597
14	MagneticDisturbanceDecayFactor	0.1597
15	AccelerometerNoise	0.1596
15	GyroscopeNoise	0.1536
15	MagnetometerNoise	0.1536
15	GyroscopeDriftNoise	0.1536
15	LinearAccelerationNoise	0.1472
15	MagneticDisturbanceNoise	0.1469
15	LinearAccelerationDecayFactor	0.1469
15	MagneticDisturbanceDecayFactor	0.1469
16	AccelerometerNoise	0.1468
16	GyroscopeNoise	0.1422
16	MagnetometerNoise	0.1422
16	GyroscopeDriftNoise	0.1422
16	LinearAccelerationNoise	0.1380
16	MagneticDisturbanceNoise	0.1378
16	LinearAccelerationDecayFactor	0.1377
16	MagneticDisturbanceDecayFactor	0.1377
17	AccelerometerNoise	0.1377
17	GyroscopeNoise	0.1352
17	MagnetometerNoise	0.1351
17	GyroscopeDriftNoise	0.1351
17	LinearAccelerationNoise	0.1351
17	MagneticDisturbanceNoise	0.1351
17	LinearAccelerationDecayFactor	0.1351
17	MagneticDisturbanceDecayFactor	0.1351
18	AccelerometerNoise	0.1351
18	GyroscopeNoise	0.1351
18	MagnetometerNoise	0.1351
18	GyroscopeDriftNoise	0.1351
18	LinearAccelerationNoise	0.1351
18	MagneticDisturbanceNoise	0.1351
18	LinearAccelerationDecayFactor	0.1350
18	MagneticDisturbanceDecayFactor	0.1350
19	AccelerometerNoise	0.1350
19	GyroscopeNoise	0.1348
19	MagnetometerNoise	0.1344
19	GyroscopeDriftNoise	0.1344
19	LinearAccelerationNoise	0.1344
19	MagneticDisturbanceNoise	0.1344
19	LinearAccelerationDecayFactor	0.1344
19	MagneticDisturbanceDecayFactor	0.1344
20	AccelerometerNoise	0.1344
20	GyroscopeNoise	0.1344
20	MagnetometerNoise	0.1344
20	GyroscopeDriftNoise	0.1344
20	LinearAccelerationNoise	0.1344
20	MagneticDisturbanceNoise	0.1344
20	LinearAccelerationDecayFactor	0.1344
20	MagneticDisturbanceDecayFactor	0.1344

Filter the sensor data using the tuned filter and show the orientation error.

```
oEstTuned = filt(sensorData.Accelerometer,sensorData.Gyroscope,sensorData.Magnetometer);
plotPerformance(oEstTuned,groundTruth.Orientation,"Tuned Filter Error - Default Configuration");
```



Use the CustomCostFcn and MATLAB Coder (R) to Accelerate and Optimize Tuning

The performance of the filter is improved after tuning but the tuning process can often take a long time. The `tunerconfig` object allows for a custom cost function to optimize this process. You can also use MATLAB Coder to create a mex function to accelerate the tuning speed. The custom cost function must have a signature `cost = fcn(params, sensorData, groundTruth)`, where `cost` is a scalar real number, `params` is a struct of noise parameters to be optimized, `sensorData` is a table of sensor data, and `groundTruth` is a table ground truth data.

From the last section, the `ahrsfilter` did not estimate the orientation very well during some of the maneuvers. Instead of using the default root-mean-squared error, the custom cost function uses higher order terms to more severely penalize outliers.

Display the details of the custom cost function. The function is attached as an m-file.

```
type customFcn.m
```

```
function c = customFcn(params, sensorData, groundTruth)
% Custom Cost function for optimizing the ahrsfilter

% Set any nontunable parameters in the constructor
decim = 10;
h = ahrsfilter('SampleRate', 200, 'DecimationFactor', decim);

% Parameterize the filter instance with the current-best parameters from
% the params struct.
h.AccelerometerNoise = params.AccelerometerNoise;
```



```

h.GyroscopeNoise = params.GyroscopeNoise;
h.MagnetometerNoise = params.MagnetometerNoise;
h.GyroscopeDriftNoise = params.GyroscopeDriftNoise;
h.LinearAccelerationNoise = params.LinearAccelerationNoise;
h.MagneticDisturbanceNoise = params.MagneticDisturbanceNoise;
h.LinearAccelerationDecayFactor = params.LinearAccelerationDecayFactor;
h.MagneticDisturbanceDecayFactor = params.MagneticDisturbanceDecayFactor;
h.ExpectedMagneticFieldStrength = params.ExpectedMagneticFieldStrength;

% Fuse sensor data
qest = h(sensorData.Accelerometer, sensorData.Gyroscope, ...
        sensorData.Magnetometer);

% Compute the orientation error
d = dist(qest, groundTruth.Orientation(1:decim:end));

% Penalize outliers heavily by using the 6th power.
c = sqrt(sqrt( mean(d(10:end,:).^6) ));

```

To create a mex file, you need exemplar input arguments. Create a parameters exemplar and copy the properties from `filt`.

```

p = {'AccelerometerNoise', 'DecimationFactor', 'ExpectedMagneticFieldStrength', ...
    'GyroscopeDriftNoise', 'GyroscopeNoise', 'InitialProcessNoise', 'LinearAccelerationDecayFactor', ...
    'LinearAccelerationNoise', 'MagneticDisturbanceDecayFactor', 'MagneticDisturbanceNoise', ...
    'MagnetometerNoise', 'OrientationFormat', 'SampleRate'}

p = 1x13 cell
    {'AccelerometerNoise'}    {'DecimationFactor'}    {'ExpectedMagneticFieldStrength'}    {'Gyroscop...

for idx=1:numel(p)
    paramEx.(p{idx}) = filt.(p{idx});
end
disp(paramEx);

```

```

AccelerometerNoise: 5.4972e-07
DecimationFactor: 10
ExpectedMagneticFieldStrength: 50
GyroscopeDriftNoise: 4.0927e-11
GyroscopeNoise: 0.0041
InitialProcessNoise: [12x12 double]
LinearAccelerationDecayFactor: 0.0050
LinearAccelerationNoise: 1.4370e-04
MagneticDisturbanceDecayFactor: 0.9872
MagneticDisturbanceNoise: 0.0360
MagnetometerNoise: 0.1278
OrientationFormat: 'quaternion'
SampleRate: 200

```

Generate code.

```
codegen customFcn.m -args {paramEx sensorData, groundTruth}
```

Code generation successful.

Use the mex function to tune quickly.

```

cfg = tunerconfig("ahrsfilter", "Cost", "Custom", ...
    "CustomCostFcn", @customFcn_mex, "MaxIterations", 20, "ObjectiveLimit", 0.0001);

```

```
reset(filt);
tune(filt, sensorData, groundTruth, cfg);
```

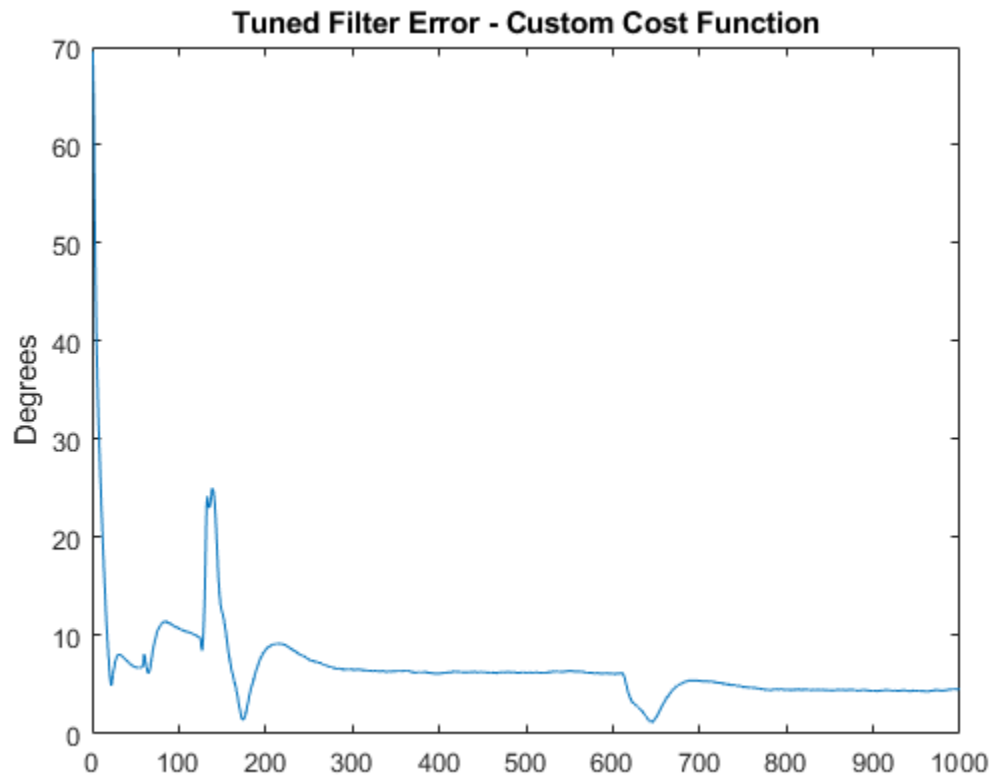
Iteration	Parameter	Metric
1	AccelerometerNoise	0.1581
1	GyroscopeNoise	0.1544
1	MagnetometerNoise	0.1544
1	GyroscopeDriftNoise	0.1544
1	LinearAccelerationNoise	0.1504
1	MagneticDisturbanceNoise	0.1498
1	LinearAccelerationDecayFactor	0.1497
1	MagneticDisturbanceDecayFactor	0.1474
2	AccelerometerNoise	0.1474
2	GyroscopeNoise	0.1437
2	MagnetometerNoise	0.1436
2	GyroscopeDriftNoise	0.1436
2	LinearAccelerationNoise	0.1395
2	MagneticDisturbanceNoise	0.1387
2	LinearAccelerationDecayFactor	0.1387
2	MagneticDisturbanceDecayFactor	0.1367
3	AccelerometerNoise	0.1367
3	GyroscopeNoise	0.1332
3	MagnetometerNoise	0.1332
3	GyroscopeDriftNoise	0.1332
3	LinearAccelerationNoise	0.1293
3	MagneticDisturbanceNoise	0.1284
3	LinearAccelerationDecayFactor	0.1284
3	MagneticDisturbanceDecayFactor	0.1271
4	AccelerometerNoise	0.1270
4	GyroscopeNoise	0.1243
4	MagnetometerNoise	0.1242
4	GyroscopeDriftNoise	0.1242
4	LinearAccelerationNoise	0.1209
4	MagneticDisturbanceNoise	0.1201
4	LinearAccelerationDecayFactor	0.1201
4	MagneticDisturbanceDecayFactor	0.1193
5	AccelerometerNoise	0.1193
5	GyroscopeNoise	0.1180
5	MagnetometerNoise	0.1178
5	GyroscopeDriftNoise	0.1178
5	LinearAccelerationNoise	0.1158
5	MagneticDisturbanceNoise	0.1152
5	LinearAccelerationDecayFactor	0.1152
5	MagneticDisturbanceDecayFactor	0.1147
6	AccelerometerNoise	0.1147
6	GyroscopeNoise	0.1147
6	MagnetometerNoise	0.1143
6	GyroscopeDriftNoise	0.1143
6	LinearAccelerationNoise	0.1132
6	MagneticDisturbanceNoise	0.1123
6	LinearAccelerationDecayFactor	0.1123
6	MagneticDisturbanceDecayFactor	0.1118
7	AccelerometerNoise	0.1118
7	GyroscopeNoise	0.1113
7	MagnetometerNoise	0.1108
7	GyroscopeDriftNoise	0.1108
7	LinearAccelerationNoise	0.1100

7	MagneticDisturbanceNoise	0.1093
7	LinearAccelerationDecayFactor	0.1093
7	MagneticDisturbanceDecayFactor	0.1093
8	AccelerometerNoise	0.1093
8	GyroscopeNoise	0.1088
8	MagnetometerNoise	0.1084
8	GyroscopeDriftNoise	0.1084
8	LinearAccelerationNoise	0.1084
8	MagneticDisturbanceNoise	0.1084
8	LinearAccelerationDecayFactor	0.1084
8	MagneticDisturbanceDecayFactor	0.1084
9	AccelerometerNoise	0.1084
9	GyroscopeNoise	0.1076
9	MagnetometerNoise	0.1072
9	GyroscopeDriftNoise	0.1072
9	LinearAccelerationNoise	0.1072
9	MagneticDisturbanceNoise	0.1072
9	LinearAccelerationDecayFactor	0.1072
9	MagneticDisturbanceDecayFactor	0.1071
10	AccelerometerNoise	0.1071
10	GyroscopeNoise	0.1068
10	MagnetometerNoise	0.1065
10	GyroscopeDriftNoise	0.1065
10	LinearAccelerationNoise	0.1062
10	MagneticDisturbanceNoise	0.1060
10	LinearAccelerationDecayFactor	0.1060
10	MagneticDisturbanceDecayFactor	0.1059
11	AccelerometerNoise	0.1059
11	GyroscopeNoise	0.1057
11	MagnetometerNoise	0.1055
11	GyroscopeDriftNoise	0.1055
11	LinearAccelerationNoise	0.1049
11	MagneticDisturbanceNoise	0.1048
11	LinearAccelerationDecayFactor	0.1048
11	MagneticDisturbanceDecayFactor	0.1048
12	AccelerometerNoise	0.1048
12	GyroscopeNoise	0.1047
12	MagnetometerNoise	0.1045
12	GyroscopeDriftNoise	0.1045
12	LinearAccelerationNoise	0.1038
12	MagneticDisturbanceNoise	0.1036
12	LinearAccelerationDecayFactor	0.1036
12	MagneticDisturbanceDecayFactor	0.1035
13	AccelerometerNoise	0.1035
13	GyroscopeNoise	0.1035
13	MagnetometerNoise	0.1033
13	GyroscopeDriftNoise	0.1033
13	LinearAccelerationNoise	0.1029
13	MagneticDisturbanceNoise	0.1027
13	LinearAccelerationDecayFactor	0.1027
13	MagneticDisturbanceDecayFactor	0.1027
14	AccelerometerNoise	0.1027
14	GyroscopeNoise	0.1024
14	MagnetometerNoise	0.1021
14	GyroscopeDriftNoise	0.1021
14	LinearAccelerationNoise	0.1019
14	MagneticDisturbanceNoise	0.1018
14	LinearAccelerationDecayFactor	0.1018

14	MagneticDisturbanceDecayFactor	0.1018
15	AccelerometerNoise	0.1018
15	GyroscopeNoise	0.1014
15	MagnetometerNoise	0.1012
15	GyroscopeDriftNoise	0.1012
15	LinearAccelerationNoise	0.1012
15	MagneticDisturbanceNoise	0.1012
15	LinearAccelerationDecayFactor	0.1011
15	MagneticDisturbanceDecayFactor	0.1011
16	AccelerometerNoise	0.1011
16	GyroscopeNoise	0.1008
16	MagnetometerNoise	0.1008
16	GyroscopeDriftNoise	0.1008
16	LinearAccelerationNoise	0.1006
16	MagneticDisturbanceNoise	0.1005
16	LinearAccelerationDecayFactor	0.1004
16	MagneticDisturbanceDecayFactor	0.1004
17	AccelerometerNoise	0.1004
17	GyroscopeNoise	0.1001
17	MagnetometerNoise	0.1001
17	GyroscopeDriftNoise	0.1001
17	LinearAccelerationNoise	0.0998
17	MagneticDisturbanceNoise	0.0998
17	LinearAccelerationDecayFactor	0.0996
17	MagneticDisturbanceDecayFactor	0.0995
18	AccelerometerNoise	0.0995
18	GyroscopeNoise	0.0992
18	MagnetometerNoise	0.0992
18	GyroscopeDriftNoise	0.0992
18	LinearAccelerationNoise	0.0990
18	MagneticDisturbanceNoise	0.0989
18	LinearAccelerationDecayFactor	0.0987
18	MagneticDisturbanceDecayFactor	0.0986
19	AccelerometerNoise	0.0986
19	GyroscopeNoise	0.0980
19	MagnetometerNoise	0.0980
19	GyroscopeDriftNoise	0.0980
19	LinearAccelerationNoise	0.0980
19	MagneticDisturbanceNoise	0.0980
19	LinearAccelerationDecayFactor	0.0978
19	MagneticDisturbanceDecayFactor	0.0975
20	AccelerometerNoise	0.0975
20	GyroscopeNoise	0.0965
20	MagnetometerNoise	0.0965
20	GyroscopeDriftNoise	0.0965
20	LinearAccelerationNoise	0.0964
20	MagneticDisturbanceNoise	0.0964
20	LinearAccelerationDecayFactor	0.0964
20	MagneticDisturbanceDecayFactor	0.0964

Filter the data and show the orientation error.

```
oEst = filt(sensorData.Accelerometer, sensorData.Gyroscope, sensorData.Magnetometer);
plotPerformance(oEst, groundTruth.Orientation, "Tuned Filter Error - Custom Cost Function");
```



Supporting Functions

plotPerformance Plot the orientation error

```
function plotPerformance(est, act, txt)
% Plot the orientation error in degrees in a new figure window.
figure;
plot(rad2deg(dist(est, act(1:10:end))));
ylabel("Degrees")
title(txt);
end
```

Wireless Data Streaming and Sensor Fusion Using BNO055

This example shows how to get data from a Bosch BNO055 IMU sensor through an HC-05 Bluetooth® module, and to use the 9-axis AHRS fusion algorithm on the sensor data to compute orientation of the device. The example creates a figure which gets updated as you move the device.

BNO055 is a 9-axis sensor with accelerometer, gyroscope, and magnetometer. Accelerometer measures acceleration, gyroscope measures angular velocity, and magnetometer measures magnetic field in x -, y - and z - axes.

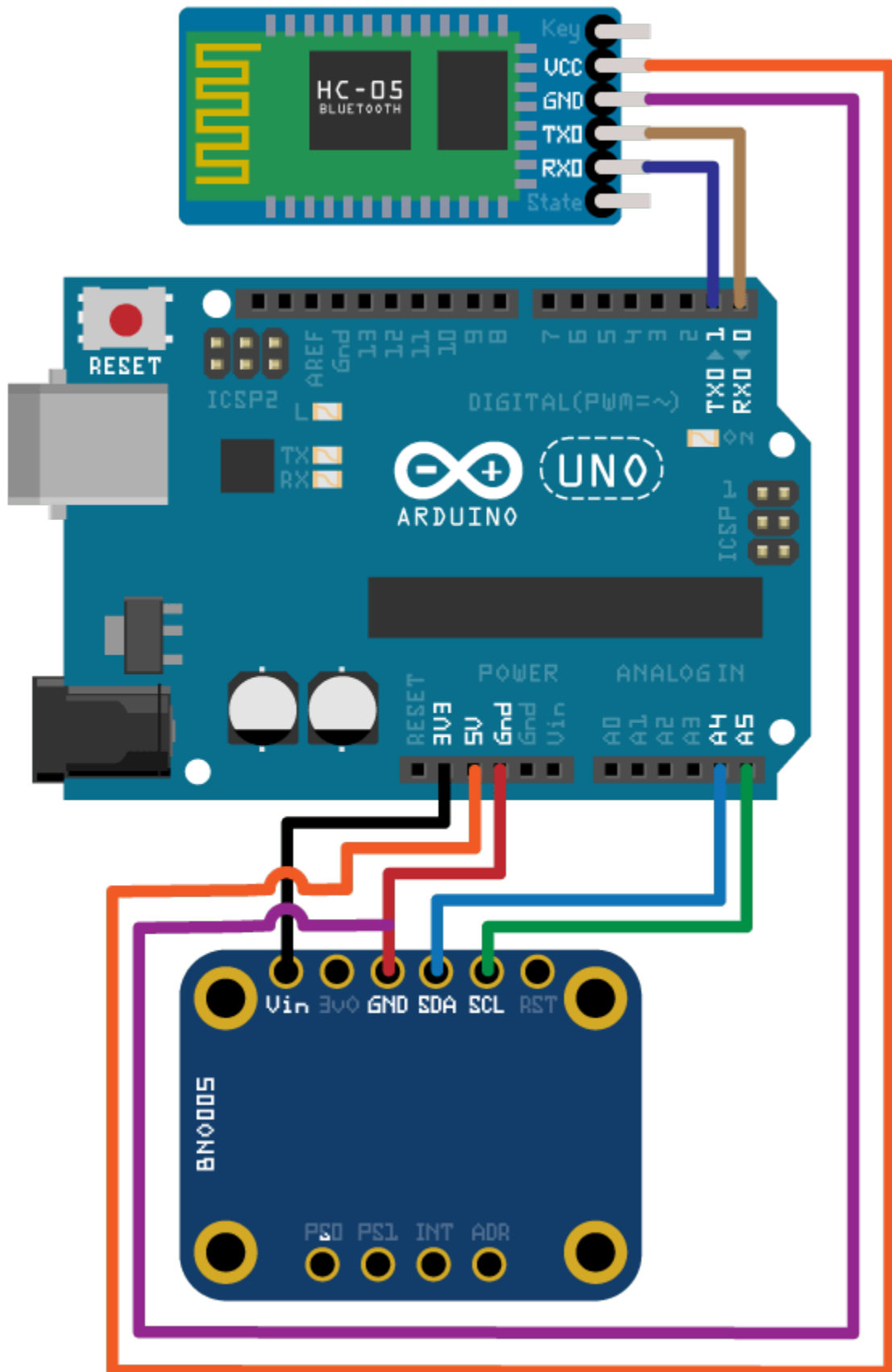
Required MathWorks® Products

- MATLAB®
- MATLAB Support Package for Arduino® Hardware
- Instrument Control Toolbox™
- Sensor Fusion and Tracking Toolbox™ or Navigation Toolbox™

Required Hardware

- Arduino® Uno
- Bosch BNO055 Sensor
- HC-05 Bluetooth® Module

Hardware Connection



Connect the SDA, SCL, GND, and the VCC pins of the BNO055 sensor to the corresponding pins on the Arduino® Uno board with the connections:

- SDA - A4
- SCL - A5
- VCC - +3.3V
- GND - GND

Connect the TX, RX, GND and VCC pins of the HC-05 module to the corresponding pins on the Arduino® Uno board. This example uses the connections:

- TX - Digital Pin 0 (RX)
- RX - Digital Pin 1 (TX)
- VCC - 5V
- GND - GND

Ensure that the connections to the sensors and Bluetooth® module are intact. It is recommended to use a BNO055 shield for Arduino Uno (Arduino 9 Axis Motion Shield). See “Troubleshooting Sensors” (MATLAB Support Package for Arduino Hardware) to debug the sensor related issues.

Setup and Configure Arduino for Bluetooth® Communication

Configure the Arduino Uno board to communicate through Bluetooth® using the `arduinsetup` command from MATLAB command prompt. See “Set up and Configure Arduino Hardware” (MATLAB Support Package for Arduino Hardware) for steps on how to configure the Arduino board for communication through Bluetooth®. Make sure to check the box for I2C Libraries to be included during the Setup.

Create Sensor Object

Create an `arduino` object.

```
a = arduino('btspp://98D33230EB9F', 'Uno');
```

Create the `bno055` sensor object in the `OperatingMode` 'amg'.

```
fs = 100; % Sample Rate in Hz  
imu = bno055(a, 'SampleRate', fs, 'OutputFormat', 'matrix', 'OperatingMode', 'amg');
```

Compensating for Hard Iron and Soft Iron Distortions

Fusion algorithms use magnetometer readings which need to be compensated 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,

- 1 Rotate the sensor from 0 to 360 degree along each axis.
- 2 Use the `magcal` function to obtain the correction coefficients.

These correction values change with the surroundings.

To obtain correction coefficients for both hard iron and soft iron distortions:


```

ts = tic;
stopTimer = 50;
magReadings=[];
while(toc(ts) < stopTimer)
    % Rotate the sensor along x axis from 0 to 360 degree.
    % Take 2-3 rotations to improve accuracy.
    % For other axes, rotate along that axes.
    [accel,gyro,mag] = read(imu);
    magReadings = [magReadings;mag];
end

[A, b] = magcal(magReadings); % A = 3x3 matrix for soft iron correction
                             % b = 3x1 vector for hard iron correction

```

Aligning the axis of BNO055 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 expect all the sensors in the object to have their axes aligned with NED convention. The sensor values have to be inverted so that they are in accordance with the NED coordinates.

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. See “Custom Tuning of Fusion Filters” on page 6-658 for more details related to tuning filter parameters.

The example uses `ahrsfilter` to demonstrate orientation estimation. See “Determine Orientation Using Inertial Sensors” for more details related to inertial fusion algorithms.

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 the 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 the orientation of the sensor and create a figure that gets updated as you move the sensor. The initial position of the sensor should be such that the device x -axis is pointing towards magnetic north, the device y -axis is pointing to east and the device z -axis is pointing downwards. You could use a cellphone or compass to determine magnetic north.

```

% GyroscopeNoise, AccelerometerNoise and MagnetometerNoise are determined from the BNO055 datasheet
% NoisePower = OutputNoisePowerDensityrms^2 * Bandwidth

GyroscopeNoiseBNO055 = 3.05e-06; % GyroscopeNoise (variance value) in units of (rad/s)^2
AccelerometerNoiseBNO055 = 67.53e-06; % AccelerometerNoise (variance value) in units of (m/s^2)^2
MagnetometerNoiseBNO055 = 1; %MagnetometerNoise (variance value) in units of uT^2

viewer = HelperOrientationViewer('Title',{'AHRS Filter'});

```

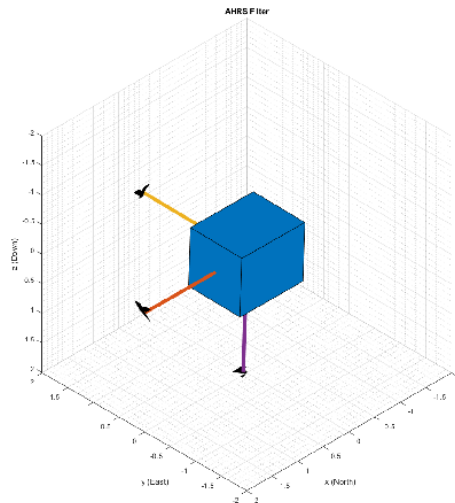
```
FUSE = ahrsfilter('SampleRate',imu.SampleRate,'GyroscopeNoise',GyroscopeNoiseBN0055,'Acceleromet
stopTimer=10;
```

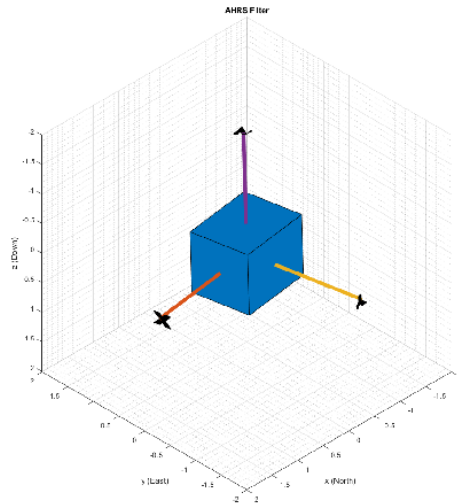
After executing the below code snippet, slowly move the sensor and check if the motion in the figure matches the motion of the sensor. Increase the `stopTimer` value, if you need to track the orientation for longer time.

```
magx_correction = b(1);
magy_correction = b(2);
magz_correction = b(3);

ts = tic;
while(toc(ts) < stopTimer)
    [accel,gyro,mag] = read(imu);
    % Align coordinates in accordance with NED convention
    accel = [-accel(:,1), accel(:,2), accel(:,3)];
    gyro = [gyro(:,1), -gyro(:,2), -gyro(:,3)];
    mag = [(mag(:,1)-magx_correction), -(mag(:,2)- magy_correction), -(mag(:,3)-magz_correction)];
    rotators = FUSE(accel,gyro,mag);
    for j = numel(rotators)
        viewer(rotators(j));
    end
end
```

If the sensor is stationary at the initial position where the device x-axis points to the magnetic north, the device y-axis points to the east, and the device z-axis points downwards, the x-axis in the figure will be parallel to and aligned with the positive x-axis, the y-axis in the figure will be parallel to and aligned with the positive y-axis, and the z-axis in the figure will be parallel to and aligned with the positive z-axis.





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, MPU-9250, and STMicroelectronics LSM9DS1.

Binaural Audio Rendering Using Head Tracking

Track head orientation by fusing data received from an IMU, and then control the direction of arrival of a sound source by applying head-related transfer functions (HRTF).

In a typical virtual reality setup, the IMU sensor is attached to the user's headphones or VR headset so that the perceived position of a sound source is relative to a visual cue independent of head movements. For example, if the sound is perceived as coming from the monitor, it remains that way even if the user turns his head to the side.

Required Hardware

- Arduino Uno
- Invensense MPU-9250

Hardware Connection

First, connect the Invensense MPU-9250 to the Arduino board. For more details, see “Estimating Orientation Using Inertial Sensor Fusion and MPU-9250” on page 6-445.

Create Sensor Object and IMU Filter

Create an arduino object.

```
a = arduino;
```

Create the Invensense MPU-9250 sensor object.

```
imu = mpu9250(a);
```

Create and set the sample rate of the Kalman filter.

```
Fs = imu.SampleRate;  
imufilt = imufilter('SampleRate',Fs);
```

Load the ARI HRTF Dataset

When sound travels from a point in space to your ears, you can localize it based on interaural time and level differences (ITD and ILD). These frequency-dependent ITD and ILD's can be measured and represented as a pair of impulse responses for any given source elevation and azimuth. The ARI HRTF Dataset contains 1550 pairs of impulse responses which span azimuths over 360 degrees and elevations from -30 to 80 degrees. You use these impulse responses to filter a sound source so that it is perceived as coming from a position determined by the sensor's orientation. If the sensor is attached to a device on a user's head, the sound is perceived as coming from one fixed place despite head movements.

First, load the HRTF dataset.

```
ARIDataset = load('ReferenceHRTF.mat');
```

Then, get the relevant HRTF data from the dataset and put it in a useful format for our processing.

```
hrtfData = double(ARIDataset.hrtfData);  
hrtfData = permute(hrtfData,[2,3,1]);
```

Get the associated source positions. Angles should be in the same range as the sensor. Convert the azimuths from [0,360] to [-180,180].

```
sourcePosition = ARIDataset.sourcePosition(:,[1,2]);
sourcePosition(:,1) = sourcePosition(:,1) - 180;
```

Load Monaural Recording

Load an ambisonic recording of a helicopter. Keep only the first channel, which corresponds to an omnidirectional recording. Resample it to 48 kHz for compatibility with the HRTF data set.

```
[heli,originalSampleRate] = audioread('Heli_16ch_ACN_SN3D.wav');
heli = 12*heli(:,1); % keep only one channel
```

```
sampleRate = 48e3;
heli = resample(heli,sampleRate,originalSampleRate);
```

Load the audio data into a `SignalSource` object. Set the `SamplesPerFrame` to 0.1 seconds.

```
sigsrc = dsp.SignalSource(heli, ...
    'SamplesPerFrame',sampleRate/10, ...
    'SignalEndAction','Cyclic repetition');
```

Set Up the Audio Device

Create an `audioDeviceWriter` with the same sample rate as the audio signal.

```
deviceWriter = audioDeviceWriter('SampleRate',sampleRate);
```

Create FIR Filters for the HRTF coefficients

Create a pair of FIR filters to perform binaural HRTF filtering.

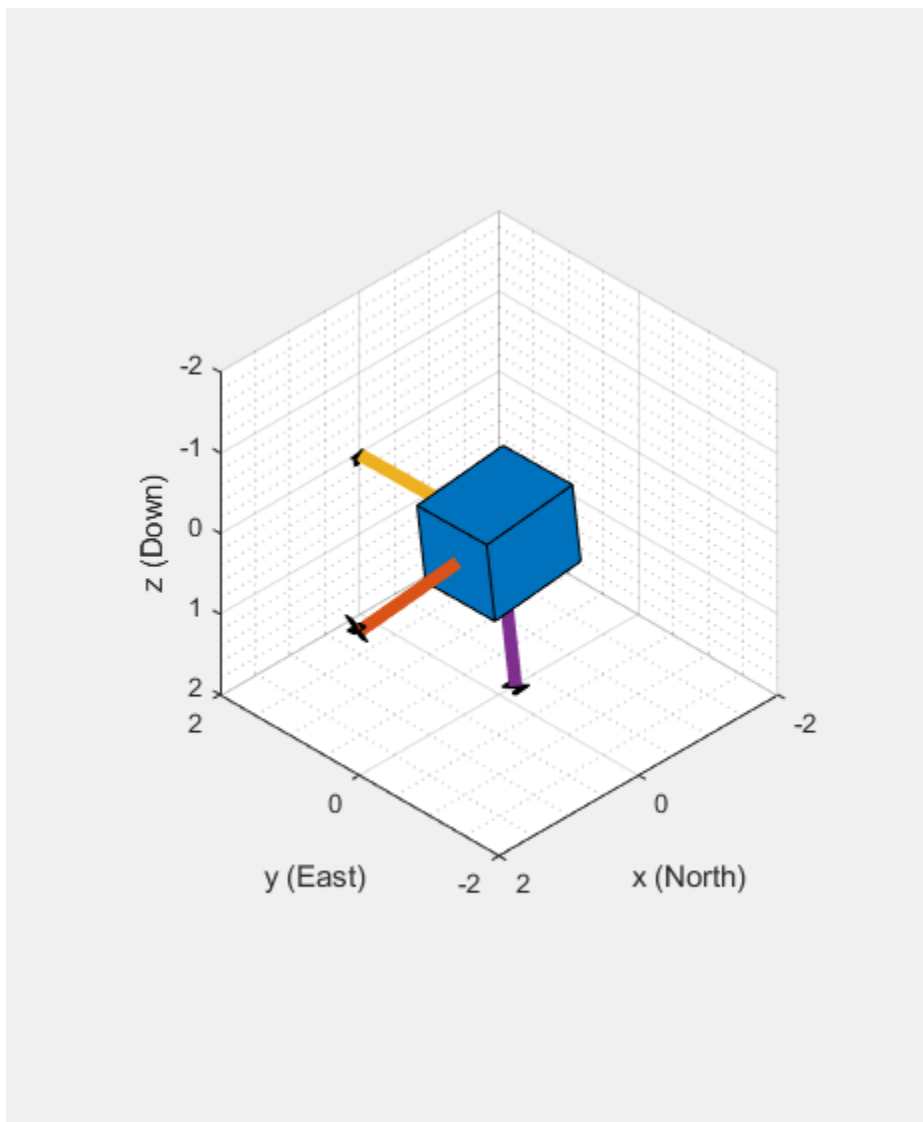
```
FIR = cell(1,2);
FIR{1} = dsp.FIRFilter('NumeratorSource','Input port');
FIR{2} = dsp.FIRFilter('NumeratorSource','Input port');
```

Initialize the Orientation Viewer

Create an object to perform real-time visualization for the orientation of the IMU sensor. Call the IMU filter once and display the initial orientation.

```
orientationScope = HelperOrientationViewer;
data = read(imu);

qimu = imufilt(data.Acceleration,data.AngularVelocity);
orientationScope(qimu);
```



Audio Processing Loop

Execute the processing loop for 30 seconds. This loop performs the following steps:

- 1 Read data from the IMU sensor.
- 2 Fuse IMU sensor data to estimate the orientation of the sensor. Visualize the current orientation.
- 3 Convert the orientation from a quaternion representation to pitch and yaw in Euler angles.
- 4 Use `interpolateHRTF` to obtain a pair of HRTFs at the desired position.
- 5 Read a frame of audio from the signal source.
- 6 Apply the HRTFs to the mono recording and play the stereo signal. This is best experienced using headphones.

```
imu0verruns = 0;  
audioUnderruns = 0;  
audioFiltered = zeros(sigsrc.SamplesPerFrame,2);
```

```
tic
while toc < 30

    % Read from the IMU sensor.
    [data,overrun] = read(imu);
    if overrun > 0
        imuOverruns = imuOverruns + overrun;
    end

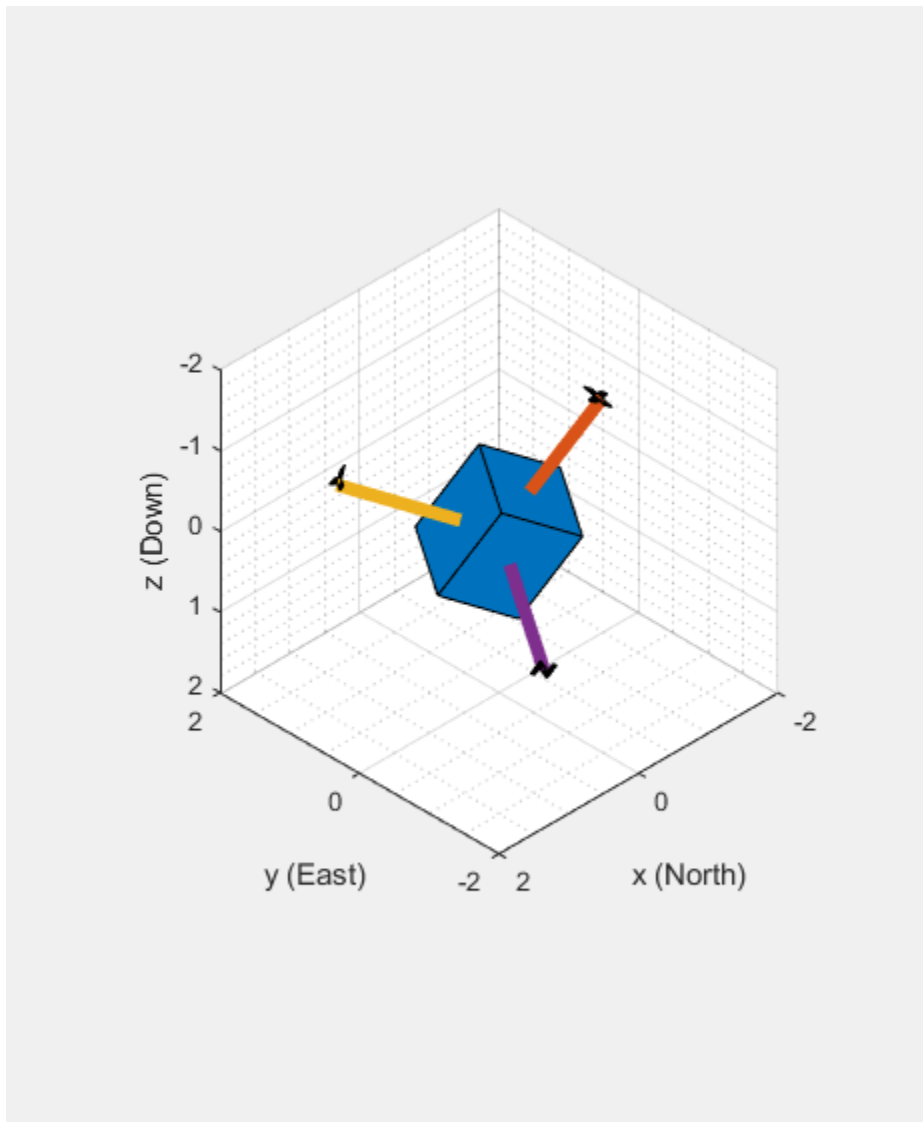
    % Fuse IMU sensor data to estimate the orientation of the sensor.
    qimu = imufilt(data.Acceleration,data.AngularVelocity);
    orientationScope(qimu);

    % Convert the orientation from a quaternion representation to pitch and yaw in Euler angles.
    ypr = eulerd(qimu,'zyx','frame');
    yaw = ypr(end,1);
    pitch = ypr(end,2);
    desiredPosition = [yaw,pitch];

    % Obtain a pair of HRTFs at the desired position.
    interpolatedIR = squeeze(interpolateHRTF(hrtfData,sourcePosition,desiredPosition));

    % Read audio from file
    audioIn = sigsrc();

    % Apply HRTFs
    audioFiltered(:,1) = FIR{1}(audioIn, interpolatedIR(1,:)); % Left
    audioFiltered(:,2) = FIR{2}(audioIn, interpolatedIR(2,:)); % Right
    audioUnderruns = audioUnderruns + deviceWriter(squeeze(audioFiltered));
end
```

**Cleanup**

Release resources, including the sound device.

```
release(sigsrc)  
release(deviceWriter)  
clear imu a
```


Motion Planning in Urban Environments Using Dynamic Occupancy Grid Map

This example shows you how to perform dynamic replanning in an urban driving scene using a Frenet reference path. In this example, you use a dynamic occupancy grid map estimate of the local environment to find optimal local trajectories.

Introduction

Dynamic replanning for autonomous vehicles is typically done with a local motion planner. The local motion planner is responsible for generating an optimal trajectory based on the global plan and information about the surrounding environment. Information about the surrounding environment can be described mainly in two ways:

- 1 Discrete set of objects in the surrounding environment with defined geometries.
- 2 Discretized grid with estimate about free and occupied regions in the surrounding environment.

In the presence of dynamic obstacles in the environment, a local motion planner requires short-term predictions of the information about the surroundings to assess the validity of the planned trajectories. The choice of environment representation is typically governed by the upstream perception algorithm. For planning algorithms, the object-based representation offers a memory-efficient description of the environment. It also allows for an easier way to define inter-object relations for behavior prediction. On the other hand, a grid-based approach allows for an object-model-free representation, which assists in efficient collision-checking in complex scenarios with large number of objects. The grid-based representation is also less sensitive to imperfections of object extraction such as false and missed targets. A hybrid of these two approaches is also possible by extracting object hypothesis from the grid-based representation.

In this example, you represent the surrounding environment as a dynamic occupancy grid map. For an example using the discrete set of objects, refer to the “Highway Trajectory Planning Using Frenet Reference Path” (Navigation Toolbox) example. A dynamic occupancy grid map is a grid-based estimate of the local environment around the ego vehicle. In addition to estimating the probability of occupancy, the dynamic occupancy grid also estimates the kinematic attributes of each cell, such as velocity, turn-rate, and acceleration. Further, the estimates from the dynamic grid can be predicted for a short-time in the future to assess the occupancy of the local environment in the near future. In this example, you obtain the grid-based estimate of the environment by fusing point clouds from six lidars mounted on the ego vehicle.

Set Up Scenario and Grid-Based Tracker

The scenario used in this example represents an urban intersection scene and contains a variety of objects, including pedestrians, bicyclists, cars, and trucks. The ego vehicle is equipped with six homogenous lidar sensors, each with a field of view of 90 degrees, providing 360-degree coverage around the ego vehicle. For more details on the scenario and sensor models, refer to the “Grid-Based Tracking in Urban Environments Using Multiple Lidars” on page 6-618 example. The definition of scenario and sensors is wrapped in the helper function `helperGridBasedPlanningScenario`.

```
% For reproducible results
rng(2020);

% Create scenario, ego vehicle and simulated lidar sensors
[scenario, egoVehicle, lidars] = helperGridBasedPlanningScenario;
```

Now, define a grid-based tracker using the `trackerGridRFS` System object™. The tracker outputs both object-level and grid-level estimate of the environment. The grid-level estimate describes the occupancy and state of the local environment and can be obtained as the fourth output from the tracker. For more details on how to set up a grid-based tracker, refer to the “Grid-Based Tracking in Urban Environments Using Multiple Lidars” on page 6-618 example.

```
% Set up sensor configurations for each lidar
sensorConfigs = cell(numel(lidars),1);

% Fill in sensor configurations
for i = 1:numel(sensorConfigs)
    sensorConfigs{i} = helperGetLidarConfig(lidars{i},egoVehicle);
end

% Set up tracker
tracker = trackerGridRFS('SensorConfigurations',sensorConfigs,...
    'HasSensorConfigurationsInput',true,...
    'GridLength',120,...
    'GridWidth',120,...
    'GridResolution',2,...
    'GridOriginInLocal',[-60 -60],...
    'NumParticles',1e5,...
    'NumBirthParticles',2e4,...
    'VelocityLimits',[-15 15;-15 15],...
    'BirthProbability',0.025,...
    'ProcessNoise',5*eye(2),...
    'DeathRate',1e-3,...
    'FreeSpaceDiscountFactor',1e-2,...
    'AssignmentThreshold',8,...
    'MinNumCellsPerCluster',4,...
    'ClusteringThreshold',4,...
    'ConfirmationThreshold',[3 4],...
    'DeletionThreshold',[4 4]);
```

Set Up Motion Planner

Set up a local motion planning algorithm to plan optimal trajectories in Frenet coordinates along a global reference path.

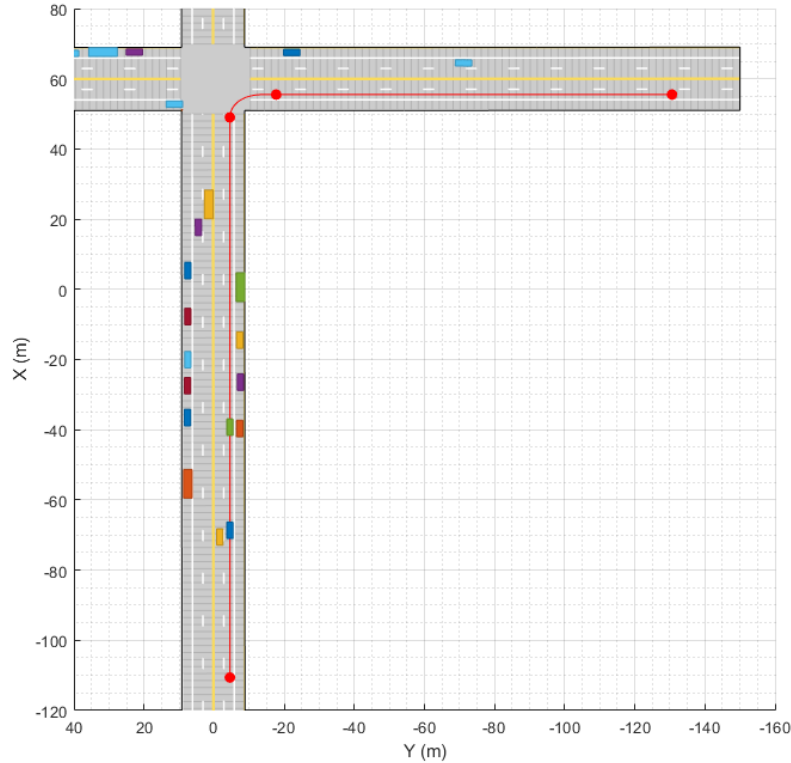
Define the global reference path using the `referencePathFrenet` (Navigation Toolbox) object by providing the waypoints in the Cartesian coordinate frame of the driving scenario. The reference path used in this example defines a path that turns right at the intersection.

```
waypoints = [-110.6 -4.5 0;
             49 -4.5 0;
             55.5 -17.7 -pi/2;
             55.5 -130.6 -pi/2]; % [x y theta]

% Create a reference path using waypoints
refPath = referencePathFrenet(waypoints);

% Visualize the reference path
fig = figure('Units','normalized','Position',[0.1 0.1 0.8 0.8]);
ax = axes(fig);
hold(ax,'on');
plot(scenario,'Parent',ax);
show(refPath,'Parent',ax);
```

```
xlim(ax, [-120 80]);
ylim(ax, [-160 40]);
```



```
snapnow;
```

The local motion planning algorithm in this example consists of three main steps:

- 1 Sample local trajectories
- 2 Find feasible and collision-free trajectories
- 3 Choose optimality criterion and select optimal trajectory

The following sections discuss each step of the local planning algorithm and the helper functions used to execute each step.

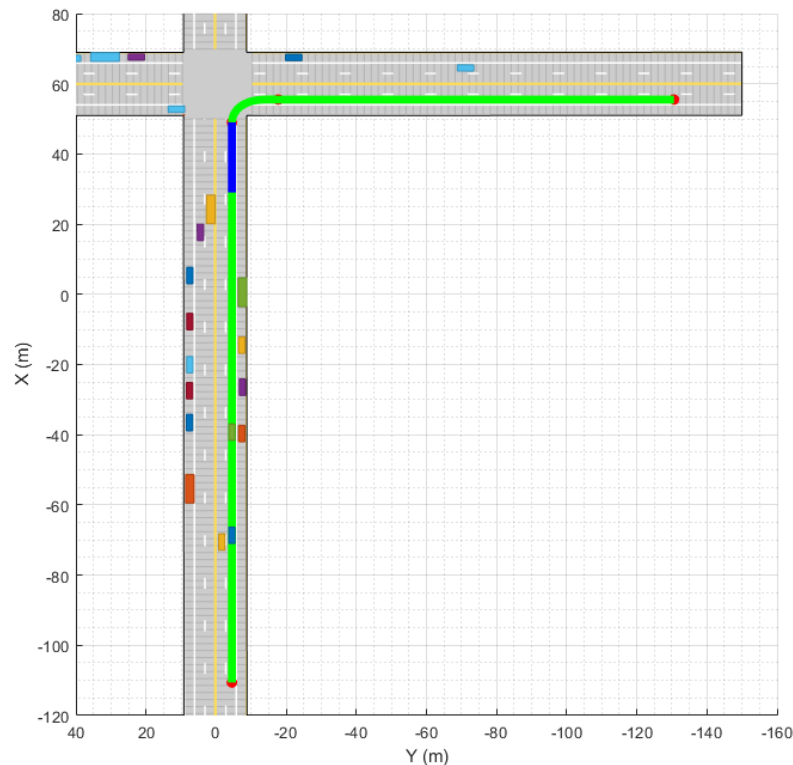
Sample Local Trajectories

At each step of the simulation, the planning algorithm generates a list of sample trajectories that the ego vehicle can choose. The local trajectories are sampled by connecting the current state of the ego vehicle to desired terminal states. Use the `trajectoryGeneratorFrenet` (Navigation Toolbox) object to connect current and terminal states for generating local trajectories. Define the object by providing the reference path and the desired resolution in time for the trajectory. The object connects initial and final states in Frenet coordinates using fifth-order polynomials.

```
connector = trajectoryGeneratorFrenet(refPath, 'TimeResolution', 0.1);
```

The strategy for sampling terminal states in Frenet coordinates often depends on the road network and the desired behavior of the ego vehicle during different phases of the global path. For more detailed examples of using different ego behavior, such as cruise-control and car-following, refer to the "Planning Adaptive Routes Through Traffic" section of the "Highway Trajectory Planning Using Frenet Reference Path" (Navigation Toolbox) example. In this example, you sample the terminal states using two different strategies, depending on the location of vehicle on the reference path, shown as blue and green regions in the following figure.

```
% Visualize path regions for sampling strategy visualization
pathPoints = closestPoint(refPath, refPath.Waypoints(:,1:2));
roadS = pathPoints(:,end);
intersectionS = roadS(2,end);
intersectionBuffer = 20;
pathGreen = [interpolate(refPath,linspace(0,intersectionS-intersectionBuffer,20));...
            nan(1,6);...
            interpolate(refPath,linspace(intersectionS,roadS(end),100))];
pathBlue = interpolate(refPath,linspace(intersectionS-intersectionBuffer,roadS(2,end),20));
hold(ax,'on');
plot(ax,pathGreen(:,1),pathGreen(:,2),'Color',[0 1 0],'LineWidth',5);
plot(ax,pathBlue(:,1),pathBlue(:,2),'Color',[0 0 1],'LineWidth',5);
```



```
snapnow;
```

When the ego vehicle is in the green region, the following strategy is used to sample local trajectories. The terminal state of the ego vehicle after ΔT time is defined as:

$$x_{\text{Ego}}(\Delta T) = [\text{NaN } \dot{s} \ 0 \ d \ 0 \ 0];$$

where discrete samples for variables are obtained using the following predefined sets:

$$\{\Delta T \in \{\text{linspace}(2, 4, 6)\}, \dot{s} \in \{\text{linspace}(0, \dot{s}_{\text{max}}, 10)\}, d \in \{0 \ w_{\text{lane}}\}\}$$

The use of NaN in the terminal state enables the `trajectoryGeneratorFrenet` object to automatically compute the longitudinal distance traveled over a minimum-jerk trajectory. This strategy produces a set of trajectories that enable the ego vehicle to accelerate up to the maximum speed limit (\dot{s}_{max}) rates or decelerate to a full stop at different rates. In addition, the sampled choices of lateral offset (d_{des}) allow the ego vehicle to change lanes during these maneuvers.

```
% Define smax and wlane
speedLimit = 15;
laneWidth = 2.975;
```

When the ego vehicle is in the blue region of the trajectory, the following strategy is used to sample local trajectories:

$$x_{\text{Ego}}(\Delta T) = [s_{\text{stop}} \ 0 \ 0 \ 0 \ 0];$$

where ΔT is chosen to minimize jerk during the trajectory. This strategy enables the vehicle to stop at the desired distance (s_{stop}) in the right lane with a minimum-jerk trajectory. The trajectory sampling algorithm is wrapped inside the helper function, `helperGenerateTrajectory`, attached with this example.

Finding Feasible and Collision-Free Trajectories

The sampling process described in the previous section can produce trajectories that are kinematically infeasible and exceed thresholds of kinematic attributes such as acceleration and curvature. Therefore, you limit the maximum acceleration and speed of the ego vehicle using the helper function `helperKinematicFeasibility` on page 6-689, which checks the feasibility of each trajectory against these kinematic constraints.

```
% Define kinematic constraints
accMax = 15;
```

Further, you set up a collision-validator to assess if the ego vehicle can maneuver on a kinematically feasible trajectory without colliding with any other obstacles in the environment. To define the validator, use the helper class `HelperDynamicMapValidator`. This class uses the `predictMapToTime` function of the `trackerGridRFS` object to get short-term predictions of the occupancy of the surrounding environment. Since the uncertainty in the estimate increases with time, configure the validator with a maximum time horizon of 2 seconds.

The predicted occupancy of the environment is converted to an inflated costmap at each step to account for the size of the ego vehicle. The path planner uses a timestep of 0.1 seconds with a prediction time horizon of 2 seconds. To reduce computational complexity, the occupancy of the surrounding environment is assumed to be valid for 5 time steps, or 0.5 seconds. As a result, only 4 predictions are required in the 2-second planning horizon. In addition to making binary decisions about collision or no collision, the validator also provides a measure of collision probability of the ego vehicle. This probability can be incorporated into the cost function for optimality criteria to account for uncertainty in the system and to make better decisions without increasing the time horizon of the planner.

```

vehDims = vehicleDimensions(egoVehicle.Length,egoVehicle.Width);
collisionValidator = HelperDynamicMapValidator('MaxTimeHorizon',2, ... % Maximum horizon for val.
'TimeResolution',connector.TimeResolution, ... % Time steps between trajectory samples
'Tracker',tracker, ... % Provide tracker for prediction
'ValidPredictionSpan',5, ... % Prediction valid for 5 steps
'VehicleDimensions',vehDims); % Provide dimensions of ego

```

Choose Optimality Criterion

After validating the feasible trajectories against obstacles or occupied regions of the environment, choose an optimality criterion for each valid trajectory by defining a cost function for the trajectories. Different cost functions are expected to produce different behaviors from the ego vehicle. In this example, you define the cost of each trajectory as

$$C = J_s + J_d + 1000P_c + 100(\dot{s}_{(\Delta T)} - \dot{s}_{\text{Limit}})^2$$

where:

J_s is the jerk in the longitudinal direction of the reference path

J_d is the jerk in the lateral direction of the reference path

P_c is the collision probability obtained by the validator

The cost calculation for each trajectory is defined using the helper function `helperCalculateTrajectoryCosts` on page 6-689. From the list of valid trajectories, the trajectory with the minimum cost is considered as the optimal trajectory.

Run Scenario, Estimate Dynamic Map, and Plan Local Trajectories

Run the scenario, generate point clouds from all the lidar sensors, and estimate the dynamic occupancy grid map. Use the dynamic map estimate and its predictions to plan a local trajectory for the ego vehicle.

```

% Close original figure and initialize a new display
close(fig);
display = helperGridBasedPlanningDisplay;

% Initial ego state
currentEgoState = [-110.6 -1.5 0 0 15 0];
helperMoveEgoVehicleToState(egoVehicle, currentEgoState);

% Initialize pointCloud outputs from each sensor
ptClouds = cell(numel(lidars),1);
sensorConfigs = cell(numel(lidars),1);

% Simulation Loop
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, ~, ~, map] = tracker(sensorData,sensorConfigs,time);

    % Update validator's future predictions using current estimate
    step(collisionValidator, currentEgoState, map, time);

    % Sample trajectories using current ego state and some kinematic
    % parameters
    [frenetTrajectories, globalTrajectories] = helperGenerateTrajectory(connector, refPath, currentEgoState, map, sensorData, sensorConfigs, time);

    % Calculate kinematic feasibility of generated trajectories
    isKinematicsFeasible = helperKinematicFeasibility(frenetTrajectories,speedLimit,accMax);

    % Calculate collision validity of feasible trajectories
    feasibleGlobalTrajectories = globalTrajectories(isKinematicsFeasible);
    feasibleFrenetTrajectories = frenetTrajectories(isKinematicsFeasible);
    [isCollisionFree, collisionProb] = isTrajectoryValid(collisionValidator, feasibleGlobalTrajectories, feasibleFrenetTrajectories, currentEgoState, map, sensorData, sensorConfigs, time);

    % Calculate costs and final optimal trajectory
    nonCollidingGlobalTrajectories = feasibleGlobalTrajectories(isCollisionFree);
    nonCollidingFrenetTrajectories = feasibleFrenetTrajectories(isCollisionFree);
    nonCollidingCollisionProb = collisionProb(isCollisionFree);
    costs = helperCalculateTrajectoryCosts(nonCollidingFrenetTrajectories, nonCollidingCollisionProb, currentEgoState, map, sensorData, sensorConfigs, time);

    % Find optimal trajectory
    [~,idx] = min(costs);
    optimalTrajectory = nonCollidingGlobalTrajectories(idx);

    % Assemble for plotting
    trajectories = helperAssembleTrajectoryForPlotting(globalTrajectories, ...
        isKinematicsFeasible, isCollisionFree, idx);

    % Update display
    display(scenario, egoVehicle, lidars, ptClouds, tracker, tracks, trajectories, collisionValidator);

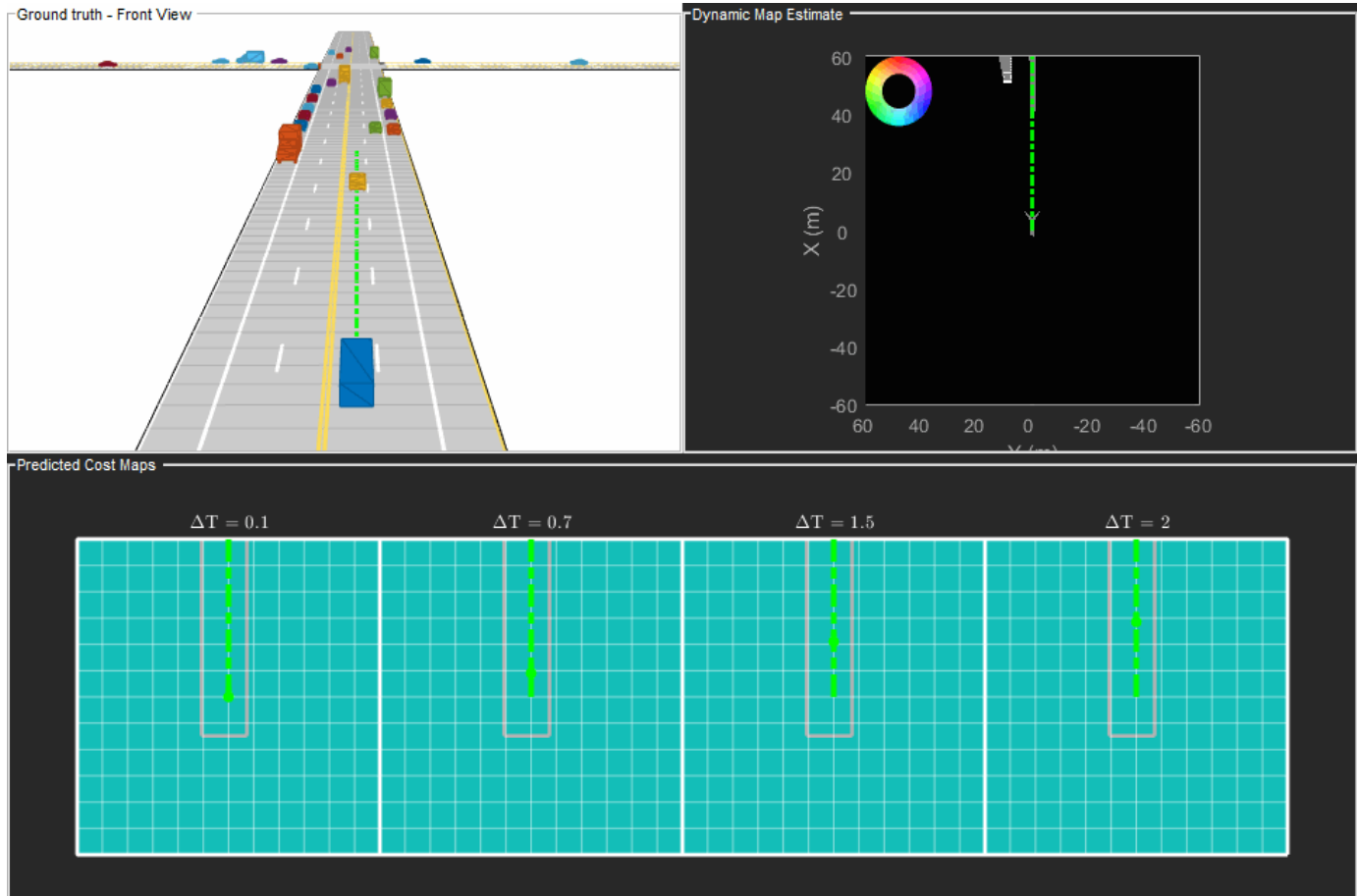
    % Move ego with optimal trajectory
    if ~isempty(optimalTrajectory)
        currentEgoState = optimalTrajectory.Trajectory(2,:);
        helperMoveEgoVehicleToState(egoVehicle, currentEgoState);
    else
        % All trajectories either violated kinematic feasibility
        % constraints or resulted in a collision. More behaviors on
        % trajectory sampling may be needed.
        error('Unable to compute optimal trajectory');
    end
end

```

Results

Analyze the results from the local path planning algorithm and how the predictions from the map assisted the planner. This animation shows the result of the planning algorithm during the entire

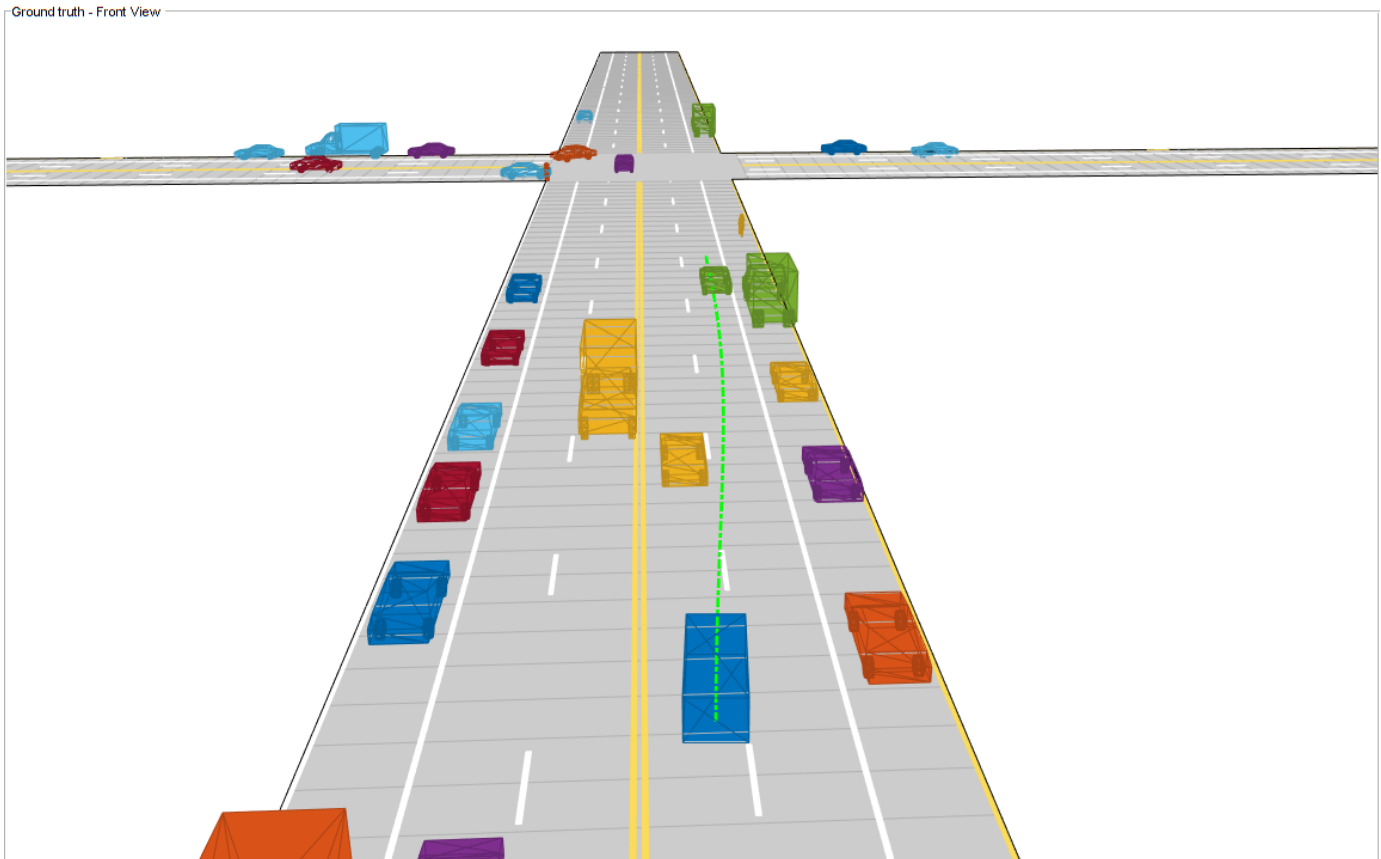
scenario. Notice that the ego vehicle successfully reached its desired destination and maneuvered around different dynamic objects, whenever necessary. The ego vehicle also came to a stop at the intersection due to the regional changes added to the sampling policy.



Next, analyze the local planning algorithm during the first lane change. The snapshots in this section are captured at time = 4.3 seconds during the simulation.

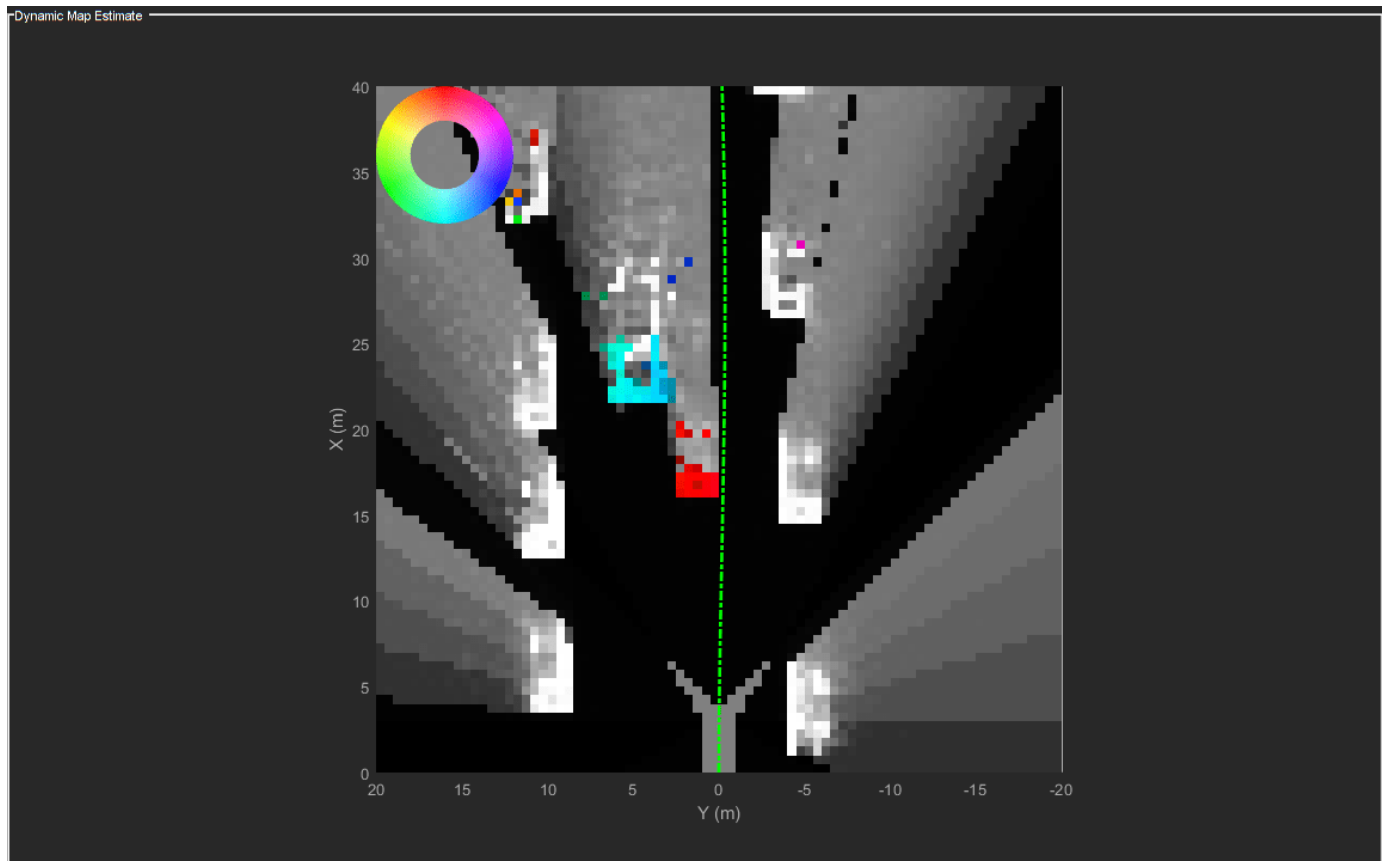
In this snapshot, the ego vehicle has just started to perform a lane change maneuver into the right lane.

```
showSnaps(display, 3, 1);
```

The snapshot that follows shows the estimate of the dynamic grid at the same time step. The color of the grid cell denotes the direction of motion of the object occupying that grid cell. Notice that the cells representing the car in front of the ego vehicle are colored red, denoting that the cells are occupied with a dynamic object. Also, the car is moving in the positive X direction of the scenario, so based on the color wheel, the color of the corresponding grid cells is red.

```
f = showSnaps(display, 2, 1);
if ~isempty(f)
    ax = findall(f, 'Type', 'Axes');
    ax.XLim = [0 40];
    ax.YLim = [-20 20];
    s = findall(ax, 'Type', 'Surf');
    s.XData = 36 + 1/3*(s.XData - mean(s.XData(:)));
    s.YData = 16 + 1/3*(s.YData - mean(s.YData(:)));
end
```



Based on the previous image, the planned trajectory of the ego vehicle passes through the occupied regions of space, representing a collision if you performed a traditional static occupancy validation. The dynamic occupancy map and the validator, however, account for the dynamic nature of the grid by validating the state of the trajectory against the predicted occupancy at each time step. The next snapshot shows the predicted costmap at different prediction steps (ΔT), along with the planned position of the ego vehicle on the trajectory. The predicted costmap is inflated to account for size of the ego vehicle. Therefore, if a point object representing the origin of the ego vehicle can be placed on the occupancy map without any collision, it can be interpreted that the ego vehicle does not collide with any obstacle. The yellow regions on the costmap denote areas with guaranteed collisions with an obstacle. The collision probability decays outside the yellow regions exponentially until the end of inflation region. The blue regions indicate areas with zero probability of collision according to the current prediction.

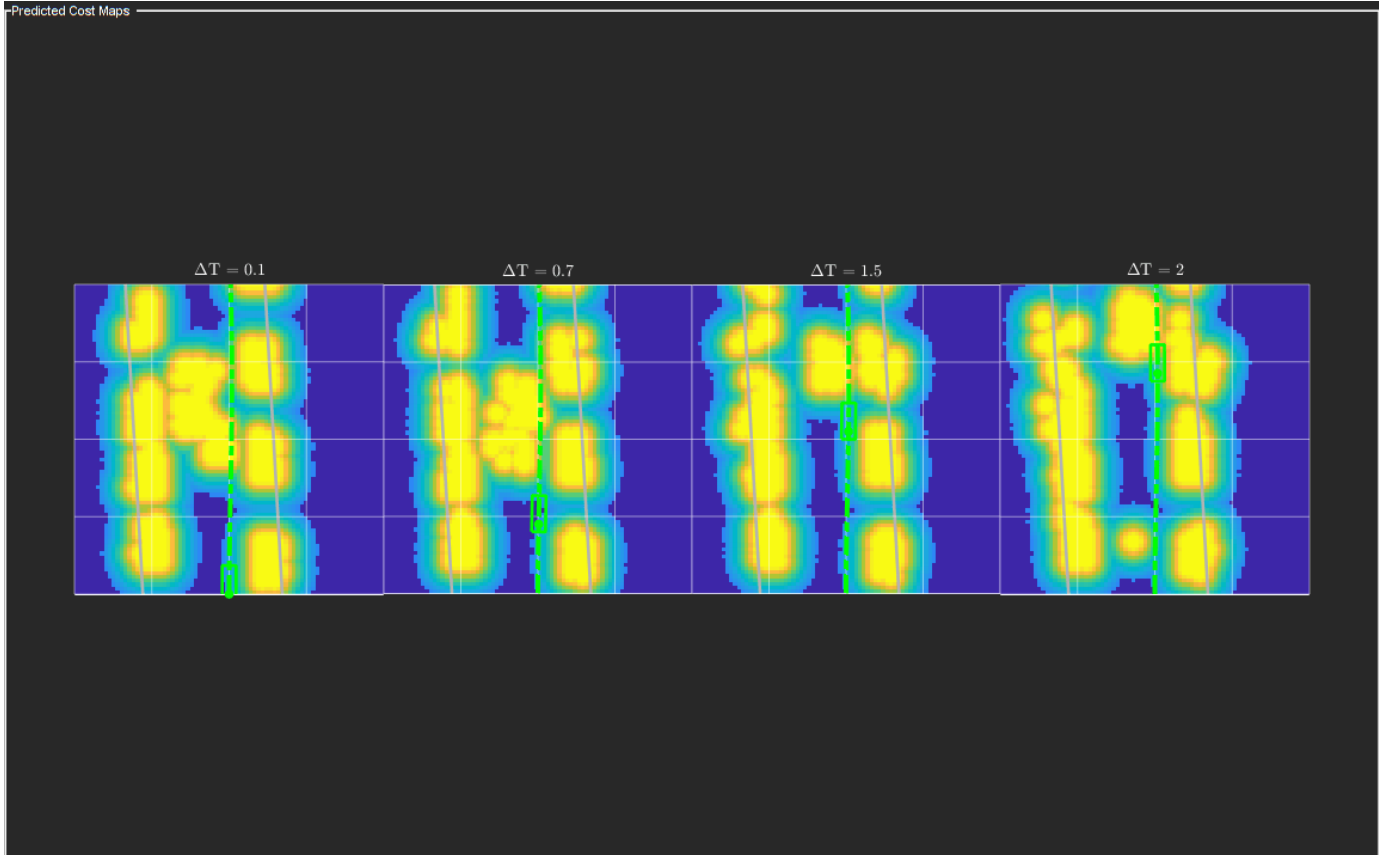
Notice that the yellow region representing the car in front of the ego vehicle moves forward on the costmap as the map is predicted in the future. This reflects that the prediction of occupancy considers the velocity of objects in the surrounding environment. Also, notice that the cells classified as static objects remained relatively static on the grid during the prediction. Lastly, notice that the planned position of the ego vehicle origin does not collide with any occupied regions in the cost map. This shows that the ego vehicle can successfully maneuver on this trajectory.

```
f = showSnaps(display, 1, 1);
if ~isempty(f)
ax = findall(f, 'Type', 'Axes');
for i = 1:numel(ax)
    ax(i).XLim = [0 40];
```

```

    ax(i).YLim = [-20 20];
end
end

```



Summary

In this example, you learned how to use the dynamic map predictions from the grid-based tracker, `trackerGridRFS`, and how to integrate the dynamic map with a local path planning algorithm to generate trajectories for the ego vehicle in dynamic complex environments. You also learned how the dynamic nature of the occupancy can be used to plan trajectories more efficiently in the environment.

Supporting Functions

```

function sensorData = packAsSensorData(ptCloud, configs, time)
% Pack the sensor data as format required by the tracker
%
% ptCloud - cell array of pointCloud object
% configs - cell array of sensor configurations
% time    - Current simulation time

%The lidar simulation returns outputs as pointCloud objects. The Location
%property of the point cloud is used to extract x,y, and z locations of
%returns and pack them as structures with information required by a tracker.
sensorData = struct('SensorIndex', {}, ...
    'Time', {}, ...

```

```

    'Measurement', {},...
    'MeasurementParameters', {});

for i = 1:numel(ptCloud)
    % This sensor's point 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;

    % Extract Measurement as a 3-by-N defining locations of points
    sensorData(i).Measurement = reshape(thisPtCloud.Location,[],3)';

    % Data is reported in the 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)
% Get configuration of the lidar sensor for tracker
%
% config - Configuration of the lidar sensor in the world frame
% lidar - lidarPointCloudGeneration object
% ego    - driving.scenario.Actor in the scenario

% 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'),'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

function helperMoveEgoVehicleToState(egoVehicle, currentEgoState)
% Move ego vehicle in scenario to a state calculated by the planner
%
% egoVehicle - driving.scenario.Actor in the scenario
% currentEgoState - [x y theta kappa speed acc]

```

```

% Set 2-D Position
egoVehicle.Position(1:2) = currentEgoState(1:2);

% Set 2-D Velocity (s*cos(yaw) s*sin(yaw))
egoVehicle.Velocity(1:2) = [cos(currentEgoState(3)) sin(currentEgoState(3))]*currentEgoState(5);

% Set Yaw in degrees
egoVehicle.Yaw = currentEgoState(3)*180/pi;

% Set angular velocity in Z (yaw rate) as v/r
egoVehicle.AngularVelocity(3) = currentEgoState(4)*currentEgoState(5);

end

function isFeasible = helperKinematicFeasibility(frenetTrajectories, speedLimit, aMax)
% Check kinematic feasibility of trajectories
%
% frenetTrajectories - Array of trajectories in Frenet coordinates
% speedLimit - Speed limit (m/s)
% aMax - Maximum acceleration (m/s^2)

isFeasible = false(numel(frenetTrajectories),1);
for i = 1:numel(frenetTrajectories)
    % Speed of the trajectory
    speed = frenetTrajectories(i).Trajectory(:,2);

    % Acceleration of the trajectory
    acc = frenetTrajectories(i).Trajectory(:,3);

    % Is speed valid?
    isSpeedValid = ~any(speed < -0.1 | speed > speedLimit + 1);

    % Is acceleration valid?
    isAccelerationValid = ~any(abs(acc) > aMax);

    % Trajectory feasible if both speed and acc valid
    isFeasible(i) = isSpeedValid & isAccelerationValid;
end

end

function cost = helperCalculateTrajectoryCosts(frenetTrajectories, Pc, smax)
% Calculate cost for each trajectory.
%
% frenetTrajectories - Array of trajectories in Frenet coordinates
% Pc - Probability of collision for each trajectory calculated by validator

n = numel(frenetTrajectories);
Jd = zeros(n,1);
Js = zeros(n,1);
s = zeros(n,1);

for i = 1:n
    % Time
    time = frenetTrajectories(i).Times;

    % resolution

```

```
dT = time(2) - time(1);

% Jerk along the path
dds = frenetTrajectories(i).Trajectory(:,3);
Js(i) = sum(gradient(dds,time).^2)*dT;

% Jerk perpendicular to path
%  $d^2L/dt^2 = d/dt(dL/ds*ds/dt)$ 
ds = frenetTrajectories(i).Trajectory(:,2);
ddL = frenetTrajectories(i).Trajectory(:,6).*(ds.^2) + frenetTrajectories(i).Trajectory(:,5)
Jd(i) = sum(gradient(ddL,time).^2)*dT;

s(i) = frenetTrajectories(i).Trajectory(end,2);
end

cost = Js + Jd + 1000*Pc(:) + 100*(s - smax).^2;

end
```

Track-to-Track Fusion for Automotive Safety Applications

This example shows how to fuse tracks from two vehicles to provide a more comprehensive estimate of the environment than 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 vision detection generator from Automated Driving Toolbox™, the radar data generator from the Radar 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 either by using 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] on page 6-703. 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, V2X protocols and cellular communication links are being developed.

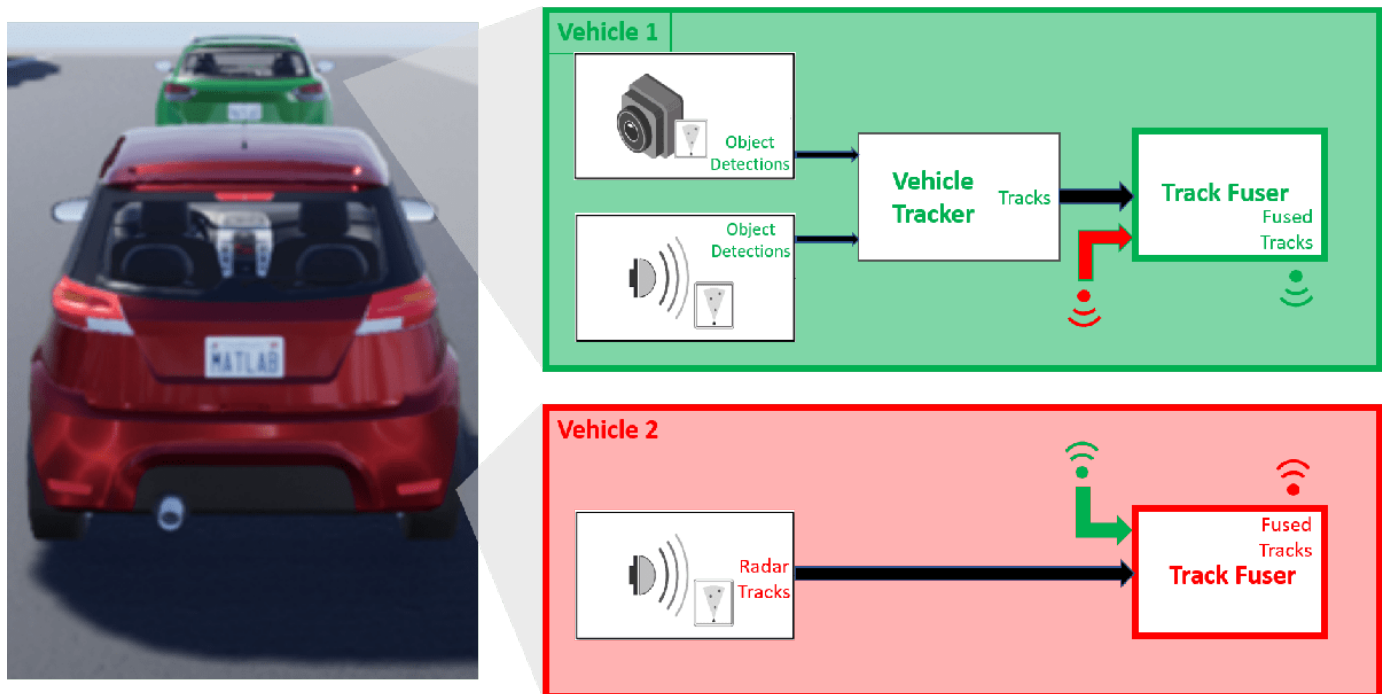
While sensor fusion across multiple vehicles is beneficial, most vehicles are required to meet certain safety requirements even if only internal sensors are available. Therefore, the vehicle is likely to be equipped with a tracker, a track fuser, or both. These tracking algorithms provide situational 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. 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, where:

- Vehicle 1 has two sensors, each providing detections to a local vehicle 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 is a tracking radar. The tracking radar outputs tracks and serves as the local tracker for vehicle 2. The tracks from the tracking radar are inputs to the vehicle 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 the vehicle 1 tracker.

```
% Create the tracker for vehicle 1
v1Tracker = trackerJPDA('TrackerIndex',1, 'DeletionThreshold', [4 4], 'AssignmentThreshold', [100 100], 'posSelector', [1 0 0 0 0 0; 0 0 1 0 0 0]);
```

In this architecture, the fused tracks from one vehicle 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 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 must be aware that the tracks of this object it gets from vehicle 2 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 ignores 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 the ego frame to the scenario frame. Similarly, the function `scenarioToEgo` transforms tracks from the scenario frame to any of the ego

frames. Both transformations rely on the `StateParameters` property of the `objectTrack` objects. 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 previously described `trackFuser` definitions, define the following sources as a `fuserSourceConfiguration` object.

```
% 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.

```
stateParams = struct('Frame','Rectangular','Position',[0 0 0],'Velocity',[0 0 0]);
v1Fuser = trackFuser('FuserIndex',3,...
    'AssignmentThreshold', [100 inf], ...
    'MaxNumSources',2,'SourceConfigurations',v1Sources,...
    'StateFusion','Intersection','DeletionThreshold',[3 3],...
    'StateParameters',stateParams);
v2Fuser = trackFuser('FuserIndex',4,...
    'AssignmentThreshold', [100 inf], ...
    'MaxNumSources',2,'SourceConfigurations',v2Sources,'StateFusion',...
    'Intersection','DeletionThreshold',[3 3],...
    'StateParameters',stateParams);
```

```
% Initialize the following variables
fusedTracks1 = objectTrack.empty(0,1);
fusedTracks2 = objectTrack.empty(0,1);
wasFuser1Updated = false;
wasFuser2Updated = false;
```

Define Scenario

The following scenario shows two vehicles driving down a street. Vehicle 1 is the lead vehicle and is equipped with two forward-looking sensors: a short-range radar sensor 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.

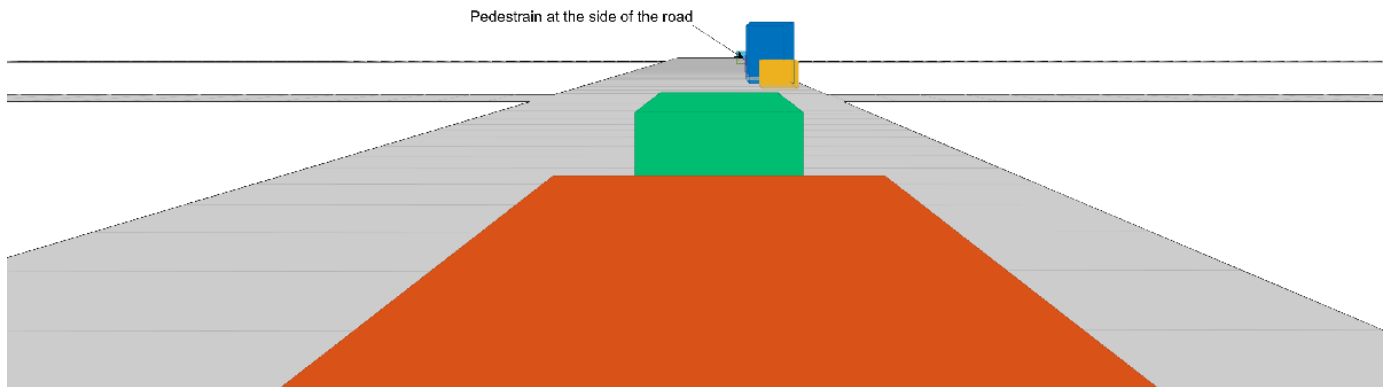
Due to the short distance between vehicle 2 and vehicle 1, most of the vehicle 2 radar sensor coverage is occluded by vehicle 1. As a result, most of the tracks that the track fuser on vehicle 2 maintains are first initialized by tracks broadcast from vehicle 1.

```
% 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] = createV2VDisplay(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.



```
% Define each vehicle as a combination of an actor, sensors, a tracker, and plotters
v1=struct('Actor',{vehicle1},'Sensors',{sensors(attachedVehicle==1)},'Tracker',{v1Tracker},'DetP
v2=struct('Actor',{vehicle2},'Sensors',{sensors(attachedVehicle==2)},'Tracker',{},{},'DetPlotter'
```

Run Simulation

The following code runs the simulation.

```
running = true;

% For repeatable results, set the random number seed
s = rng;
rng(2019)
snaptimes = [0.5, 2.8, 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];
```

```

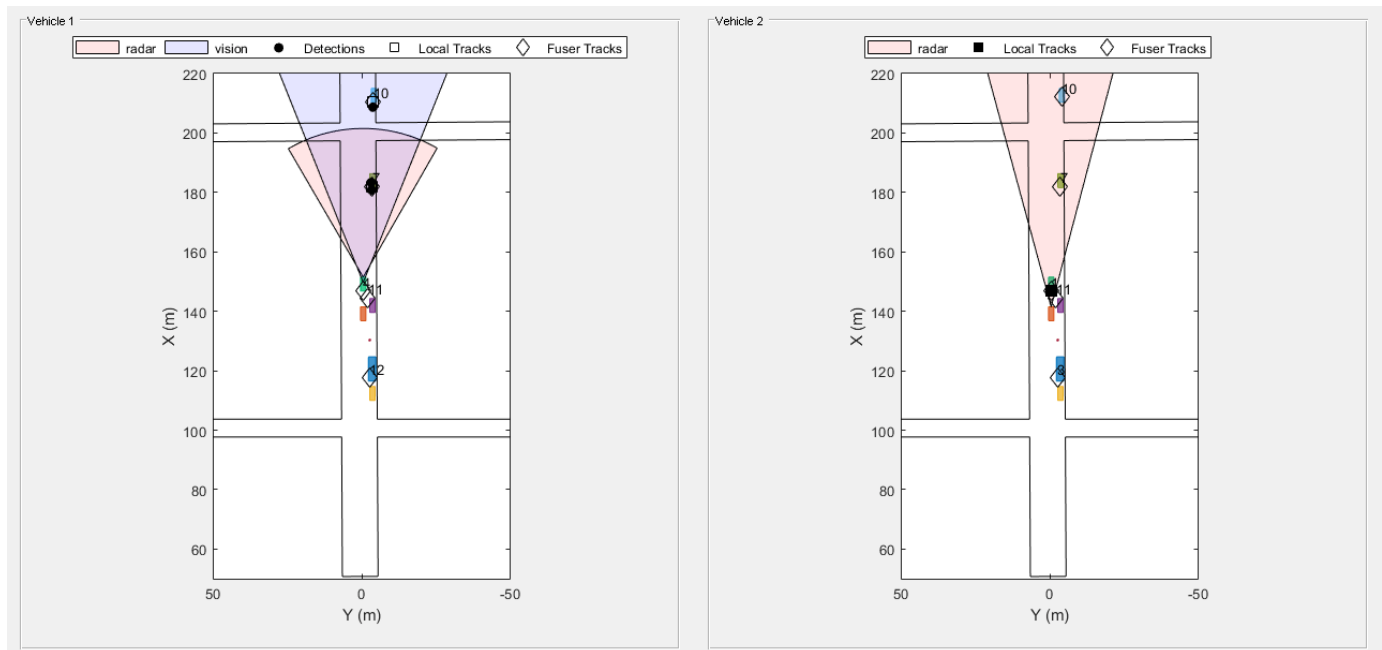
    if isLocked(v1Fuser) || ~isempty(tracksToFuse1)
        [fusedTracks1,~,~,info1] = v1Fuser(tracksToFuse1,time);
        wasFuser1Updated = true;
        pos = getTrackPositions(fusedTracks1,posSelector);
        ids = string([fusedTracks1.TrackID]');
        plotTrack(plotters.veh1FusePlotter,pos,ids);
    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
updateV2VDisplay(plotters, scenario, sensors, attachedVehicle)

% Advance the scenario one time step and exit the loop if the scenario is complete
running = advance(scenario);

% Capture an image of the frame at specified times
if time >= snaptimes(i)
    snaps{i} = takesnap(f);
    i = i + 1;
end
end

```

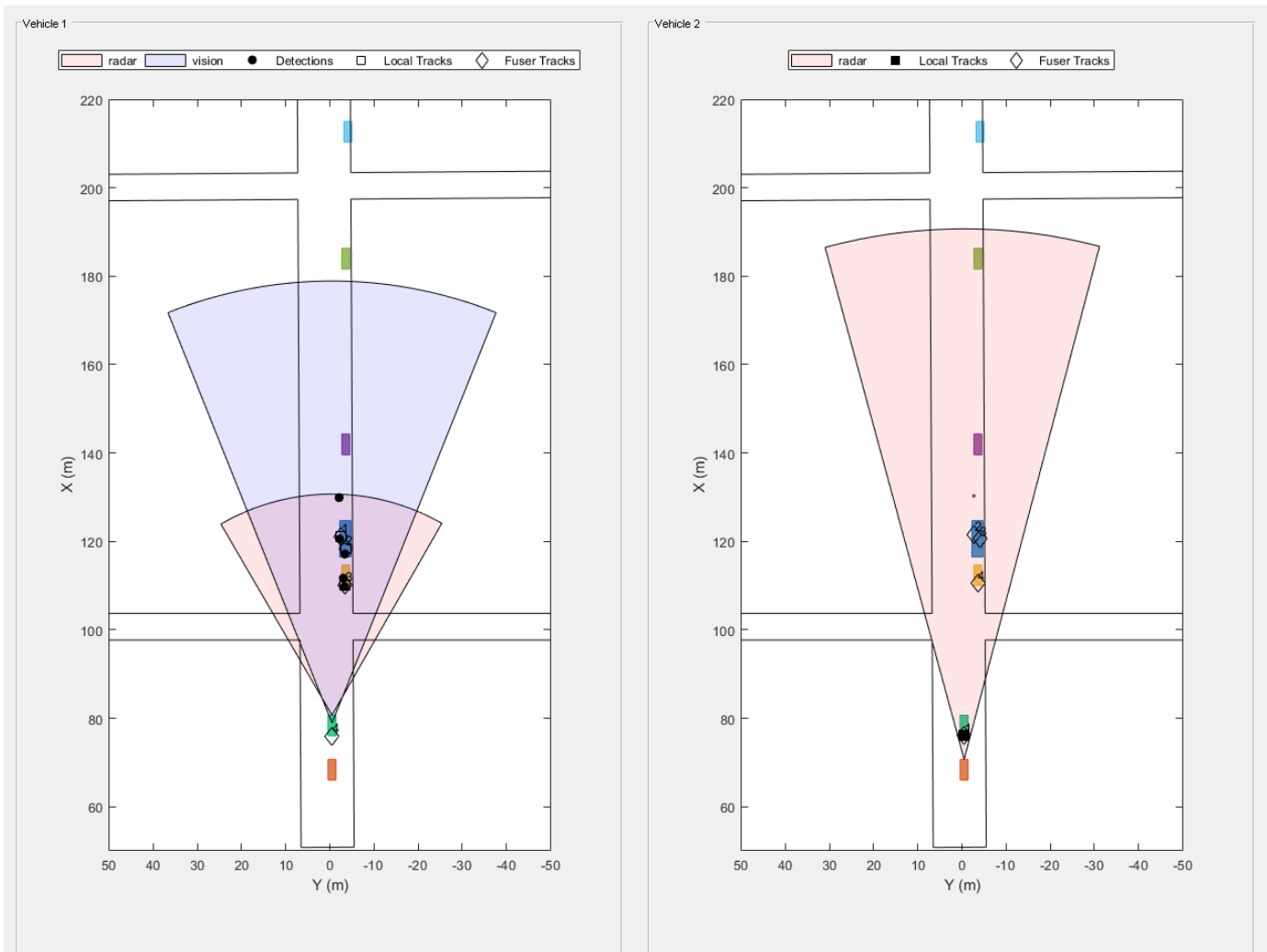


The figure shows the scene and tracking results at the end of the scenario. Subsequent sections of this example analyze the tracking results at key times.

Analyze Tracking at Beginning of Simulation

When the simulation begins, vehicle 1 detects the vehicles parked on the right side of the street. Then, vehicle 1 tracker confirms the tracks associated with the parked vehicles. At this time, the only object detected and tracked by vehicle 2 tracker is vehicle 1, which is 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)
```

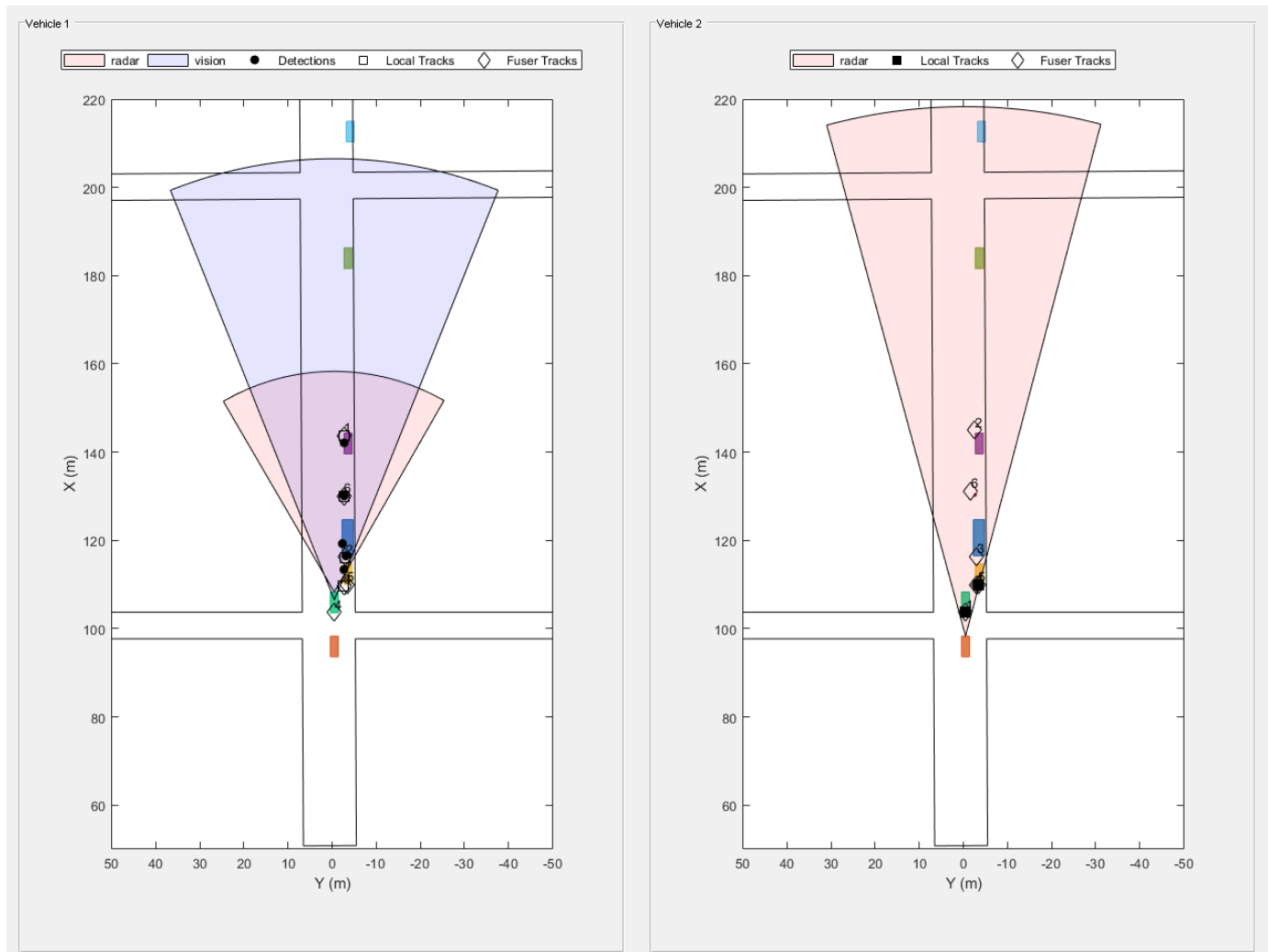


Analyze Tracking of Pedestrian at Side of Street

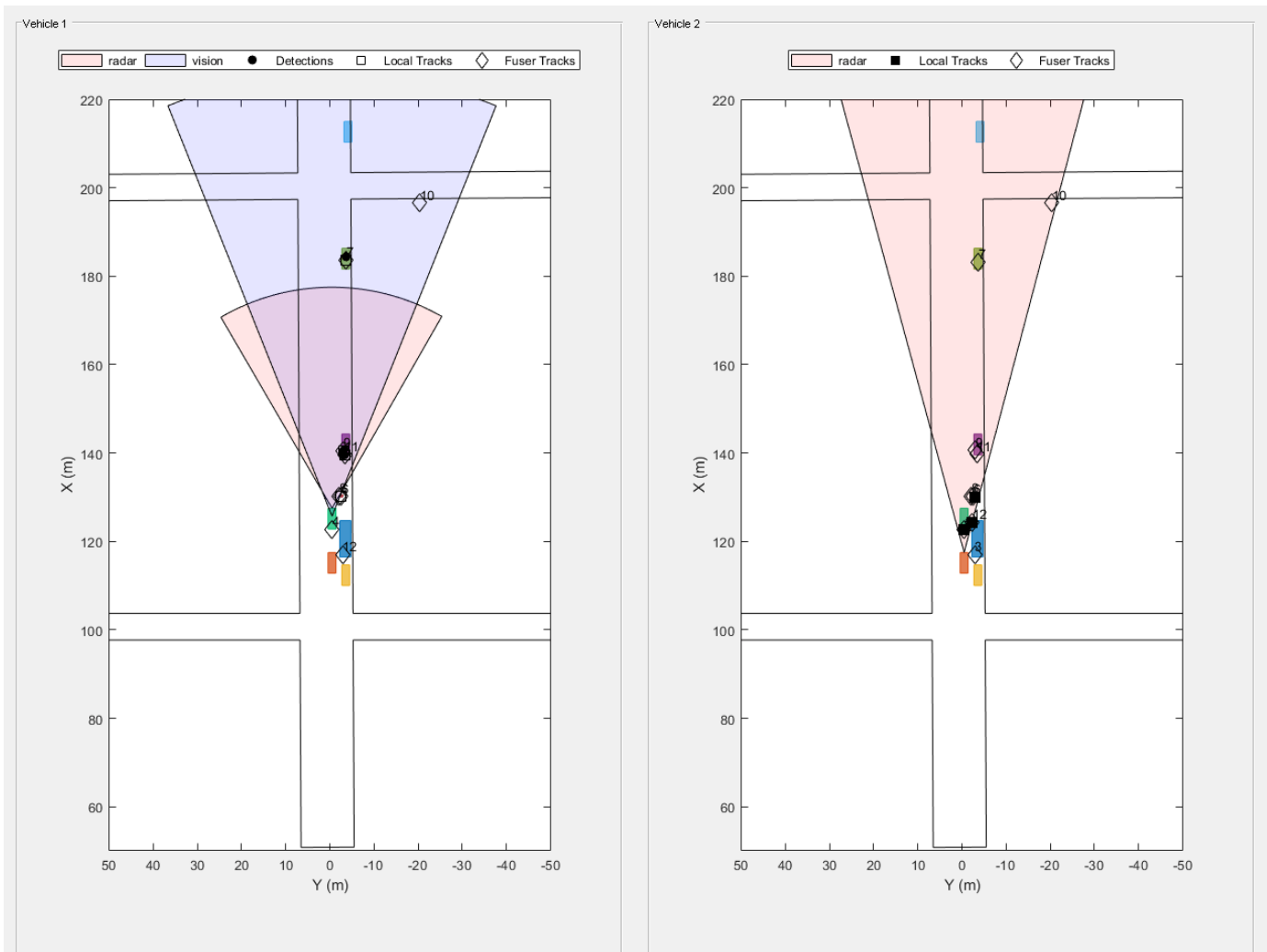
As the simulation continues, vehicle 2 is able to detect and track the vehicles parked at the side as well and fuses them with the tracks coming from vehicle 1. Vehicle 2 is able to detect and track the pedestrian about 4 seconds into the simulation, and vehicle 2 fuses the track associated with the 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). Detecting a pedestrian in the street two seconds earlier can markedly improve safety.

showsnap(snaps, 2)

6 Featured Examples



`showsnap(snaps, 3)`

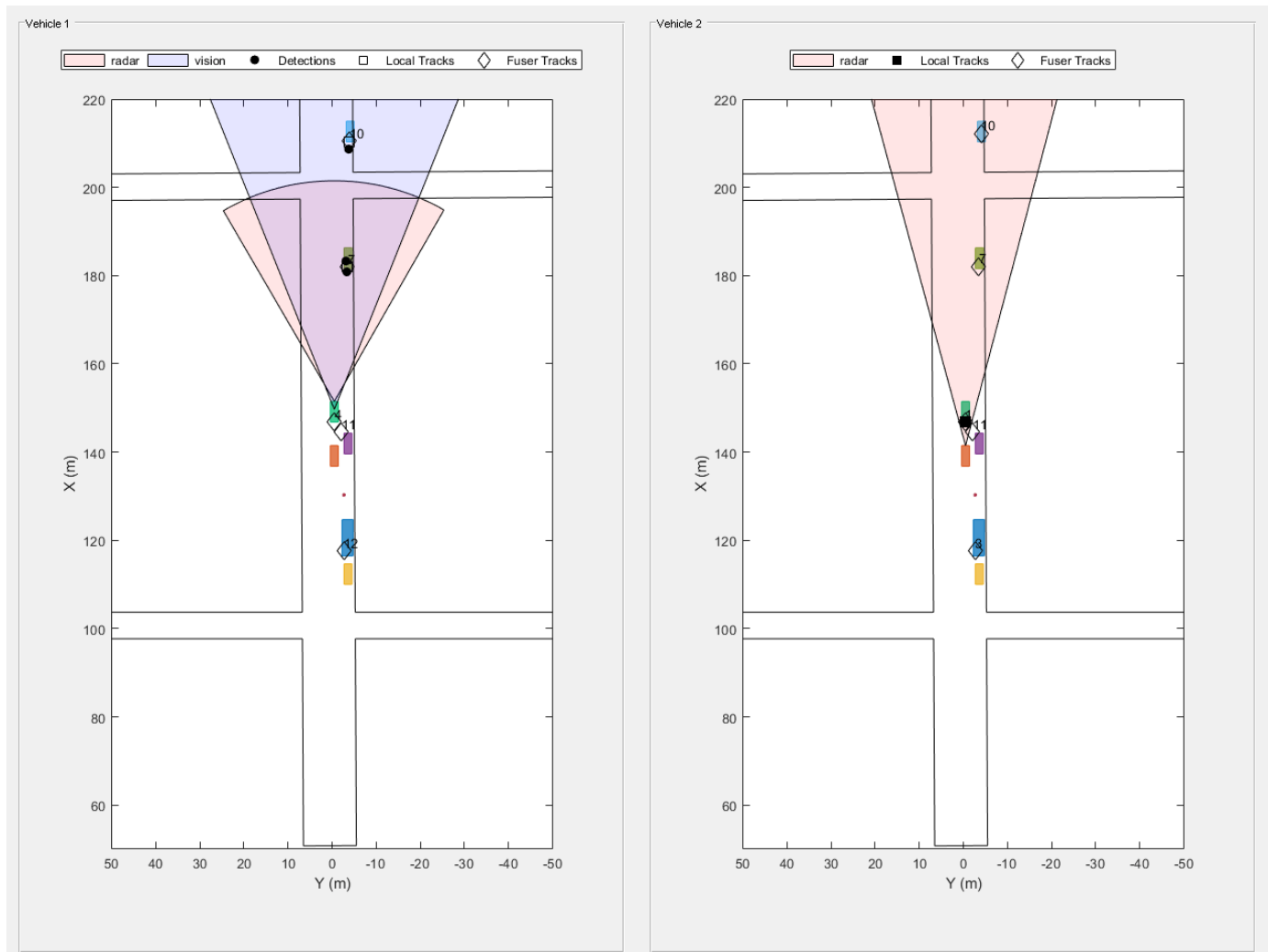


Avoiding Rumor Propagation

When the two vehicles communicate tracks to each other, there is a risk that they will continue communicating information about objects that they do not detect anymore just by repeating what the other vehicle communicated. This situation is called rumor propagation.

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] Duraisamy, B., T. Schwarz, and C. Wohler. "Track Level Fusion Algorithms for Automotive Safety Applications." In *2013 International Conference on Signal Processing, Image Processing & Pattern Recognition*, 179–84, 2013. <https://doi.org/10.1109/ICSIPR.2013.6497983>.

Supporting Functions

createDrivingScenario

Create a driving scenario defined in the **Driving Scenario Designer** app.

```
function [scenario, egoVehicle, secondVehicle] = createDrivingScenario
% Construct a drivingScenario object
scenario = drivingScenario('SampleTime', 0.1);

% 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], 'PlotColor', [0 0.7410 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

Create the sensors used in the scenario and list their attachments to vehicles.

```

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

% Assign into each sensor the physical and radar profiles for all actors
profiles = actorProfiles(scenario);

% Vehicle 1 radar reports clustered detections
sensors{1} = radarDataGenerator('No scanning', 'SensorIndex', 1, 'UpdateRate', 10, ...
    'MountingLocation', [3.7 0 0.2], 'RangeLimits', [0 50], 'FieldOfView', [60 5], ...
    'RangeResolution', 2.5, 'AzimuthResolution', 4, ...
    'Profiles', profiles, 'HasOcclusion', true, 'HasFalseAlarms', false, ...
    'TargetReportFormat', 'Clustered detections');

% Vehicle 2 radar reports tracks
sensors{2} = radarDataGenerator('No scanning', 'SensorIndex', 2, 'UpdateRate', 10, ...
    'MountingLocation', [3.7 0 0.2], 'RangeLimits', [0 120], 'FieldOfView', [30 5], ...
    'RangeResolution', 2.5, 'AzimuthResolution', 4, ...
    'Profiles', profiles, 'HasOcclusion', true, 'HasFalseAlarms', false, ...
    'TargetReportFormat', 'Tracks', 'DeletionThreshold', [3 3]);

% Vehicle 1 vision sensor reports detections
sensors{3} = visionDetectionGenerator('SensorIndex', 3, ...
    'MaxRange', 100, 'SensorLocation', [1.9 0], 'DetectorOutput', 'Objects only', ...
    'ActorProfiles', profiles);
attachedVehicle = [1;2;1];
numSensors = numel(sensors);
end

```

scenarioToEgo

Perform coordinate transformation from scenario to ego coordinates.

trackInScenario has StateParameters defined to transform it from scenario coordinates to ego coordinates.

The state uses the constant velocity model $[x;vx;y;vy;z;vz]$.

```

function trackInEgo = scenarioToEgo(trackInScenario)
egoPosInScenario = trackInScenario.StateParameters.OriginPosition;
egoVelInScenario = trackInScenario.StateParameters.OriginVelocity;
stateInScenario = trackInScenario.State;
stateShift = [egoPosInScenario(1);egoVelInScenario(1);egoPosInScenario(2);egoVelInScenario(2);egoPosInScenario(3);egoVelInScenario(3)];
stateInEgo = stateInScenario - stateShift;
trackInEgo = objectTrack('UpdateTime', trackInScenario.UpdateTime, 'State', stateInEgo, 'StateCovariance', trackInScenario.StateCovariance);
end

```

egoToScenario

Perform coordinate transformation from ego to scenario coordinates.

trackInEgo has StateParameters defined to transform it from ego coordinates to scenario coordinates.

The state uses the constant velocity model $[x;vx;y;vy;z;vz]$.

```
function trackInScenario = egoToScenario(trackInEgo)
egoPosInScenario = trackInEgo.StateParameters.OriginPosition;
egoVelInScenario = trackInEgo.StateParameters.OriginVelocity;
stateInScenario = trackInEgo.State;
stateShift = [egoPosInScenario(1);egoVelInScenario(1);egoPosInScenario(2);egoVelInScenario(2);egoVelInScenario(2);egoVelInScenario(2)];
stateInEgo = stateInScenario + stateShift;
trackInScenario = objectTrack('UpdateTime',trackInEgo.UpdateTime,'State',stateInEgo,'StateCovariance',stateInEgo.StateCovariance);
end
```

detectAndTrack

This function is used for collecting all the detections from the sensors in one vehicle and updating the tracker with them.

The agent is a structure that contains the actor information and the sensors, tracker, and plotter to plot detections and vehicle tracks.

```
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.Detector);

% Update the tracker to get tracks from sensors that reported detections
if isValid
    agent.Tracker.StateParameters = struct(...
        'Frame','Rectangular', ...
        'OriginPosition', agent.Actor.Position, ...
        'OriginVelocity', agent.Actor.Velocity);
    tracks = agent.Tracker(detections,time);
    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

% Get additional tracks from tracking sensors
[sensorTracks,wasSensorTrackerUpdated] = vehicleTracks(agent.Actor,agent.Sensors,poses,time,agent.Detector);
tracks = vertcat(tracks,sensorTracks);
wasTrackerUpdated = wasTrackerUpdated || wasSensorTrackerUpdated;
end
```

vehicleDetections

Collect the detections from all the sensors attached to this vehicle that return detections.

```
function [objectDetections,isValid] = vehicleDetections(position, sensors, poses, time, plotter)
numSensors = numel(sensors);
objectDetections = {};
isValidTime = false(1, numSensors);
```

```

% Generate detections for each sensor
for sensorIndex = 1:numSensors
    sensor = sensors{sensorIndex};
    if isa(sensor, 'visionDetectionGenerator') || ~strcmpi(sensor.TargetReportFormat, 'Tracks')
        [objectDets, ~, sensorConfig] = sensor(poses, time);
        if islogical(sensorConfig)
            isValidTime(sensorIndex) = sensorConfig;
        else
            isValidTime(sensorIndex) = sensorConfig.IsValidTime;
        end
        objectDets = cellfun(@(d) setAtt(d), objectDets, 'UniformOutput', false);
        numObjects = numel(objectDets);
        objectDetections = [objectDetections; objectDets(1:numObjects)]; %#ok<AGROW>
    end
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 a structure
d.ObjectAttributes = struct;
% Keep only the position measurement and remove velocity
if numel(d.Measurement)==6
    d.Measurement = d.Measurement(1:3);
    d.MeasurementNoise = d.MeasurementNoise(1:3,1:3);
    d.MeasurementParameters{1}.HasVelocity = false;
end
end

```

vehicleTracks

Collect all the tracks from sensors that report tracks on the vehicle.

```

function [tracks,wasTrackerUpdated] = vehicleTracks(actor, sensors, poses, time, plotter)
% Create detections from the vehicle
numSensors = numel(sensors);
tracks = objectTrack.empty;
isValidTime = false(1, numSensors);

% Generate detections for each sensor
for sensorIndex = 1:numSensors
    sensor = sensors{sensorIndex};
    if isa(sensor, 'radarDataGenerator') && strcmpi(sensor.TargetReportFormat, 'Tracks')
        [sensorTracks, ~, sensorConfig] = sensor(poses, time);
        if islogical(sensorConfig)
            isValidTime(sensorIndex) = sensorConfig;
        else
            isValidTime(sensorIndex) = sensorConfig.IsValidTime;
        end
        numObjects = numel(sensorTracks);
        tracks = [tracks; sensorTracks(1:numObjects)]; %#ok<AGROW>
    end
end

```

```
        end
    end
    wasTrackerUpdated = any(isValidTime);

    if ~wasTrackerUpdated % No vehicle tracking sensor updated
        return
    end

    % Add vehicle position and velocity to track state parameters
    for i = 1:numel(tracks)
        tracks(i).StateParameters.OriginPosition = tracks(i).StateParameters.OriginPosition + actor.V
        tracks(i).StateParameters.OriginVelocity = tracks(i).StateParameters.OriginVelocity + actor.V
    end

    % Plot tracks
    if numel(tracks)>0
        trPos = arrayfun(@(t)t.State([1,3]), tracks, 'UniformOutput', false);
        trPos = cell2mat(trPos') + actor.Position(1:2);
        plotTrack(plotter, trPos);
    end
end
```

Detect and Track LEO Satellite Constellation with Ground Radars

This example shows how to import a Two-Line Element (TLE) file of a satellite constellation, simulate radar detections of the constellation, and track the constellation.

The task of populating and maintaining a catalog of space objects orbiting Earth is crucial in space surveillance. This task consists of several processes: detecting and identifying new objects and adding them to the catalog, updating known objects orbits in the catalog, tracking orbit changes throughout their lifetime, and predicting reentries in the atmosphere. In this example, we are study how to detect and track new satellites and add them to a catalog.

To guarantee safe operations in space and prevent collisions with other satellites or known debris, it important to correctly detect and track newly launched satellites. Space agencies typically share prelaunch information, which can be used to select a search strategy. A Low Earth Orbit (LEO) satellite search strategy consisting of fence-type radar systems is commonly used. A fence-type radar system searches a finite volume in space and detects satellites as they pass through its field of view. This strategy can detect and track newly launched constellation quickly. [1]

Importing a satellite constellation from a TLE file

Two-Line Element sets are a common data format to save orbital information of satellites. You can use the `satelliteScenario` object to import satellite orbits defined in a TLE file. By default, the imported satellite orbits are propagated using the SGP4 orbit propagation algorithm which provides good accuracy for LEO objects. In this example, these orbits provide with the ground truth to test the radar tracking system capability to detect newly launched satellites.

```
% Create a satellite scenario
satscene = satelliteScenario;
% Add satellites from TLE file.
tleFile = "leoSatelliteConstellation.tle";
constellation = satellite(satscene, tleFile);
```

Use the satellite scenario viewer to visualize the constellation.

```
play(satscene);
```

Simulating synthetic detections and track constellation

Modeling space surveillance radars

Define two stations with fan-shaped radar beams looking into space. The fans cut through the satellite orbits to maximize the number of detections. The radar stations located on North America form an East-West fence.

```
% First station coordinates in LLA
station1 = [48 -80 0];

% Second station coordinates in LLA
station2 = [50 -117 0];
```

Each station is equipped with a radar, which is modeled by using a `fusionRadarSensor` object. In order to detect satellites 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 field of view of 120 degrees in azimuth and 30 degrees in elevation
- Looking up into space

```

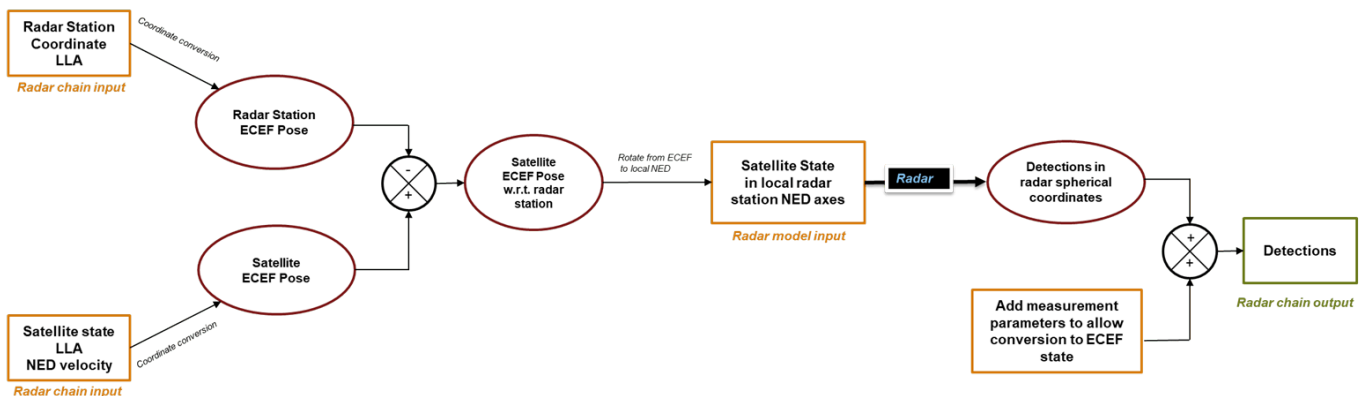
% Create fan-shaped monostatic radars
fov = [120;40];
radar1 = fusionRadarSensor(1,...
    'UpdateRate',0.1,... 10 sec
    'ScanMode','No scanning',...
    'MountingAngles',[0 90 0],... look up
    'FieldOfView',fov,... degrees
    'ReferenceRange',2000e3,... m
    'RangeLimits', [0 2000e3], ... m
    'ReferenceRCS', 10,... dBsm
    'HasFalseAlarms',false,...
    'HasNoise', true,...
    'HasElevation',true,...
    'AzimuthResolution',0.03,... degrees
    'ElevationResolution',0.03,... degrees
    'RangeResolution',2000, ... m % accuracy ~= 2000 * 0.05 (m)
    'DetectionCoordinates','Sensor Spherical',...
    'TargetReportFormat','Detections');
    
```

```

radar2 = clone(radar1);
radar2.SensorIndex = 2;
    
```

Radar Processing Chain

In this example, several coordinate conversions and axes transformation are performed to properly run the radar tracking chain. The diagram below illustrates how the inputs defined above are transformed and passed to the radar.



In the first step, you calculate each satellite pose in the local radar station NED axes. You achieve this by first obtaining the ground station ECEF pose and converting the satellite position and velocity to the ECEF coordinates. The radar input is obtained by taking the differences between the satellite pose and the ground station pose and rotating the differences into ground station local NED axes. See the **assembleRadarInputs** supporting function for the implementation details.

In the second step, you add the required information to the detection object so that the tracker can operate with an ECEF state. You use the `MeasurementParameters` property in each object detection to achieve that purpose, as shown in the `addMeasurementParams` supporting function.

Defining a tracker

The radar models you defined above output detections. To estimate the satellite orbits, you use a tracker. The Sensor Fusion and Tracking Toolbox™ provides a variety of multi-object trackers. In this example, you choose a Joint Probabilistic Data Association (JPDA) tracker because it offers a good balance of tracking performance and computational cost.

You need to define a tracking filter for the tracker. You can use a lower fidelity model than SGP4, such as a Keplerian integration of the equation of motion, to track the satellite. Often, the lack of fidelity in the motion model of targets is compensated by measurement updates and incorporating process noise in the filter. The supporting function `initKeplerUKF` defines the tracking filter.

```
% Define the tracker
tracker = trackerJPDA('FilterInitializationFcn',@initKeplerUKF,...
    'HasDetectableTrackIDsInput',true,...
    'ClutterDensity',1e-40,...
    'AssignmentThreshold',1e4,...
    'DeletionThreshold',[5 8],...
    'ConfirmationThreshold',[5 8]);
```

Running the simulation

In the remainder of this example, you step through the scenario to simulate radar detections and track satellites. This section uses the `trackingGlobeViewer` for visualization. You use this class to display sensor and tracking data with uncertainty ellipses and show the true position of each satellite.

```
viewer = trackingGlobeViewer('ShowDroppedTracks',false, 'PlatformHistoryDepth',700);

% Define coverage configuration of each radar and visualize it on the globe
ned1 = dcmecef2ned(station1(1), station1(2));
ned2 = dcmecef2ned(station2(1), station2(2));
covcon(1) = coverageConfig(radar1,lla2ecef(station1),quaternion(ned1,'rotmat','frame'));
covcon(2) = coverageConfig(radar2,lla2ecef(station2),quaternion(ned2, 'rotmat', 'frame'));
plotCoverage(viewer, covcon, 'ECEF');
```

You first generate the entire history of the states of the constellation over 5 hours. Then, you simulate radar detections and generate tracks in a loop.

```
satscene.StopTime = satscene.StartTime + hours(5);
satscene.SampleTime = 10;
numSteps = ceil(seconds(satscene.StopTime - satscene.StartTime)/satscene.SampleTime);

% Get constellation positions and velocity over the course of the simulation
plats = repmat(...
    struct('PlatformID',0,'Position',[0 0 0], 'Velocity', [0 0 0]),...
    numSteps, 40);
for i=1:numel(constellation)
    [pos, vel] = states(constellation(i),'CoordinateFrame','ECEF');
    for j=1:numSteps
        plats(j,i).Position = pos(:,j)';
        plats(j,i).Velocity = vel(:,j)';
        plats(j,i).PlatformID = i;
    end
end
```



```

end

% Initialize tracks and tracks log
confTracks = objectTrack.empty(0,1);
trackLog = cell(1,numSteps);

% Initialize radar plots
radarplt = helperRadarPlot(fov);

% Set random seed for reproducible results
s = rng;
rng(2020);
step = 0;
while step < numSteps
    time = step*satscene.SampleTime;
    step = step + 1;

    % Generate detections of the constellation following the radar
    % processing chain
    targets1 = assembleRadarInputs(station1, plats(step,:));
    [dets1,numdets1] = radar1(targets1, time);
    dets1 = addMeasurementParams(dets1,numdets1,station1);

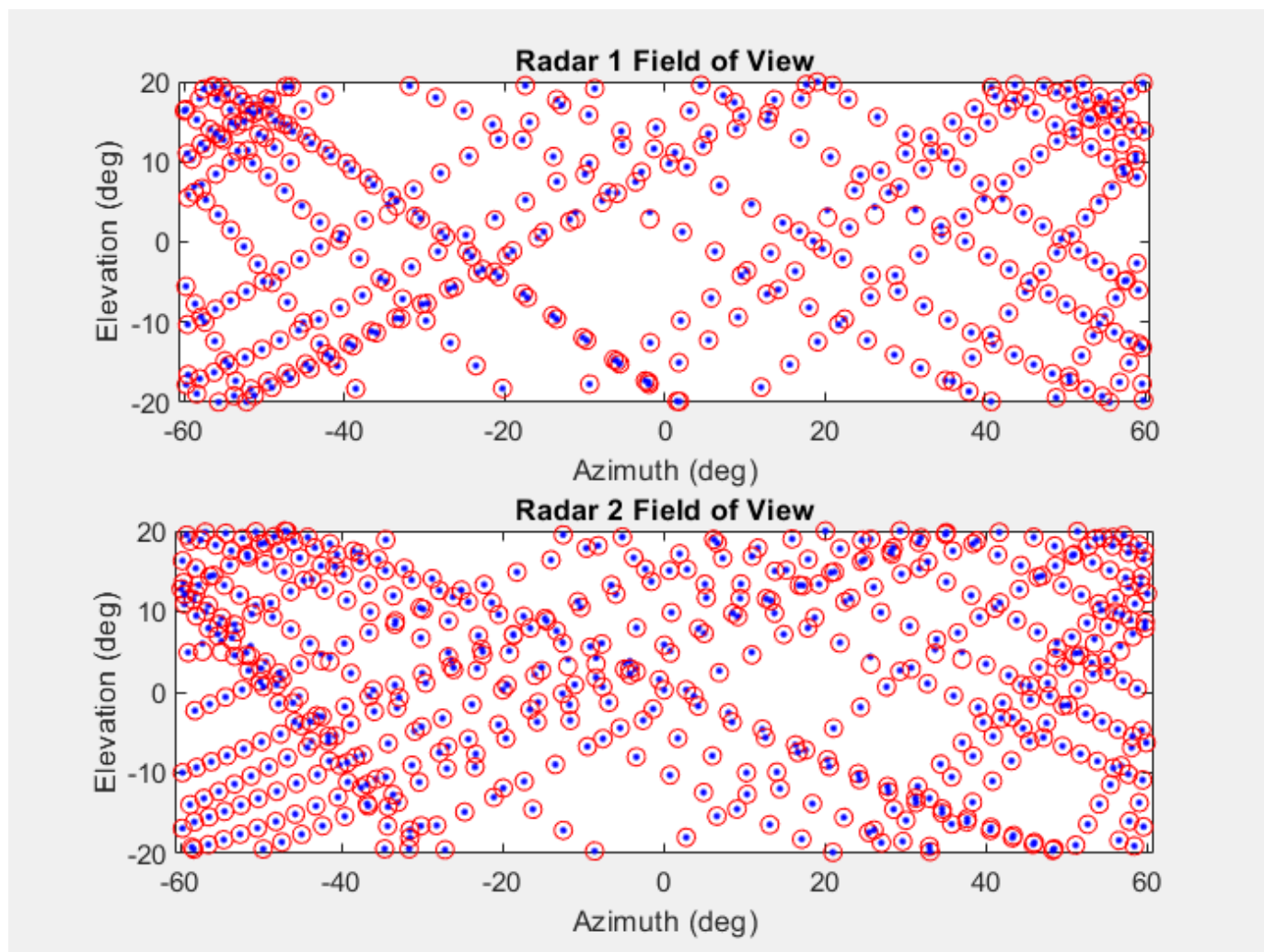
    targets2 = assembleRadarInputs(station2, plats(step,:));
    [dets2, numdets2] = radar2(targets2, time);
    dets2 = addMeasurementParams(dets2, numdets2, station2);

    detections = [dets1; dets2];
    updateRadarPlots(radarplt,targets1, targets2 ,dets1, dets2)

    % Generate and update tracks
    detectableInput = isDetectable(tracker,time, covcon);
    if ~isempty(detections) || isLocked(tracker)
        [confTracks,~,~,info] = tracker(detections,time,detectableInput);
    end
    trackLog{step} = confTracks;

    % Update viewer
    plotPlatform(viewer, plats(step,:), 'ECEF', 'Color',[1 0 0], 'LineWidth',1);
    plotDetection(viewer, detections, 'ECEF');
    plotTrack(viewer, confTracks, 'ECEF', 'Color',[0 1 0], 'LineWidth',3);
end

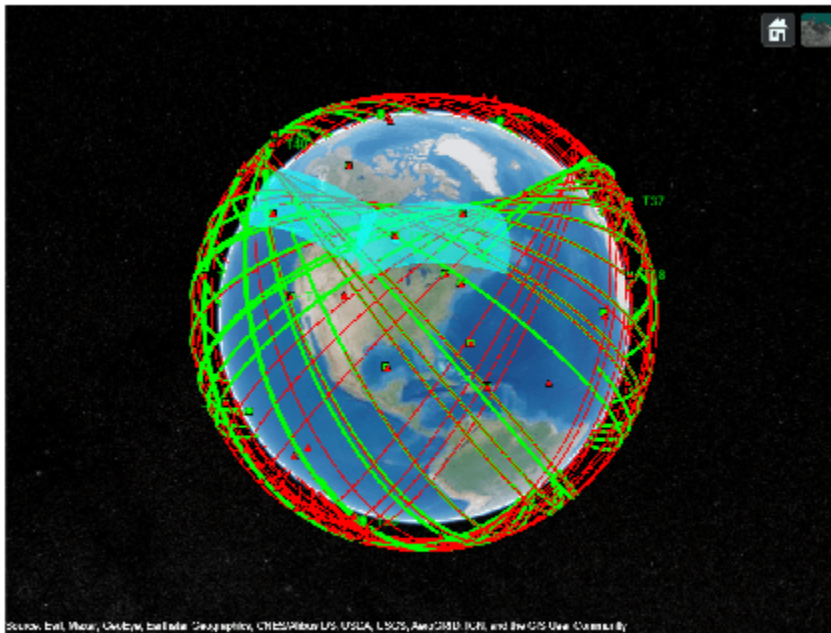
```



The plot above shows the orbits (blue dots) and the detections (red circles) from the point of view of each radar.

```
% Restore previous random seed state  
rng(s);
```

```
figure;  
snapshot(viewer);
```



After 5 hours of tracking, about half the constellation is tracked successfully. Maintaining tracks with a partial orbit coverage is challenging since satellites can often stay undetected for long periods of time in this configuration. In this example, there are only two radar stations. Additional tracking stations are expected to generate better tracking performance. The assignment metrics, which evaluate the tracking performance by comparing between the true objects and the tracks, are shown below.

```
% Show Assignment metrics
truthIdFcn = @(x)[x.PlatformID];

tam = trackAssignmentMetrics('DistanceFunctionFormat','custom',...
    'AssignmentDistanceFcn',@distanceFcn,...
    'DivergenceDistanceFcn',@distanceFcn,...
    'TruthIdentifierFcn',truthIdFcn,...
    'AssignmentThreshold',1000,...
    'DivergenceThreshold',2000);

for i=1:numSteps
    % Extract the tracker and ground truth at the i-th tracker update
    tracks = trackLog{i};
    truths = plats(i,:);
    % Extract summary of assignment metrics against tracks and truths
    [trackAM,truthAM] = tam(tracks, truths);
end

% Show cumulative metrics for each individual recorded truth object
results = truthMetricsTable(tam);
results(:,{'TruthID','AssociatedTrackID','BreakLength','EstablishmentLength'})

ans=40x4 table
    TruthID    AssociatedTrackID    BreakLength    EstablishmentLength
```

1	52	5	232
2	24	0	594
3	4	0	56
4	55	11	492
5	18	0	437
6	48	8	811
7	21	0	513
8	27	0	661
9	39	0	1221
10	50	0	1504
11	43	0	1339
12	37	0	1056
13	NaN	0	1800
14	NaN	0	1800
15	NaN	0	1800
16	NaN	0	1800
:			

The table above lists 40 satellites in the launched constellation and shows the tracked satellites with associated track IDs. A track ID of value NaN indicates that the satellite is not tracked by the end of the simulation. This either means that the orbit of the satellite did not pass through the field of view of one of the two radars or the track of the satellite has been dropped. The tracker can drop a track due to an insufficient number of initial detections, which leads to a large uncertainty on the estimate. Alternately, the tracker can drop the track if the satellite is not re-detected soon enough, such that the lack of updates leads to divergence and eventually deletion.

Summary

In this example, you have learned how to use the `satelliteScenario` object from the Aerospace Toolbox™ to import orbit information from TLE files. You propagated the satellite trajectories using SGP4 and visualized the scenario using the Satellite Scenario Viewer. You learned how to use the radar and tracker models from the Sensor Fusion and Tracking Toolbox™ to model a space surveillance radar tracking system. The constructed tracking system can predict the estimated orbit of each satellite using a low fidelity model.

Supporting functions

`initKeplerUKF` initializes an Unscented Kalman filter using the Keplerian motion model. Set `Alpha = 1`, `Beta = 0`, and `Kappa = 0` to ensure robustness of the unscented filter over long prediction period.

```
function filter = initKeplerUKF(detection)

radarsphmeas = detection.Measurement;
[x, y, z] = sph2cart(deg2rad(radarsphmeas(1)),deg2rad(radarsphmeas(2)),radarsphmeas(3));
radarcartmeas = [x y z];
Recef2radar = detection.MeasurementParameters.Orientation;
ecefmeas = detection.MeasurementParameters.OriginPosition + radarcartmeas*Recef2radar;
% This is equivalent to:
% Ry90 = [0 0 -1 ; 0 1 0; 1 0 0]; % frame rotation of 90 deg around y axis
% nedmeas(:) = Ry90' * radarcartmeas(:);
% ecefmeas = lla2ecef(lla) + nedmeas * dcmecef2ned(lla(1),lla(2));
initState = [ecefmeas(1); 0; ecefmeas(2); 0; ecefmeas(3); 0];
```

```

sigpos = 2;% m
sigvel = 0.5;% m/s^2

filter = trackingUKF(@keplerorbit,@cvmeas,initState,...
    'Alpha', 1, 'Beta', 0, 'Kappa', 0, ...
    'StateCovariance', diag(repmat([1000, 10000].^2,1,3)),...
    'ProcessNoise',diag(repmat([sigpos, sigvel].^2,1,3)));

end

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 4th order 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

```

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.FieldOfView(2)
            isDetectable(i,j) = true;
        else
            isDetectable(i,j) = false;
        end
    end
end
end

detectInput = alltrackid(any(isDetectable,2))';
end
end

```

assembleRadarInput is used to construct the constellation poses in each radar body frame. This is the first step described in the diagram.

```

function targets = assembleRadarInputs(station, constellation)
% For each satellite in the constellation, derive its pose with respect to
% the radar frame.
% inputs:
%     - station      : LLA vector of the radar ground station
%     - constellation : Array of structs containing the ECEF position
%                       and ECEF velocity of each satellite
% outputs:
%     - targets      : Array of structs containing the pose of each
%                       satellite with respect to the radar, expressed in the local
%                       ground radar frame (NED)

% Template structure for the outputs which contains all the field required
% by the radar step method
targetTemplate = struct( ...
    'PlatformID', 0, ...
    'ClassID', 0, ...
    'Position', zeros(1,3), ...
    'Velocity', zeros(1,3), ...
    'Acceleration', zeros(1,3), ...
    'Orientation', quaternion(1,0,0,0), ...
    'AngularVelocity', zeros(1,3), ...
    'Dimensions', struct( ...
        'Length', 0, ...
        'Width', 0, ...
        'Height', 0, ...
        'OriginOffset', [0 0 0]), ...
    'Signatures', {{rcsSignature}});

% First fill in the current satellite ECEF pose
targetPoses = repmat(targetTemplate, 1, numel(constellation));

```

```

for i=1:numel(constellation)
    targetPoses(i).Position = constellation(i).Position;
    targetPoses(i).Velocity = constellation(i).Velocity;
    targetPoses(i).PlatformID = constellation(i).PlatformID;
    % Orientation and angular velocity are left null, assuming satellite to
    % be point targets with a uniform rcs
end

% Then derive the radar pose in ECEF based on the ground station location
Recef2station = dcmecef2ned(station(1), station(2));
radarPose.Orientation = quaternion(Recef2station, 'rotmat', 'frame');
radarPose.Position = lla2ecef(station);
radarPose.Velocity = zeros(1,3);
radarPose.AngularVelocity = zeros(1,3);

% Finally, take the difference and rotate each vector to the ground station
% NED axes
targets = targetPoses;
for i=1: numel(targetPoses)
    thisTgt = targetPoses(i);
    pos = Recef2station*(thisTgt.Position(:) - radarPose.Position(:));
    vel = Recef2station*(thisTgt.Velocity(:) - radarPose.Velocity(:)) - cross(radarPose.AngularV
    angVel = thisTgt.AngularVelocity(:) - radarPose.AngularVelocity(:);
    orient = radarPose.Orientation' * thisTgt.Orientation;

    % Store into target structure array
    targets(i).Position(:) = pos;
    targets(i).Velocity(:) = vel;
    targets(i).AngularVelocity(:) = angVel;
    targets(i).Orientation = orient;
end
end

```

addMeasurementParam implements step 2 in the radar chain process as described in the diagram.

```

function dets = addMeasurementParams(dets, numdets, station)
% Add radar station information to the measurement parameters
Recef2station = dcmecef2ned(station(1), station(2));
for i=1:numdets
    dets{i}.MeasurementParameters.OriginPosition = lla2ecef(station);
    dets{i}.MeasurementParameters.IsParentToChild = true; % parent = ecef, child = radar
    dets{i}.MeasurementParameters.Orientation = dets{i}.MeasurementParameters.Orientation' * Recef2
end
end

```

distanceFcn is used with the assignment metrics to evaluate the tracking assignment.

```

function d = distanceFcn(track, truth)

estimate = track.State([1 3 5 2 4 6]);
true = [truth.Position(:) ; truth.Velocity(:)];
cov = track.StateCovariance([1 3 5 2 4 6], [1 3 5 2 4 6]);
d = (estimate - true)' / cov * (estimate - true);
end

```

Reference

[1] Sridharan, Ramaswamy, and Antonio F. Pensa, eds. *Perspectives in Space Surveillance*. MIT Press, 2017.

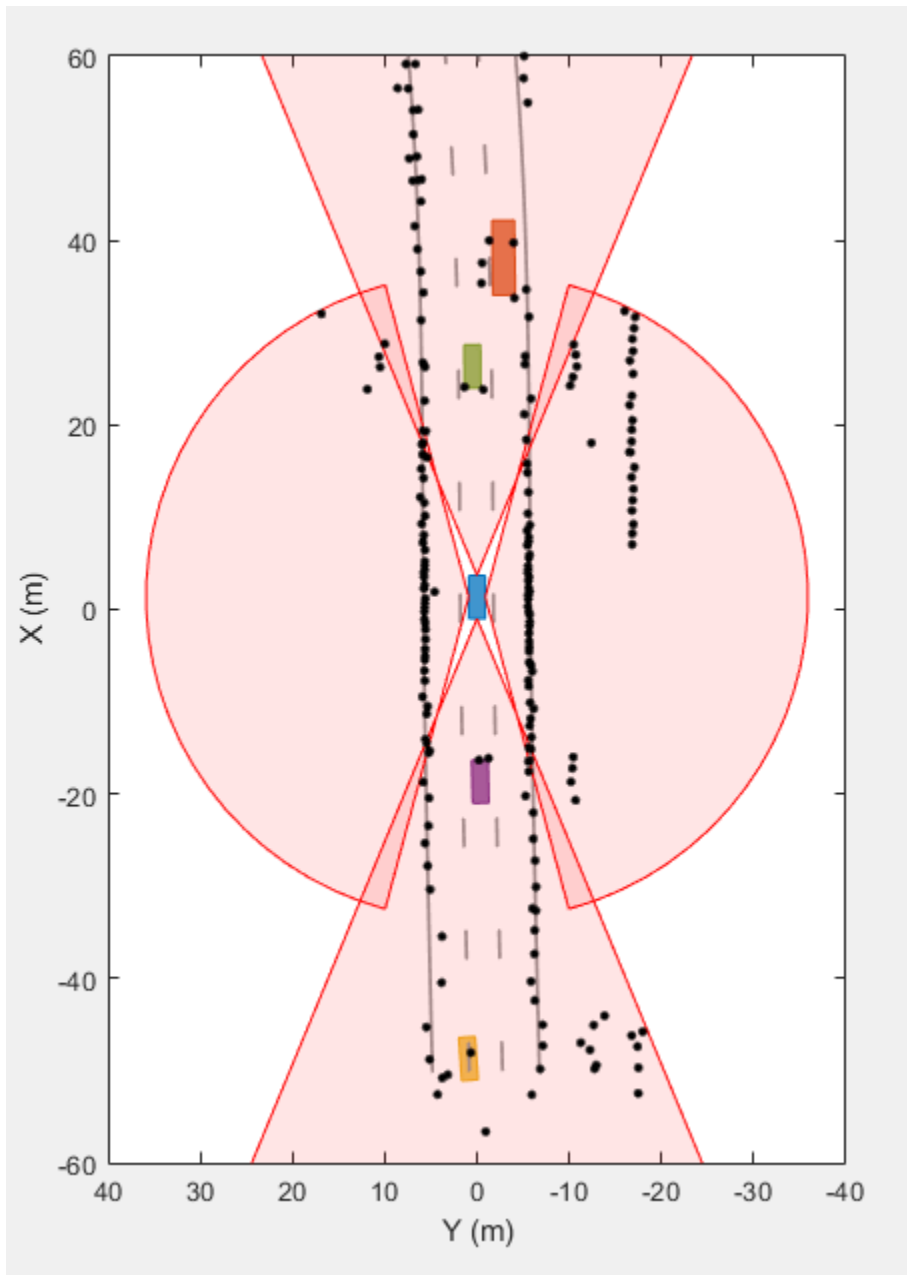
Highway Vehicle Tracking with Multipath Radar Reflections

This example shows the challenges associated with tracking vehicles on a highway in the presence of multipath radar reflections. It also shows a ghost filtering approach used with an extended object tracker to simultaneously filter ghost detections and track objects.

Introduction

Automotive radar sensors are robust against adverse environment conditions encountered during driving, such as fog, snow, rain, and strong sunlight. Automotive radar sensors have this advantage because they operate at substantially large wavelengths compared to visible-wavelength sensors, such as lidar and camera. As a side effect of using large wavelengths, surfaces around the radar sensor act like mirrors and produce undesired detections due to multipath propagation. These detections are often referred to as *ghost detections* because they seem to originate from regions where no target exists. This example shows you the impact of these multipath reflections on designing and configuring an object tracking strategy using radar detections. For more details regarding multipath phenomenon and simulation of ghost detections, see the “Simulate Radar Ghosts Due to Multipath Return” (Radar Toolbox) example.

In this example, you simulate the multipath detections from radar sensors in an urban highway driving scenario. The highway is simulated with a barrier on both sides of the road. The scenario consists of an ego vehicle and four other vehicles driving on the highway. The ego vehicle is equipped with four radar sensors providing 360-degree coverage. This image shows the configuration of the radar sensors and detections from one scan of the sensors. The red regions represent the field of view of the radar sensors and the black dots represent the detections.



The radar sensors report detections from the vehicles and from the barriers that are on both sides of the highway. The radars also report detections that do not seem to originate from any real object in the scenario. These are ghost detections due to multipath propagation of radar signals. Object trackers assume all detections originate from real objects or uniformly distributed random clutter in the field of view. Contrary to this assumption, the ghost detections are typically more persistent than clutter and behave like detections from real targets. Due to this reason, an object tracking algorithm is very likely to generate false tracks from these detections. It is important to filter out these detections before processing the radar scan with a tracker.

Generate Sensor Data

The scenario used in this example is created using the `drivingScenario` class. You use the `radarDataGenerator` (Radar Toolbox) System object™ to simulate radar returns from a direct path and from reflections in the scenario. The `HasGhosts` property of the sensor is specified as `true` to simulate multipath reflections. The creation of the scenario and the sensor models is wrapped in the helper function `helperCreateMultipathDrivingScenario`, which is attached with this example. The data set obtained by sensor simulation is recorded in a MAT file that contains returns from the radar and the corresponding sensor configurations. To record the data for a different scenario or sensor configuration, you can use the following command:

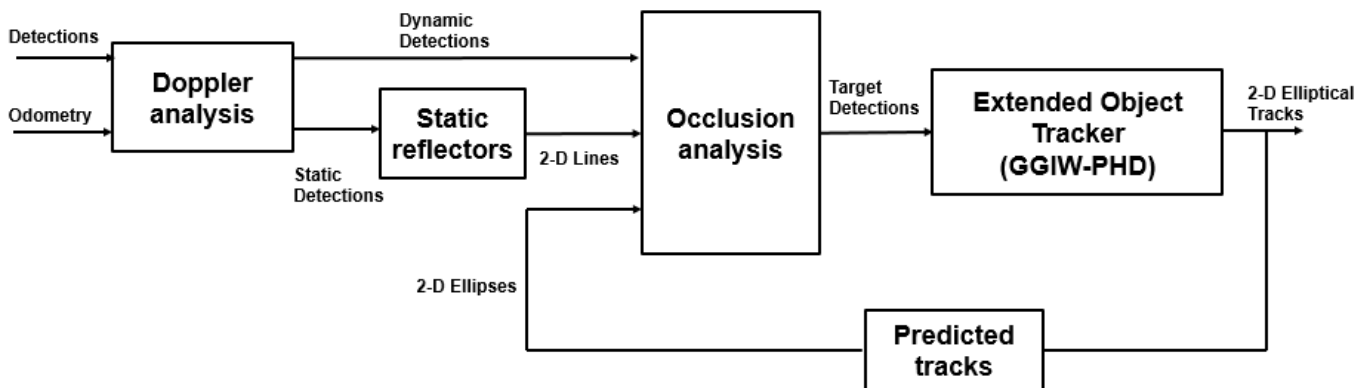
```
helperRecordData(scenario, egoVehicle, sensors, fName);

% Create the scenario
[scenario, egoVehicle, sensors] = helperCreateMultipathDrivingScenario;

% Load the recorded data
load('MultiPathRadarScenarioRecording.mat', 'detectionLog', 'configurationLog');
```

Radar Processing Chain: Radar Detections to Track List

In this section, you set up an integrated algorithm to simultaneously filter radar detections and track extended objects. The block diagram illustrates the radar processing chain used in this example.



Next, you learn about each of these steps and the corresponding helper functions.

Doppler analysis

The radar sensors report the measured relative radial velocity of the reflected signals. In this step, you utilize the measured radial velocity of the detections to determine if the target is static or dynamic [1]. In the previous radar scan, a large percentage of the radar detections originate from the static environment around the ego vehicle. Therefore, classifying each detection as static or dynamic greatly helps to improve the understanding of the scene. You use the helper function `helperClassifyStaticDynamic` to classify each detection as static or dynamic.

Static reflectors

The static environment is also typically responsible for a large percentage of the ghost reflections reported by the radar sensors. After segmenting the data set and finding static detections, you process them to find 2-D line segments in the coordinate frame of the ego vehicle. First, you use the

DBSCAN algorithm to cluster the static detections into different clusters around the ego vehicle. Second, you fit a 2-D line segment on each cluster. These fitted line segments define possible reflection surfaces for signals propagating back to the radar. You use the helper function `helperFindStaticReflectors` to find these 2-D line segments from static detections.

Occlusion analysis

Reflection from surfaces produce detections from the radar sensor that seem to originate behind the reflector. After segmenting the dynamic detections from the radar, you use simple occlusion analysis to determine if the radar detection is occluded behind a possible reflector. Because signals can be reflected by static or dynamic objects, you perform occlusion analysis in two steps. First, dynamic detections are checked against occlusion with the 2-D line segments representing the static reflectors. You use the helper function `helperClassifyGhostsUsingReflectors` to classify if the detection is occluded by a static reflector. Second, the algorithm uses information about predicted tracks from an extended tracking algorithm to check for occlusion against dynamic objects in the scene. The algorithm uses only confirmed tracks from the tracker to prevent overfiltering of the radar detections in the presence of tentative or false tracks. You use the helper function `helperClassifyGhostsUsingTracks` to classify if the detection is occluded by a dynamic object.

This entire algorithm for processing radar detections and classifying them is then wrapped into a larger helper function, `helperClassifyRadarDetections`, which classifies and segments the detection list into four main categories:

- 1** Target detections - These detections are classified to originate from real dynamic targets in the scene.
- 2** Environment detections - These detections are classified to originate from the static environment.
- 3** Ghost (Static) - These detections are classified to originate from dynamic targets but reflected via the static environment.
- 4** Ghost (Dynamic) - These detections are classified to originate from dynamic targets but reflected via other dynamic objects.

Setup GGIW-PHD Extended Object Tracker

The target detections are processed with an extended object tracker. In this example, you use a gamma Gaussian inverse Wishart probability hypothesis density (GGIW-PHD) extended object tracker. The GGIW-PHD tracker models the target with an elliptical shape and the measurement model assumes that the detections are uniformly distributed inside the extent of the target. This model allows a target to accept two-bounce ghost detections, which have a higher probability of being misclassified as a real target. These two-bounce ghost detections also report a Doppler measurement that is inconsistent with the actual motion of the target. When these ghost detections and real target detections from the same object are estimated to belong to the same partition of detections, the incorrect Doppler information can potentially cause the track estimate to diverge.

To reduce this problem, the tracker processes the range-rate measurements with higher measurement noise variance to account for this imperfection in the target measurement model. The tracker also uses a combination of a high assignment threshold and low merging threshold. A high assignment threshold allows the tracker to reduce the generation of new components from ghost targets detections, which are misclassified as target detections. A low merging threshold enables the tracker to discard corrected components (hypothesis) of a track, which might have diverged due to correction with ghost detections.

You set up the tracker using the `trackerPHD` System object™. For more details about extended object trackers, refer to the “Extended Object Tracking of Highway Vehicles with Radar and Camera” (Automated Driving Toolbox) example.

```
% Configuration of the sensors from the recording to set up the tracker
[~, sensorConfigurations] = helperAssembleData(detectionLog{1}, configurationLog{1});

% Configure the tracker to use the GGIW-PHD filter with constant turn-rate motion model
for i = 1:numel(sensorConfigurations)
    sensorConfigurations{i}.FilterInitializationFcn = @helperInitGGIWFilter;
    sensorConfigurations{i}.SensorTransformFcn = @ctmeas;
end

% Create the tracker using trackerPHD with Name-value pairs
tracker = trackerPHD(SensorConfigurations = sensorConfigurations, ...
    PartitioningFcn = @(x)helperMultipathExamplePartitionFcn(x,2,5), ...
    AssignmentThreshold = 450, ...
    ExtractionThreshold = 0.8, ...
    ConfirmationThreshold = 0.85, ...
    MergingThreshold = 25, ...
    DeletionThreshold = 1e-3, ...
    BirthRate = 1e-4, ...
    HasSensorConfigurationsInput = true...
);
```

Run Scenario and Track Objects

Next, you advance the scenario, use the recorded measurements from the sensors, and process them using the previously described algorithm. You analyze the performance of the tracking algorithm by using the generalized optimal subpattern assignment (GOSPA) metric. You also analyze the performance of the classification filtering algorithm by estimating a confusion matrix between true and estimated classification of radar detections. You obtain the true classification information about the detections using the `helperTrueClassificationInfo` helper function.

```
% Create trackGOSPAMetric object to calculate GOSPA metric
gospaMetric = trackGOSPAMetric(Distance = 'custom', ...
    DistanceFcn = @helperGOSPADistance, ...
    CutoffDistance = 35);

% Create display for visualization of results
display = helperMultiPathTrackingDisplay;

% Predicted track list for ghost filtering
predictedTracks = objectTrack.empty(0,1);

% Confusion matrix
confMat = zeros(5,5,numel(detectionLog));

% GOSPA metric
gospa = zeros(4,numel(detectionLog));

% Ground truth
groundTruth = scenario.Operators(2:end);

for i = 1:numel(detectionLog)
    % Advance scene for visualization of ground truth
    advance(scenario);
```

```
% Current time
time = scenario.SimulationTime;

% Detections and sensor configurations
[detections, configurations] = helperAssembleData(detectionLog{i},configurationLog{i});

% Predict confirmed tracks to current time for classifying ghosts
if isLocked(tracker)
    predictedTracks = predictTracksToTime(tracker,'confirmed',time);
end

% Classify radar detections as targets, ghosts, or static environment
[targets, ghostStatic, ghostDynamic, static, reflectors, classificationInfo] = helperClassifyRadarDetections(detections, configurations, time);

% Pass detections from target and sensor configurations to the tracker
confirmedTracks = tracker(targets, configurations, time);

% Visualize the results
display(egoVehicle, sensors, targets, confirmedTracks, ghostStatic, ghostDynamic, static, reflectors);

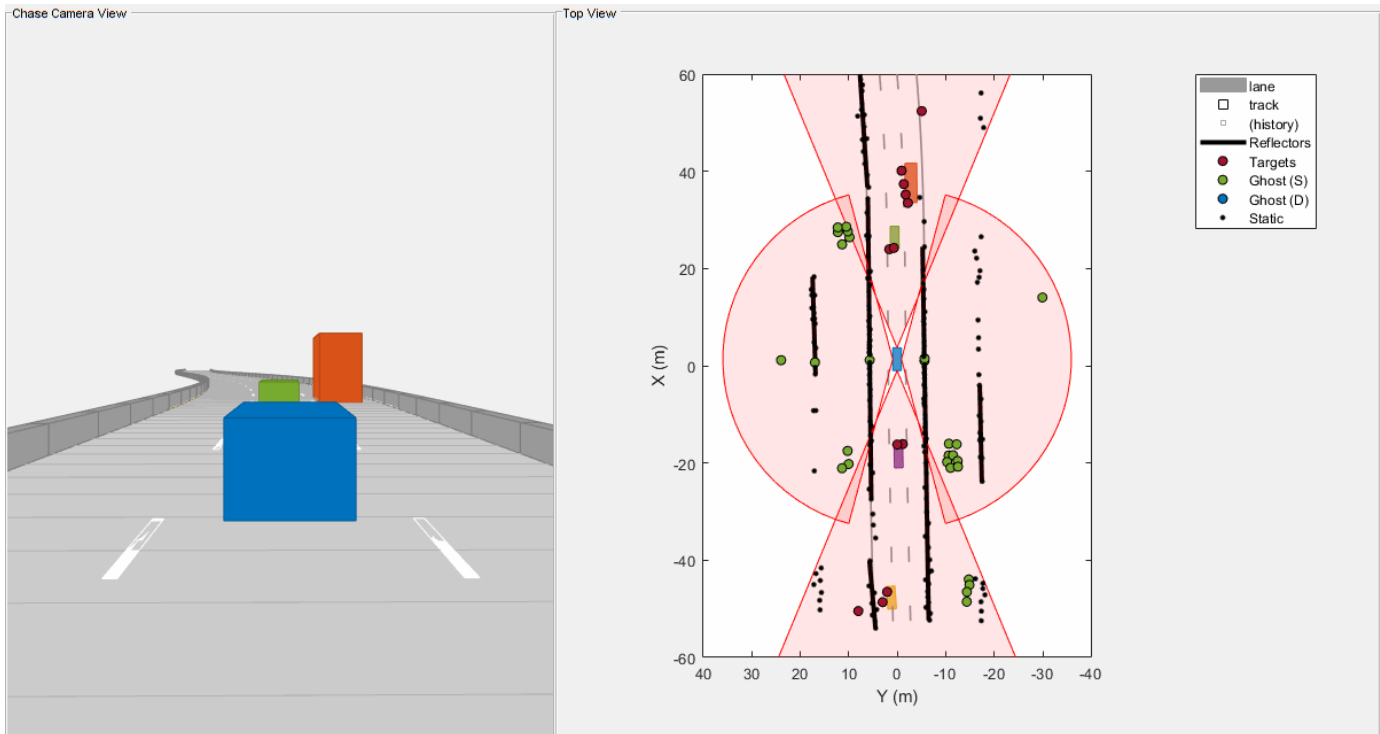
% Calculate GOSPA metric
[gospa(1, i),~,~,gospa(2,i),gospa(3,i),gospa(4,i)] = gospaMetric(confirmedTracks, groundTruthTracks{i});

% Get true classification information and generate confusion matrix
trueClassificationInfo = helperTrueClassificationInfo(detections);
confMat(:, :, i) = helperConfusionMatrix(trueClassificationInfo, classificationInfo);
end
```

Results

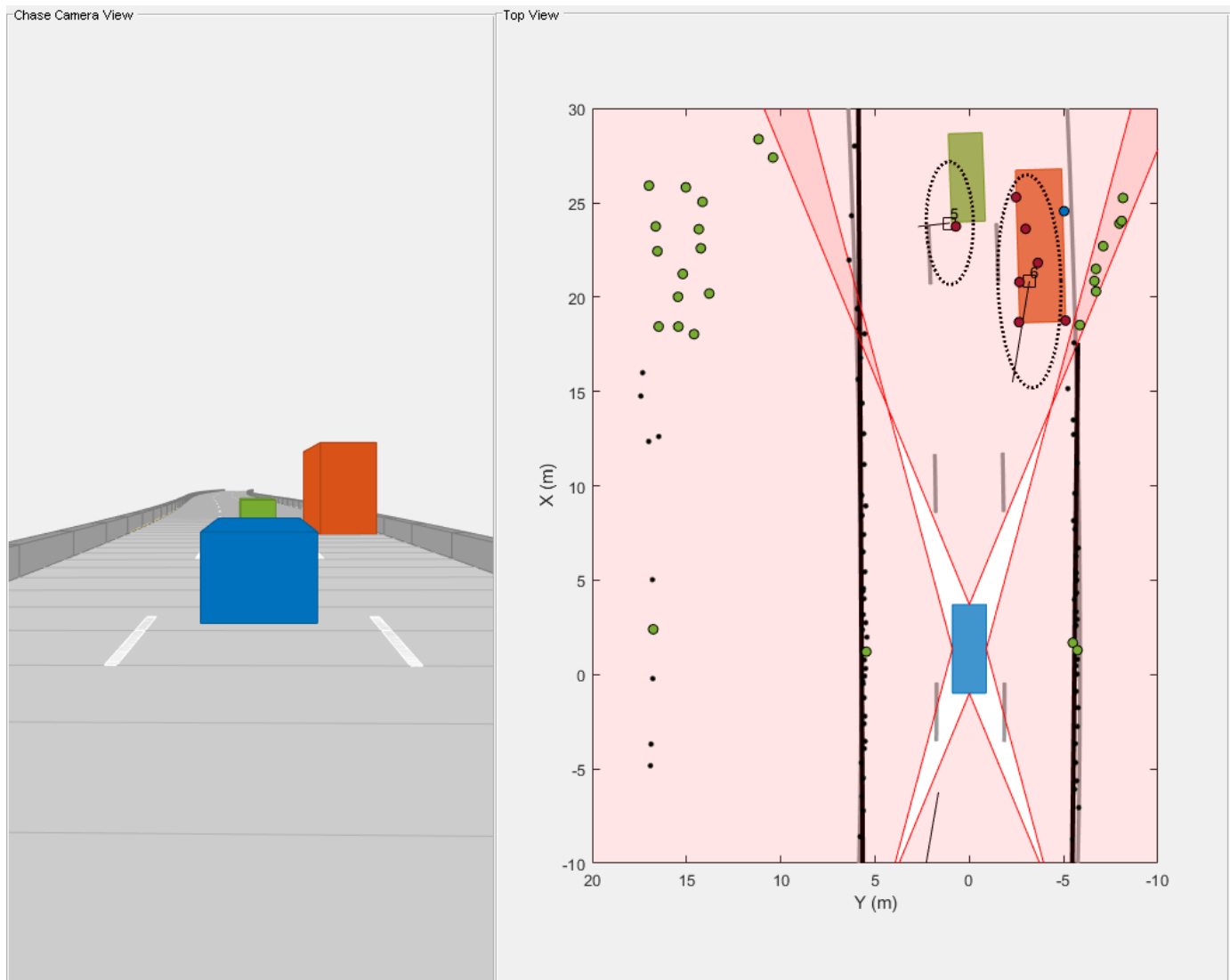
Animation and snapshot analysis

The animation that follows shows the result of the radar data processing chain. The black ellipses around vehicles represent estimated tracks. The radar detections are visualized with four different colors depending on their predicted classification from the algorithm. The black dots in the visualization represent static radar target detections. Notice that these detections are overlapped by black lines, which represent the static reflectors found using the DBSCAN algorithm. The maroon markers represent the detections processed by the extended object tracker, while the green and blue markers represent radar detections classified as reflections via static and dynamic objects, respectively. Notice that the tracker is able to maintain a track on all four vehicles during the scenario.



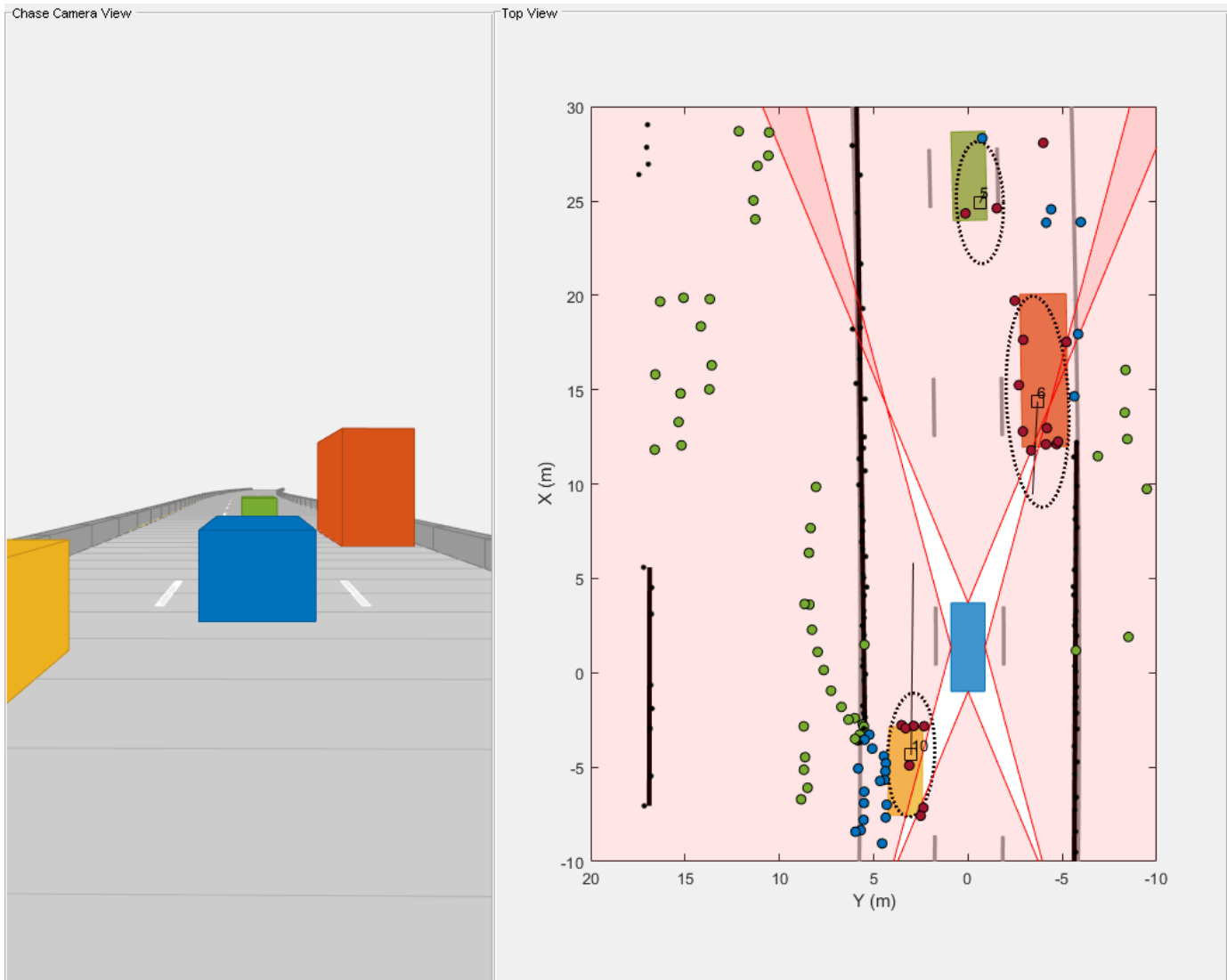
Next, you analyze the performance of the algorithm using different snapshots captured during the simulation. The snapshot below is captured at 3 seconds and shows the situation in front of the ego vehicle. At this time, the ego vehicle is approaching the slow-moving truck, and the left radar sensor observes reflections of these objects via the left barrier. These detections appear as mirrored detections of these objects in the barrier. Notice that the black line estimated as a 2-D reflector is in the line of sight of these detections. Therefore, the algorithm is able to correctly classify these detections as ghost targets reflected off static objects.

```
f = showSnaps(display,1:2,1);
if ~isempty(f)
    ax = findall(f,'Type','Axes','Tag','birdsEyePlotAxes');
    ax.XLim = [-10 30];
    ax.YLim = [-10 20];
end
```



Next, analyze the performance of the algorithm using the snapshot captured at 4.3 seconds. At this time, the ego vehicle is even closer to the truck and the truck is approximately halfway between the green vehicle and the ego vehicle. During these situations, the left side of the truck acts as a strong reflector and generates ghost detections. The detections on the right half of the green vehicle are from two-bounce detections off of the green vehicle as the signal travels back to the sensor after reflecting off the truck. The algorithm is able to classify these detections as ghost detections generated from dynamic object reflections because the estimated extent of the truck is in the direct line of sight of these detections.

```
f = showSnaps(display,1:2,2);
if ~isempty(f)
    ax = findall(f,'Type','Axes','Tag','birdsEyePlotAxes');
    ax.XLim = [-10 30];
    ax.YLim = [-10 20];
end
```

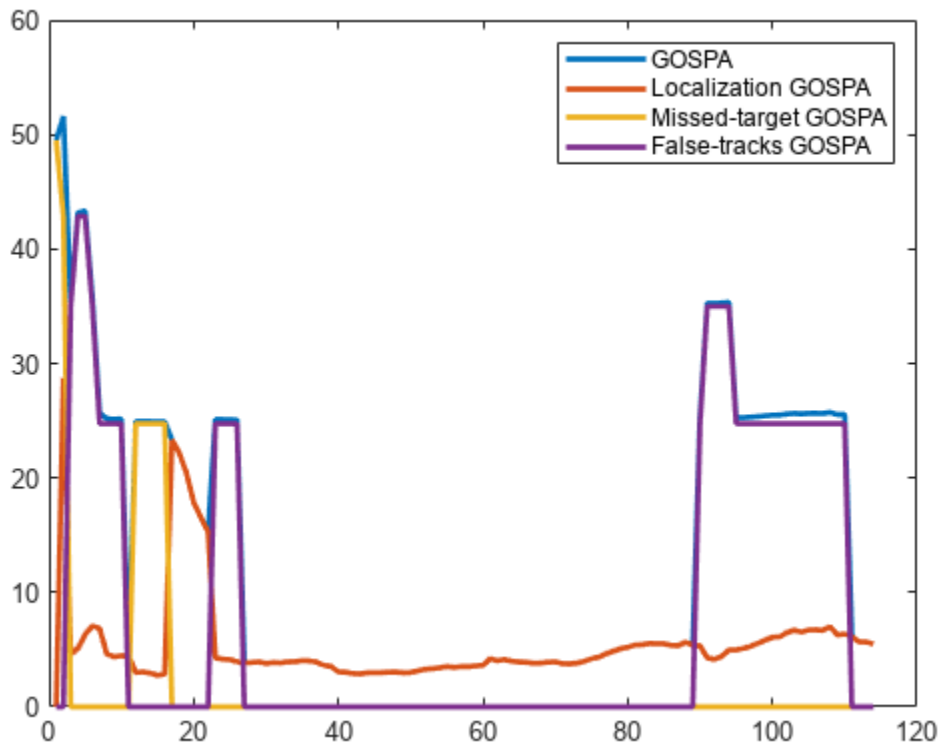
Also notice the passing vehicle denoted by the yellow car on the left of the ego vehicle. The detections, which seem to originate from the nonvisible surface of the yellow vehicle, are two-bounce detections of the barriers, reflected via the front face of the passing vehicle. These ghost detections are misclassified as target detections because they seem to originate from inside the estimated extent of the vehicle. At the same location, the detections that lie beyond the barrier are also two-bounce detections of the front face when the signal is reflected from the barrier and returns to the sensor. Since these detections lie beyond the extent of the track and the track is in the direct line of sight, they are classified as ghost detections from reflections off dynamic objects.

Performance analysis

Quantitatively assess the performance of the tracking algorithm by using the GOSPA metric and its associated components. A lower value of the metric denotes better tracking performance. In the figure below, the *Missed-target* component of the metric remains zero after a few steps in the beginning, representing establishment delay of the tracker as well as an occluded target. The zero value of the component shows that no targets were missed by the tracker. The *False-tracks* component of the metric increased around for 1 second around 85th time step. This denotes a false

track confirmed by the tracker for a short duration from ghost detections incorrectly classified as a real target.

```
figure;
plot(gospa', 'LineWidth', 2);
legend('GOSPA', 'Localization GOSPA', 'Missed-target GOSPA', 'False-tracks GOSPA');
```



Similar to the tracking algorithm, you also quantitatively analyze the performance of the radar detection classification algorithm by using a confusion matrix [2]. The rows shown in the table denote the true classification information of the radar detections and the columns represent the predicted classification information. For example, the second element of the first row defines the percentage of target detections predicted as ghosts from static object reflections.

More than 90% of the target detections are classified correctly. However, a small percentage of the target detections are misclassified as ghosts from dynamic reflections. Also, approximately 3% of ghosts from static object reflections and 20% of ghosts from dynamic object reflections are misclassified as targets and sent to the tracker for processing. A common situation when this occurs in this example is when the detections from two-bounce reflections lie inside the estimated extent of the vehicle. Further, the classification algorithm used in this example is not designed to find false alarms or clutter in the scene. Therefore, the fifth column of the confusion matrix is zero. Due to spatial distribution of the false alarms inside the field of view, the majority of false alarm detections are either classified as reflections from static objects or dynamic objects.

```
% Accumulate confusion matrix over all steps
confusionMatrix = sum(confMat,3);
numElements = sum(confusionMatrix,2);
```

```
numElemsTable = array2table(numElements, 'VariableNames', {'Number of Detections'}, 'RowNames', {'True Information'});
disp('True Information'); disp(numElemsTable);
```

```
True Information
```

	Number of Detections
Targets	1969
Ghost (S)	3155
Ghost (D)	849
Environment	27083
Clutter	138

```
% Calculate percentages
```

```
percentMatrix = confusionMatrix./numElements*100;
```

```
percentMatrixTable = array2table(round(percentMatrix,2), 'RowNames', {'Targets', 'Ghost (S)', 'Ghost (D)', 'Environment', 'Clutter'}, 'VariableNames', {'Targets', 'Ghost (S)', 'Ghost (D)', 'Environment', 'Clutter'});
```

```
disp('True vs Predicted Confusion Matrix (%)'); disp(percentMatrixTable);
```

```
True vs Predicted Confusion Matrix (%)
```

	Targets	Ghost (S)	Ghost (D)	Environment	Clutter
Targets	90.76	0.56	8.28	0.41	0
Ghost (S)	3.2	83.77	12.68	0.35	0
Ghost (D)	18.26	0.24	81.51	0	0
Environment	1.05	2.71	4.06	92.17	0
Clutter	21.74	62.32	15.22	0.72	0

Summary

In this example, you simulated radar detections due to multipath propagation in an urban highway driving scenario. You configured a data processing algorithm to simultaneously filter ghost detections and track vehicles on the highway. You also analyzed the performance of the tracking algorithm and the classification algorithm using the GOSPA metric and confusion matrix.

References

[1] Prophet, Robert, et al. "Instantaneous Ghost Detection Identification in Automotive Scenarios." *2019 IEEE Radar Conference (RadarConf)*. IEEE, 2019.

[2] Kraus, Florian, et al. "Using machine learning to detect ghost images in automotive radar." *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2020.

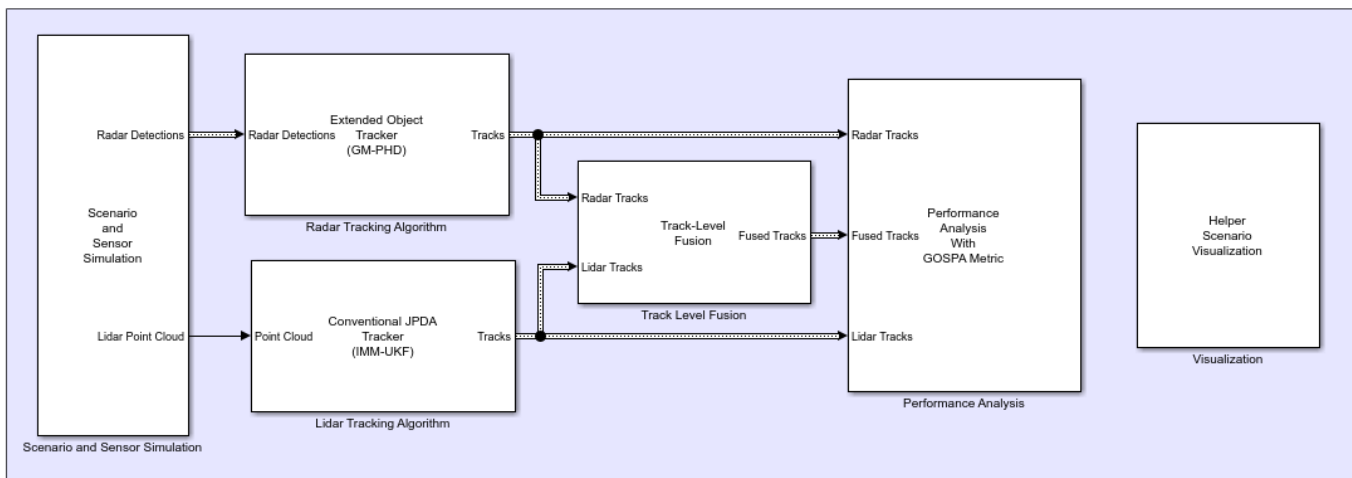
Track-Level Fusion of Radar and Lidar Data in Simulink

Autonomous systems require precise estimation of their surroundings to support decision making, planning, and control. High-resolution sensors such as radar and lidar are frequently used in autonomous systems to assist in estimation of the surroundings. These sensors generally output tracks. Outputting tracks instead of detections and fusing the tracks together in a decentralized manner provide several benefits, including low false alarm rates, higher target estimation accuracy, a low bandwidth requirement, and low computational costs. This example shows you how to track objects from measurements of a radar and a lidar sensor and how to fuse them using a track-level fusion scheme in Simulink®. You process the radar measurements using a Gaussian Mixture Probability Hypothesis Density (GM-PHD) 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 example closely follows the “Track-Level Fusion of Radar and Lidar Data” on page 6-496 MATLAB® example.

Overview of Model

```
load_system('TrackLevelFusionOfRadarAndLidarDataInSimulink');
set_param('TrackLevelFusionOfRadarAndLidarDataInSimulink','SimulationCommand','update');
open_system('TrackLevelFusionOfRadarAndLidarDataInSimulink');
```

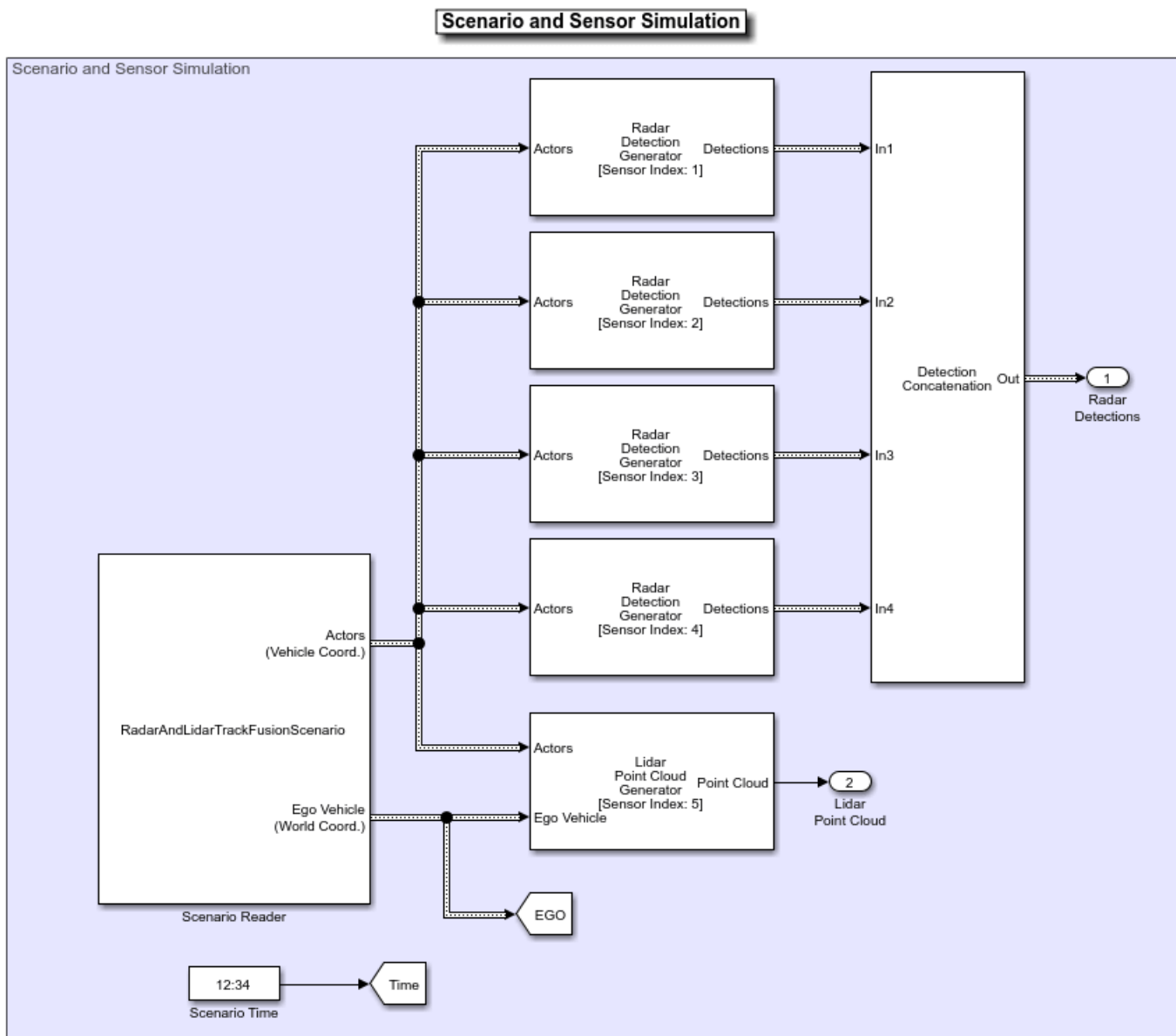
Track-Level Fusion of Radar and Lidar Data in Simulink



The model has six subsystems, each implementing a part of the algorithm.

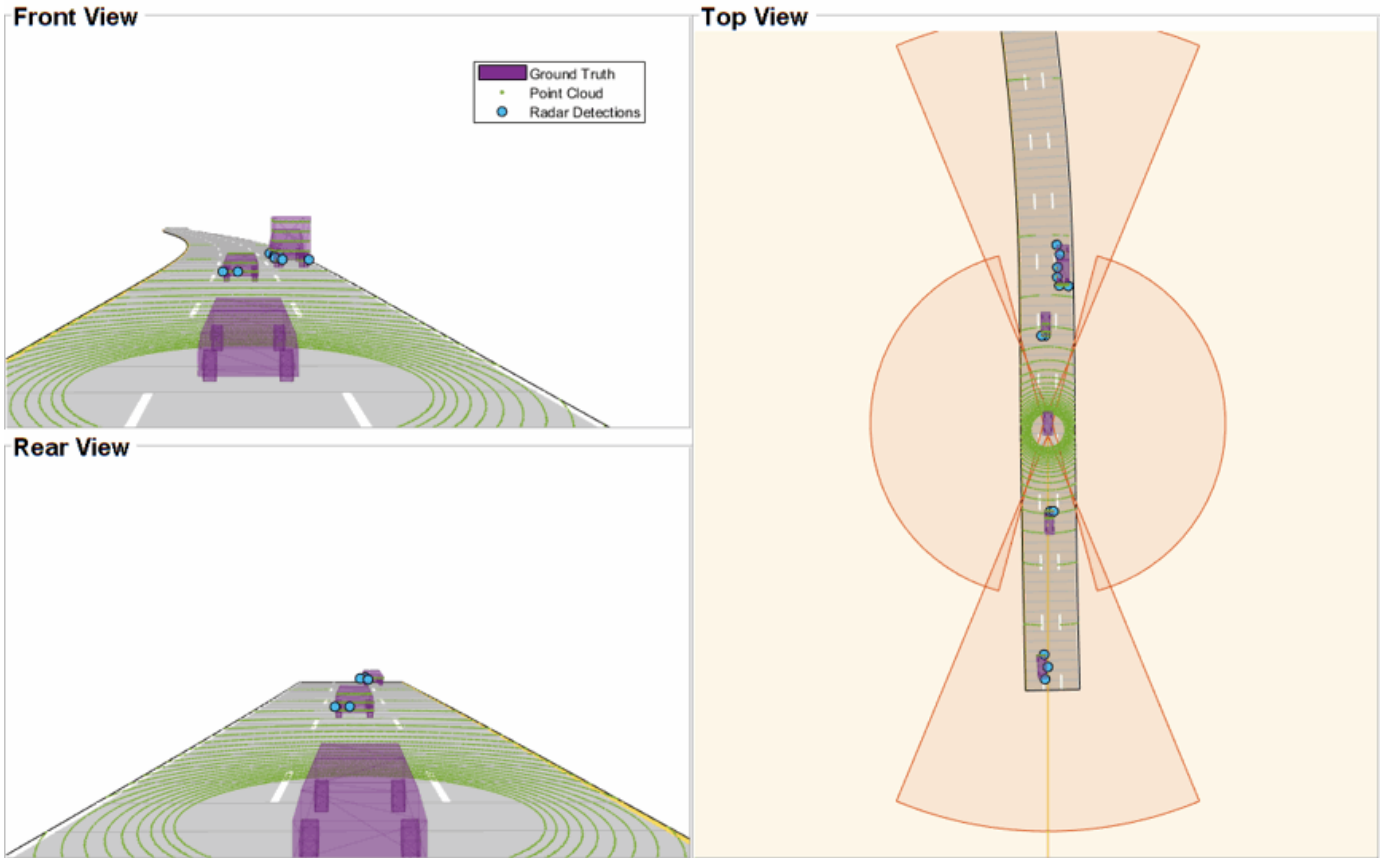
- Scenario and Sensor Simulation
- Radar Tracking Algorithm
- Lidar Tracking Algorithm
- Track Level Fusion
- Performance Analysis
- Visualization

Scenario and Sensor Simulation



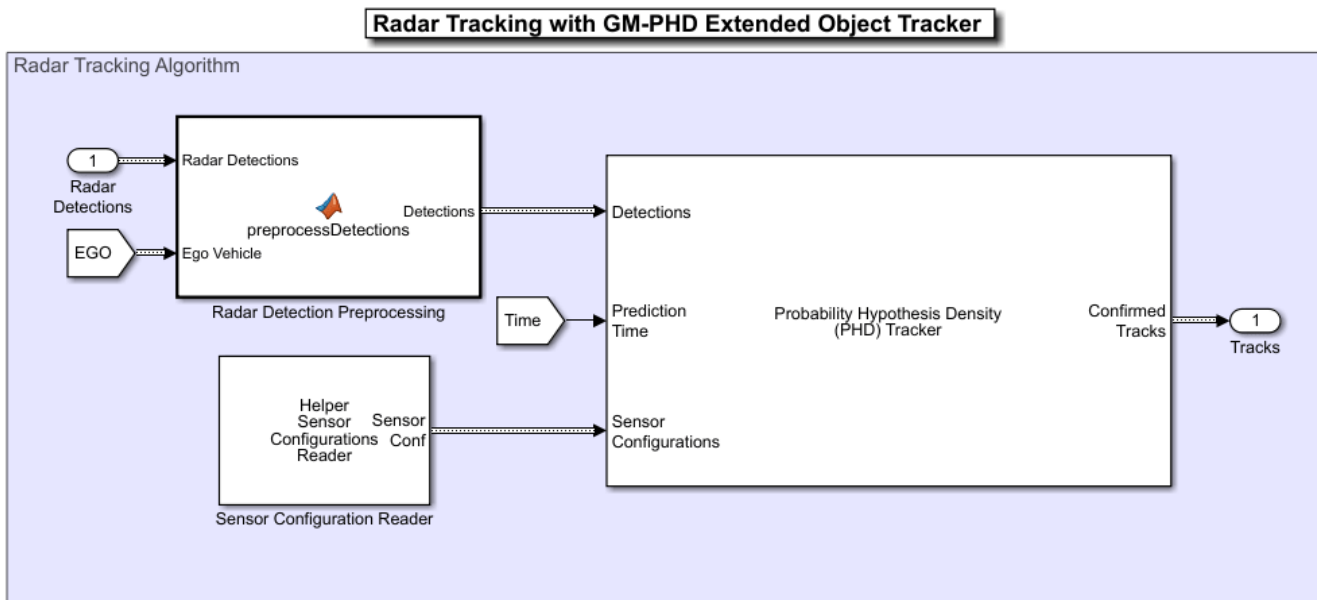
The scenario recording for this example is captured from the scenario described in “Track-Level Fusion of Radar and Lidar Data” on page 6-496 MATLAB example. The Scenario Reader block reads a prerecorded scenario file and generates actors and ego vehicle position data as `Simulink.Bus` (Simulink) objects. In this scenario, 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 vehicle 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). The Radar Detection Generator block generates radar detections and the Lidar Point Cloud Generator block generates point clouds. Detections from all four radar sensors are grouped together with the Detection Concatenation block and the Digital Clock

block is used to simulate time. Sensor configurations and the simulated sensor data is visualized in the animation that follows. Notice that the radars have higher resolution than objects and therefore return multiple measurements per object. Also notice that the lidar interacts with the actors as well as the road surface to return multiple points.



- Ground Truth
- Radar Tracks
- Lidar Tracks
- Fused Tracks
- Point Cloud
- Radar Detections
- ▲ Lidar Bounding Box Detections

Radar Tracking Algorithm

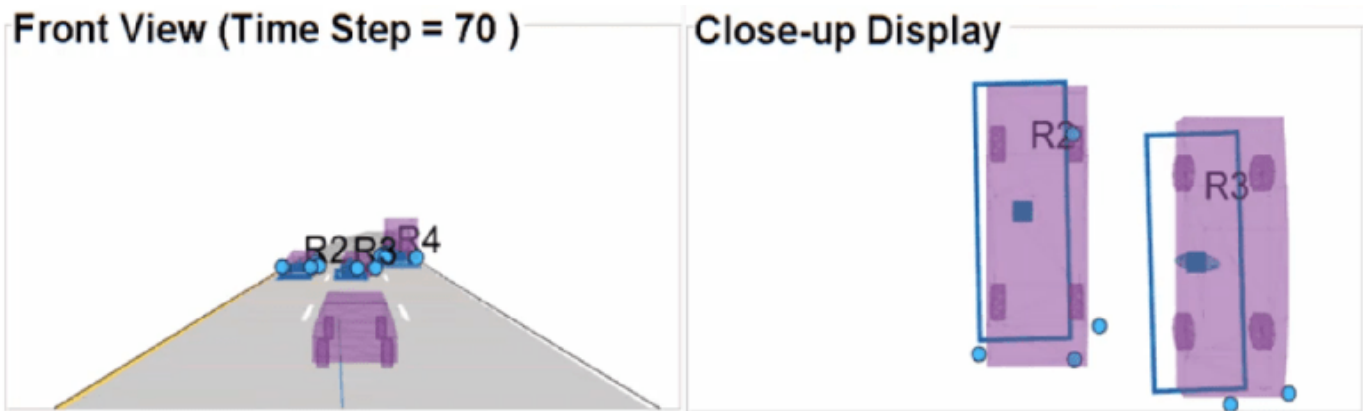


Radars generally have higher resolution than the objects and return multiple detections per object. Conventional trackers such as Global Nearest Neighbor (GNN) and Joint Probabilistic Data Association (JPDA) assume that the sensors return at most one detection per object per scan. Therefore, the detections from high-resolution sensors must be either clustered before processing them with conventional trackers or must be processed using extended object trackers. Extended

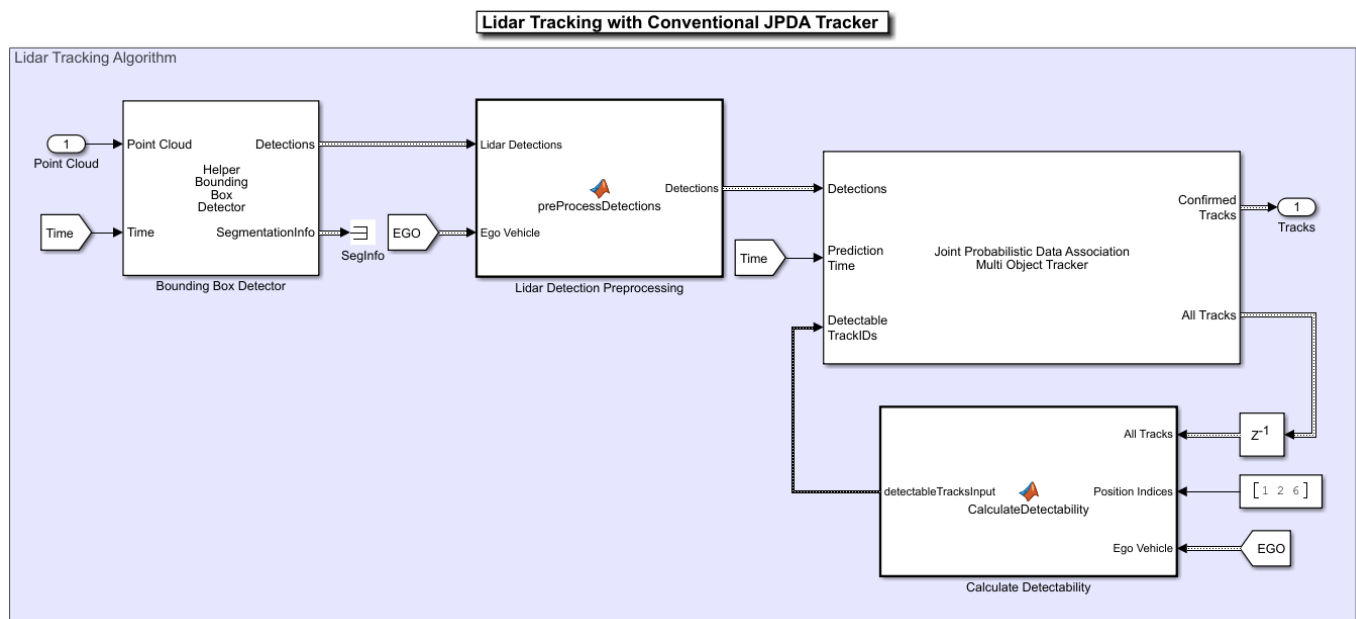
object trackers do not require preclustering of detections. Generally, extended object trackers offer better estimation of objects since they handle clustering and data association simultaneously.

In this example, you use a Gaussian mixture probability hypothesis density (GM-PHD) extended object tracker for radar tracking. The tracker block takes detections, prediction time, and sensor configurations as input and outputs confirmed tracks as a `Simulink.Bus` (Simulink) object. Detections from the radar are preprocessed to include ego vehicle INS information in the Radar Detection Preprocessing MATLAB Function (Simulink) Block. The Sensor Configuration Reader block provides Sensor Configuration to the tracker block. The block is implemented by using MATLAB System (Simulink) block. Code for this block is defined in the `HelperSourceConfigReader` class.

This visualization shows radar tracking at a single time step. Notice that the radar sensors report multiple detections per object and the GM-PHD tracker forms two-dimensional rectangular tracks corresponding to each object.



Lidar Tracking Algorithm

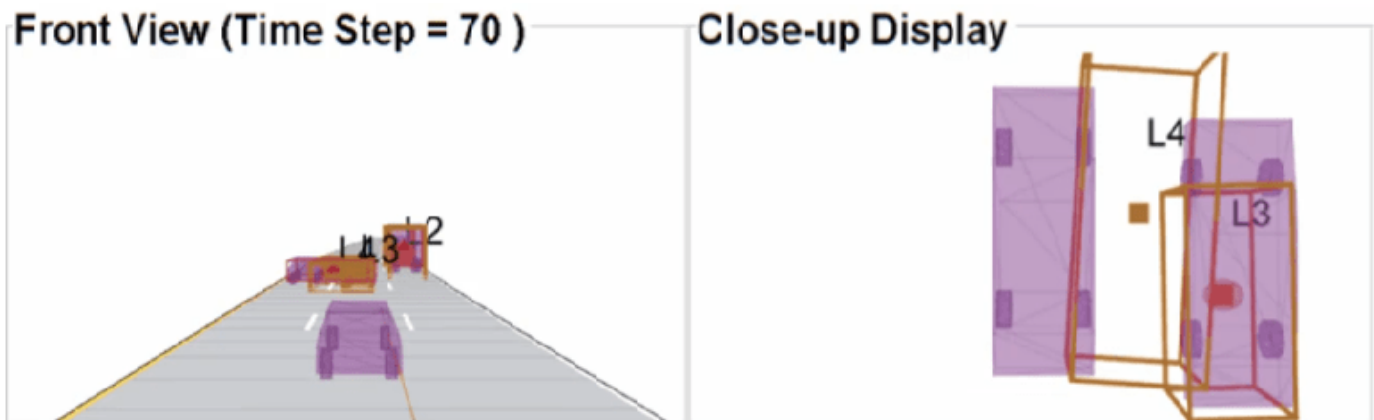


Lidar sensors have high resolution capabilities, and each scan from the sensor contains many points, commonly known as a *point cloud*. This raw data must be preprocessed to extract objects. The preprocessing is performed using a RANSAC-based plane-fitting algorithm and bounding boxes are fitted using 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 6-352 example.

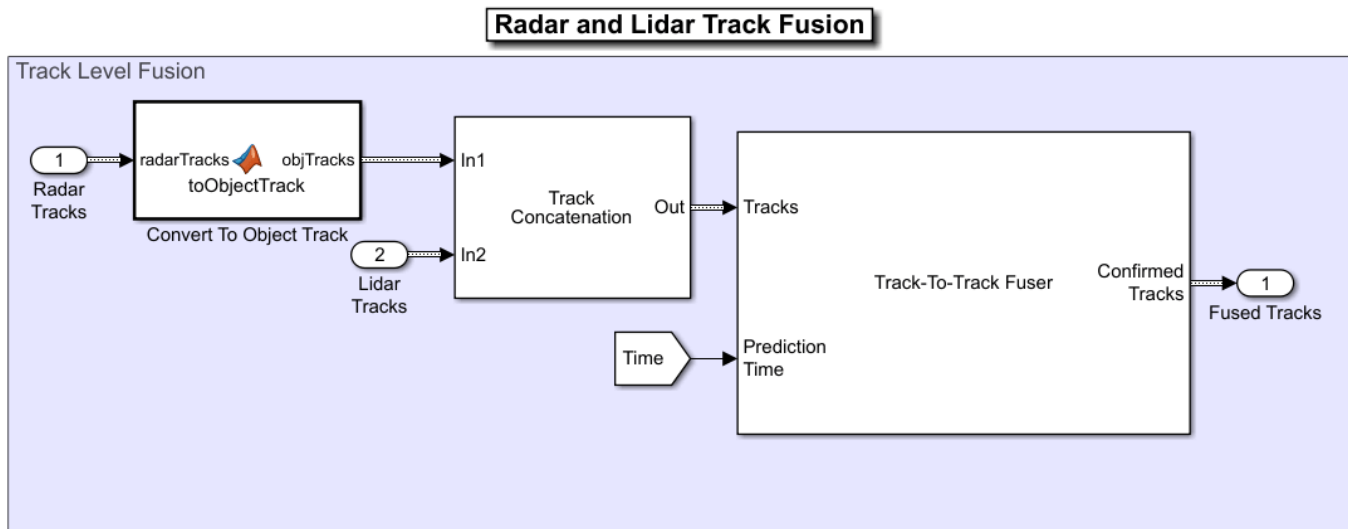
The Bounding Box Detector block is implemented using a MATLAB System (Simulink) block. Code for this block is defined in a helper class, `HelperBoundingBoxDetectorBlk`. The block accepts point cloud locations and a prediction time as input and outputs bounding box detections corresponding to obstacles and segmentation information. Detections are processed using a conventional JPDA tracker, configured with an interacting multiple model (IMM) filter. The IMM filter is implemented using the helper function `helperInitIMMUKFfilter`, which is specified as the **Filter initialization function** parameter of the block. Detections from lidar sensor are preprocessed to include ego vehicle INS information in the Lidar Detection Preprocessing MATLAB Function (Simulink) block.

The Calculate Detectability block calculates the `Detectable TrackIDs` input for the tracker and outputs an two-column array. The first column represents the TrackIDs of the tracks and the second column specifies their detection probability by the sensor and bounding box detector. The block is implemented using a MATLAB Function (Simulink) block.

This visualization shows the lidar tracking at a single time step. The Bounding Box Detector block generates detections from the point cloud, and the JPDA tracker forms three-dimensional cuboid tracks corresponding to each object.



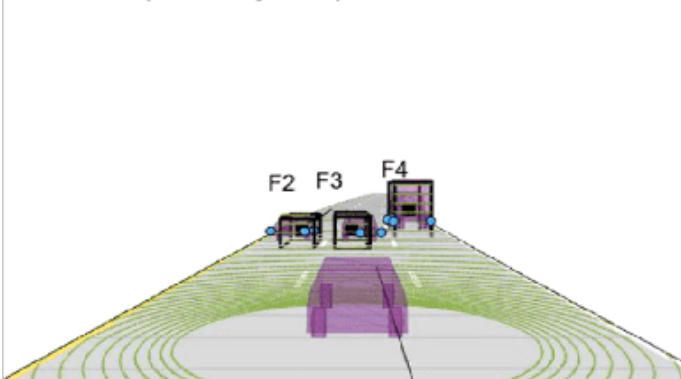
Track-Level Fusion



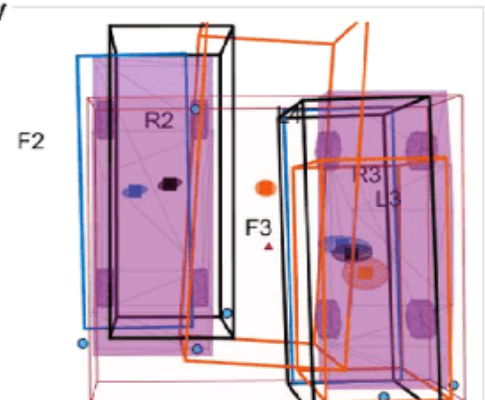
The track fusion algorithm is implemented using the Track-To-Track Fuser block. The block takes a prediction time, rectangular radar tracks, and cuboid lidar tracks as input and outputs fused tracks. It uses a traditional track-to-track association-based fusion scheme and GNN assignment to create a single hypothesis. The Track Concatenation block combines tracks from both sources and generates a single track bus. The fuser source configuration for radar and lidar is set using the `SourceConfig` variable through the `PreLoadFcn` callback. See “Model Callbacks” (Simulink) for more information about callback functions. The state fusion algorithm for this example is implemented in the `helperRadarLidarFusionFunction` helper function and specified as the 'custom fusion function' property.

This visualization shows track fusion at a single time step. The fused tracks are closer to the actual objects than the individual sensor tracks, which indicates that track accuracy increased after fusion of track estimates from both sensors.

Front View (Time Step = 70)

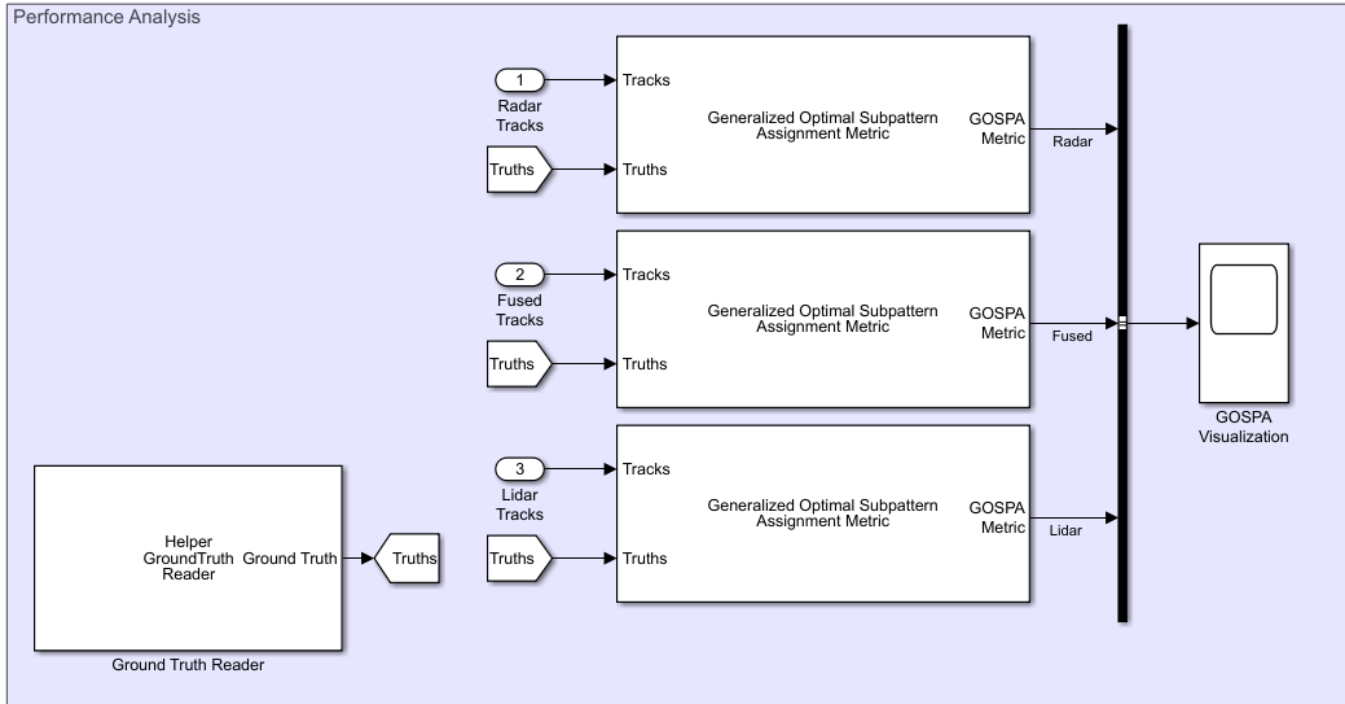


Close-up Display



Performance Analysis

Tracking Performance Analysis with GOSPA Metric



In this example, you assess the performance of each algorithm using the Generalized Optimal Subpattern Assignment (GOSPA) metric. The GOSPA metric aims to evaluate the performance of a tracking system with a scalar cost.

The GOSPA metric can be calculated with the following equation

$$GOSPA = [\sum_{i=0}^m (\min(d_b, c))^p + \frac{c^p}{\alpha} (n - m)]^{1/p}$$

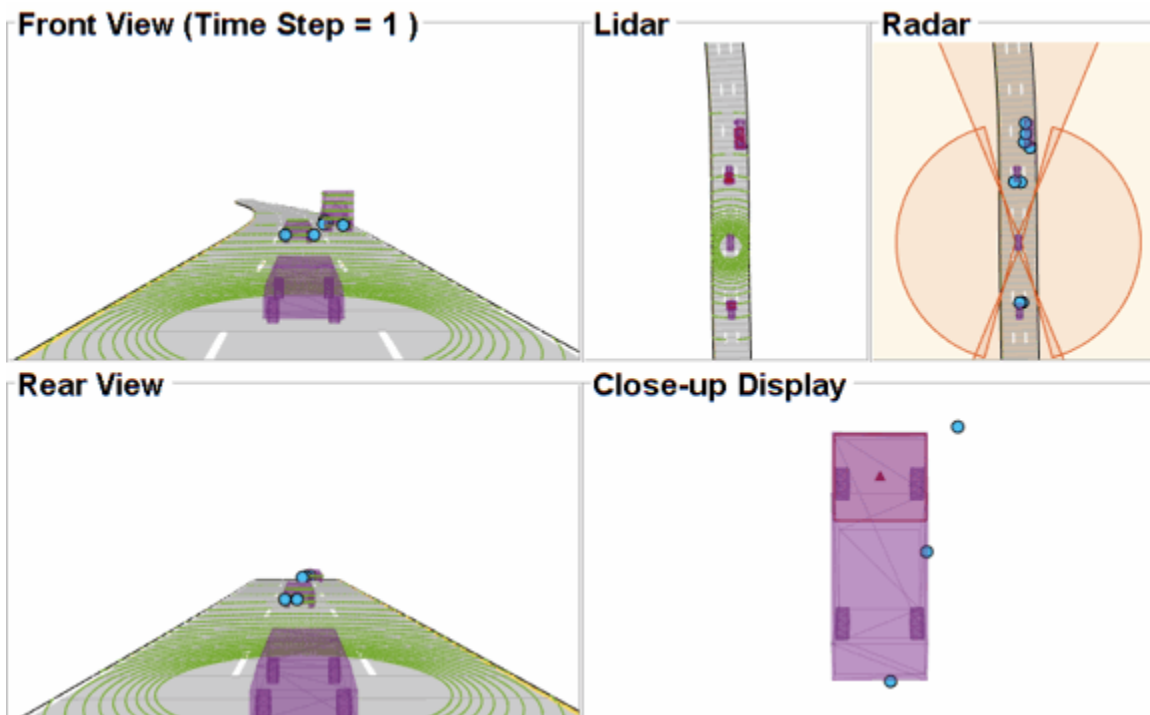
where m is the number of ground truths, n ($n \geq m$) is the number of estimated tracks, c is the cutoff distance threshold, d_b is the base distance between track and truth calculated by a distance function specified in the Distance property, p is order of the metric and α is the alpha parameter of the metric, defined from the block mask.

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 are defined in `helperRadarDistance` and `helperLidarDistance` helper files. The Ground Truth Reader block provides truth data at each time step. The block is implemented using a MATLAB System (Simulink) block and code for this block is defined in the `HelperGroundTruthReader` class. Finally, the GOSPA score for radar tracking, lidar tracking, and track fusion algorithms are grouped together.

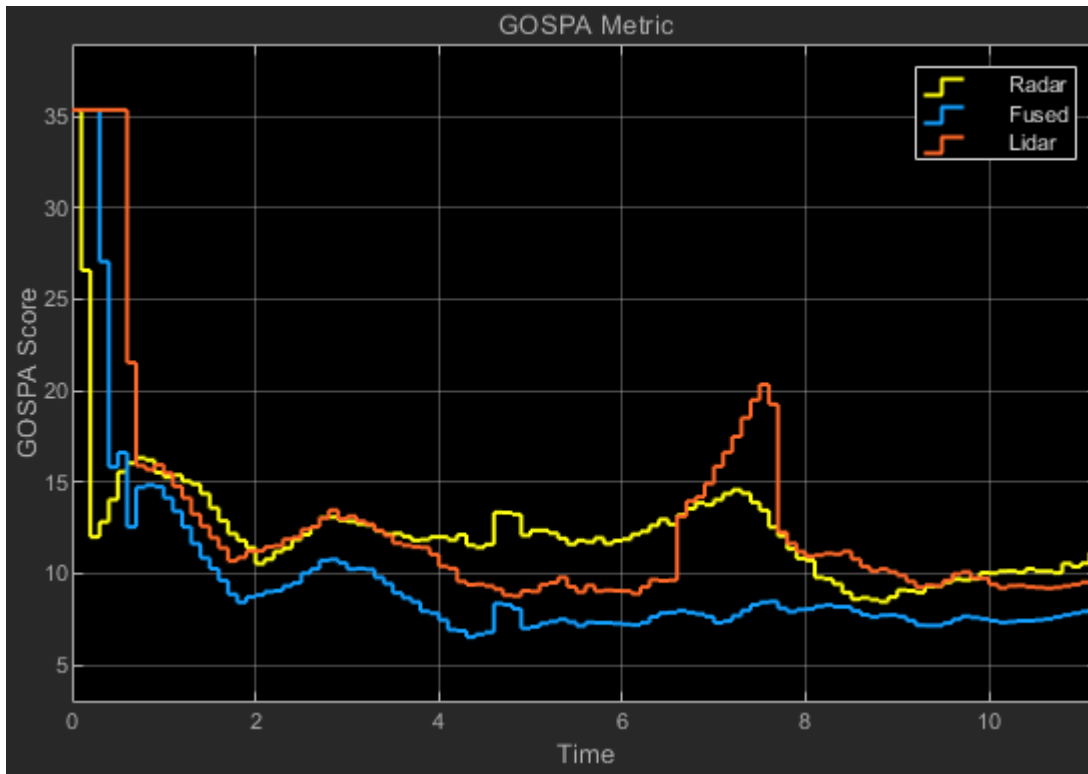
Visualization

The Visualization block is implemented using a MATLAB System (Simulink) block. Code for this block is defined in the helper class `helperLidarRadarTrackFusionDisplayBlock`. The block uses the `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.

This animation shows the entire run every three time steps. Each three tracking system (radar, lidar, and track-level fusion) was able to track all four vehicles in the scenario without confirming any false tracks.



The GOSPA metric is visualized using the scope block. The x-axis in the figure that follows represents time and y-axis represents the GOSPA score. Each unit on the x-axis represents 10 time steps in the scenario. The track-level localization accuracy of each tracker can be quantitatively assessed with 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 the state estimation of each vehicle. Each component of the GOSPA metric can also be selected as a block output and visualized separately.



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 both sensors.

```
close_system('TrackLevelFusionOfRadarAndLidarDataInSimulink');
```

Summary

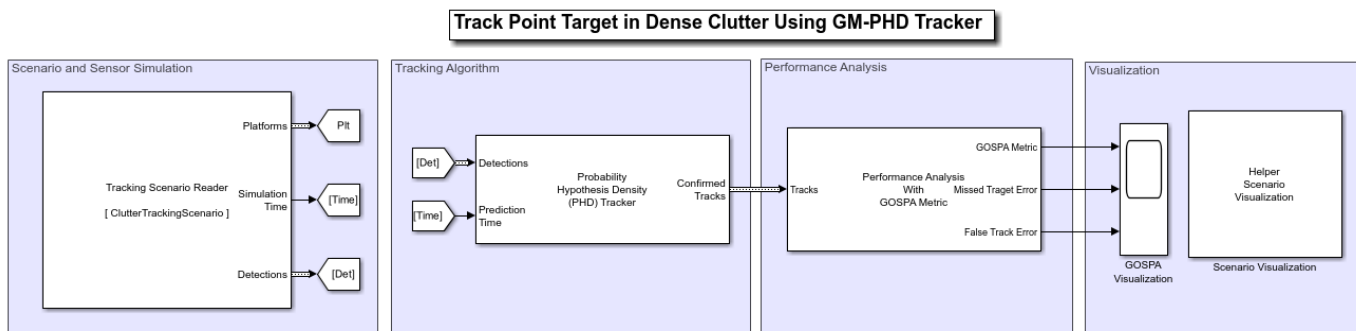
In this example, you learned how to track radar measurements using an extended object tracker with a two-dimensional rectangular model, how to track lidar measurements using a conventional JPDA tracker with a three-dimensional cuboid model, and how to set up a track-level fusion algorithm for fusing tracks from radar and lidar sensors in Simulink. You also learned how to evaluate performance of a tracking algorithm using the Generalized Optimal Subpattern Assignment metric. The simulation results show that tracking by fusing tracks from radar and lidar is more accurate than tracking by each individual sensor.

Track Point Targets in Dense Clutter Using GM-PHD Tracker in Simulink

Radars generally receive echoes from all surfaces in the signal path. These unwanted back-scattered signals or echoes generated from physical objects are called clutter. In a densely cluttered environment, missed detections and false alarms make tracking objects a challenging task for conventional trackers such as Global Nearest-Neighbor (GNN) tracker. In such an environment a PHD tracker provides better estimation of objects as it can handle multiple detections per object per sensor without clustering them first. This example shows you how to track point targets in dense clutter using a Gaussian mixture probability hypothesis density (GM-PHD) tracker with a constant velocity model in Simulink. The example closely follows the “Track Point Targets in Dense Clutter Using GM-PHD Tracker” on page 6-466 MATLAB® example.

Overview of the model

```
load_system('TrackPointTargetsInDenseClutterSimulinkExample');
set_param('TrackPointTargetsInDenseClutterSimulinkExample', 'SimulationCommand', 'update');
open_system('TrackPointTargetsInDenseClutterSimulinkExample');
```

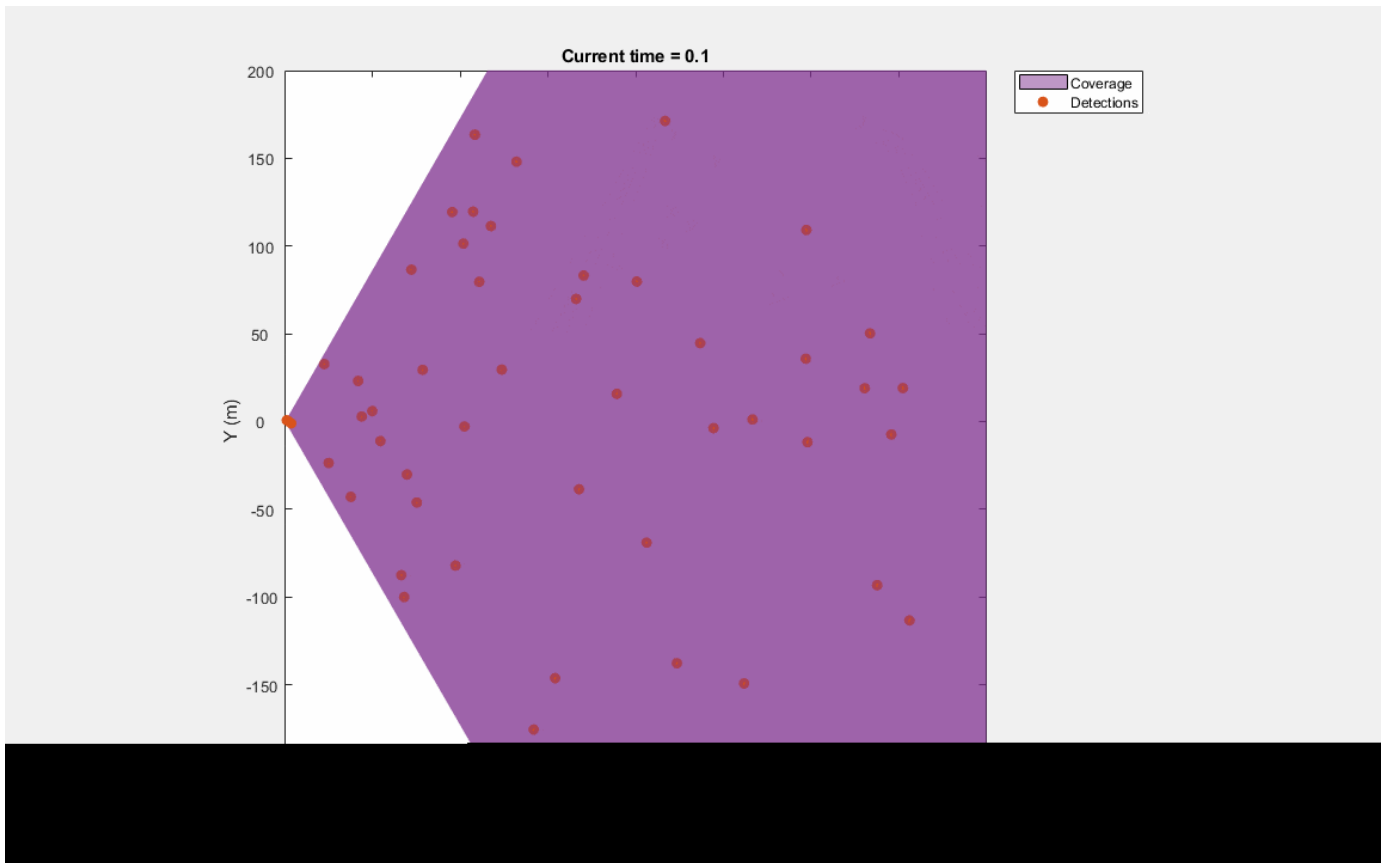


The model consists of the four sections, each implementing a part of the workflow:

- Scenario and Sensor Simulation
- Tracking Algorithm
- Performance Analysis
- Visualization

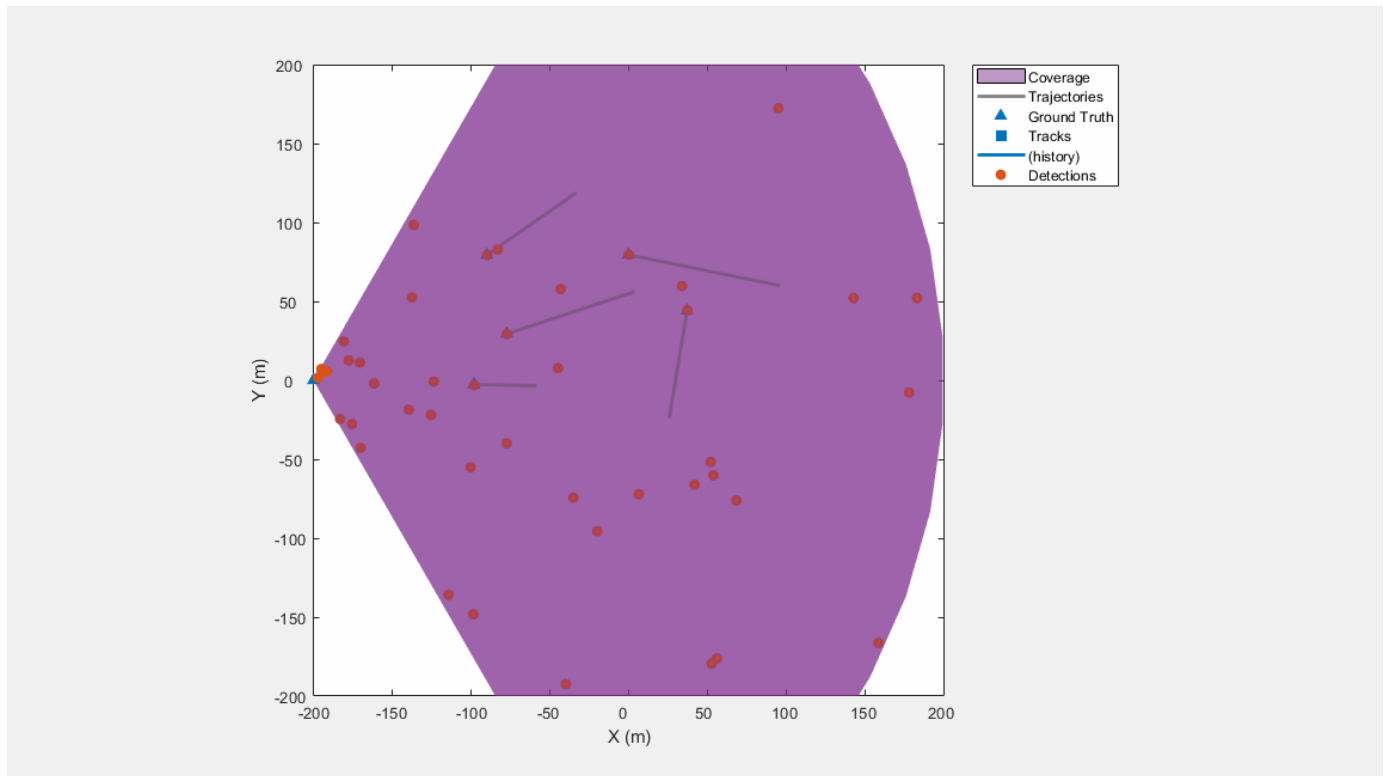
Scenario and Sensor Simulation

In this example the scenario is created using `trackingScenario`. The scenario consists of five point targets moving at a constant velocity. You use the `fusionRadarSensor` to simulate radar detections in the scenario. The targets move within the field of view of the sensor. 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 $1e-3$ and the resolution of the sensor defined in this example, there are approximately 53 false alarms generated per step. The scenario for this example is defined in helper file `helperCreateClutterTrackingScenario`. You use the Tracking Scenario Reader block to read the scenario object from the workspace. You configure the block to output detections along with platform poses and simulation time. The block output platform poses and detections as `Simulink.Bus` (Simulink) objects.

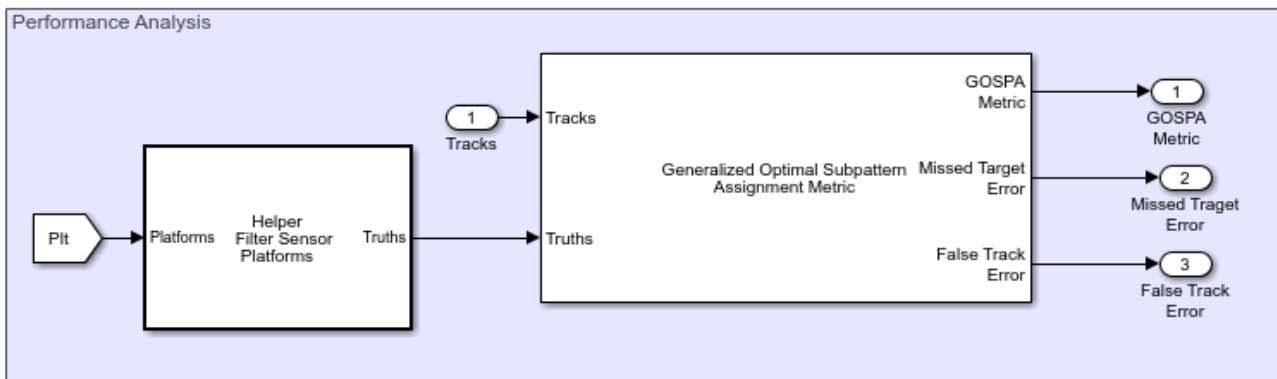


Tracking Algorithm

In this example you use the Probability Hypothesis Density (PHD) Tracker block with the `gmphd` filter to track targets. The first step towards configuring a PHD tracker is to define the configuration of the sensor. You define the sensor configuration as a structure with fields same as `trackingSensorConfiguration`. You set the `SensorIndex` of the configuration to 1 to match the index 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` field of the configuration as 1. 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. You define the `ClutterDensity` field in the configuration, which refers to false alarm rate per-unit volume from the sensor. You set the `FilterInitializationFcn` to `initcvgmphd`, which creates a constant-velocity GM-PHD filter. You specify the birth rate of new targets from block mask to define the expected number of targets appearing in the field of view per unit time. Code for creating the sensor configuration is defined in the helper function `helperCreateSensorConfig`. You call this function in the `PreLoadFcn` callback. See “Model Callbacks” (Simulink) for more information about callback functions.



Performance Analysis



To evaluate the performance of the tracker, you use the Generalized Optimal Subpattern Assignment Metric block. The GOSPA metric aims to evaluate the performance of a tracker using a single combined score for errors in assignment and distance. The GOSPA metric can be calculated by the following equation:

$$GOSPA = [\sum_{i=0}^m (\min(d_b, c))^p + \frac{c^p}{\alpha} (n - m)]^{1/p}$$

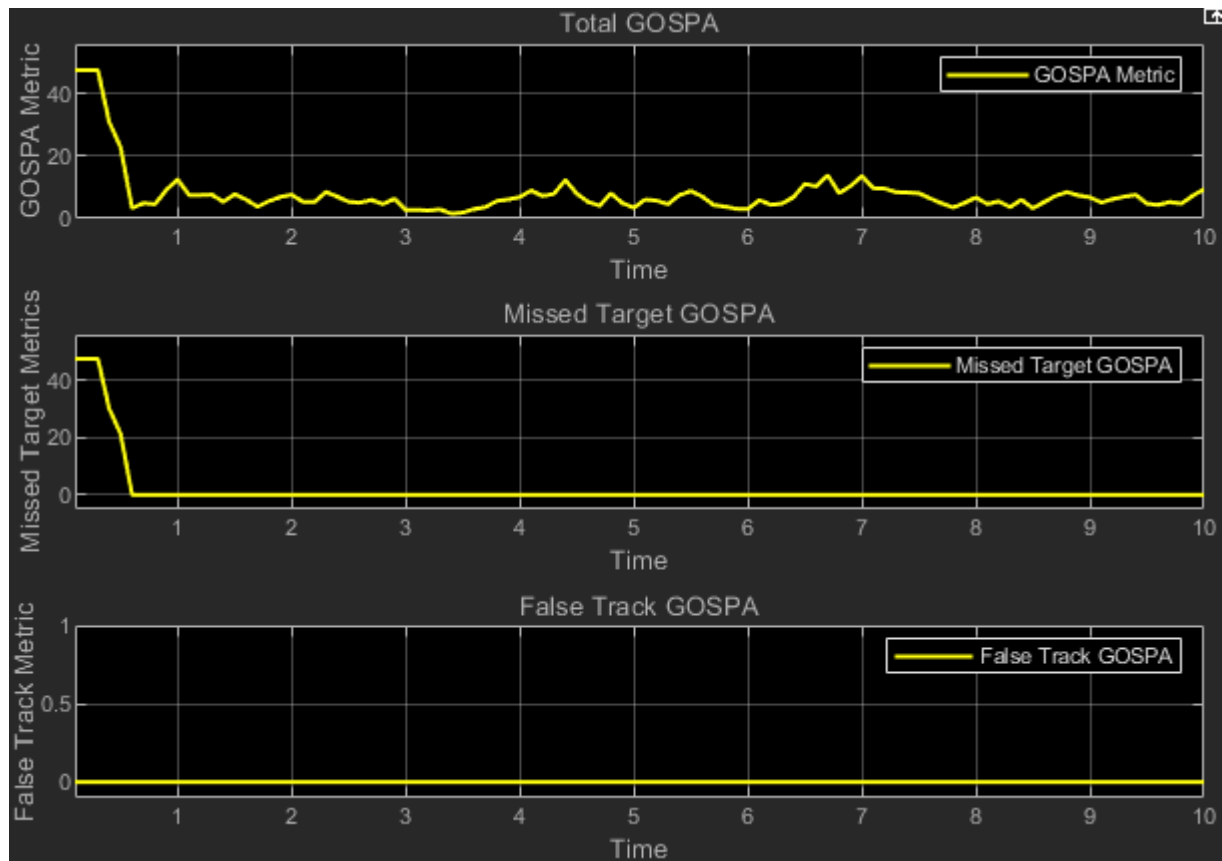
Where m is the number of ground truths and $n(n \geq m)$ is the number of estimated tracks. c is the cutoff distance threshold, and d_b is the base distance between track and truth calculated by a

distance function specified in the Distance property. P is order of the metric and α is the alpha parameter of the metric, defined from block mask.

The lower GOSPA cost represents better tracking performance. A value of zero represents a perfect tracking. You enable Missed target error and False track error components of GOSPA metric as block outputs and visualize them. Before passing the truth information to the GOSPA metric block you remove the sensor platform from the truth information. To do this, you use the Helper Filter Sensor Platform helper block, implemented using MATLAB Function (Simulink) block.

Visualization

In this example the GOSPA metric is visualized using the scope block and the scenario is visualized using Scenario Visualization block. The Scenario Visualization block is implemented using MATLAB System (Simulink) block. Code for this block is defined in the helper class `helperClutterTrackingDisplayBlock`. 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.



Notice that GOSPA metric decreases after a few steps. The initial value of GOSPA metric is higher due to the establishment delay for each track. The GOSPA metric results show that the GM-PHD tracker performed well in the densely cluttered scenario with zero false alarms and zero missed tracks.

```
close_system('TrackPointTargetsInDenseClutterSimulinkExample');
```

Summary

In this example you learned how to use a PHD tracker to track point objects in dense clutter in Simulink. You also learned how to evaluate performance of a tracking algorithm using GOSPA metric and its associated components. The simulation results show that the GM-PHD tracker does not miss targets or create false alarms. The lower overall GOSPA score also indicates desirable tracking performance.

Define and Test Tracking Architectures for System-of-Systems

This example shows how to define the tracking architecture of a system-of-systems that includes multiple detection-level multi-object trackers and track-level fusion algorithms. You can use the tracking architectures to compare different tracking system designs and find the best solution for your system.

Introduction

The “Simulate and Track En-Route Aircraft in Earth-Centered Scenarios” on page 6-564 example shows how to track aircraft using multiple long-range radar systems and fuse data from Automatic Dependent Surveillance Broadcast (ADS-B) transponders to get an accurate air situation picture. In that example, radar detections are fused centrally by a tracker before these tracks are fused with tracks from the ADS-B reports. You first load scenario data from that example and set up a globe-based visualization.

```
% Create the scenario and globe viewer
load('recordedScenario','scenarioData');
viewer = createGlobe(scenarioData);
```

The architecture described above is only one possible architecture and tracking system engineers often consider various other options. There are many factors that determine how a system can be architected, some of them are:

- 1 Sensor outputs: Some sensor manufacturers design their sensors to output detection lists while others track objects internally and only provide sensor tracks as an output.
- 2 Communication networks: The amount of data that is required to be transmitted can be reduced if sensors report tracks instead of detections. Additionally, latency, physical distance, and other limitations may require that sensors report tracks.
- 3 Computational resources: Processing all the detections in a single central tracker usually requires more memory and computations than if processing data distributively between different nodes.

However, there are reasons to prefer central processing over decentralized architectures. First, centralized tracking systems can be optimal, because all the available data is processed in one place and there are no constraints on the data being processed. In addition, the architecture is much simpler as there is only one tracker.

Centralized Air Surveillance Architecture

As a baseline for comparison, you define an architecture that reflects the centralized tracking system described in the “Simulate and Track En-Route Aircraft in Earth-Centered Scenarios” on page 6-564 example. To do that, you first define the same centralized `trackerGNN` for the radars. Note that the radars update every 12 seconds and the ADS-B receiver updates every second. To accommodate different update rates, you wrap the tracker inside a `helperScheduledTracker` object.

```
gnn = trackerGNN(...
    'TrackerIndex',2,...
    'FilterInitializationFcn',@initfilter,...
    'ConfirmationThreshold',[3 5],...
    'DeletionThreshold',[5 5],...
    'AssignmentThreshold',[1000 Inf]);

tracker = helperScheduledTracker(gnn, 12);
```

You define the ADS-B and tracker as two sources that are fused by a `trackFuser`. Note that the `SourceIndex` value for each source configuration must match the `SourceIndex` value of the ADS-B tracks (1) and the `TrackerIndex` value of the tracker (2). Additionally, the `FuserIndex` value must be unique as well. In this case you set it to 3.

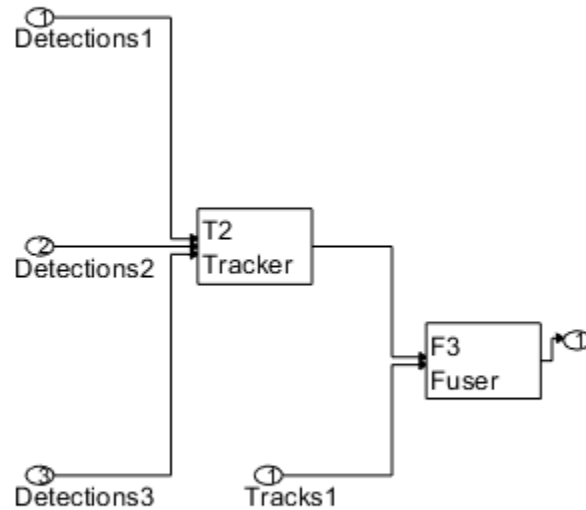
```
sources = {...
    fuserSourceConfiguration("SourceIndex", 1);... % ADS-B
    fuserSourceConfiguration("SourceIndex", 2) ... % Tracker
};

fuser = trackFuser("FuserIndex",3, ...
    "MaxNumSources", 2,...
    "SourceConfigurations", sources, ...
    "AssignmentThreshold", [1000 Inf],...
    "StateFusion", "Intersection",...
    "StateFusionParameters", "trace",...
    "ProcessNoise", 10*eye(3));
```

With the tracker and track fuser defined, you define the `trackingArchitecture`. You add the tracker to it, with sensors 1, 2, and 3 reporting to the tracker. You also add the track fuser to the `trackingArchitecture`. The `trackingArchitecture` directs fuser sources to the fuser based on the fuser's `SoucreConfigurations` property. Finally, you use the `show` object function to display the architecture.

```
centralizedArchitecture = trackingArchitecture;
addTracker(centralizedArchitecture,tracker,'Name','Tracker','SensorIndices',[1 2 3],'ToOutput',f);
addTrackFuser(centralizedArchitecture,fuser,'Name','Fuser','ToOutput',true);
show(centralizedArchitecture)
```

Tracking Architecture: centralizedArchitecture



Run the scenario and track using the tracking architecture

```

% Clear previous viewer data and reset the architecture if previously used
clear(viewer);
plotTruth(scenarioData, viewer);
reset(centralizedArchitecture);

for i = 1:numel(scenarioData)
    % Read the timestamp from the recorded scenario data
    time = scenarioData(i).Time;

    % Read detections from all radars in the scenario
    detections = scenarioData(i).Detections;

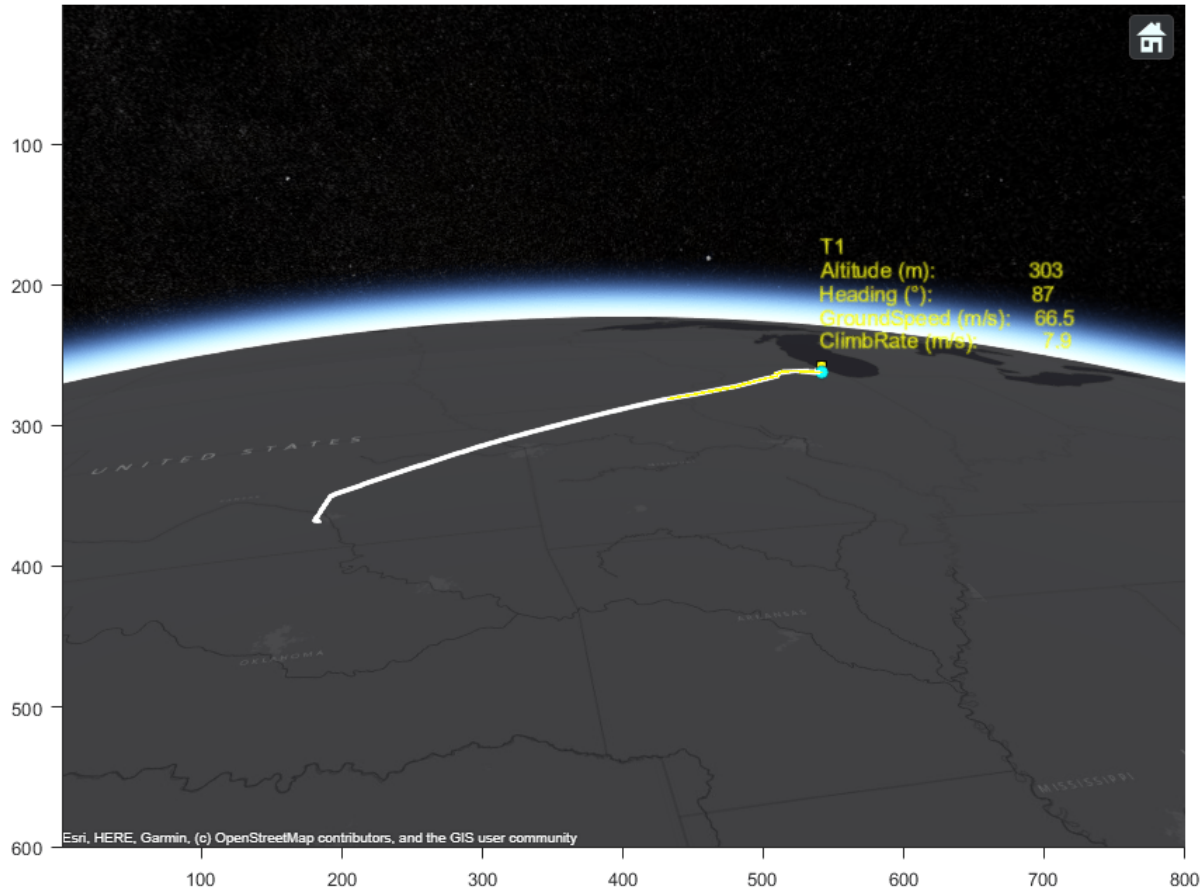
    % Update detections on the globe
    plotDetection(viewer, detections);

    % Read ADS-B tracks
    truePose = scenarioData(i).TruePose;
    adsbTracks = scenarioData(i).ADSBTrack;

    % Pass the detections and ADS-B tracks to the tracking architecture
    tracks = centralizedArchitecture(detections,adsbTracks,time);

    % Update display
    plotTrack(viewer, tracks);
end
  
```

```
% Provide an output image
snap(viewer);
```



Decentralized Architecture

You want to explore how the system works when each radar reports sensor-level tracks that are fused with the ADS-B tracks by a track-level fusion algorithm. You define the `trackingArchitecture` that represents this decentralized architecture.

```
decentralizedArchitecture = trackingArchitecture;
```

Next, you define a tracker for each radar. There are three radar sensors in the original example, so you define three trackers. For simplicity, assume that these trackers are defined in the same way as the central case. Since the `SourceIndex` of the ADS-B tracks is 1, you define the trackers with `TrackerIndex` of 2, 3, and 4. Meanwhile, the radar detections have `SensorIndex` = 1, 2, and 3.

```
sensorTracker = trackerGNN(...
    'TrackerIndex',2,...
    'FilterInitializationFcn',@initfilter,...
    'ConfirmationThreshold',[3 5],...
    'DeletionThreshold',[5 5],...
    'AssignmentThreshold',[1000 Inf]);
```

```

% Add one tracker to each radar
for i = 2:4
    tracker = helperScheduledTracker(clone(sensorTracker),12);
    tracker.TrackerIndex = i; % Specify each radar tracker with a different index
    addTracker(decentralizedArchitecture, tracker, 'Name', strcat('Radar',num2str(i-1)), 'Sensor');
end

```

You add a trackFuser that fuses tracks from four sources, the ADS-B and the three radar trackers.

```

% Define the sources
sources = {...
    fuserSourceConfiguration(1);... % ADS-B
    fuserSourceConfiguration(2);... % Tracker for radar 1
    fuserSourceConfiguration(3);... % Tracker for radar 2
    fuserSourceConfiguration(4);... % Tracker for radar 3
};

% Add the fuser
fuser = trackFuser("FuserIndex", 5, ...
    "MaxNumSources", 4,...
    "SourceConfigurations", sources, ...
    "AssignmentThreshold", [1000 Inf],...
    "StateFusion", "Intersection", ...
    "StateFusionParameters", "trace", ...
    "ProcessNoise", 10*eye(3));
addTrackFuser(decentralizedArchitecture, fuser, 'Name', 'Fuser');

% Show the tracking architecture in a figure
show(decentralizedArchitecture)

```

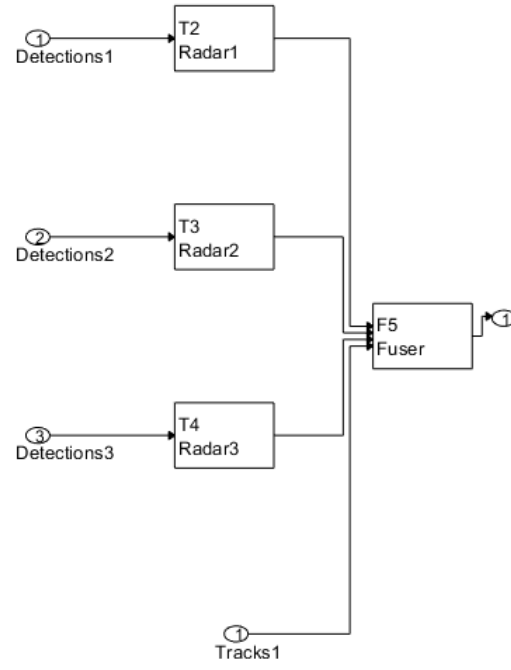
The `replayScenario` function reads the recorded scenario data, steps the new architecture, and displays the results.

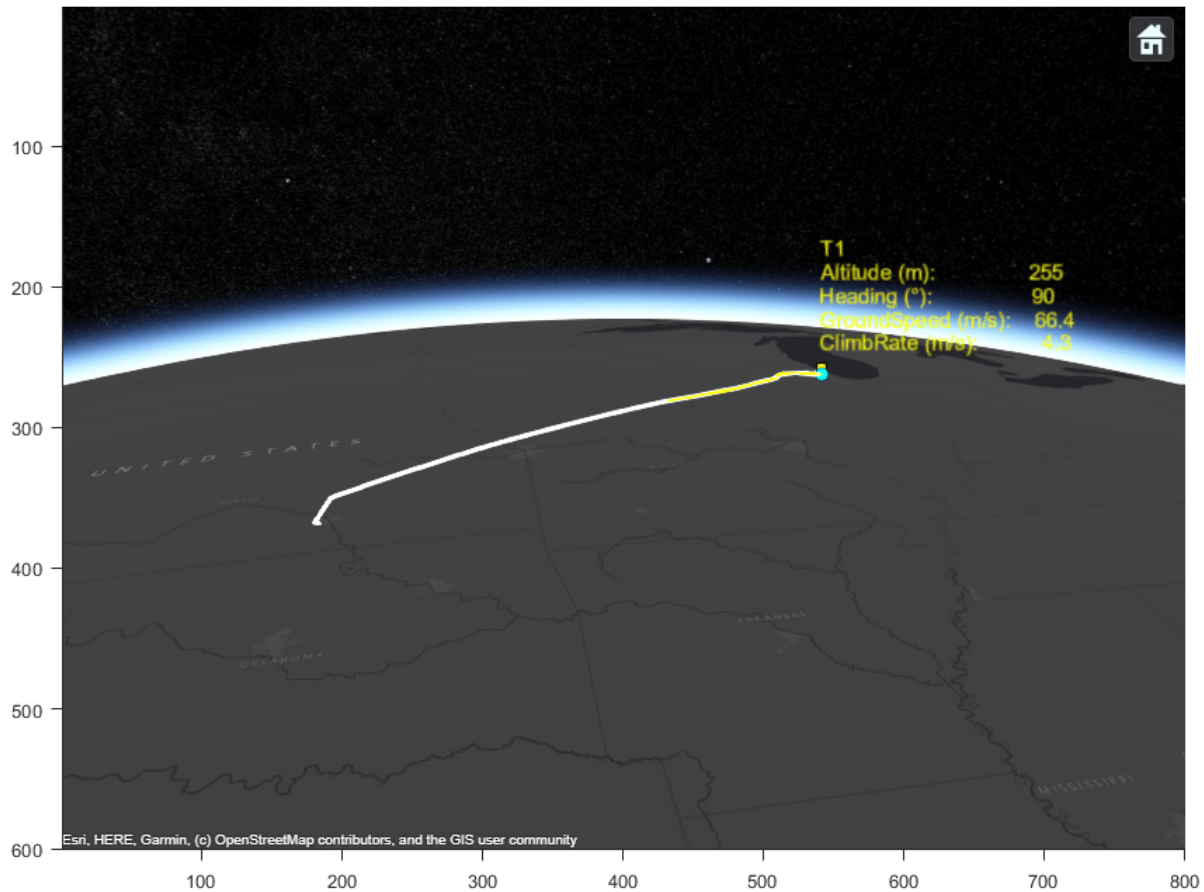
```

% Rerun the scenario and observe tracking results
replayScenario(decentralizedArchitecture,scenarioData,viewer)

```

Tracking Architecture: decentralizedArchitecture





Regional Air Surveillance Architecture

Another possible configuration can fall between the centralized and decentralized architectures above. In many cases, there is no need for an air traffic control center to track aircraft that are too far away from it. In that case, regional air traffic control centers can be used, where one or more radars report to each region.

```
regionalArchitecture = trackingArchitecture;
for i = 2:4
    tracker = helperScheduledTracker(clone(sensorTracker),12);
    tracker.TrackerIndex = i; % Specify each radar with a different index
    addTracker(regionalArchitecture, tracker, 'Name', strcat('Radar',num2str(i-1)), 'SensorIndices', i);
end
```

You define two regions and attach two radar sensors to each. Additionally, each region only fuses ADS-B tracks if the reporting aircraft is close enough to the region.

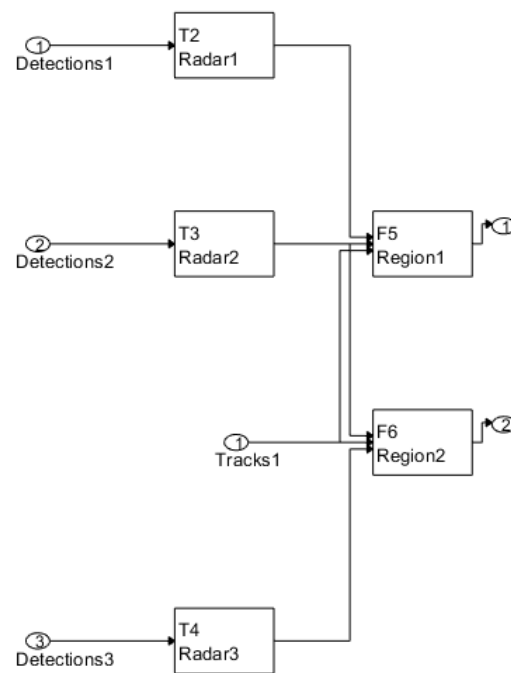
```
region1Fuser = trackFuser('FuserIndex',5, 'MaxNumSources',3, ...
    'SourceConfigurations',sources([1,2,3]),... % Two radars and ADS-B
    'AssignmentThreshold',[1000 Inf],...
    'StateFusion','Intersection',...
    'StateFusionParameters','trace',...
    'ProcessNoise',10*eye(3));
```

```

region2Fuser = trackFuser('FuserIndex',6, 'MaxNumSources',3, ...
    'SourceConfigurations',sources([1,3,4]),... % Two radars and ADS-B
    'AssignmentThreshold',[1000 Inf],...
    'StateFusion','Intersection',...
    'StateFusionParameters','trace',...
    'ProcessNoise',10*eye(3));
addTrackFuser(regionalArchitecture, region1Fuser, 'Name', 'Region1');
addTrackFuser(regionalArchitecture, region2Fuser, 'Name', 'Region2');
show(regionalArchitecture)

```

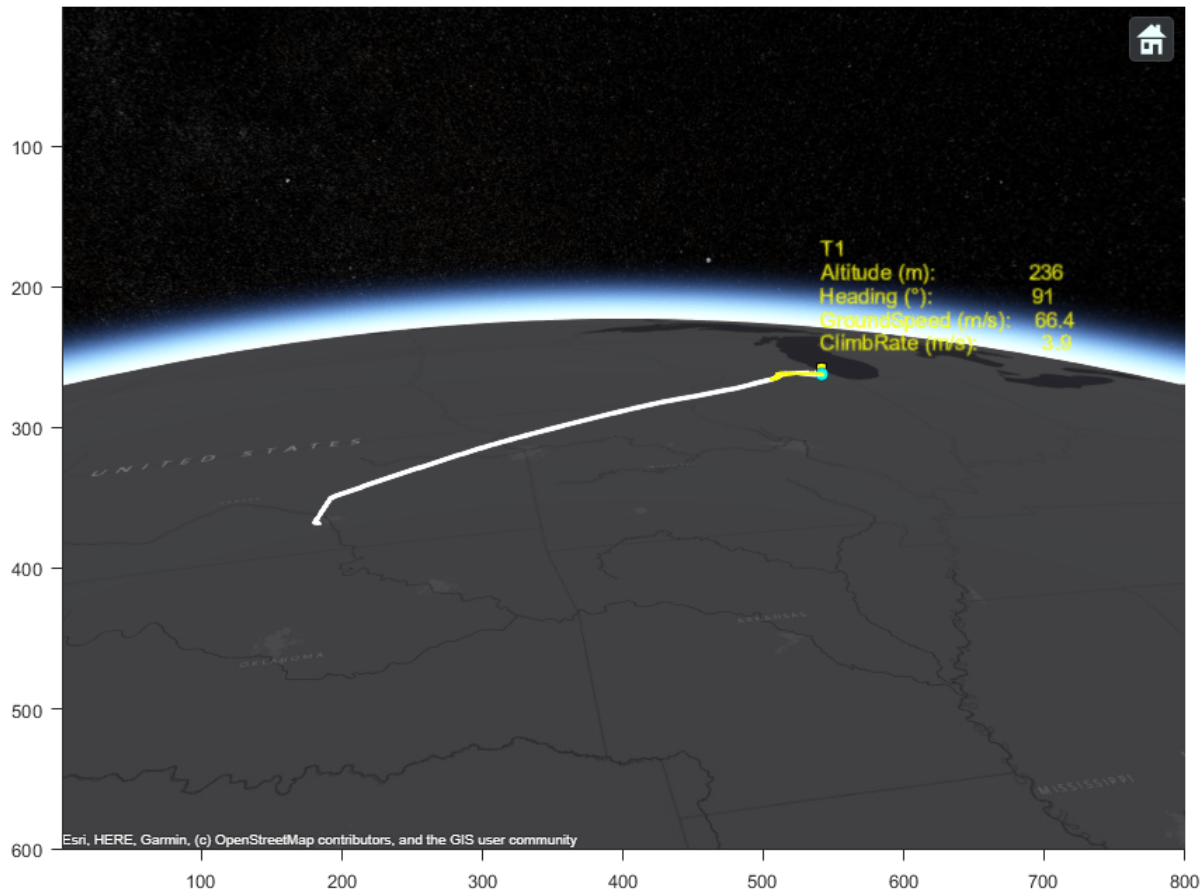
Tracking Architecture: regionalArchitecture



```

% Rerun the scenario and observe tracking results
replayScenario(regionalArchitecture,scenarioData,viewer)

```



Summary

In this example, you learned how to define tracking architectures for system-of-systems that use multiple trackers and track-to-track fusion algorithms. You saw how to pass detections and tracks as inputs to the tracking architecture and how to output tracks from it.

Supporting Functions

createGlobe Creates the globe visualization

```
function gl = createGlobe(scenarioData)
% 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 flight route
plotTruth(scenarioData, gl);
end
```

plotTruth plots the ground truth position

```
function plotTruth(scenarioData, viewer)
poses = [scenarioData.TruePose];
positions = vertcat(poses.Position);
plotLines(viewer, positions(:,1), positions(:,2), positions(:,3))
end
```

initfilter defines the extended Kalman filter used by the tracker

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 = 10*filter.StateCovariance; % initcvekf uses measurement noise as the de
filter.ProcessNoise = eye(3);
end
```

replayScenario Replays the recorded scenario and generates results

```
function replayScenario(arch,scenarioData,viewer)
clear(viewer);
plotTruth(scenarioData, viewer);
reset(arch);
numFusers = numel(arch.TrackFusers);
tracks = cell(1,numFusers);
for i = 1:numel(scenarioData)
    time = scenarioData(i).Time;

    % Create detections from all radars in the scenario
    detections = scenarioData(i).Detections;

    % Update detections on the globe
    plotDetection(viewer, detections);

    % Generate ADS-B tracks
    adsbTracks = scenarioData(i).ADSBTrack;

    % Pass the detections and ADS-B tracks to the tracking architecture
    [tracks{1:numFusers}] = arch(detections,adsbTracks,time);

    % Update display
    plotTrack(viewer, vertcat(tracks{:}));
end
snap(viewer);
end
```

Estimate Phone Orientation Using Sensor Fusion

MATLAB Mobile™ reports sensor data from the accelerometer, gyroscope, and magnetometer on Apple or Android mobile devices. Raw data from each sensor or fused orientation data can be obtained. This examples shows how to compare the fused orientation data from the phone with the orientation estimate from the `ahrsfilter` (Navigation Toolbox) object.

Read the Accelerometer, Gyroscope, Magnetometer, and Euler Angles

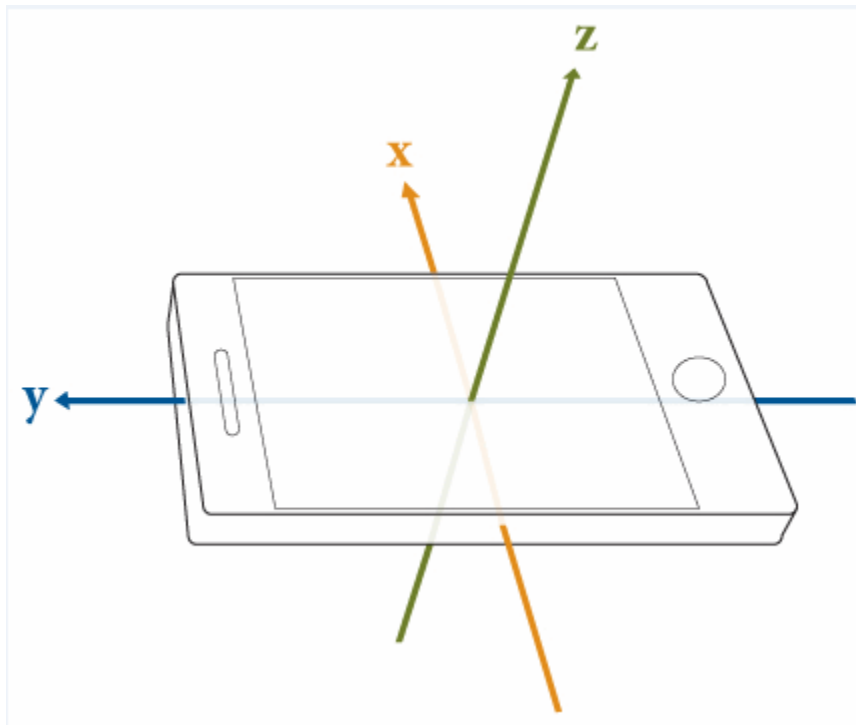
Read the logged phone sensor data. The MAT-file `samplePhoneData.mat` contains sensor data logged on an iPhone at a 100 Hz sampling rate. To run this example with your own phone data, refer to “Sensor Data Collection with MATLAB Mobile or MATLAB Online”.

```
matfile = 'samplePhoneData.mat';
SampleRate = 100; % This must match the data rate of the phone.
```

```
[Accelerometer, Gyroscope, Magnetometer, EulerAngles] ...
    = exampleHelperProcessPhoneData(matfile);
```

Convert to North-East-Down (NED) Coordinate Frame

MATLAB Mobile uses the convention shown in the following image. To process the sensor data with the `ahrsfilter` object, convert to NED, a right-handed coordinate system with clockwise motion around the axes corresponding to positive rotations. Swap the x- and y-axis and negate the z-axis for the various sensor data. Note that the accelerometer readings are negated since the readings have the opposite sign in the two conventions.



```
Accelerometer = -[Accelerometer(:,2), Accelerometer(:,1), -Accelerometer(:,3)];
Gyroscope = [Gyroscope(:,2), Gyroscope(:,1), -Gyroscope(:,3)];
```

```
Magnetometer = [Magnetometer(:,2), Magnetometer(:,1), -Magnetometer(:,3)];
qTrue = quaternion([EulerAngles(:,3), -EulerAngles(:,2), EulerAngles(:,1)], ...
    'eulerd', 'ZYX', 'frame');
```

Correct Phone Initial Rotation

The phone may have a random rotational offset. Without knowing the offset, you cannot compare the `ahrsfilter` object and the phone results. Use the first four samples to determine the rotational offset, then rotate the phone data back to desirable values.

```
% Get a starting guess at orientation using ecompass. No coefficients
% required. Use the initial orientation estimates to figure out what the
% phone's rotational offset is.
q = ecompass(Accelerometer, Magnetometer);
```

```
Navg = 4;
qfix = meanrot(q(1:Navg))./meanrot(qTrue(1:Navg));
Orientation = qfix*qTrue; % Rotationally corrected phone data.
```

Tune the AHRS Filter

To optimize the noise parameters for the phone, tune the `ahrsfilter` object. The parameters on the filter need to be tuned for the specific IMU on the phone that logged the data in the MAT-file. Use the `tune` (Navigation Toolbox) function with the logged orientation data as ground truth.

```
orientFilt = ahrsfilter('SampleRate', SampleRate);
groundTruth = table(Orientation);
sensorData = table(Accelerometer, Gyroscope, Magnetometer);

tc = tunerconfig('ahrsfilter', "MaxIterations", 30, ...
    'Objectivelimit', 0.001, 'Display', 'none');
tune(orientFilt, sensorData, groundTruth, tc);
```

Fuse Sensor Data With Filter

Estimate the device orientation using the tuned `ahrsfilter` object.

```
reset(orientFilt);
qEst = orientFilt(Accelerometer, Gyroscope, Magnetometer);
```

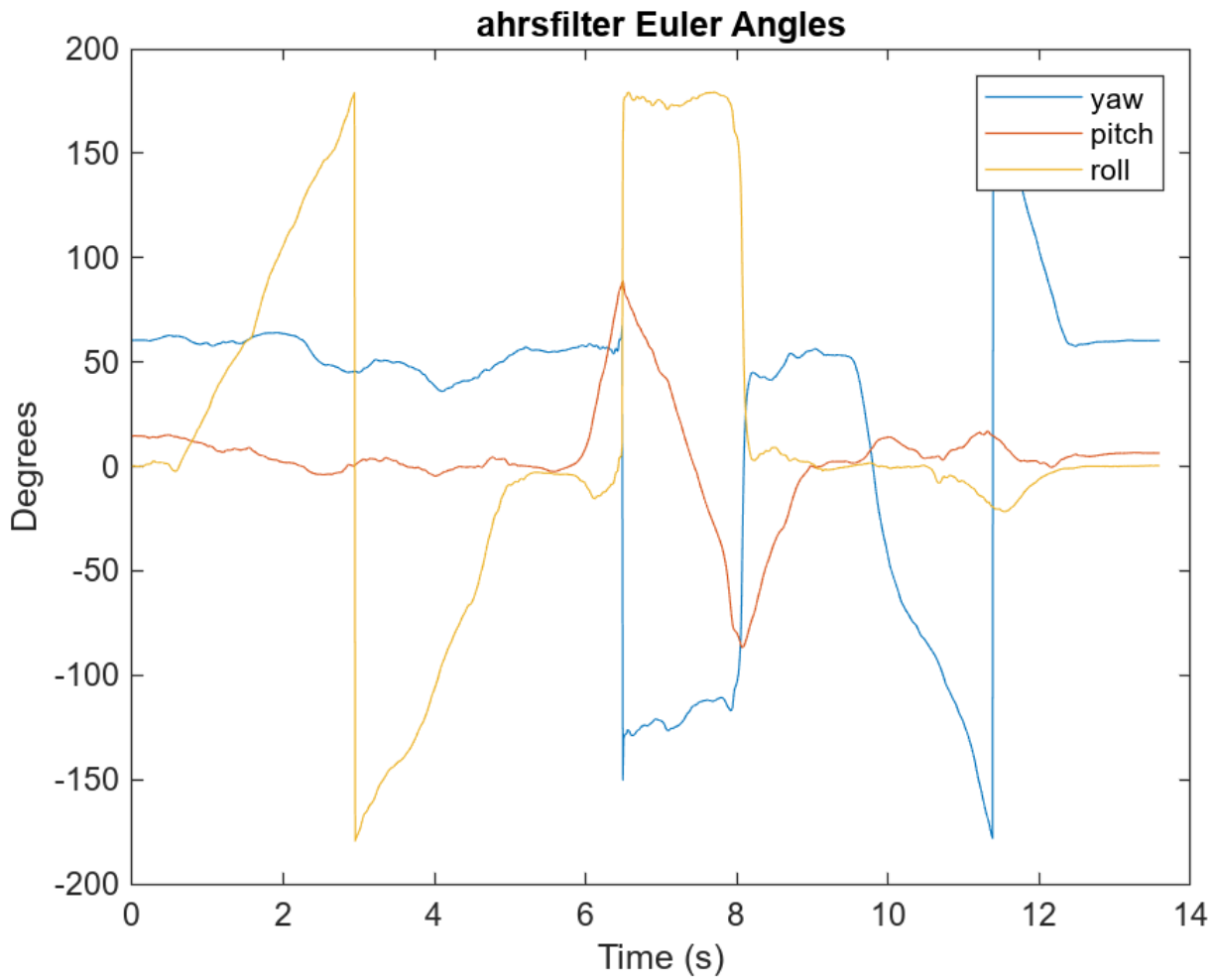
Plot Results

Plot the Euler angles for each orientation estimate and the quaternion distance between the two orientation estimates. Quaternion distance is measured as the angle between two quaternions. This distance can be used as an error metric for orientation estimation.

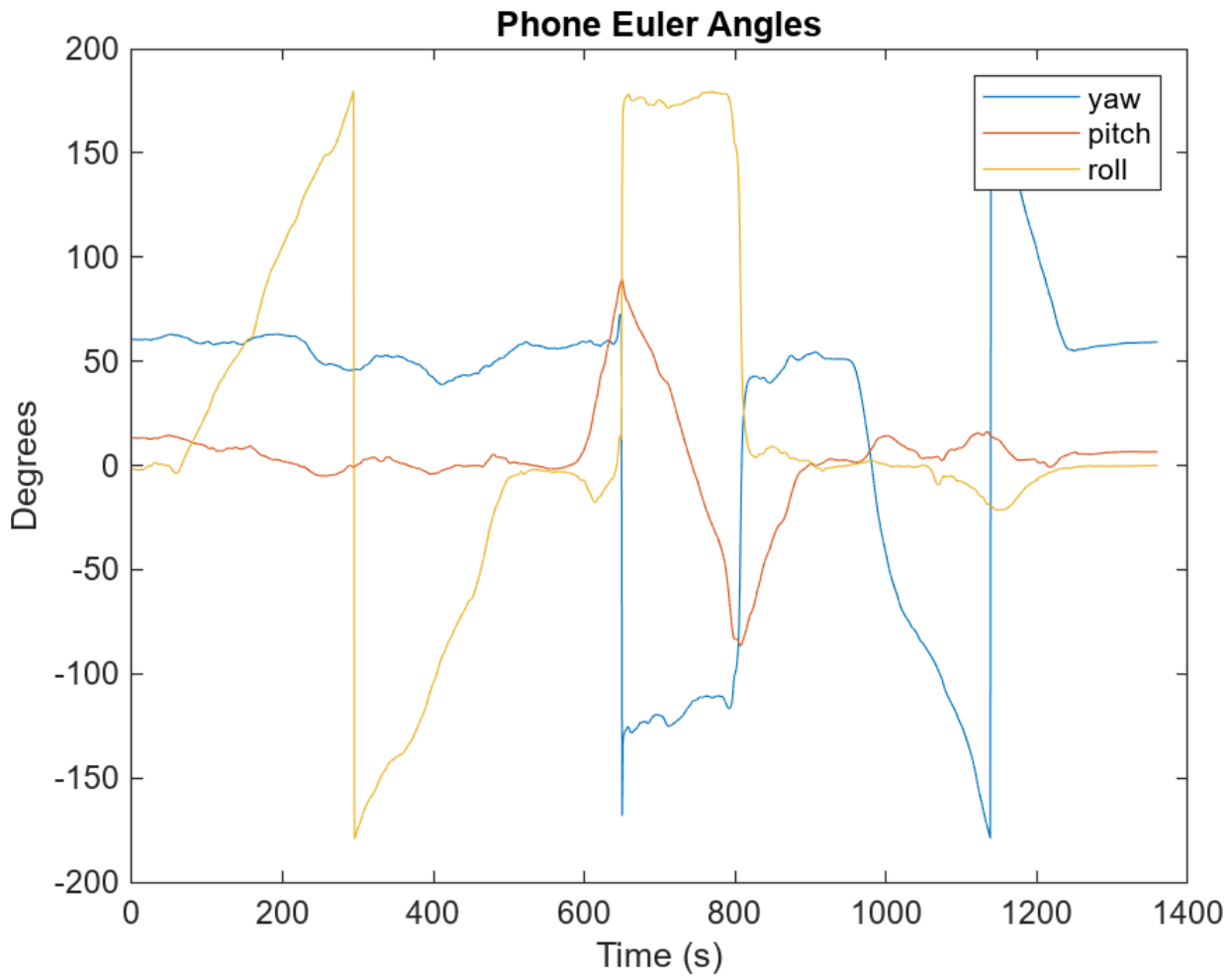
```
numSamples = numel(Orientation);
t = (0:numSamples-1).'/SampleRate;

d = rad2deg(dist(qEst, Orientation));

figure
plot(t, eulerd(qEst, 'ZYX', 'frame'))
legend yaw pitch roll
title('ahrsfilter Euler Angles')
ylabel('Degrees')
xlabel('Time (s)')
```



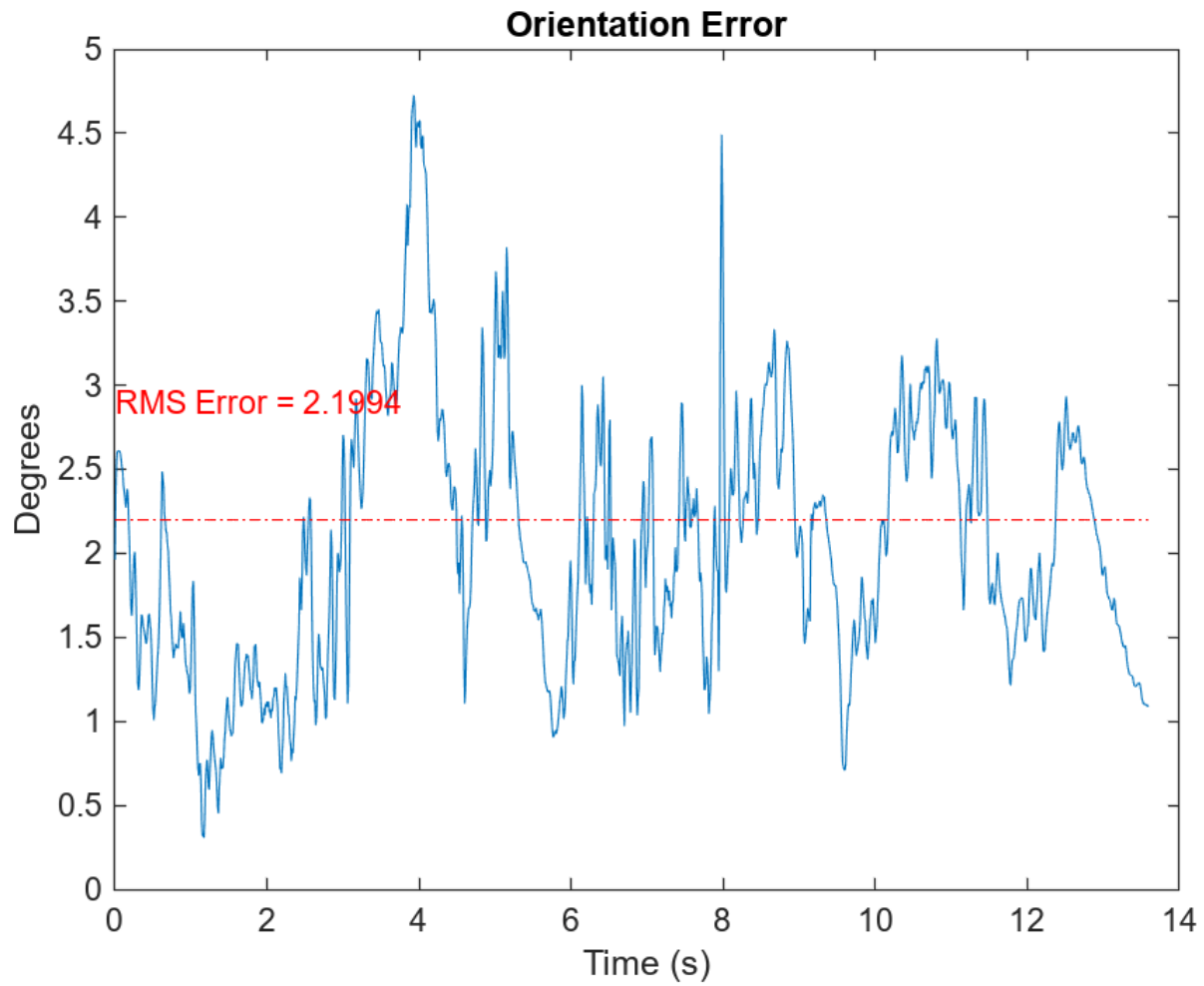
```
figure
plot(eulerd(Orientation, 'ZYX', 'frame'))
legend yaw pitch roll
title('Phone Euler Angles')
ylabel('Degrees')
xlabel('Time (s)')
```



```

figure
plot(t, d)
title('Orientation Error')
ylabel('Degrees')
xlabel('Time (s)')
% Add RMS error
rmsval = sqrt(mean(d.^2));
line(t, repmat(rmsval,size(t)), 'LineStyle', '-.', 'Color', 'red');
text(t(1), rmsval + 0.7, "RMS Error = " + rmsval, 'Color', 'red')

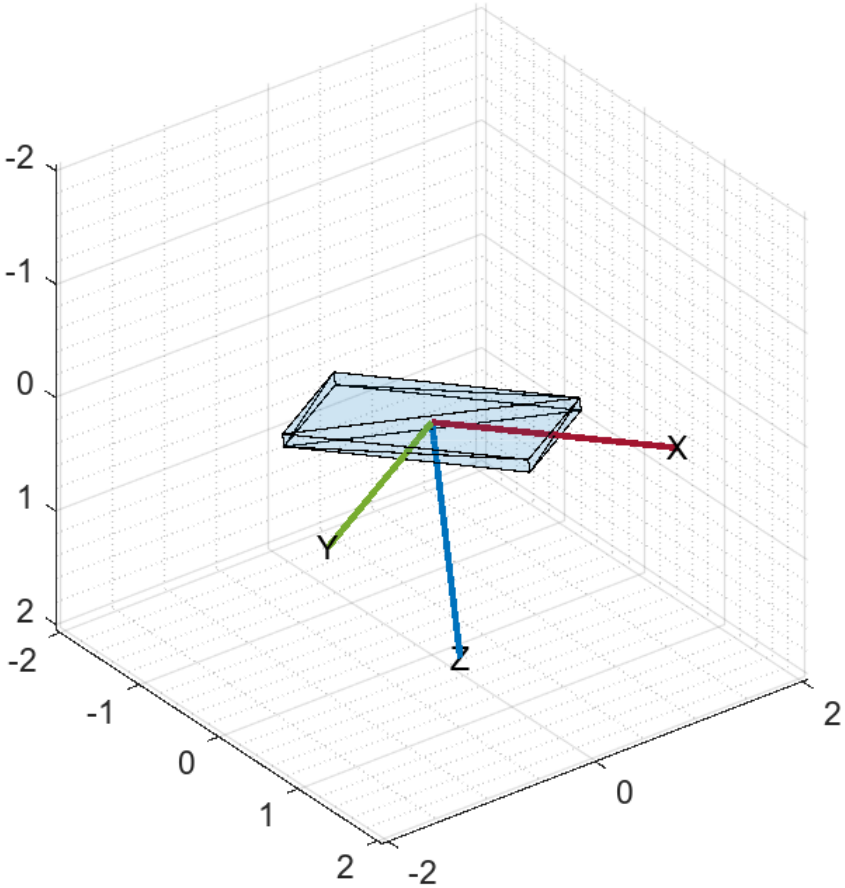
```

Use `poseplot` to view the orientation estimates of the phone as a 3-D rectangle.

```
figure
pp = poseplot("MeshFileName", "phoneMesh.stl");

for i = 1:numel(qEst)
    set(pp, "Orientation", qEst(i));
    drawnow
end
```



Handle Out-of-Sequence Measurements in Multisensor Tracking Systems

This example shows how to handle out-of-sequence measurements (OOSM) in a multisensor tracking system. The example compares tracking results when OOSM are present using various handling techniques. For more information about OOSM handling techniques see “Handle Out-of-Sequence Measurements with Filter Retrodiction” on page 6-771 example.

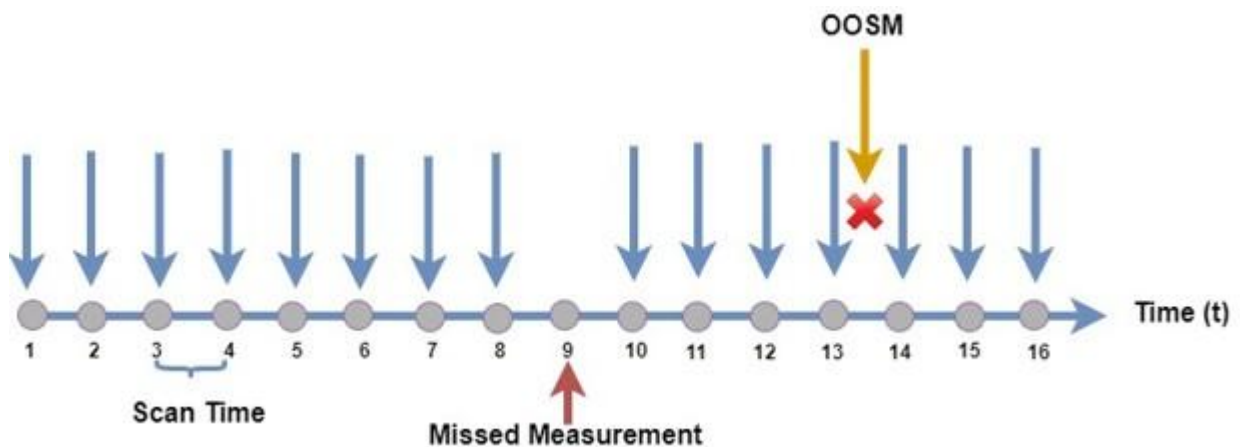
Introduction

In a multisensor system, when multiple sensors report to the same tracker, the measurements may arrive at the tracker with a time delay relative to the time when they are generated by the sensor. The delay can be caused by any of the following reasons:

- 1 The sensor may require a significant amount of time to process the data. For example, a vision sensor may require tens of milliseconds to detect objects in a frame it captures.
- 2 If the sensor and the tracker are connected by a network, there may be a communication delay.
- 3 The tracker may update at a different rate from the sensor scan rate. For example, if the tracker is updated just before the sensor measurements arrive, these measurements are considered as out-of-sequence.

There are various OOSM handling techniques. In this example, you explore two techniques:

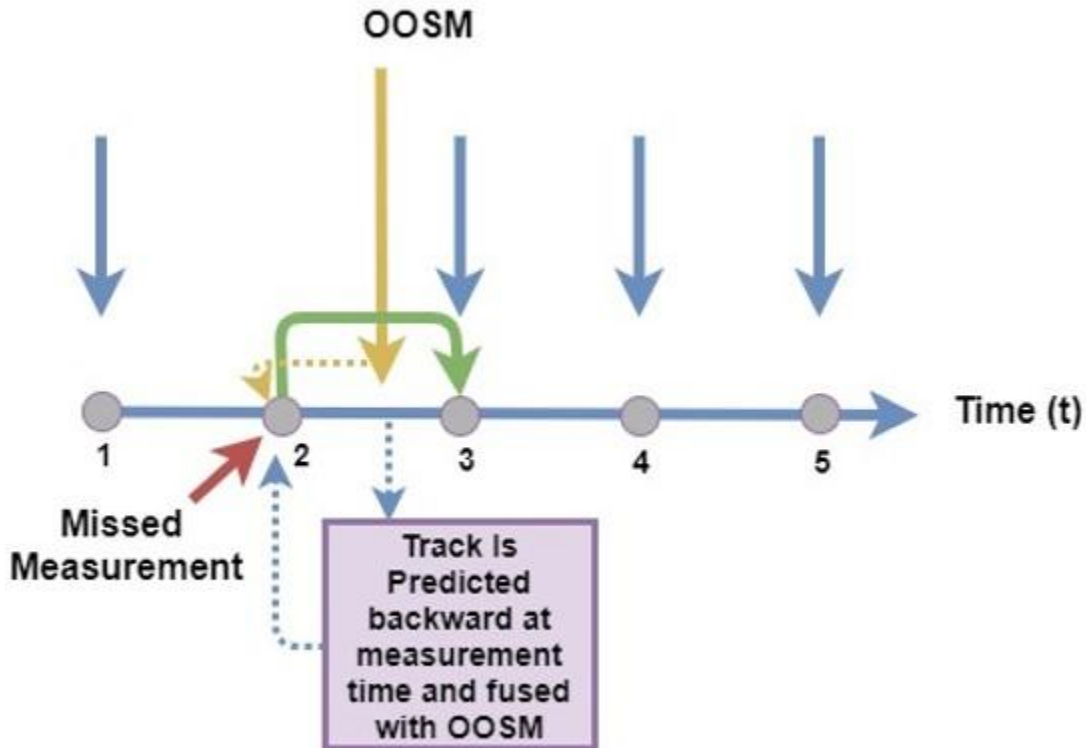
Neglect: In this technique, any OOSM is simply ignored and not used to update the tracker, as shown in the figure below. This technique is the easiest and is useful in cases where the OOSM is not expected to contain data that would significantly modify the filter state and uncertainty. It is also the most efficient technique, in terms of memory and processing.



Retrodiction: In this technique, the tracker saves its state for the last n steps. When a new set of detections is sent to the tracker, the tracker:

- Uses the detection time to classify the detections as: in-sequence (after the last tracker update time) or out-of-sequence (before the last tracker update time).
- The tracker neglects any detections with timestamps that are older than the tracker “MaxNumOOSMSteps” property allows.

- The tracker retrodict existing tracks to the time of the out-of-sequence detections and calculates the cost of assignment for each combination of track and detection.
- The tracker attempts to assign the out-of-sequence detections to the tracks.
- The tracker retrodicts the track to the time of each assigned out-of-sequence detection and retroCorrect the track with the OOSM, see the picture below.
- The tracker initializes new tracks from unassigned out-of-sequence detections.
- Then, the tracker resumes its usual processing with the in-sequence detections.



The retrodiction technique is more expensive in terms of memory and processing time than the neglect technique. However, it is important to handle OOSM in the following cases:

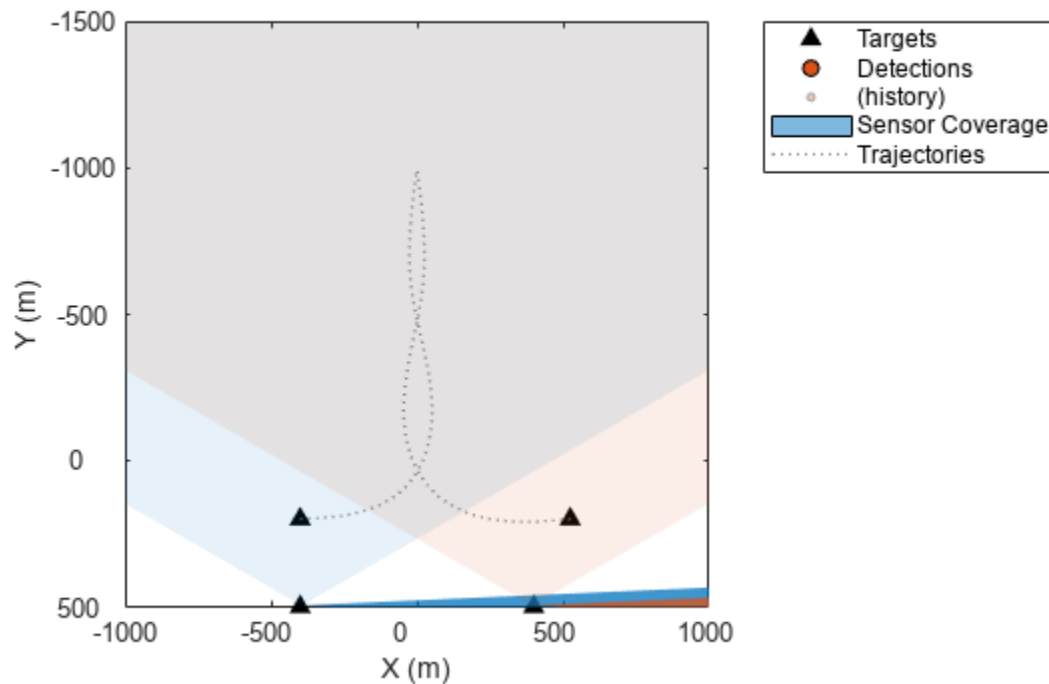
- The sensor that provides OOSM covers areas that are not covered by any other sensor. In that case, neglecting the OOSM would result in complete loss of coverage in the area.
- The OOSM contain novel information that other sensors cannot provide. For example, if the OOSM are provided by the only sensor that reports object classification, neglecting the OOSM would result in tracks that have no classification data.
- The sensors that report to the tracker operate at a low scan rate. In that case, every detection is important and neglecting OOSM would result in low tracking accuracy.

Define the Scenario and Sensor Lag

To explore OOSM handling techniques and their impact on tracking, you create a simple scenario. In the scenario, there are two moving platforms that approach from left and right to the middle of the scenario, where they move close to each other. There are two radars, one on the left and one on the right. The radar on the left reports detections directly to the tracker with no time delay. The radar on the right reports detections through a network that has a time delay. Note that initially only the radar on the right covers the platform on the right until it reaches an area covered by the left sensor.

The following lines of code create and visualize the scenario.

```
scenario = createScenario();
[tp, platp, trajp, detp, covp, trp] = createPlotters(scenario);
```



In this code section you define the time delay between the radar on the right and the tracker. You use the `objectDetectionDelay` object to add the network time delay to data coming from the right radar sensor, with `SensorIndex = 2`, to the tracker. Use the slider on line 5 to select the time delay. Valid values range from 0 (no delay) to 3 seconds with a default value of 2 seconds. Once you set the value on the slider, the radar updates its time delay.

```
% Set the time delay of the sensor
sensorDelay = objectDetectionDelay(SensorIndices = 2, DelaySource = "Property", DelayDistribution
sensorDelay.DelayParameters = 2 
sensorDelay =
  objectDetectionDelay with properties:
    SensorIndices: 2
    Capacity: Inf
    DelaySource: 'Property'
    DelayDistribution: 'Constant'
    DelayParameters: 2
```

Define the Tracker and OOSM Handling Technique

Both `trackerGNN` and `trackerJPDA` System objects allow you to choose the OOSM handling technique by setting the `OOSMHandling` property. Use the first drop down below to choose between GNN and JPDA. Then use the next drop down to choose between "Neglect" and "Retrodiction". Once selected, the tracker uses the value you choose.

```

tracker =  ;
tracker.OOSMHandling = 

tracker =
  trackerGNN with properties:

        TrackerIndex: 0
  FilterInitializationFcn: @initfilter
        MaxNumTracks: 10
        MaxNumDetections: Inf
        MaxNumSensors: 20

        Assignment: 'MatchPairs'
  AssignmentThreshold: [30 Inf]
  AssignmentClustering: 'off'

        OOSMHandling: 'Retrodiction'
        MaxNumOOSMSteps: 3

        TrackLogic: 'History'
  ConfirmationThreshold: [2 4]
  DeletionThreshold: [5 5]

  HasCostMatrixInput: false
  HasDetectableTrackIDsInput: false
        StateParameters: [1x1 struct]

        NumTracks: 0
  NumConfirmedTracks: 0

        ClassFusionMethod: 'None'

  EnableMemoryManagement: false

```

Run the Scenario and Analyze Metrics

In this section, you run the scenario with the time delay you set for the right radar and the OOSM handling technique you defined for the tracker.

You define a Generalized Optimal Sub-Pattern Association (GOSPA) metric to evaluate the tracker performance in terms of overall GOSPA, localization, missed targets, and false tracks. For more information about GOSPA, see `trackGOSPAMetric`.

```

gospa = trackGOSPAMetric;
lgospa = zeros(1,33);
localization = zeros(1,33);
missTarget = zeros(1,33);

```

```
falseTracks = zeros(1,33);
i = 0;
```

The next code section initializes the scenario and the visualization, resets the tracker, and sets the random seed for repeatable results.

```
s = rng(2021, "twister");
h = onCleanup(@() rng(s));
delayedDets = {};
detBuffer = {};
restart(scenario);
reset(tracker);
clearPlotterData(tp);
plotTrueTrajectories(trajp, scenario);
```

Next, you run the main simulation loop. You can see the results in the figure below the block of code. Because the tracker is connected to the left radar, it updates when the left radar finishes a scan, or every 1.7 seconds.

```
% Main simulation loop
while advance(scenario) && ishghandle(tp.Parent)
    % Generate sensor data
    [dets, configs, sensorConfigPIDs] = detect(scenario);

    % Apply time delay to the detections
    time = scenario.SimulationTime;
    if isLocked(sensorDelay) || ~isempty(dets)
        delayedDets = sensorDelay(dets,time);
    end

    detBuffer = vertcat(detBuffer, delayedDets); %#ok<AGROW>

    [truePosition, meas, measCov] = readData(scenario, detBuffer);

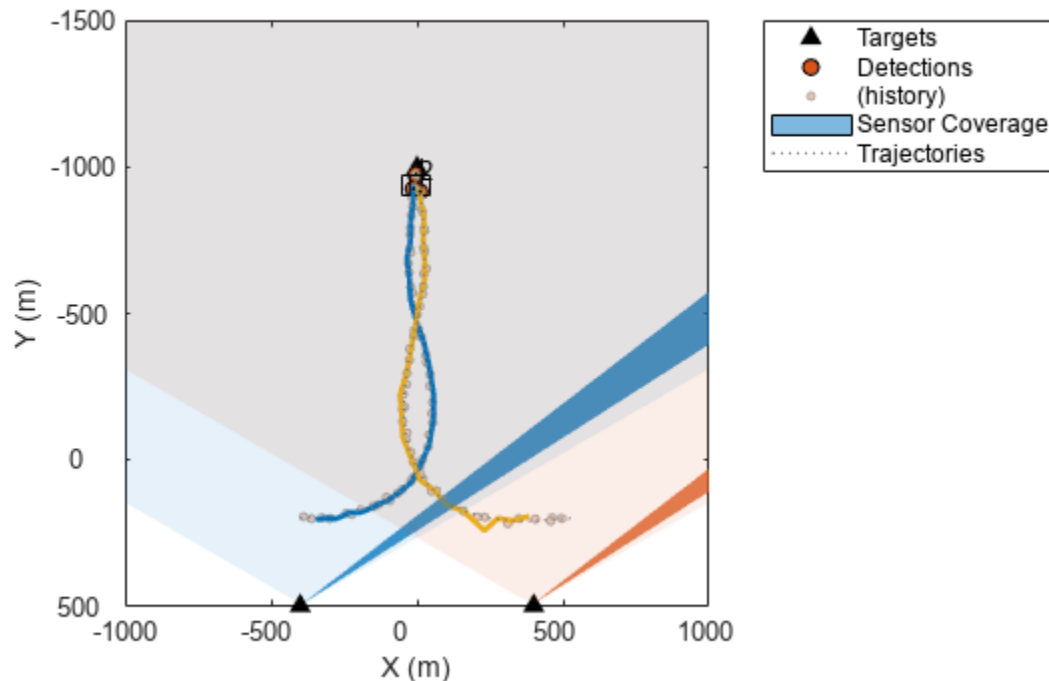
    % Tracker update
    if configs(1).IsScanDone
        if isLocked(tracker) || ~isempty(detBuffer)
            [tracks,~,~,info] = tracker(detBuffer, time);
        else
            tracks = objectTrack.empty;
        end
        detBuffer = {};

        % Update the trackPlotter
        posSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 1 0];
        trpos = getTrackPositions(tracks, posSelector);
        plotTrack(trp, trpos, string([tracks.TrackID]));

        % Update GOSPA metric
        i = i + 1;
        truths = platformPoses(scenario);
        [lgospa(i), ~, ~, localization(i), missTarget(i), falseTracks(i)] = gospa(tracks,truths(i));
    end

    % Update plots
    plotPlatform(platp,truePosition);
    plotDetection(detp,meas,measCov);
    plotCoverage(covp,coverageConfig(scenario));
```

```
drawnow limitrate
end
```



You want to analyze the results of the tracker for the selected radar time delay. The following code block shows the four GOSPA metrics. Remember that a lower GOSPA metric value indicates better tracking.

For the default selection of 2 seconds radar time delay and "Retrodiction" OOSM handling technique, the GOSPA metrics show that there are no false tracks and that both platforms are being tracked after 3 tracker updates. After the third update, only the localization errors contribute to the overall GOSPA metric.

Use the time delay slider and the tracker OOSM handling drop down menu in the previous section to choose other combinations and compare them with these values. For example, if the tracker OOSM handling is set to "Neglect", and the radar data delay is 2 seconds, the platform on the right will not be tracked until it enters the coverage area of the left radar. As a result, the missed targets GOSPA remains high for the first part of the scenario until the platform enters the coverage area of the left radar sensor.

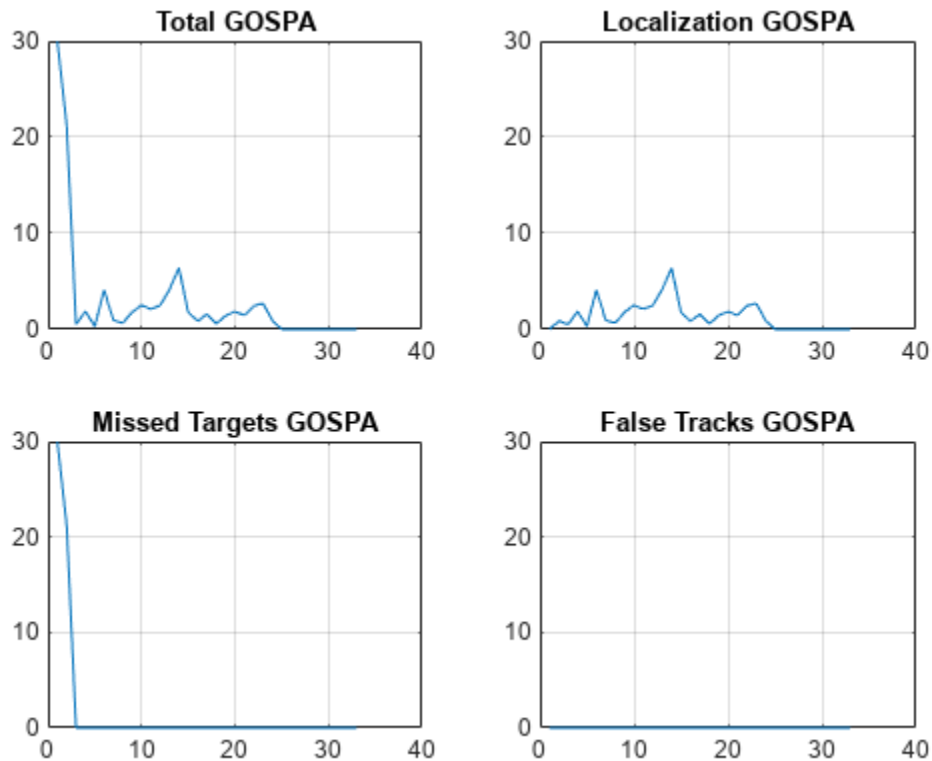
```
% Display GOSPA metrics
figure;
subplot(2,2,1),plot(lgospa);
ylim([0 30]);
grid
title("Total GOSPA")
subplot(2,2,2),plot(localization);
```



```

ylim([0 30]);
grid
title("Localization GOSPA")
subplot(2,2,3),plot(missTarget);
ylim([0 30]);
grid
title("Missed Targets GOSPA")
subplot(2,2,4),plot(falseTracks);
ylim([0 30]);
grid
title("False Tracks GOSPA");

```



Summary

In this example you learned the importance of handling out-of-sequence measurements (OOSM). You used the retrodiction handling to process detections that arrived late to the tracker and compared the results of retrodiction to the results of neglecting the OOSM.

Supporting Functions

The `readData` function prepares detection and platform data to update the plotters.

```

function [position, meas, measCov] = readData(scenario,dets)
allDets = [dets{:}];

if ~isempty(allDets)
    % extract column vector of measurement positions
    meas = cat(2,allDets.Measurement)';

```

```

    % extract measurement noise
    measCov = cat(3,allDets.MeasurementNoise);
else
    meas = zeros(0,3);
    measCov = zeros(3,3,0);
end

truePoses = platformPoses(scenario);
position = vertcat(truePoses(:).Position);
end

```

The createPlotters function creates the plotters and sets up the initial display.

```

function [tp, platp, trajp, detp, covp, trp] = createPlotters(scenario)
% Create plotters
tp = theaterPlot(XLim = [-1000 1000], YLim = [-1500 500], ZLim = [-1500 200]);
set(tp.Parent, YDir = "reverse", ZDir = "reverse");

% Change to 2-D view
view(2)

platp = platformPlotter(tp, DisplayName = "Targets", MarkerFaceColor = "k");
detp = detectionPlotter(tp, DisplayName = "Detections", MarkerSize = 6, MarkerFaceColor = [0.85 0.85 0.85]);
covp = coveragePlotter(tp, DisplayName = "Sensor Coverage");
trp = trackPlotter(tp, ConnectHistory = "on", ColorizeHistory = "on");

% Plot ground truth trajectories for the moving objects
poses = platformPoses(scenario);
plotPlatform(platp, vertcat(poses.Position));
trajp = trajectoryPlotter(tp, DisplayName = "Trajectories", LineWidth = 1);
plotTrueTrajectories(trajp, scenario);

% Plot coverage
covcon = coverageConfig(scenario);
plotCoverage(covp,covcon);
end

```

The createScenario function creates the scenario, platforms, and radars.

```

function scenario = createScenario
% Create Scenario
scenario = trackingScenario;
scenario.StopTime = Inf;
scenario.UpdateRate = 0;

% Create platforms
Tower = platform(scenario, ClassID = 3);
Tower.Dimensions = struct( ...
    Length = 10, ...
    Width = 10, ...
    Height = 60, ...
    OriginOffset = [0 0 30]);
Tower.Trajectory.Position = [-400 500 0];

Tower1 = platform(scenario, ClassID = 3);
Tower1.Dimensions = struct( ...

```

```

    Length = 10, ...
    Width = 10, ...
    Height = 60, ...
    OriginOffset = [0 0 30]);
Tower1.Trajectory.Position = [400 500 0];

Plane = platform(scenario, ClassID = 1);
Plane.Dimensions = struct( ...
    Length = 1, ...
    Width = 1, ...
    Height = 1, ...
    OriginOffset = [0 0 0]);
Plane.Signatures = {...
    rcsSignature(...
        Pattern = [20 20;20 20], ...
        Azimuth = [-180 180], ...
        Elevation = [-90;90], ...
        Frequency = [0 1e+20])});
Plane.Trajectory = waypointTrajectory( ...
    [-400 200 0;-50 100 0;-20 -600 0; 0 -1000 0], ...
    [0;22;45;60], ...
    AutoPitch = true, ...
    AutoBank = true);

Plane1 = platform(scenario,ClassID = 1);
Plane1.Dimensions = struct( ...
    Length = 1, ...
    Width = 1, ...
    Height = 1, ...
    OriginOffset = [0 0 0]);
Plane1.Signatures = {...
    rcsSignature(...
        Pattern = [20 20;20 20], ...
        Azimuth = [-180 180], ...
        Elevation = [-90;90], ...
        Frequency = [0 1e+20])});
Plane1.Trajectory = waypointTrajectory( ...
    [525 200 0;50 100 0;20 -600 0; 0 -1000 0], ...
    [0;22;45;60], ...
    AutoPitch = true, ...
    AutoBank = true);

% Create sensors
Sector = fusionRadarSensor(SensorIndex = 1, ...
    UpdateRate = 10, ...
    MountingLocation = [4.85 -4.98 0], ...
    MountingAngles = [0 0 0], ...
    FieldOfView = [5 1], ...
    HasINS = true, ...
    ReferenceRange = 1000, ...
    DetectionCoordinates = "Scenario", ...
    MechanicalAzimuthLimits = [-150 -30]);

Sector1 = fusionRadarSensor(SensorIndex = 2, ...
    UpdateRate = 10, ...
    MountingLocation = [-5.01 -5.04 0], ...
    FieldOfView = [5 1], ...

```

```
HasINS = true, ...  
ReferenceRange = 1000, ...  
DetectionCoordinates = "Scenario", ...  
MechanicalAzimuthLimits = [-150 -30]);
```

```
% Assign sensors to platforms  
Tower.Sensors = Sector;  
Tower1.Sensors = Sector1;  
end
```

The `initfilter` function initializes a nearly constant velocity tracking EKF filter configured to allow a higher process noise.

```
function ekf = initfilter(detection)  
ekf = initcvekf(detection);  
ekf.ProcessNoise = ekf.ProcessNoise*9;  
end
```

The `plotTrueTrajectories` function plots ground truth trajectories on the theater plot.

```
function plotTrueTrajectories(trajp, scenario)  
trajpos = cell(1,2);  
for i = 3:4  
    trajpos{i-2} = lookupPose(scenario.Platforms{i}.Trajectory, (0:0.1:60));  
end  
plotTrajectory(trajp, trajpos);  
end
```

Handle Out-of-Sequence Measurements with Filter Retrodiction

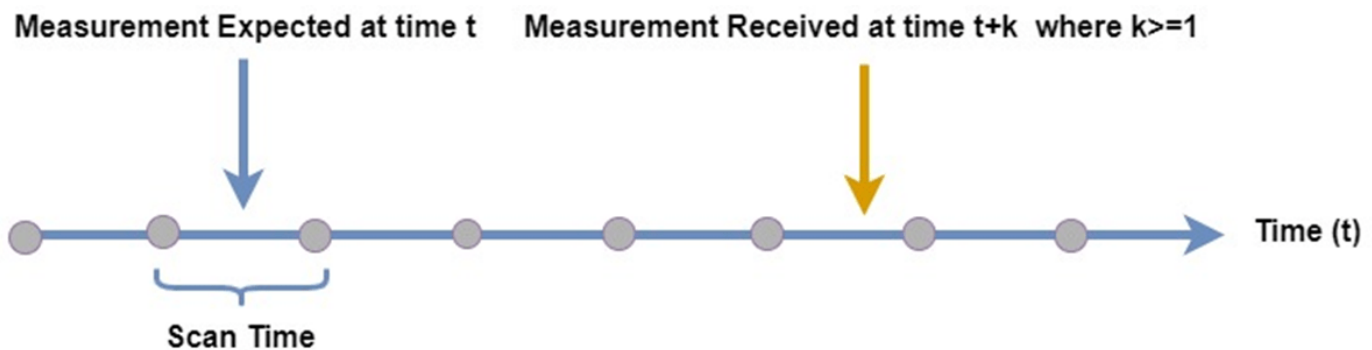
This example shows how to handle out-of-sequence measurements using the retrodiction technique at the filter level.

Introduction

In a tracking system, when multiple sensors report to the same tracker, the measurements may arrive at the tracker with a time delay relative to the time when they are generated from the sensor. The delay can be caused by any of the following reasons:

- 1 The sensor may require a significant amount of time to process the data. For example, a vision sensor may require tens of milliseconds to detect objects in a frame it captures.
- 2 If the sensor and the tracker are connected by a network, there may be a communications delay.
- 3 The filter may update in a different rate from the sensor scan rate. For example, if the filter is updated just before the sensor measurements arrive, these measurements are considered as out-of-sequence.

The following figure depicts the general case, in which a measurement is expected at time t , but is received at time $t+k$, where k is the number of filter updates. After the filter updates its state at time $t+k$, the arrived measurement from time t is an out-of-sequence measurement (OOSM) for the filter. The number of steps, k , is called the *lag*.



There are several techniques in the literature on how to handle an OOSM:

- 1 **Neglect:** In this technique, any OOSM is simply ignored and not used to update the filter state. This technique is the easiest and is useful in cases where the OOSM is not expected to contain data that would significantly modify the filter state and uncertainty. It is also the most efficient technique in terms of memory and processing.
- 2 **Reprocessing:** In this technique, the filter state and all the measurements are kept for the last n updates. Whenever a new measurement arrives, whether in-sequence or out-of-sequence, all the measurements are reordered by their measurement time and reprocessed to obtain the current state. This technique is guaranteed to be the most accurate, but is also the most expensive in terms of memory and processing.
- 3 **Retrodiction:** In this technique, the filter state is saved for the last n steps. If an OOSM arrives with a lag that is less than n steps, the filter is predicted backwards in time, or *retrodicted*, to the OOSM time. Then, the OOSM is used to correct the filter's current state estimate. If the lag of

the OOSM is greater than n , it is neglected. This technique is more efficient than reprocessing but nearly as accurate.

There are several ways to implement the retrodiction technique. The technique you use in this example is known as algorithm B/1, where algorithm B is an approximated algorithm, and the filter incorporates an OOSM with an l -steps lag and 1 "giant leap"[1].

Initialize the Filter and Enable Retrodiction

In this example, you follow the example shown in section IX of [1]. Consider an object moves along the x-axis with a nearly constant velocity model.

```
q = 0.5; % The power spectral density of the continuous time process noise.
```

The sensor measures both the position and the speed of the object along the x-axis, with the following measurement covariance matrix.

```
R = diag([1,0.1]);
```

You define that the object follows a constant velocity at the speed of 10 m/s along the x axis.

```
dt = 1; % The time step, in seconds.
v = 10; % The speed along the x axis, in meters per second.
```

The following code initializes a 1-D constant velocity extended Kalman filter used in this example. See the utility functions `oneDmotion`, `oneDmeas`, `oneDmotionJac`, and `oneDmeasJac` provided at the end of this script.

```
ekf = trackingEKF(@oneDmotion, @oneDmeas, ...
    'StateTransitionJacobianFcn', @oneDmotionJac, ...
    'MeasurementJacobianFcn', @oneDmeasJac, ...
    'HasAdditiveProcessNoise', false, ...
    'ProcessNoise', q, ...
    'State', [0;10], ... % x=0, v=10
    'StateCovariance', R,...
    'MeasurementNoise', R);
```

To enable retrodiction, you must set the number of OOSM steps using the `MaxNumOOSMSteps` property of the filter so that it prepares the filter history used by the retrodiction algorithm.

```
ekf.MaxNumOOSMSteps = 5;
```

Compare OOSM Handling Techniques

In this section, you compare the results of reprocessing, neglect, and retrodiction for a 1-lag measurement delay.

The in-sequence measurements are obtained at timesteps 1, 2, 3, and 4. The OOSM is obtained at timestep 3.5, which falls within the first lag interval, between timesteps 3 and 4.

```
t = [1, 2, 3, 4, 3.5];
x = v * t;
allStates = [x; repmat(v, 1, numel(t))];
allMeasurements = oneDmeasWithNoise(allStates, R);
```

You use the *neglect* technique. To do that, you run the filter only with the in-sequence measurements from timesteps 1, 2, 3, and 4 and you ignore the OOSM at time 3.5.

```

neglectEKF = clone(ekf); % Clone the EKF to preserve its initial state
for i = 1:4
    predict(neglectEKF, dt); % Predict
    correct(neglectEKF, allMeasurements(:,i)); % Correct the filter
end

```

To compare the different techniques, you observe the state covariance. The state covariance represents the level of uncertainty about the state estimate. Higher values in the state covariance mean higher uncertainty or less certainty about the state estimate. A common technique to compare the magnitude of values in the state covariance is by using the trace or the determinant of the matrix. You use the trace here.

```

disp(neglectEKF.StateCovariance);

    0.3142    0.0370
    0.0370    0.0834

disp(trace(neglectEKF.StateCovariance));

    0.3976

```

For the reprocessing technique, you use the OOSM at timestep 3.5 as if it were given in the right order with the rest of the in-sequence measurements.

```

reprocessingEKF = clone(ekf); % Clone the EKF to preserve its initial state
indices = [1 2 3 5 4]; % Reorder the measurements
for i = 1:numel(indices)
    if i <= 3 % Before t=3
        dt = 1;
    else % For 3 -> 3.5 and 3.5 -> 4
        dt = 0.5;
    end
    predict(reprocessingEKF, dt);
    correct(reprocessingEKF, allMeasurements(:,indices(i)));
end
disp(reprocessingEKF.StateCovariance);

    0.2287    0.0225
    0.0225    0.0759

disp(trace(reprocessingEKF.StateCovariance));

    0.3046

```

You observe that the reprocessing technique provides a much smaller state covariance, which means a more certain state estimate. The result is expected, because the OOSM at $t=3.5$ was reprocessed in the right sequence and the new information it contains helped reduce the uncertainty.

You now use the retrodiction technique. First, you process all the in-sequence measurements. Then, you retrodict the filter to the OOSM time and then retro-correct the filter with the OOSM.

```

retroEKF = clone(ekf); % Clone the EKF to preserve its initial state
dt = 1;
for i = 1:4
    predict(retroEKF, dt); % Predict
    correct(retroEKF, allMeasurements(:,i)); % Correct the filter
end
retrodict(retroEKF, -0.5); % Retrodict from t=4 to t=3.5

```

```
retroCorrect(retroEKF, allMeasurements(:,5)); % The measurement at t=3.5
disp(retroEKF.StateCovariance);

    0.2330    0.0254
    0.0254    0.0779

disp(trace(retroEKF.StateCovariance));

    0.3109
```

As expected, the retrodiction technique provides a state covariance matrix that is about the same magnitude as the one obtained using the ideal reprocessing technique. The matrix trace for the retrodiction technique is only 2% greater than the reprocessing state covariance trace. It is significantly smaller than the state covariance trace obtained by using the neglect technique.

Compare the Results for Various Lag Values

To understand the impact of the lag on OOSM handling, you define four levels of lag from 1-step to 4-steps lag. These lags correspond to generating the OOSM at times 3.5, 2.5, 1.5, and 0.5, respectively.

You organize the results in a tabular form as shown in the code below.

```
for lag = 1:4
    timestamps = [0, 1, 2, 3, 4, 4.5-lag];
    allStates = [v*timestamps, repmat(v, 1, numel(timestamps))];
    allMeasurements = oneDmeasWithNoise(allStates, R);
    oneLagStruct(lag) = runOneLagValue(ekf, allMeasurements, timestamps); %#ok<SAGROW>
end
displayTable(oneLagStruct)
```

Lag	Neglect	Reprocessing	Retrodiction
1	$\begin{bmatrix} 0.3142 & 0.0370 \\ 0.0370 & 0.0834 \end{bmatrix}$	$\begin{bmatrix} 0.2287 & 0.0225 \\ 0.0225 & 0.0759 \end{bmatrix}$	$\begin{bmatrix} 0.2330 & 0.0254 \\ 0.0254 & 0.0779 \end{bmatrix}$
2	$\begin{bmatrix} 0.3142 & 0.0370 \\ 0.0370 & 0.0834 \end{bmatrix}$	$\begin{bmatrix} 0.2597 & 0.0381 \\ 0.0381 & 0.0832 \end{bmatrix}$	$\begin{bmatrix} 0.2667 & 0.0389 \\ 0.0389 & 0.0830 \end{bmatrix}$
3	$\begin{bmatrix} 0.3142 & 0.0370 \\ 0.0370 & 0.0834 \end{bmatrix}$	$\begin{bmatrix} 0.2854 & 0.0387 \\ 0.0387 & 0.0833 \end{bmatrix}$	$\begin{bmatrix} 0.2955 & 0.0403 \\ 0.0403 & 0.0828 \end{bmatrix}$
4	$\begin{bmatrix} 0.3142 & 0.0370 \\ 0.0370 & 0.0834 \end{bmatrix}$	$\begin{bmatrix} 0.2983 & 0.0381 \\ 0.0381 & 0.0833 \end{bmatrix}$	$\begin{bmatrix} 0.3070 & 0.0393 \\ 0.0393 & 0.0826 \end{bmatrix}$

The first three rows in the table above are equal to the results shown in Table I in [1] for the three techniques. You make the following observations:

- 1 Using the neglect technique, there is no difference in the results as a function of lag. This result is expected because the neglect technique does not use the OOSM at all. It is also the worst technique of the three as seen by its largest state covariance values.
- 2 Using the reprocessing technique, which is the best of the three, the state covariance values increase as the lag increases. This result is expected, because as the lag becomes longer, the OOSM provides a smaller impact on the current state estimate and uncertainty.
- 3 Using the retrodiction technique, the results are bound by the results obtained by the reprocessing technique and the neglect technique for each lag value. As a result, as the lag becomes longer, introducing the OOSM provides a smaller benefit. This result is important,

because it shows the diminishing returns of keeping more history to support retrodiction beyond 3 or 4 steps. Another interesting result is that the trace of the state covariances obtained by the retrodiction technique is only 2-3% higher than the corresponding state covariance using the reprocessing technique.

Summary

This example introduced the topic of out-of-sequence measurements, often known by the abbreviation OOSM. The example showed three common techniques of handling an OOSM at the filter level: neglecting it, reprocessing it with all the measurements kept in a buffer, or retrodicting the filter and introducing the OOSM to improve the current estimate. Of the three techniques, neglect is the most efficient in memory and processing, but provides the worst state estimate. The reprocessing technique is the most expensive in terms of memory and processing but provides the most accurate result. The retrodiction technique is a good compromise of processing and memory vs. accuracy. You also saw that the maximum number of OOSM steps should be limited to 3 or 4 because the benefit of introducing an OOSM becomes smaller when the number of steps increases.

References

[1] Yaakov Bar-Shalom, Huimin Chen, and Mahendra Mallick, "One-Step Solution for the Multistep Out-of-Sequence-Measurement Problem in Tracking", IEEE Transactions on Aerospace and Electronic Systems, Vol. 40, No. 1, January 2004.

Supporting functions

oneDmotion

1-D constant velocity state transition function.

```
function state = oneDmotion(state, ~, dt)
state = [1 dt; 0 1]*state;
end
```

oneDmotionJac

1-D constant velocity state transition function Jacobian. It provides the process noise used in [1].

```
function [dfdx,dfdv] = oneDmotionJac(~, ~, dt)
dfdx = [1 dt; 0 1];
dfdv = chol([dt^3/3 dt^2/2; dt^2/2 dt], 'lower');
end
```

oneDmeas

1-D constant velocity measurements function. It provides the measurement including position and velocity without noise.

```
function z = oneDmeas(state)
z = state;
end
```

oneDmeasJac

1-D constant velocity measurements function Jacobian.

```
function H = oneDmeasJac(state)
H = eye(size(state,1));
end
```

oneDmeasWithNoise

1-D constant velocity measurements function. It provides the measurement including position and velocity with Gaussian noise and covariance R.

```
function z = oneDmeasWithNoise(state,R)
z = state + R * randn(size(R,1), size(state,2));
end
```

runNeglect

Runs the neglect technique for various values of lags.

```
function [x,P] = runNeglect(ekf, allMeasurements, timestamps)
neglectEKF = clone(ekf); % Clone the EKF to preserve its initial state
dt = diff(timestamps);
for i = 1:numel(dt)-1
    predict(neglectEKF, dt(i)); % Predict
    correct(neglectEKF, allMeasurements(:,i)); % Correct the filter
end
x = neglectEKF.State;
P = neglectEKF.StateCovariance;
end
```

runReprocessing

Runs the reprocessing technique for various values of lags.

```
function [x, P] = runReprocessing(ekf, allMeasurements, timestamps)
reprocessingEKF = clone(ekf); % Clone the EKF to preserve its initial state
[timestamps, indices] = sort(timestamps); % Reorder the timestamps
allMeasurements = allMeasurements(:, indices(2:end)-1); % Reorder the measurements
dt = diff(timestamps);
for i = 1:numel(dt)
    predict(reprocessingEKF, dt(i));
    correct(reprocessingEKF, allMeasurements(:,i));
end
x = reprocessingEKF.State;
P = reprocessingEKF.StateCovariance;
end
```

runRetrodiction

Runs the retrodiction technique for various values of lags.

```
function [x, P] = runRetrodiction(ekf, allMeasurements, timestamps)
retrodictionEKF = clone(ekf); % Clone the EKF to preserve its initial state
dt = diff(timestamps);
for i = 1:numel(dt)-1
    predict(retrodictionEKF, dt(i));
    correct(retrodictionEKF, allMeasurements(:,i));
end
retrodict(retrodictionEKF, dt(end));
retroCorrect(retrodictionEKF, allMeasurements(:,end));
x = retrodictionEKF.State;
P = retrodictionEKF.StateCovariance;
end
```

runOneLagValue

Runs and collects results from the three techniques: neglect, reprocessing, and retrodiction.

```
function oneLagStruct = runOneLagValue(ekf, allMeasurements, timestamps)
oneLagStruct = struct('Lag',ceil(timestamps(end-1)-timestamps(end)),...
    'Neglect',zeros(2,2),...
    'Reprocessing',zeros(2,2),...
    'Retrodiction',zeros(2,2));
```

```
% Neglect the OOSM
```

```
[~, P] = runNeglect(ekf, allMeasurements, timestamps);
oneLagStruct.Neglect = P;
```

```
% Reprocess all the measurements according to time
```

```
[~, P] = runReprocessing(ekf, allMeasurements, timestamps);
oneLagStruct.Reprocessing = P;
```

```
%
```

```
% Use retrodiction
```

```
[~, P] = runRetrodiction(ekf, allMeasurements, timestamps);
oneLagStruct.Retrodiction = P;
```

```
end
```

displayTable

Displays the results in an easy to read tabular form.

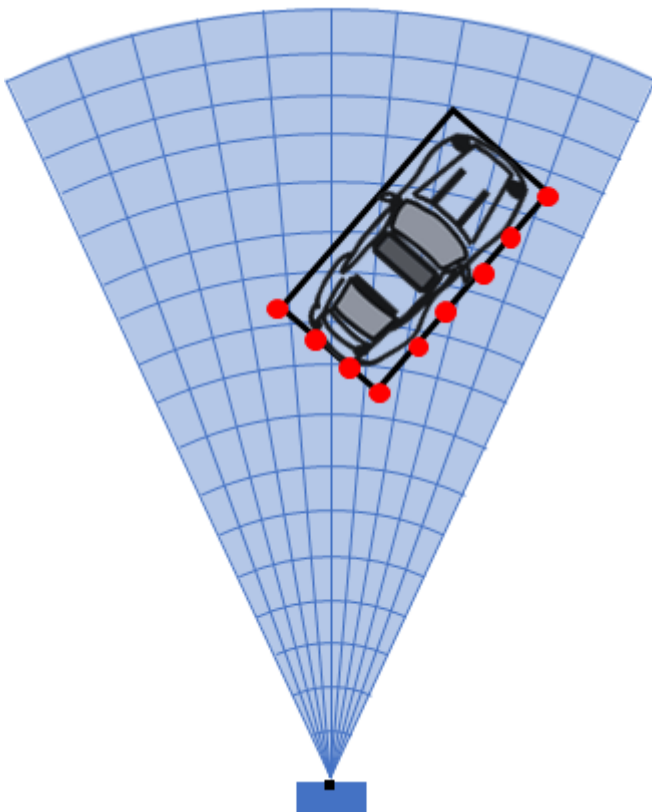
```
function displayTable(t)
varNames = fieldnames(t);
fprintf('<strong>%6s </strong>', 'Lag');
fprintf('<strong>%13s </strong>', string(varNames(2:4)));
fprintf('\n===== \n');
for i = 1:numel(t)
    fprintf('    %d', t(i).Lag);
    for j = 2:numel(varNames)
        fprintf('    %c %1.4f %1.4f %c ', 9121, t(i).(varNames{j})(1,1:2), 9124);
    end
    fprintf('\n    ');
    for j = 2:numel(varNames)
        fprintf('    %c %1.4f %1.4f %c', 9123, t(i).(varNames{j})(2,1:2), 9126);
    end
    fprintf('\n');
    fprintf('===== \n');
end
end
```

Extended Object Tracking of Highway Vehicles with Radar and Camera in Simulink

This example shows you how to track highway vehicles around an ego vehicle in Simulink. In this example, you use multiple extended object tracking techniques to track highway vehicles and evaluate their tracking performance. This example closely follows the “Extended Object Tracking of Highway Vehicles with Radar and Camera” on page 6-148 MATLAB® example.

Extended Objects and Extended Object Tracking

In the sense of object tracking, extended objects are objects, whose dimensions span multiple sensor resolution cells. As a result, the sensors report multiple detections per objects in a single scan. 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. For example, the image below depicts multiple detections for a single vehicle that spans multiple radar resolution cells.



In conventional tracking approaches such as global nearest neighbor, joint probabilistic data association and multi-hypothesis tracking, tracked objects are assumed to return one detection per sensor scan. High resolution sensors that report multiple returns per object in a scan present new challenges to conventional tracker. 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.

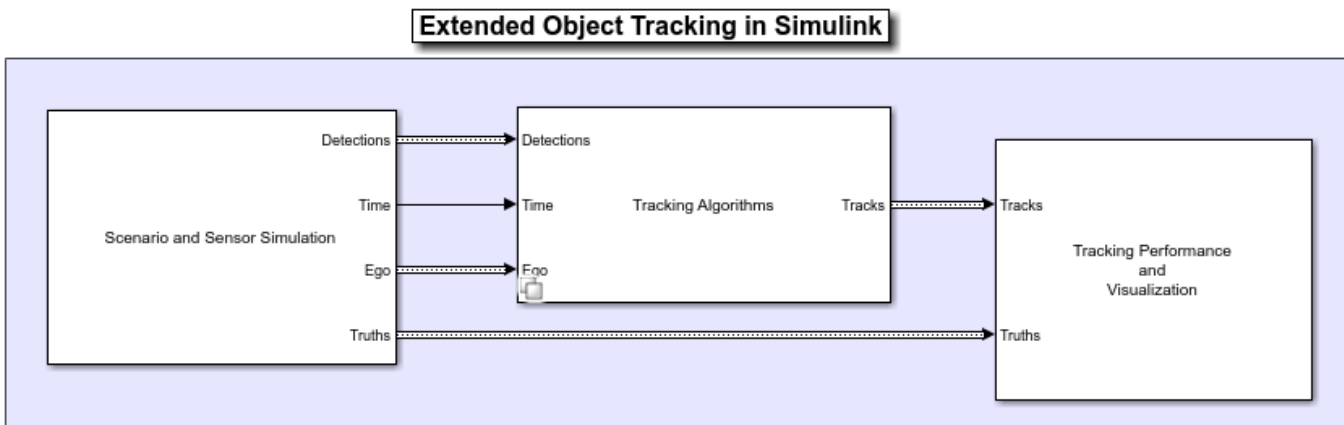
Extended object trackers can handle multiple detections per object. In addition, these trackers can estimate not only the kinematic states, such as position and velocity of the object, but also the dimensions and orientation of the object. In this example, you track vehicles around the ego vehicle using the following trackers:

- A conventional multi-object tracker using a point-target model, Multi-Object Tracker (Automated Driving Toolbox).
- A GGIW-PHD tracker, Probability Hypothesis Density (PHD) Tracker with Gamma Gaussian Inverse Wishart (ggiwphd) filter.
- A GM-PHD tracker, Probability Hypothesis Density (PHD) Tracker with Gaussian Mixture (gmphd) filter using a rectangular target model.

You will evaluate the results using the Optimal Subpattern Assignment Metric, which provides a single combined score accounting for errors in both assignment and distance. A lower score means better tracking.

Overview of the Model

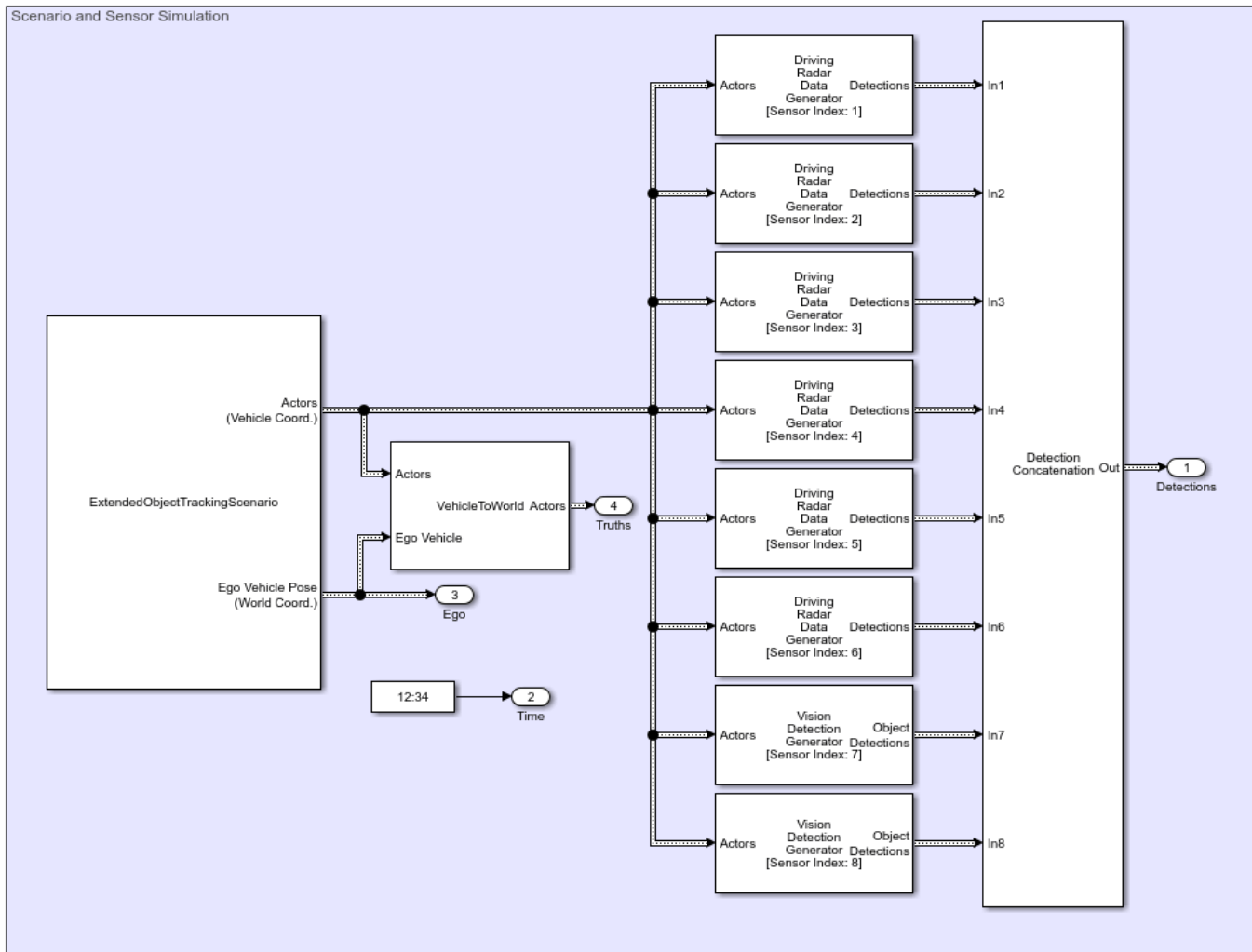
```
load_system('ExtendedObjectTrackingInSimulink');
set_param('ExtendedObjectTrackingInSimulink','SimulationCommand','update');
open_system('ExtendedObjectTrackingInSimulink');
```



The model has three sub-systems, each implementing a part of the workflow:

- Scenario and Sensor Simulation
- Tracking Algorithms
- Tracking Performance and Visualization

Scenario and Sensor Simulation

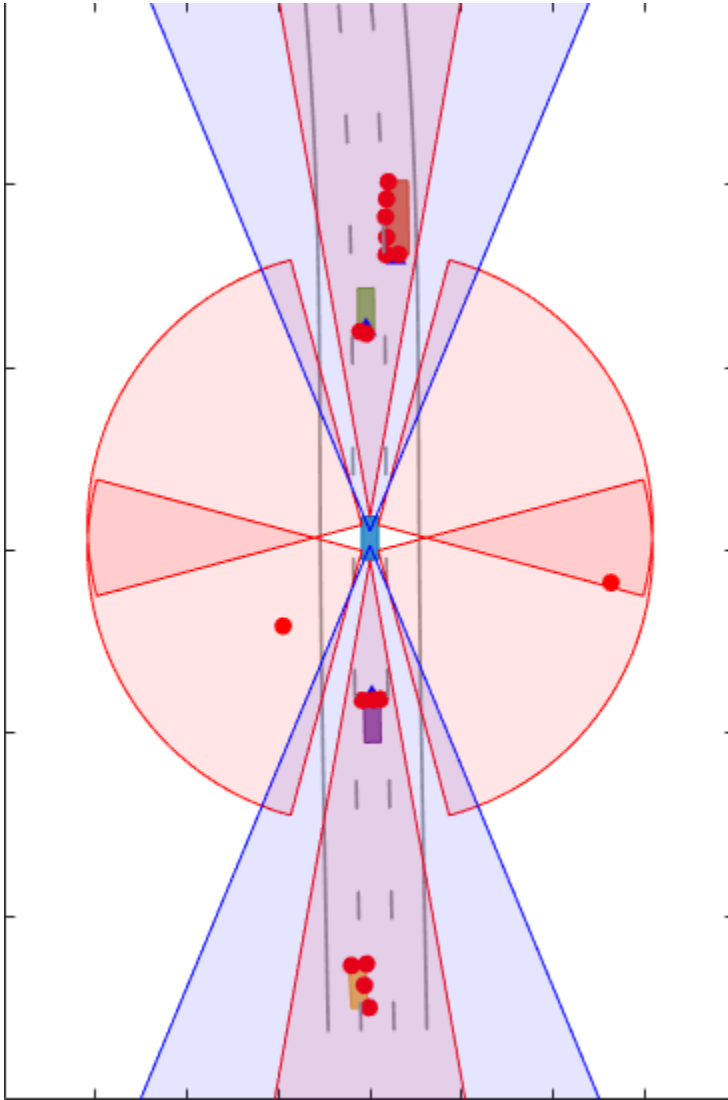


The Scenario Reader (Automated Driving Toolbox) block reads a `drivingScenario` (Automated Driving Toolbox) object from workspace and generates Actors and Ego vehicle position data as “Explore Simulink Bus Capabilities” (Simulink) Objects. The Vehicle To World block converts the actor position from vehicle coordinates to world coordinates. The Driving Radar Data Generator (Automated Driving Toolbox) block simulates radar detections and Vision Detection Generator (Automated Driving Toolbox) simulates camera detections. Detections from all the sensors are grouped together using the Detection Concatenation (Automated Driving Toolbox) block.

In the scenario 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 six radar sensors and two vision sensors covering the 360-degree field of view. The sensors have some coverage overlaps and gaps. The ego vehicle is equipped with a long-range radar sensor and a vision sensor on the front and back of the vehicle. On each side of the vehicle two short-range radar sensors cover 90 degrees respectively. One

of the two sensors cover from the middle of the vehicle to the back, and the other sensor covers from the middle of the vehicle to the front.



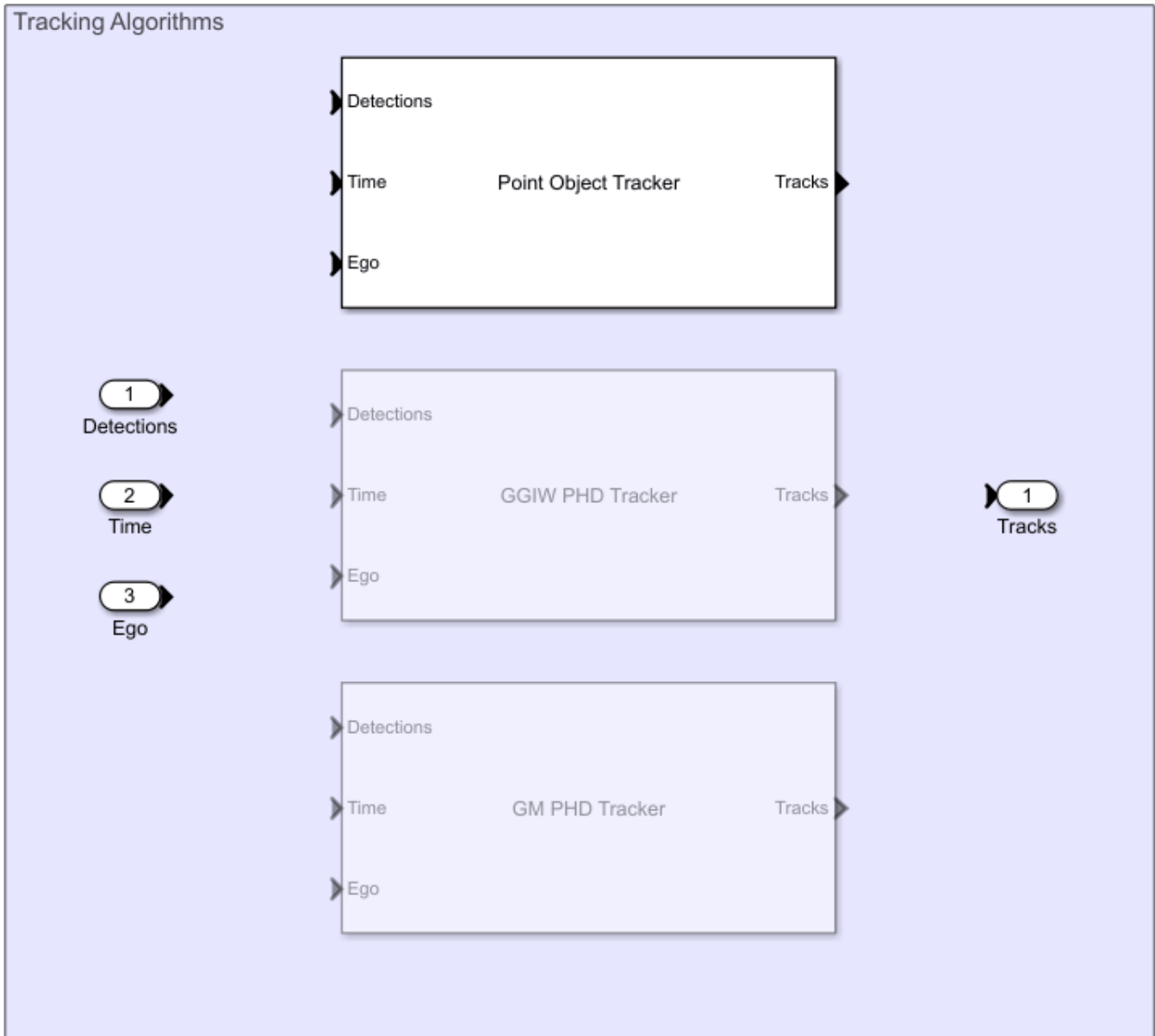
Tracking Algorithms

You implement three different extended object tracking algorithms using a variant sub-system. See "Implement Variations in Separate Hierarchy Using Variant Subsystems" (Simulink) for more information. The variant sub-system has one Subsystem (Simulink) for each tracking algorithm. You can select the tracking algorithm by changing the value of the workspace variable TRACKER. The default value of TRACKER is 1.

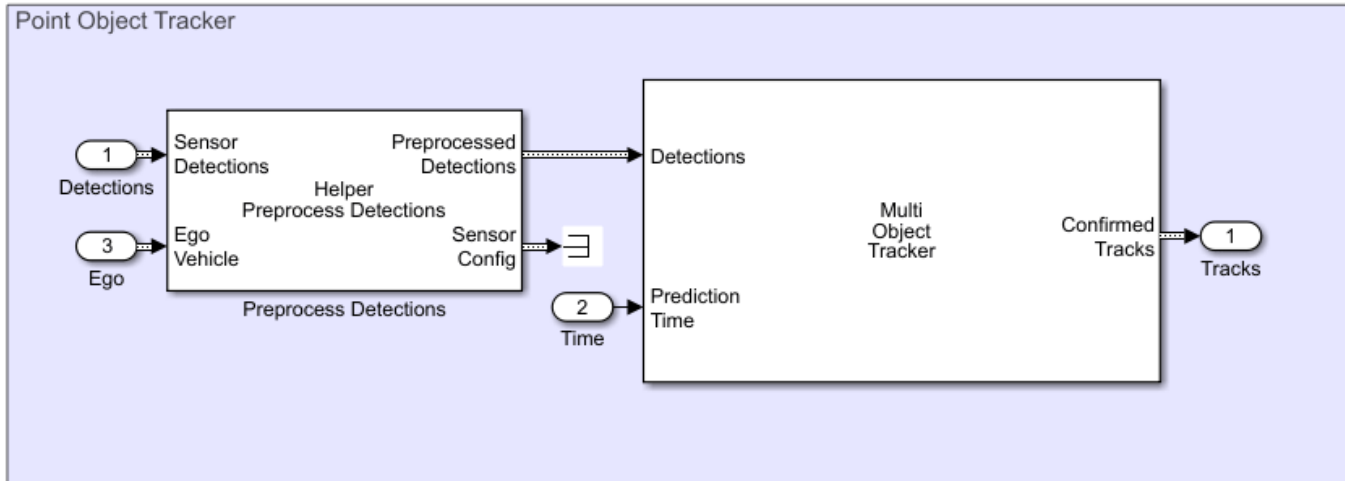
TRACKER

Tracking Algorithm

- 1 Conventional Multi-Object Tracker with a point-target model
- 2 Probability Hypothesis Density Tracker with ggiwphd filter
- 3 Probability Hypothesis Density Tracker with gmphd filter



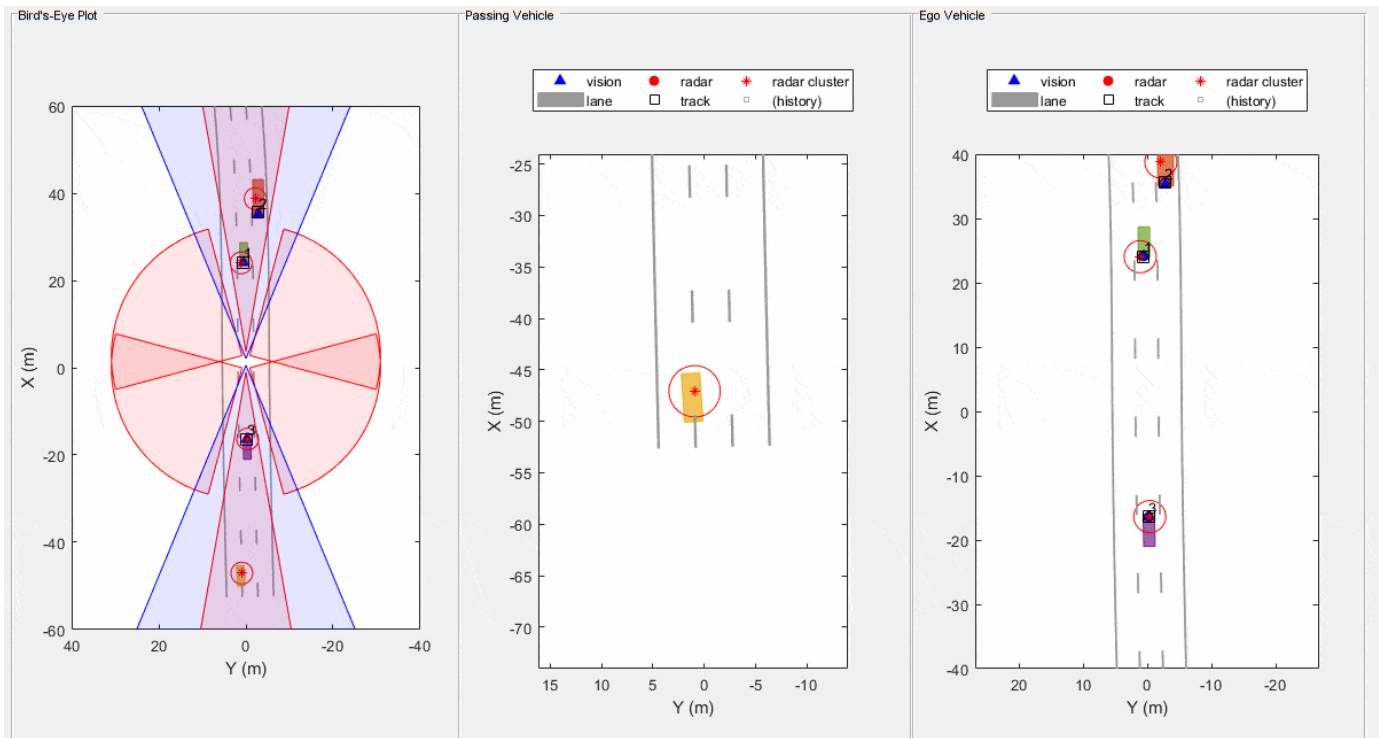
Point Object Tracker



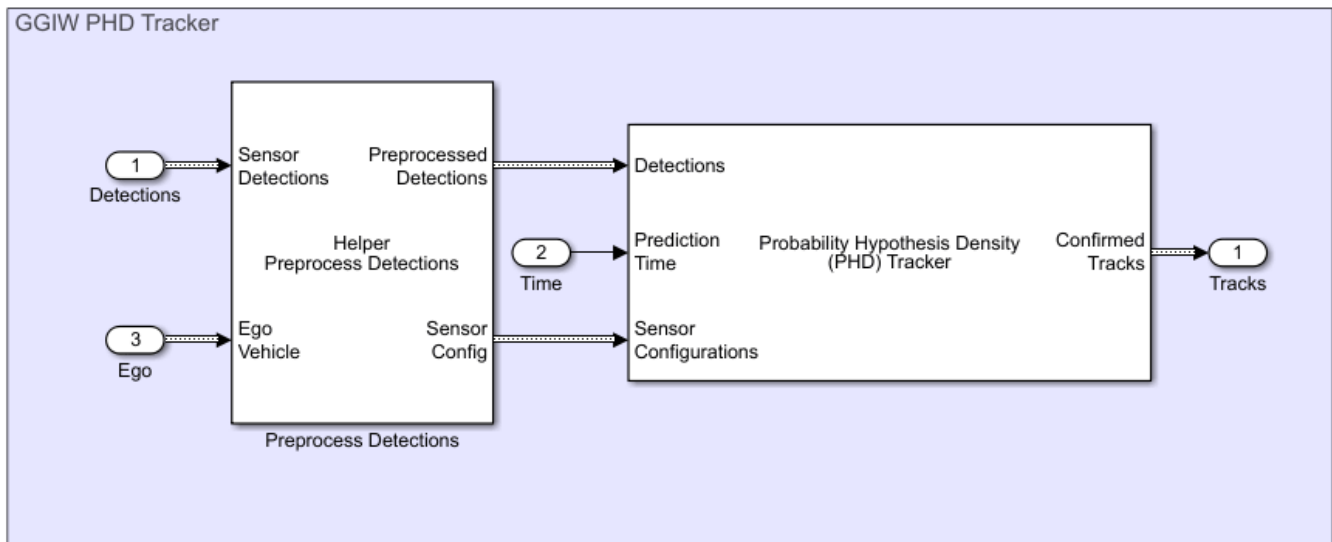
In this section you use the Multi-Object Tracker (Automated Driving Toolbox) block to implement the tracking algorithm based on a point target model. Detections from the radar are preprocessed to include ego vehicle INS information using the Helper Preprocess Detection block. The block is implemented using the MATLAB System (Simulink) block. Code for this block is defined in the helper class `helperPreProcessDetections`. The Multi-Object Tracker 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. However, the simulated radar sensors have a high enough resolution and 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.

To cluster the radar detections, you configure the Driving Radar Data Generator block to output Clustered Detections instead of detections. To do this you set the `TargetReportFormat` parameter on the block as `Clustered detections`. In the model, this is achieved by specifying the block parameters in the `InitFcn` callback of the Point Object Tracker subsystem. See "Model Callbacks" (Simulink) for more information about callback functions.

The animation below shows that, with clustering, the tracker can keep track of the objects in the scene. 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. You can also observe 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.



GGIW-PHD Extended Object Tracker



In this section you use a Probability Hypothesis Density (PHD) Tracker tracker block to implement the extended object tracking algorithm with `ggiwphd` filter to track objects. Detections from the radar are preprocessed to include ego vehicle INS information using the Helper Preprocess Detection block. The block is implemented using the MATLAB System (Simulink) block. Code for this block is defined in the helper class `helperPreProcessDetections`. It also outputs the sensor configurations required by the tracker for calculating the detectability of each component in the density.

You specify the `SensorConfigurations` parameter of the PHD tracker block as a structure with fields same as `trackingSensorConfiguration` and set the `FilterInitializationFcn` field as `helperInitGGIWFilter` and `SensorTransformFcn` field as `ctmeas`. In the model this is achieved by specifying the `InitFcn` callback of the GGIW PHD Tracker subsystem. See “Model Callbacks” (Simulink) for more information about callback functions.

Unlike Multi-Object Tracker, which maintains one hypothesis per track, the GGIW-PHD is a multi-target filter which describes the probability hypothesis density (PHD) of the scenario. GGIW-PHD filter uses these distributions to model the extended targets:

Gamma: Represents the expected number of detections on a sensor from the extended object.

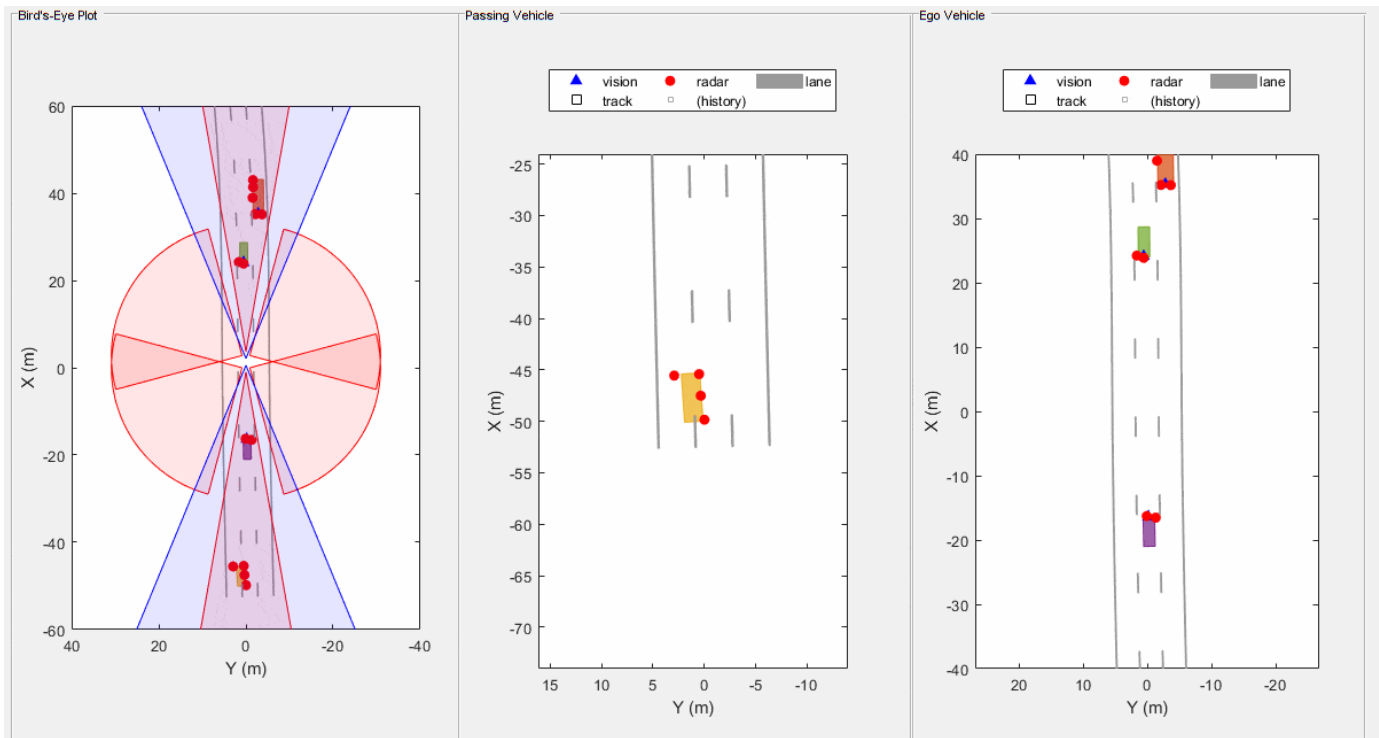
Gaussian: Represents the kinematic state of the extended object.

Inverse-Wishart: Represents the spatial extent of the target. In 2-D space, the extent is represented by a 2-by-2 random positive definite matrix, which corresponds to a 2-D ellipse description. In 3-D space, the extent is represented by a 3-by-3 random matrix, which corresponds to a 3-D ellipsoid description. The probability density of these random matrices is given as an Inverse-Wishart distribution.

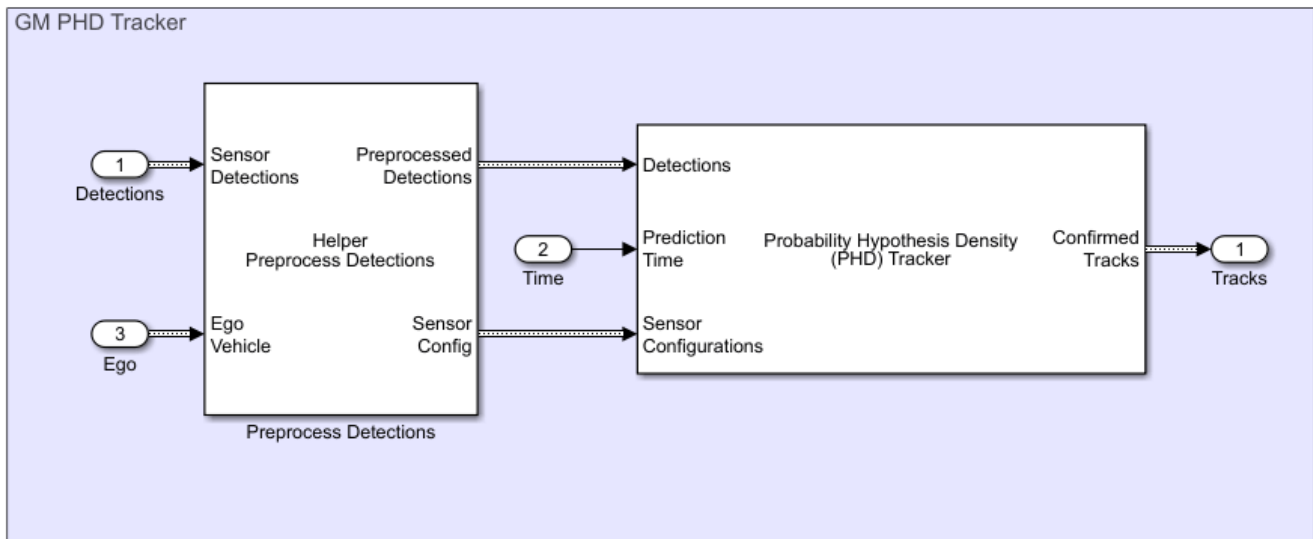
The model assumes that each distribution is independent of each other. Thus, the probability hypothesis density (PHD) in GGIW-PHD filter is described by a weighted sum of the probability density functions of several GGIW components. In contrast to a point object tracker, which assumes one partition of detections, the PHD tracker creates multiple possible partitions of a set of detections and evaluates it against the current components in the PHD filter. You specify a `PartitioningFcn` to create detection partitions, which provides multiple hypotheses about the clustering.

The animation below shows 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, dimension, and orientation of each object. The dashed elliptical shape in the figure demonstrates the expected extent of the target.

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. The tracker can better approximate the length and width of vehicles that are nearby using the ellipse. In the simulation, for example, the tracker produces a better ellipse overlap with the actual size of the passing vehicle.



GM-PHD Rectangular Object Tracker

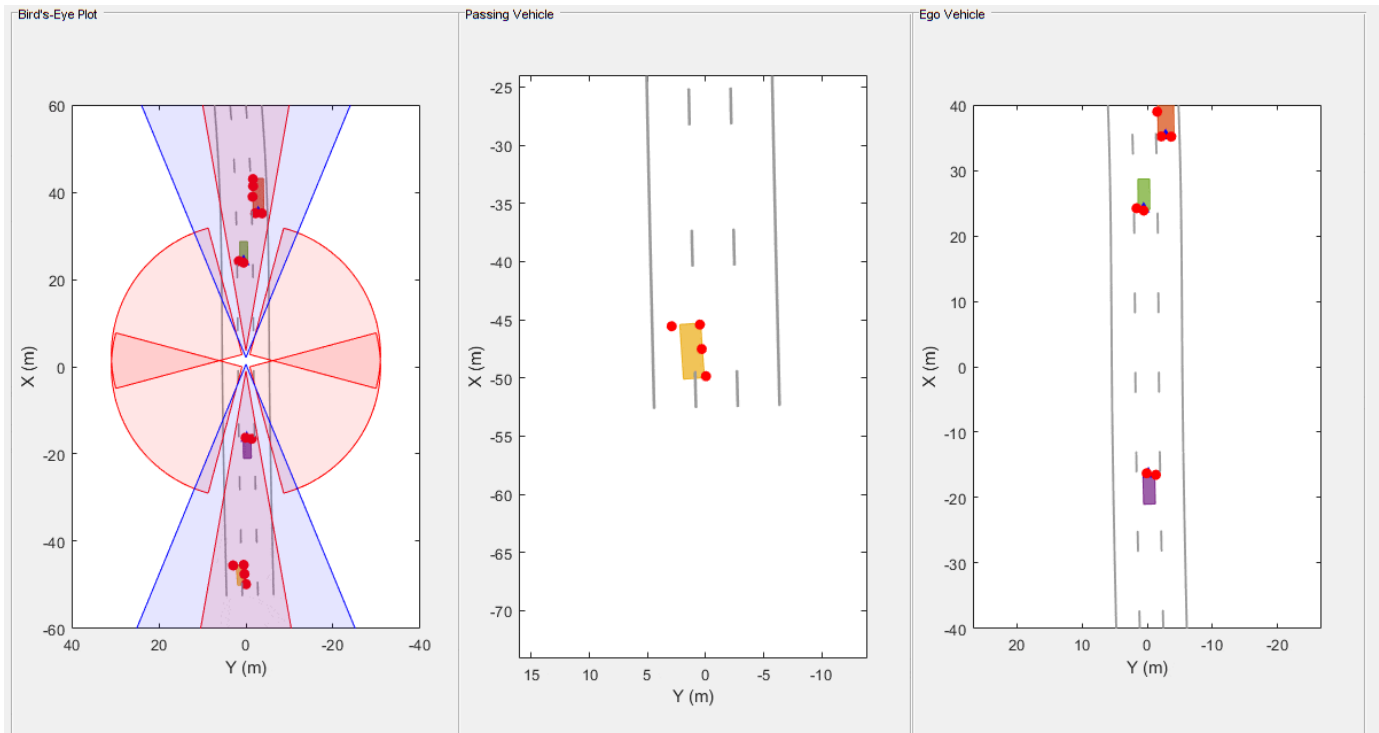


In this section you use a Probability Hypothesis Density (PHD) Tracker tracker block with a gmphd filter to track objects using a rectangular target model. Unlike ggiwphd, which uses an elliptical shape to estimate the object extent, gmphd allows you to use a Gaussian distribution to define the shape of your choice. You define a rectangular target model by using motion models, `ctrect` and `ctrectjac` and measurement models, `ctrectmeas` and `ctrectmeasjac`.

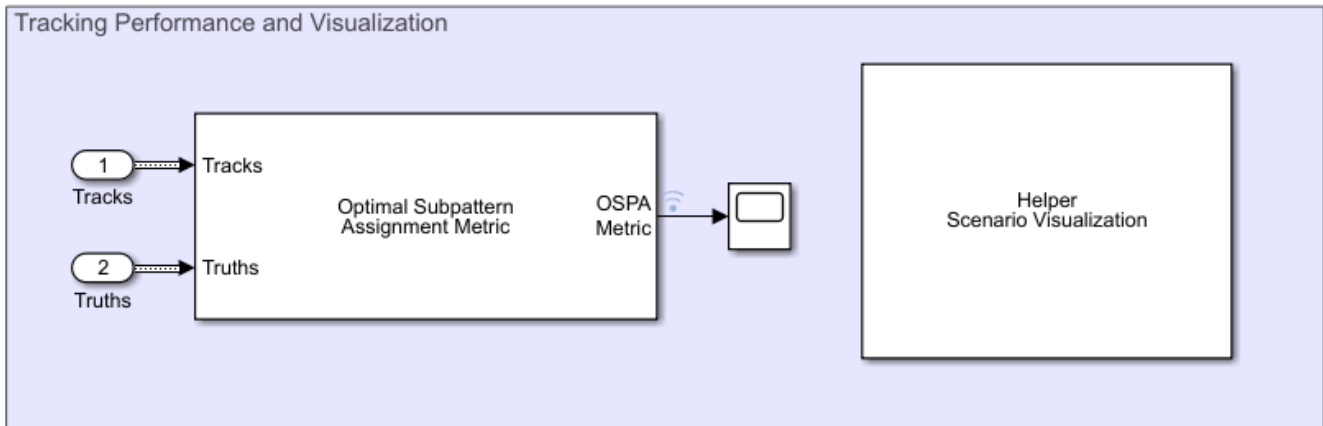
The sensor configurations defined for PHD Tracker remain the same except for definition of the `SensorTransformFcn` and `FilterInitializationFcn` fields. You set the `FilterInitializationFcn` field as `helperInitRectangularFilter` and the `SensorTransformFcn` field as `ctrectcorners`. In the model this is achieved by specifying the `InitFcn` callback of the GM PHD Tracker subsystem. See “Model Callbacks” (Simulink) for more information about callback functions.

The animation below shows 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 of different sizes.

You can see that the estimated tracks, 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 vehicles along with their kinematic centers.



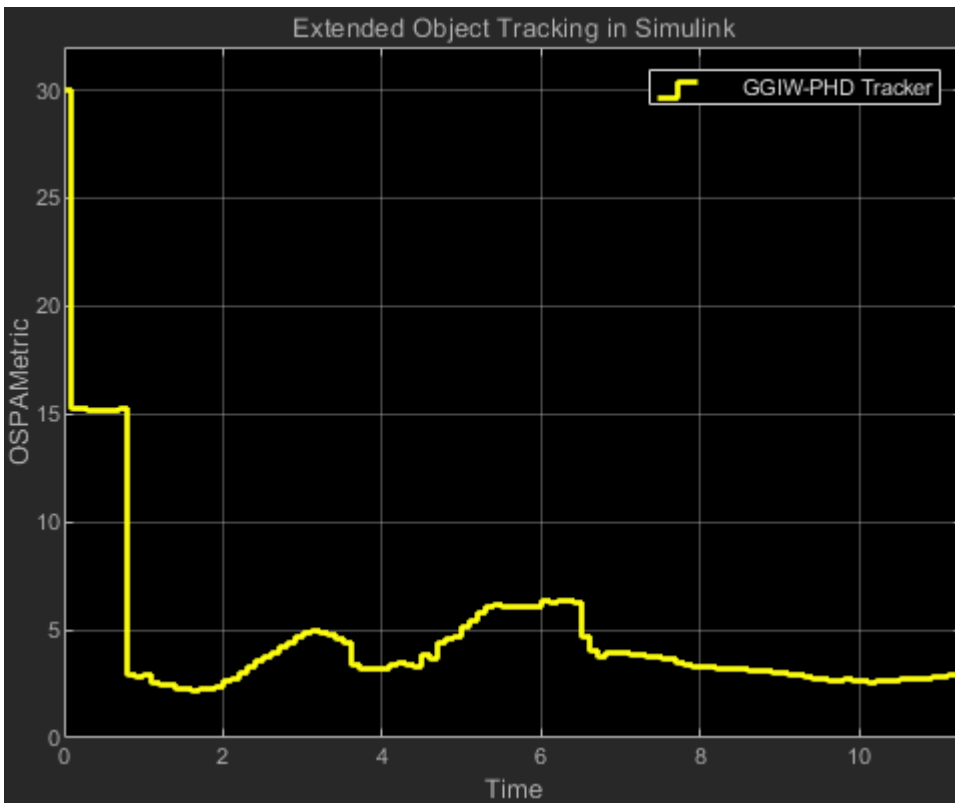
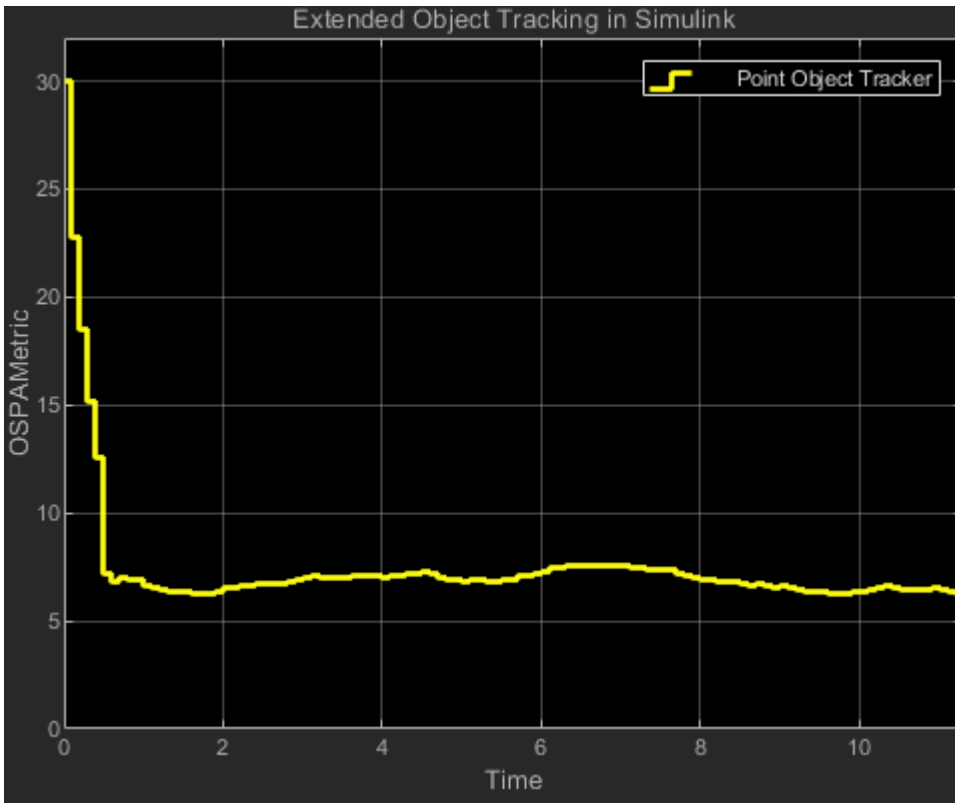
Tracking Performance and Visualization

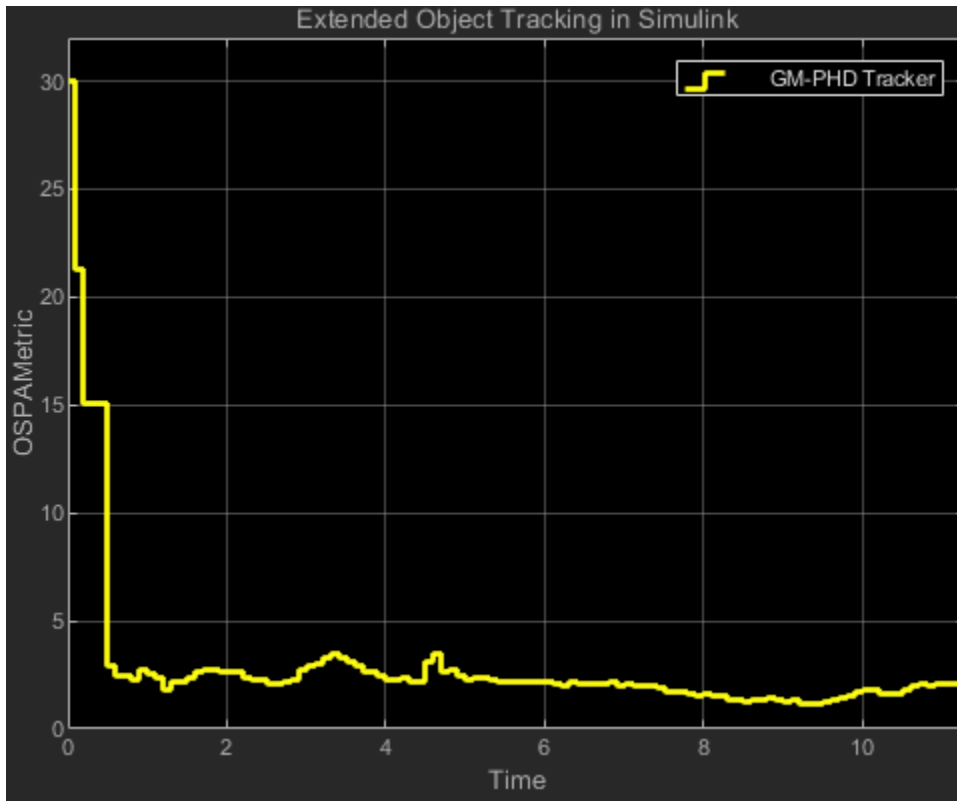


You assess the performance of each algorithm using the Optimal Subpattern Assignment (OSPA) metric. The OSPA metric aims to evaluate the performance of a tracking system with a scalar cost by combining different error components.

$$\text{OSPA} = \left(d_{\text{loc}}^p + d_{\text{card}}^p + d_{\text{lab}}^p \right)^{\frac{1}{p}}$$

where d_{loc} , d_{card} , and d_{lab} are localization, cardinality and labeling error components and p is the order of the OSPA metric. See `trackOSPAMetric` for more information.





You set the `Distance` type parameter to `custom` and define the distance function between a track and its associated ground truth as the `helperExtendedTargetDistance` helper function. This distance helper function captures position, velocity, dimension and yaw error between a track and an associated truth. The OSPA metric is shown in the scope block. Each unit on the x-axis represents 10 time steps in the scenario. Notice that the OSPA metric decreases and thus shows performance improvement when you switch from point object tracker to GGIW-PHD tracker and from GGIW-PHD tracker to GM-PHD tracker. The scenario is visualized using the Helper Scenario Visualization block, implemented using the MATLAB System (Simulink) block. Code for this block is defined in the helper class `helperExtendedTargetTrackingDisplayBlk`.

```
bdclose('ExtendedObjectTrackingInSimulink');
```

Summary

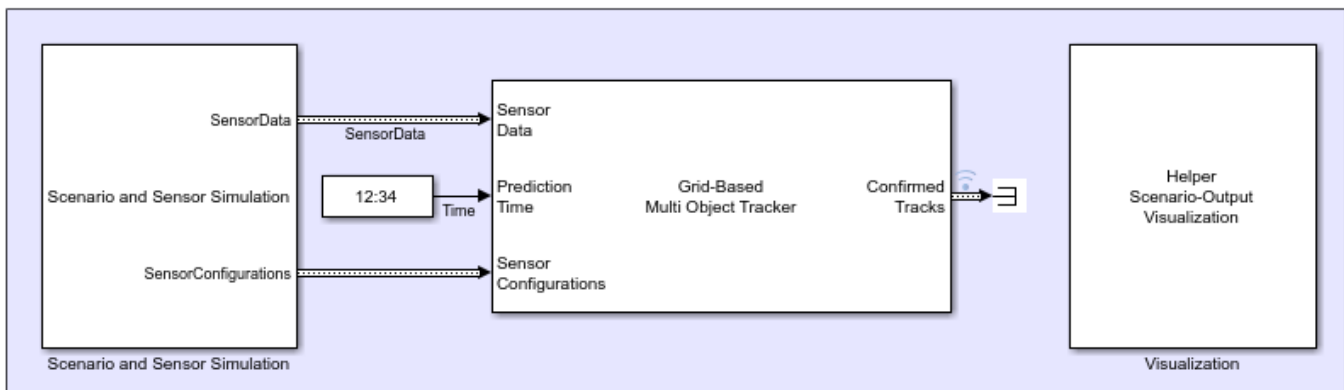
In this example you learned how to track objects that return multiple detections in a single sensor scan using different tracking approaches in Simulink environment. You also learned how to evaluate the performance of a tracking algorithm using the OSPA metric.

Grid-based Tracking in Urban Environments Using Multiple Lidars in Simulink

This example shows how to track moving objects with multiple lidars using a grid-based tracker in Simulink. You use the Grid-Based Multi Object Tracker Simulink block to define the grid-based tracker. This Grid-based tracker uses dynamic occupancy grid map as an intermediate representation of the environment. This example closely follows the “Grid-Based Tracking in Urban Environments Using Multiple Lidars” on page 6-618 MATLAB® example.

Overview of the model

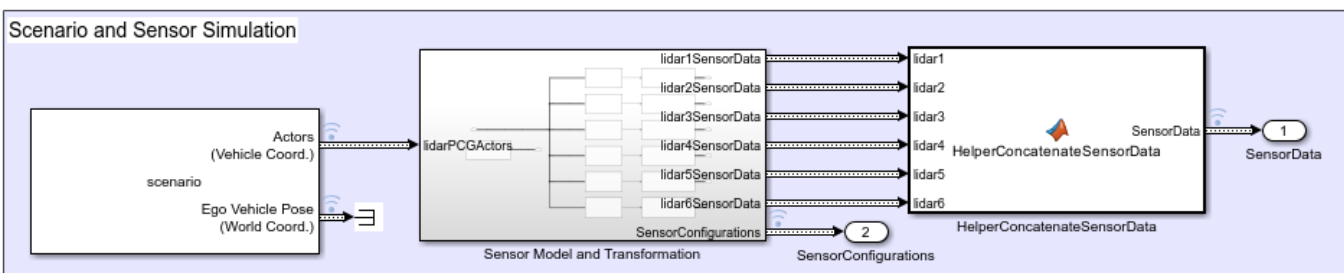
Grid-based Tracking in Urban Environments Using Multiple Lidars in Simulink



The model is composed of three parts:

- Scenario and Sensor Simulation
- Grid-Based Multi Object Tracker
- Visualization

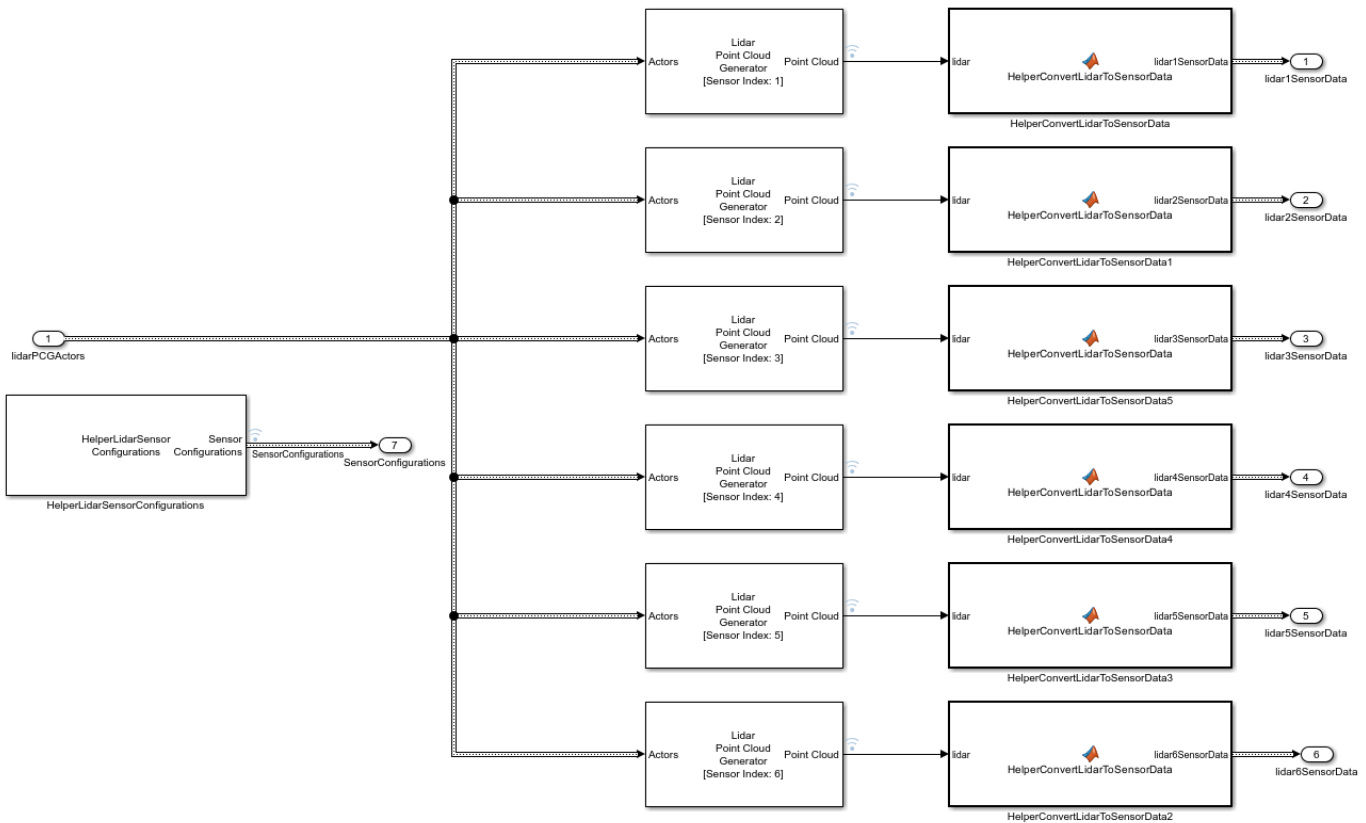
Scenario and Sensor Simulation



The Scenario Reader block reads a drivingScenario (Automated Driving Toolbox) object from workspace and generates Actors and Ego vehicle position data as Simulink.Bus (Simulink) objects. The subsystem Sensor Model and Transformation helps to generate multiple lidar sensor data and transform that data into a high-resolution sensor data. The HelperConcatenateSensorData

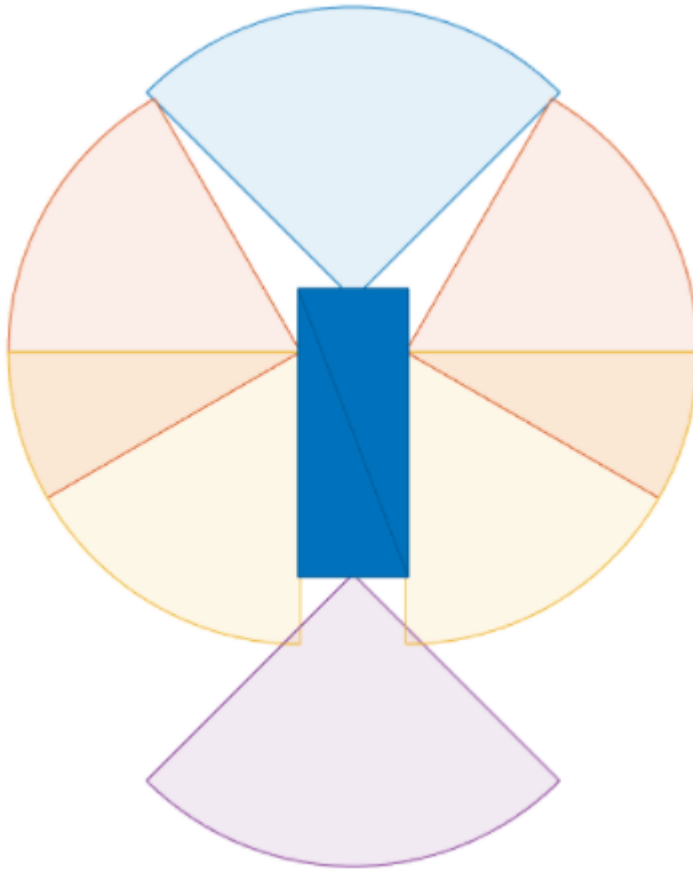
block is implemented using a MATLAB Function (Simulink) block. Code for this block is defined in the `HelperSensorData` file. This block groups the data received from `Sensor Model` and `Transformation` subsystem, which is used as an input to the tracker block.

Sensor Model and Transformation



The `HelperLidarSensorConfigurations` block generates real time lidar sensors configuration with the help of Runtime objects obtained from the `Scenario Reader` and `Lidar Point Cloud Generator` blocks. These 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. As the sensors move in the scenario system, their configurations must be updated each time by specifying the configurations as an input to the tracker block. The `HelperConvertLidarToSensorData` block is implemented using a MATLAB Function (Simulink) block. Code for this block is defined in the `HelperLidarToSensorData` file. This block transforms the data coming from lidar sensors and sensor configurations Runtime object from the `HelperLidarSensorConfigurations` block into a high resolution sensor data.

The scenario used in this example was created using the `Driving Scenario Designer (Automated Driving Toolbox)` app and then exported to a MATLAB® function. The scenario represents an urban intersection scene and contains a variety of objects including pedestrians, bicyclists, cars, and trucks. The ego vehicle is equipped with six homogeneous lidars, each with a horizontal field of view of 90 degrees and a vertical field of view of 40 degrees. 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.



Grid-Based Multi Object Tracker

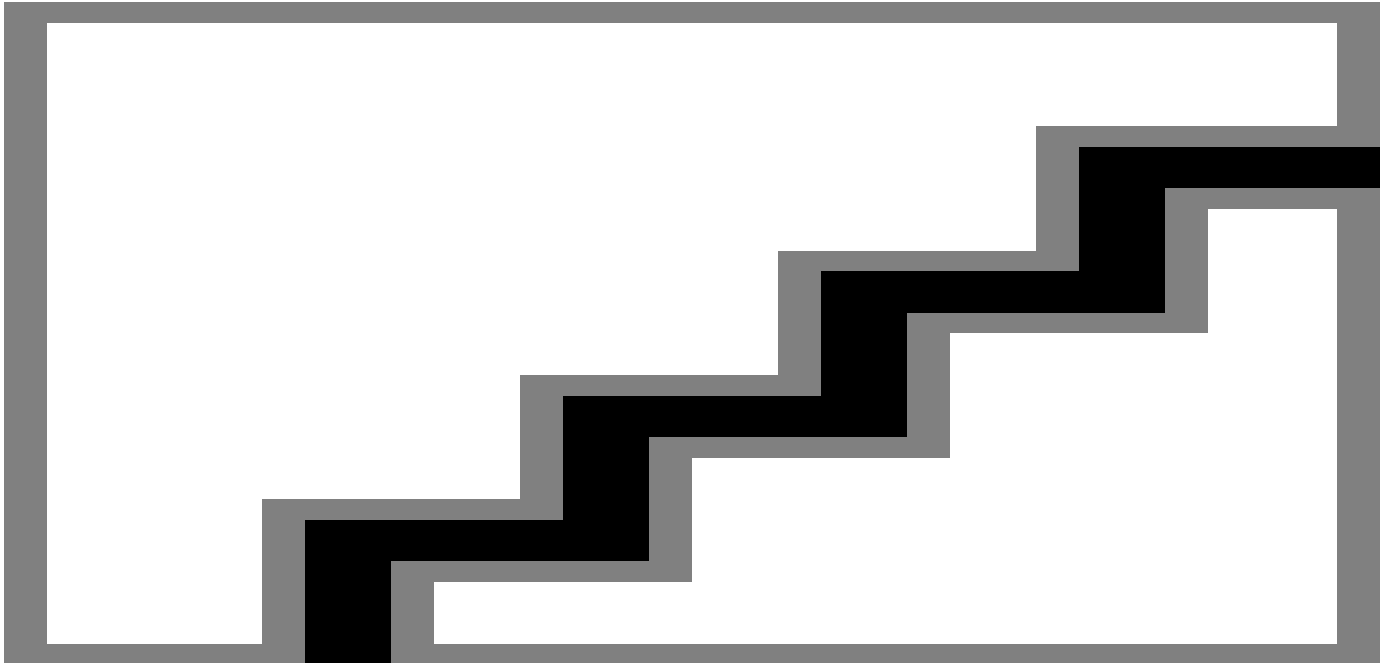
You use the Grid-Based Multi Object Tracker block to implement the tracking algorithm to track dynamic objects in the scene. You define all the parameters in the block mask based on the scenario requirement. To visualize the dynamic grid map, make sure you select the `Enable dynamic grid map visualization` parameter on the `visualization` tab of the tracker block.

Visualization

The visualization used for this example is defined using a helper class, `HelperVisualization` a MATLAB System (Simulink) block, attached with this example. In the step call of visualization block make sure you set the "Parent" property to the current axes. This allows you to visualize the dynamic grid map on the current figure axes. The color disc classifies the motion of the grid cells object. You can see that the grid cells motion in the positive x-direction are classified with red color, objects moving in the negative x-direction are classified with the blue color, objects moving in the negative y-direction are classified with purple and objects moving in the positive y-direction are classified with light-green color. 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 in grayscale according to their occupancy. The degree of grayness denotes the probability that the space occupied by the grid cell is 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.



Results and Analysis

You can analyze the performance of the tracker based on the visualization results. The Grid-based tracker panel shows the estimated dynamic map as well as the estimated tracks of the objects. It also shows the configurations of sensors, which are mounted on the ego vehicle and shown as blue circular sectors. Notice that the grey area shown in the dynamic grid map is not observable from any of the ego vehicle sensors since the view direction is blocked. You can also find that the tracks are only extracted from the dynamic cells and hence the tracker is able to filter out static objects.

Summary

In this example you learned how to construct a grid-based multi object tracking system and how to track and visualize dynamic objects in a complex urban driving environment in Simulink.

Simulate INS Block

In this example, you simulate an INS block by using the pose information of a vehicle undertaking a left-turn trajectory.

Load Vehicle Trajectory Data

First, you load the trajectory information of the vehicle to the workspace.

```
load leftTurnTrajectory.mat
```

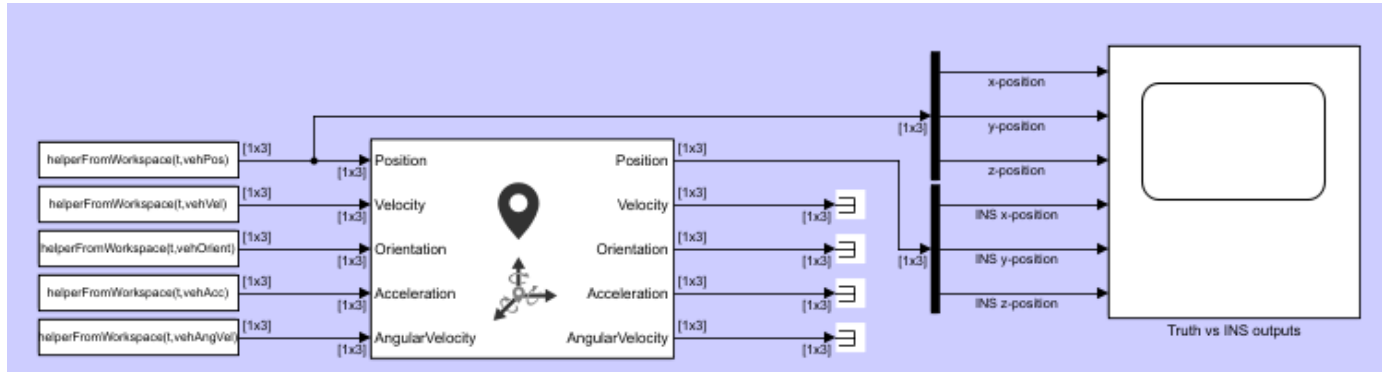
You notice that seven new variables appear in MATLAB workspace.

- `dt` — The time step size of 0.4 seconds.
- `t` — The total time span of 7.88 seconds.
- `vehPos`, `vehVel`, `vehAcc`, `vehOrient`, `vehAngVel` — The history of position, velocity, acceleration, orientation, and angular velocity, each specified as a 198-by-3 matrix, where 198 is the total number of steps.

Open Simulink Model

Next, you open the Simulink model.

```
open simulateINS.slx
```



The model contains three parts: the data importing part, the INS block, and the scope block to compare the true positions with the INS outputs.

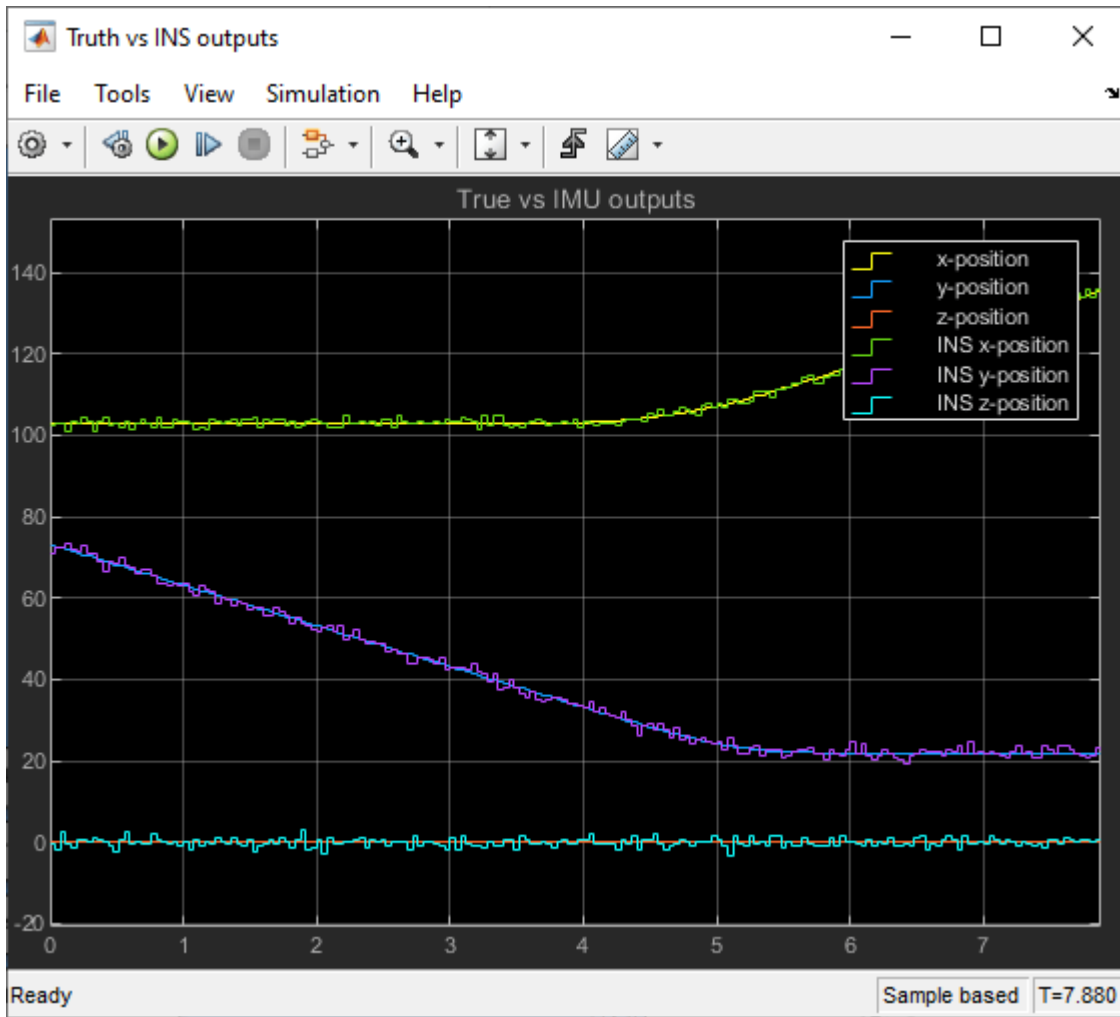
The data importing part imports the vehicle trajectory data into Simulink using the From Workspace (Simulink) block. You use a helper function `helperFromWorkspace`, attached in the example folder, to convert the trajectory data into a structure format required by the From Workspace block.

Run the Model

Run the Simulink model.

```
results = sim('simulateINS');
```

Click on the scope block and see the results. The INS block position outputs closely follow the truth with the addition of noise.



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

This example shows how to use an interactive motion-model based filter along with a smoother to reconstruct the ground truth trajectory based on interpolation techniques.

Using Tracking Filters

Many trajectories can be segmented into a series of discrete maneuvers across time, where each maneuver is governed by a small set of parameters that control how the pose of the object changes over time. These maneuvers often consist of very simple models, for example, a constant velocity, constant turn, or constant acceleration model. These simple models can be used with a tracking filter that is tolerant to small perturbations of the parameters.

While these perturbations should be small, they do not necessarily need to be random. For example, a constant velocity filter can often track an object with a small constant acceleration at the expense of a small bias in the tracking result. Filter performance can drop severely when these perturbations are significant.

In the “Tracking Maneuvering Targets” on page 6-193 example, you learn how to use an interacting multiple-model (IMM) filter to track a target whose trajectory can be divided into three segments: a single constant velocity, a constant turn, and a constant acceleration segment. The IMM filter is a “hybrid” filter that runs a set of filters in parallel and returns a result that weights the estimates of each filter by its performance. You can see the result of the example reproduced below:

```
n = 1000;
[trueState, time, fig1] = helperGenerateTruthLoop(n);
dt = diff(time(1:2));
numSteps = numel(time);

rng(2021); % for repeatable results

% Select position from 9-dimensional state [x;vx;ax;y;vy;ay;z;vz;az]
positionSelector = [1 0 0 0 0 0 0 0 0;
                   0 0 0 1 0 0 0 0 0;
                   0 0 0 0 0 0 1 0 0];
truePos = positionSelector*trueState;
sigma = 10; % Measurement noise standard deviation
measNoise = sigma* randn(size(truePos));
measPos = truePos + measNoise;
initialState = positionSelector'*measPos(:,1);
initialCovariance = diag([1 1e4 1e4 1 1e4 1e4 1 1e4 1e4]); % Velocity and acceleration are not measured

detection = objectDetection(0,[0; 0; 0], 'MeasurementNoise',sigma^2 * eye(3));
f1 = initcaekf(detection); % Constant acceleration EKF
f1.ProcessNoise = 0.3*eye(3);
f2 = initctekf(detection); % Constant turn EKF
f2.ProcessNoise = diag([1 1 100 1]);
f3 = initcvkfk(detection); % Constant velocity EKF
imm = trackingIMM({f1; f2; f3}, 'TransitionProbabilities',0.99, ...
    'ModelConversionFcn',@switchimm);
initialize(imm, initialState, initialCovariance);

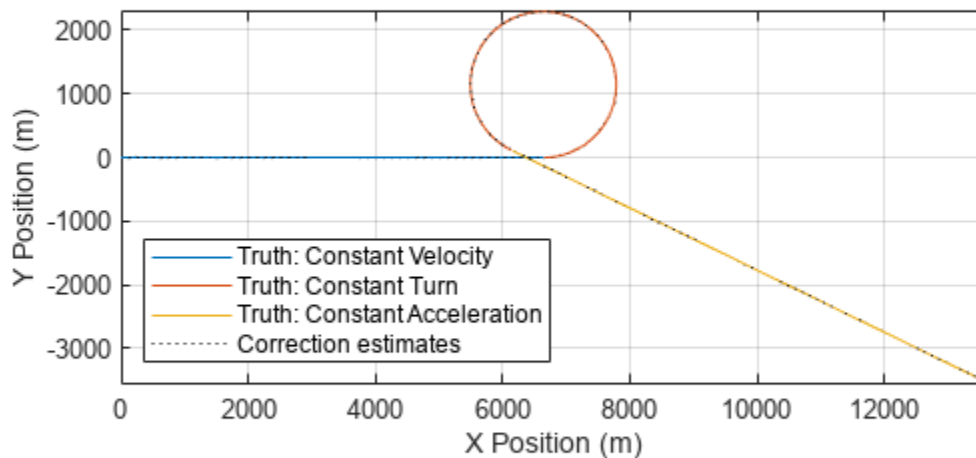
estState = zeros(9,numSteps);
```

```

for i = 2:size(measPos,2)
    predict(imm,dt);
    estState(:,i) = correct(imm,measPos(:,i));
end

figure(fig1);
ax = gca(fig1);
line(ax,estState(1,:),estState(4,:),estState(7,:), 'Color', 'k', 'LineStyle', ':', 'DisplayName', 'Corr');
axis(ax, 'image');

```

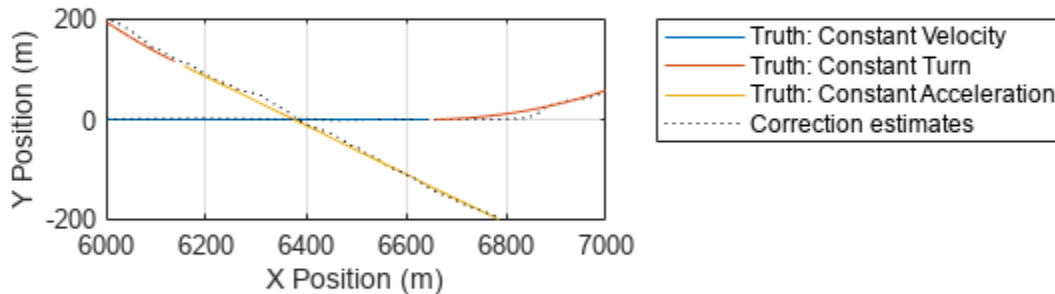


Zoom into the transition regions between each model:

```

axis(ax,[6000 7000 -200 200])
legend(ax, 'Location', 'bestoutside')

```

The IMM filter faces challenges in the time intervals after the motion model changes from one to another. In each transition region the centripetal acceleration is not accounted for in the constant velocity and constant acceleration models. Therefore, the estimation bias persists for a small duration until the IMM filter has enough data to determine the correct motion model. The bias vanishes thereafter until the next transition region.

Smooth Trajectory Estimates

Smoothing is an effective technique to retroactively improve the trajectory estimates. Smoothing is performed by applying both a forwards and backwards filter. The smoother can then choose the "best" of the forwards and backwards predicted regions that mitigate prediction errors in the transition region between maneuvers. You use the same data and filter setup as above, but with the smoothing capability enabled.

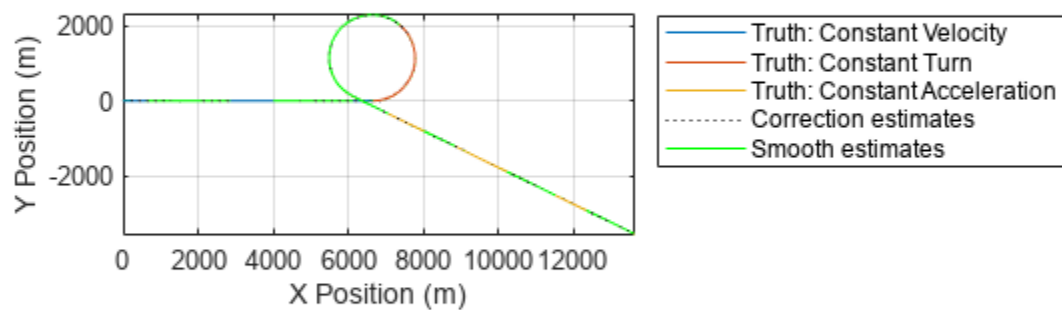
```
imm = trackingIMM({f1; f2; f3}, 'TransitionProbabilities', 0.99, ...
    'ModelConversionFcn', @switchimm, ...
    'EnableSmoothing', true, ...
    'MaxNumSmoothingSteps', size(measPos, 2) - 1);
initialize(imm, initialState, initialCovariance);

estState = zeros(9, numSteps);

for i = 2:size(measPos, 2)
    predict(imm, dt);
    estState(:, i) = correct(imm, measPos(:, i));
end
```

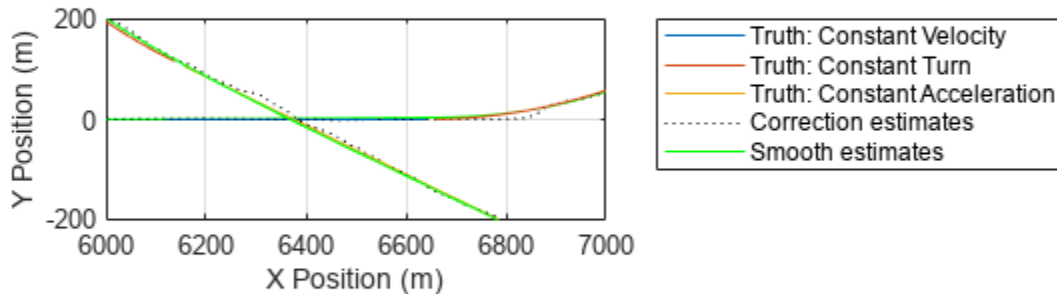
```
smoothState = smooth(imm);
```

```
line(ax,smoothState(1,:),smoothState(4,:),smoothState(7,:), 'Color','g','LineStyle','-','DisplayN  
axis(ax,'image');
```



Zoom in the same region as above and you can see that the smoothed estimates are considerably more successful at recovering the ground truth trajectory.

```
axis(ax,[6000 7000 -200 200])
```



From the results, you can see that smoothing is an effective tool to recover trajectory data when transitioning from one motion model to another with an abrupt change. This forward-backward smoothing often reasonably well in estimating a ground truth trajectory.

Using Low-Order Polynomials for Densely Sampled Data

When comparing tracking algorithms, it can be advantageous to present the tracking results into a format so that pose information can be queried for an arbitrary time instant. When data is sampled at a sufficiently high rate, you can use a set of low-order piecewise polynomials to approximate the data. These polynomials can be differentiated to provide velocity and acceleration information. Additionally, you can use a set of algorithms to infer the orientation of an object based on certain assumptions. For example, assume an object always faces the direction of travel and banks to counteract centripetal forces. The following code shows how to use the smoothed positions and velocities to infer the total acceleration of the object by using a cubic spline.

```
smoothPos = smoothState([1 4 7],:);
smoothVel = smoothState([2 5 8],:);

% Smoothed state vector does not include acceleration
% Create piecewise polynomial from velocity information
velPP = pchip(time(2:end),smoothVel);

% Construct a new piecewise polynomial with the derivative
[breaks,coefs,~,k,dim]=unmkpp(velPP);
coefs = coefs(:,1:k-1).*(k-1:-1:1);
accPP=mkpp(breaks,coefs,dim);
```

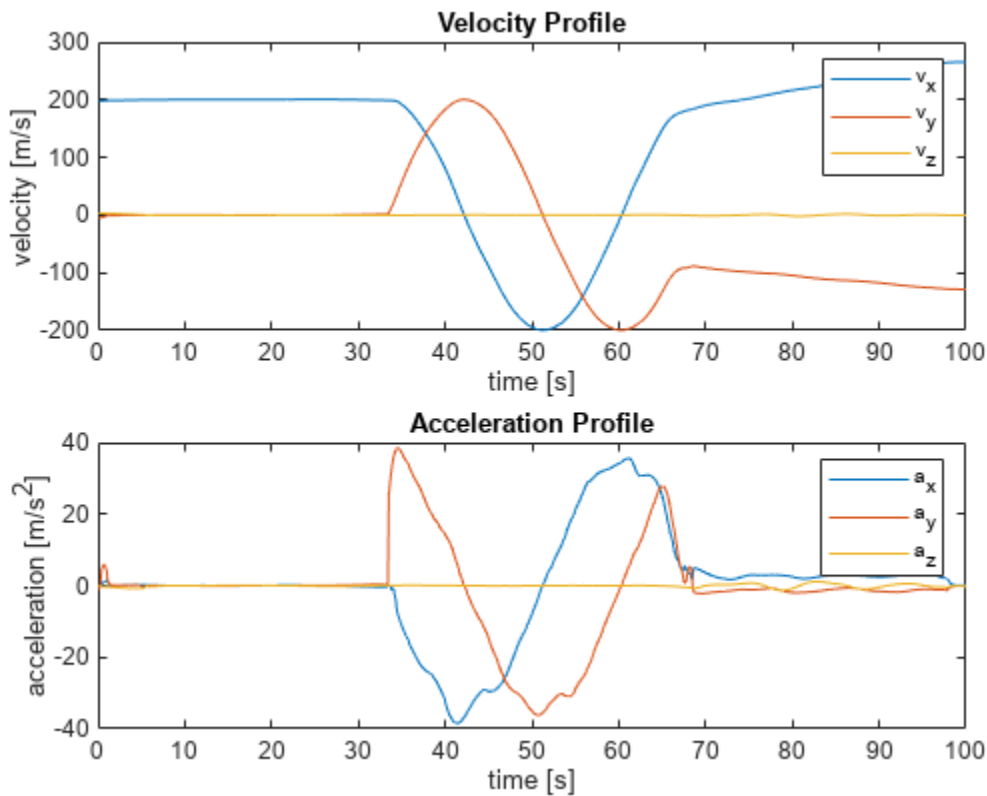
```

% Evaluate the piecewise polynomial to get the acceleration
smoothAcc = ppval(accPP,time(2:end));

% Display the velocity and resulting acceleration
fig = figure;
ax1 = subplot(2,1,1,'Parent',fig);
plot(ax1,time(2:end),smoothVel(1,:),time(2:end),smoothVel(2,:),time(2:end),smoothVel(3,:));
xlabel('time [s]');
ylabel('velocity [m/s]');
title('Velocity Profile');
legend('v_x','v_y','v_z');

ax2 = subplot(2,1,2,'Parent',fig);
plot(ax2,time(2:end),smoothAcc(1,:),time(2:end),smoothAcc(2,:),time(2:end),smoothAcc(3,:));
xlabel('time [s]');
ylabel('acceleration [m/s^2]');
title('Acceleration Profile');
legend('a_x','a_y','a_z');

```



The computed acceleration is not purely sinusoidal while in the constant turn maneuver, which is due to small variations in the smoothed output. Nevertheless, the result provides information to identify the distinct regions of each maneuver.

Though the cubic polynomials are numerically efficient to interpolate the position, they rely on having dense sampling to faithfully represent circular arcs. If greater accuracy is required, you need to use data sampled at a higher rate, which requires an increase of memory resources and a larger lookup table.

Using High-Order Polynomials

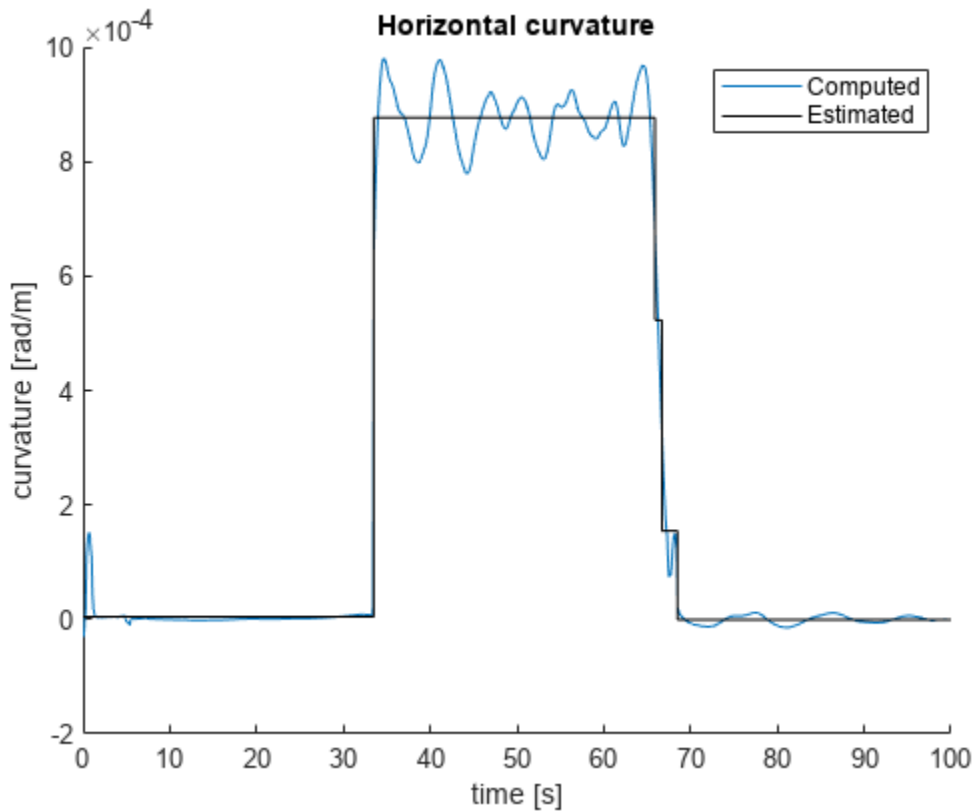
In the previous section you used cubic polynomials to compute the acceleration at each point. For trajectories that consist mainly of circular and straight motion, using an interpolant that favors constant curvature works considerably better than low-order polynomials. The `waypointTrajectory` object can model trajectories with constant turn (constant curvature). It can also faithfully represent trajectories with gradual changes in both curvature and acceleration.

A simple technique to reduce the memory footprint of the interpolant and reduce the magnitude of transients is to use changepoint estimation to locate significant changes in curvature of the trajectory. After that, use those points to compute the interpolant.

```
% Obtain curvature of trajectory projected into x-y plane
planarCurvature = cross(smoothVel,smoothAcc) ./ vecnorm(smoothVel,2).^3;
horizontalCurvature = planarCurvature(3,:);

% Obtain the estimates of the changepoints
threshold = var(horizontalCurvature);
[tf,m] = ischange(horizontalCurvature,'Threshold',threshold);

fig=figure;
hAx=gca(fig);
line(hAx,time(2:end),horizontalCurvature,'DisplayName','Computed');
[xx,yy]=stairs(time(2:end),m);
line(hAx,xx,yy,'DisplayName','Estimated','Color','k')
legend(hAx)
xlabel('time [s]');
ylabel('curvature [rad/m]')
title('Horizontal curvature')
```

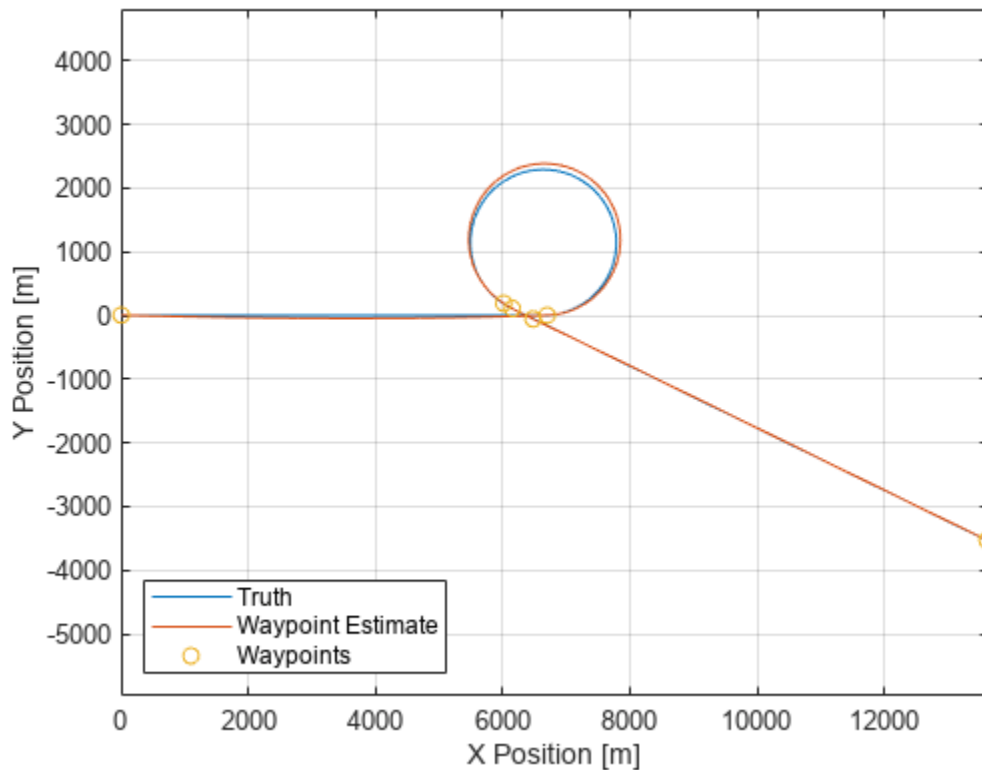


You can then take these changepoints as starting points for constructing the waypoints of the trajectory:

```
idx = 1+[0 find(tf) numel(tf)-1];

waypoints = smoothPos(:,idx)';
wayvels = smoothVel(:,idx)';
timeOfArrival = time(idx);
traj = waypointTrajectory(waypoints,timeOfArrival,'Velocities',wayvels);
[wayPos, ~, wayVel] = lookupPose(traj,time(1:end-1));

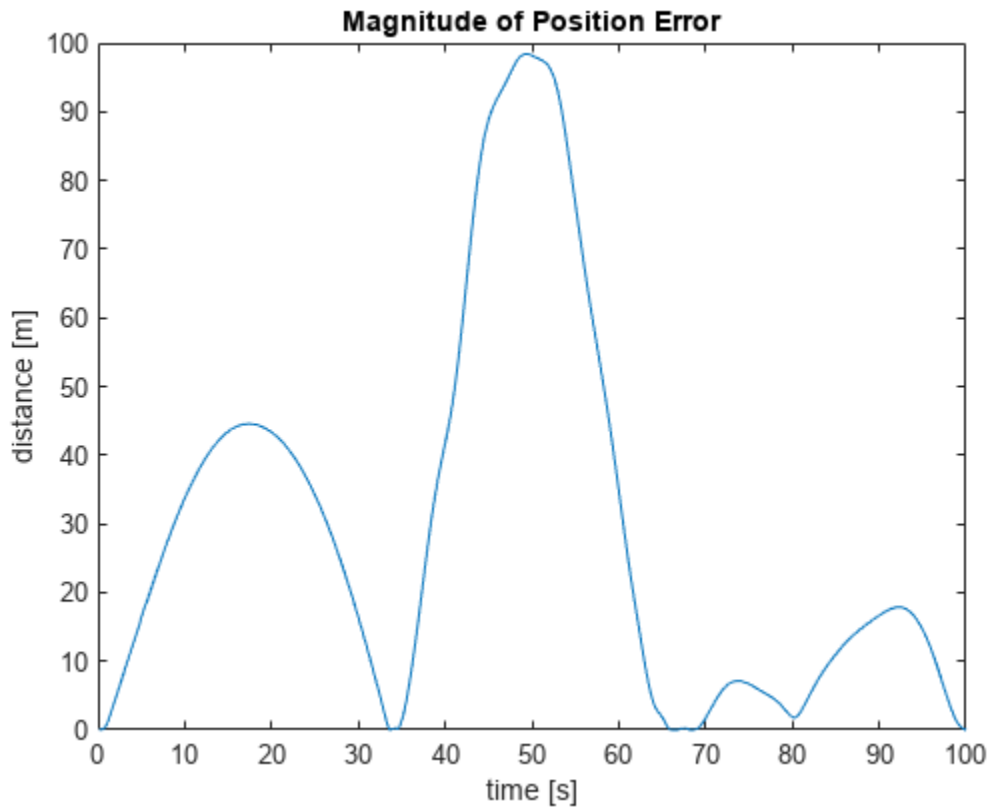
clf
plot(trueState(1,:),trueState(4,:),'-');
hold on;
plot(wayPos(:,1),wayPos(:,2),'-');
plot(waypoints(:,1),waypoints(:,2),'o ');
grid on;
xlabel('X Position [m]');
ylabel('Y Position [m]');
axis equal;
legend('Truth', 'Waypoint Estimate', 'Waypoints', 'Location','southwest')
```



From the results, you recovered a circular trajectory by using just a few waypoints and their corresponding velocities. However, the radius is sensitive to the tangent angles of the velocities estimated within the transition region - which can be a difficult region for a tracking filter to predict.

If you plot the magnitude of the position error of the reconstructed trajectory against the smoothed estimates, you can see localized regions with larger errors:

```
figure;
plot(time(2:end), vecnorm(smoothPos-wayPos',2))
xlabel('time [s]')
ylabel('distance [m]')
title('Magnitude of Position Error');
```



Note the close agreement in regions where the waypoints are close to their smoothed counterparts and the larger deviations in the regions further away from the waypoints. A simple way to reduce the error is to repeatedly add additional waypoints at the positions of maximum error until a desired objective is achieved. In this example, if you set an objective of 10 meters for the maximum error, you only need a few more waypoints.

```

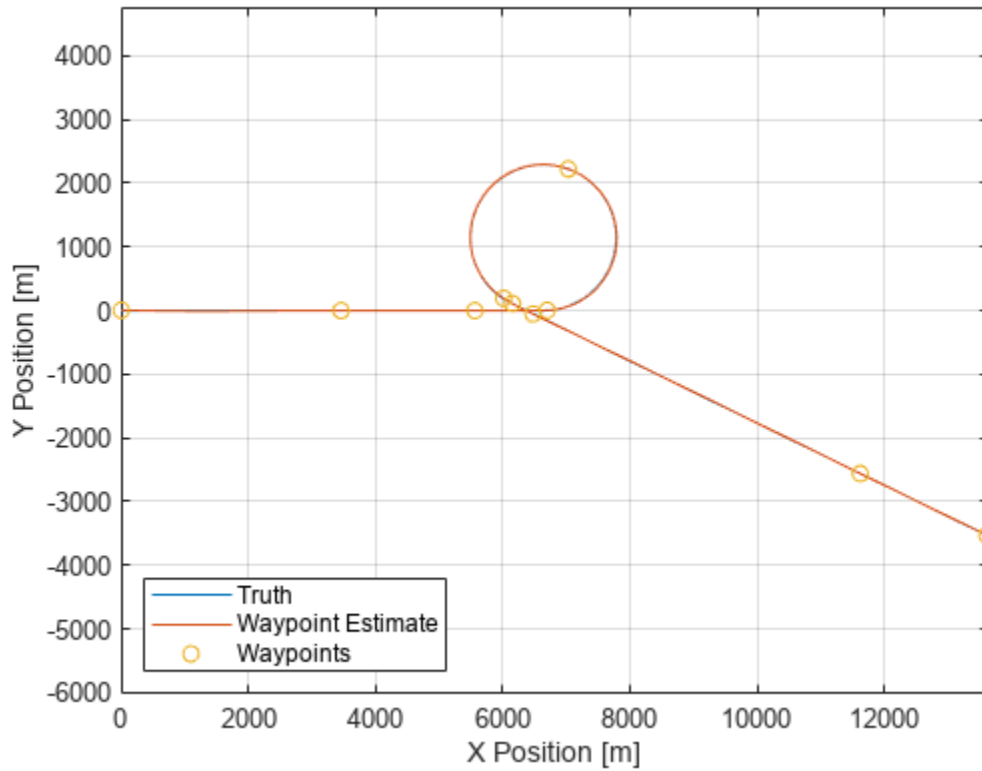
objective = 10; % use 10 m for objective maximal position error
maxError = Inf; % pre-assign error to a maximal value
while maxError > objective
    [~,maxi] = max(vecnorm(smoothPos-wayPos',2));
    idx = unique(sort([maxi idx]));
    waypoints = smoothPos(:,idx)';
    wayvels = smoothVel(:,idx)';
    timeOfArrival = time(idx);
    traj = waypointTrajectory(waypoints,timeOfArrival,'Velocities',wayvels);
    [wayPos, ~, wayVel, wayAcc] = lookupPose(traj,time(1:end-1));
    maxError = max(vecnorm(smoothPos-wayPos',2));
end

figure;
plot(trueState(1,:),trueState(4:5,:), '-');
hold on;
plot(wayPos(:,1),wayPos(:,2), '-');
plot(waypoints(:,1),waypoints(:,2), 'o ');
grid on;
xlabel('X Position [m]');
ylabel('Y Position [m]');

```

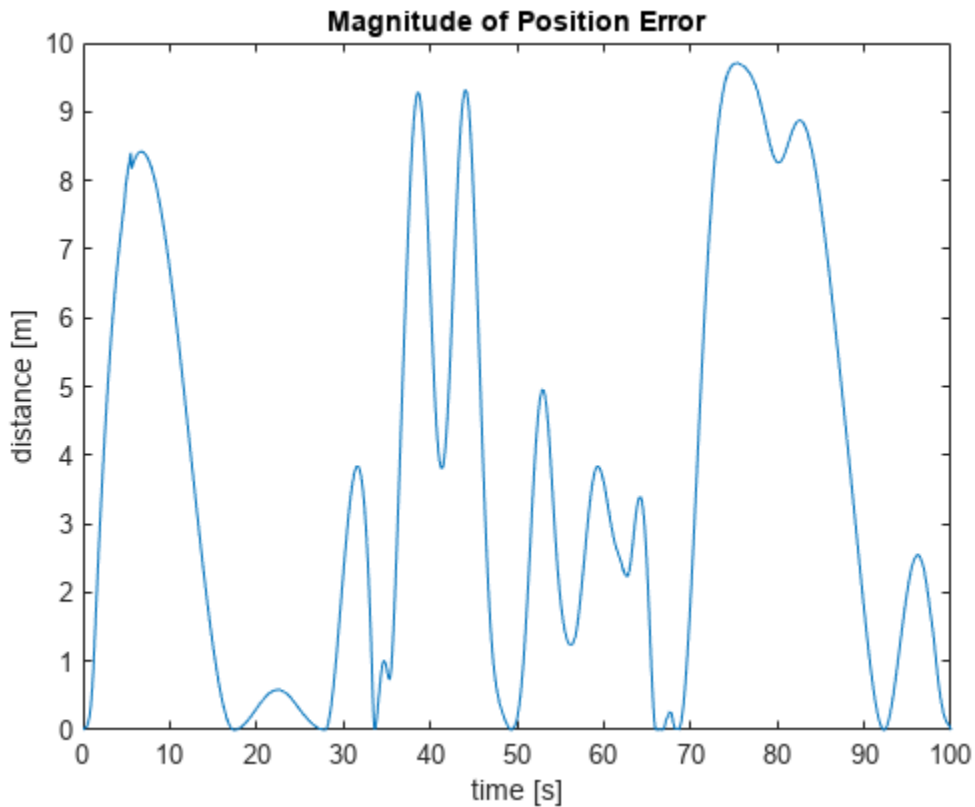


```
axis equal;
legend('Truth', 'Waypoint Estimate', 'Waypoints', 'Location', 'southwest')
```



You can then observe the reduced position error.

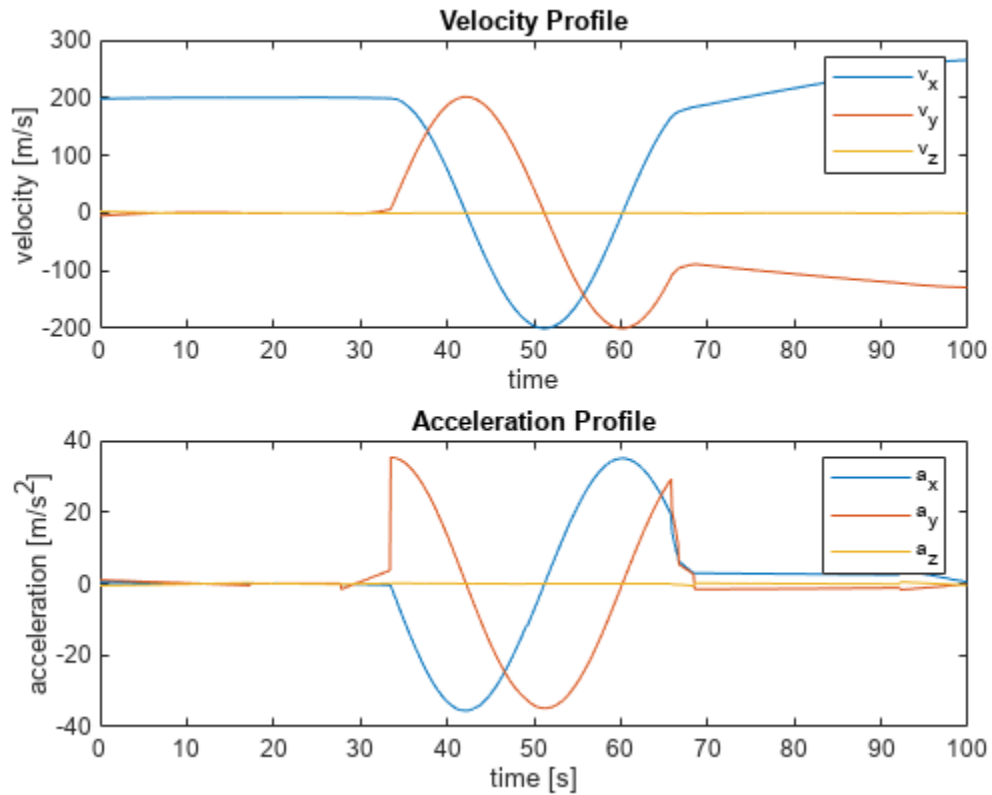
```
figure;
plot(time(2:end), vecnorm(smoothPos-wayPos',2))
xlabel('time [s]')
ylabel('distance [m]')
title('Magnitude of Position Error');
```



Now that you have a reasonable position profile, you can inspect the resulting velocity and acceleration profiles from the trajectory. Note the overall improvement in the recovery of the sinusoidal acceleration profile.

```
fig = figure;
ax1 = subplot(2,1,1,'Parent',fig);
plot(ax1,time(1:end-1),wayVel);
title('Velocity Profile')
xlabel('time');
ylabel('velocity [m/s]')
legend('v_x','v_y','v_z')

ax2 = subplot(2,1,2,'Parent',fig);
plot(ax2,time(1:end-1),wayAcc);
title('Acceleration Profile')
xlabel('time [s]');
ylabel('acceleration [m/s^2]')
legend('a_x','a_y','a_z')
```



Summary

In this example you learned how to reconstruct a ground truth trajectory using an interacting multiple-model smoother and low order polynomials. You also used a simple scheme that used a high-order interpolant (contained in the `waypointTrajectory` object) along with critical sampling to recover higher-order derivatives that would otherwise be masked by noise.

Asynchronous Sensor Fusion and Tracking with Retrodiction

This example shows how to construct an asynchronous sensor fusion and tracking model in Simulink®.

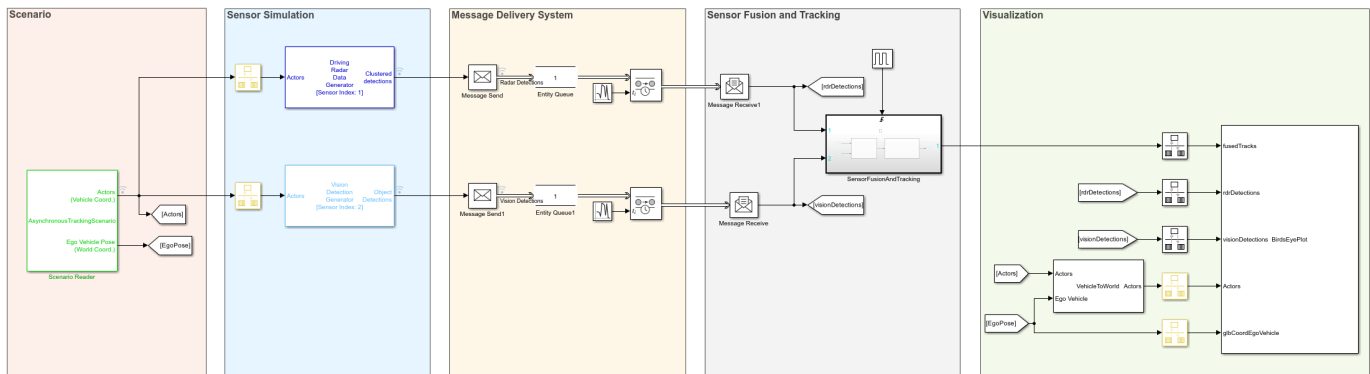
Introduction

In this example you create a model for sensor fusion and tracking by simulating radar and vision camera, each running at a different update rate. The sensors and the tracker run on separate electronic control units (ECUs). The tracker runs asynchronously from the sensors at a different update rate.

Model Description

Open the Simulink model using the `open_system` command.

```
open_system('AsynchronousTrackingModel.slx');
```



The model consists of five parts.

The Scenario part of the model consists of a Scenario Reader block, which loads the scenario saved in `AsynchronousTrackingScenario.mat`. In the scenario, there are 4 vehicles: the ego vehicle, a car in front of it, a passing car, and a car behind the ego car.

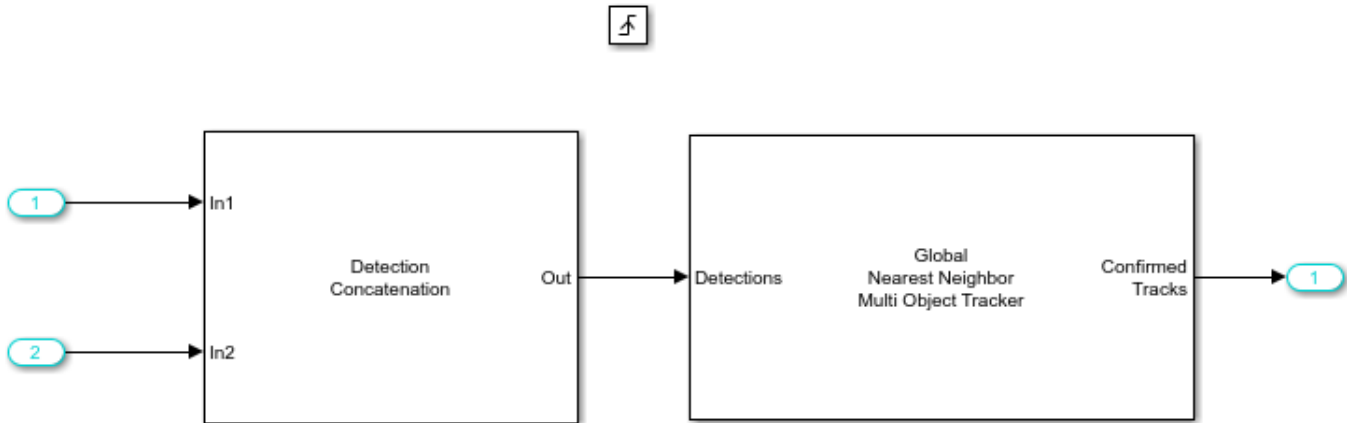
The ego car has two sensors: a radar and a vision camera. The radar is simulated using the Driving Radar Data Generator block, running at 25 Hz, or every 40 milliseconds. The camera is simulated using a Vision Detection Generator block, running every 44 milliseconds. Due to the combination of sensor rates, the scenario needs to run at a rate of at least 250 Hz, or every 4 milliseconds. Both sensor models are shown in the Sensor Simulation part of the model.

The Message Delivery System part of the model simulates the asynchronous communication system between the sensors and the tracker. Each sensor outputs a bus of detections that is packed into by a Message Send block and delivered to an Entity Queue. The queues are organized as a last-in-first-out (LIFO) queue with a capacity of 1. The queues store only latest data from sensors. The Entity Transport Delay blocks are used to simulate communication delays in the network. The blocks delay an entity for a period of time received at its second input port. In this example you use the Random Number block to generate mean delay values of 40 and 44 milliseconds for radar and vision sensors respectively.

The Sensor Fusion and Tracking part of the model consists of the Message Receive blocks and a Triggered Subsystem block. The Message Receive blocks read the messages and pass their payload to

the subsystem. The subsystem is triggered using an external signal at 20Hz or 50 milliseconds. This means that the tracker is updated every 50 milliseconds. The trigger signal is generated using the Pulse Generator block.

```
open_system('AsynchronousTrackingModel/SensorFusionAndTracking');
```



In the subsystem, the Detection Concatenation block concatenates detections from both sensors and passes them to the tracker.

The final part of the model is the Visualization, where all the scenario, sensor, and tracking data are visualized by a helper block.

Configure the Tracker to Use Retrodiction

For the tracker, you use a Global Nearest Neighbor Tracker block. You modify the tracker to use the retrodiction technique for out-of-sequence measurement (OOSM) handling. When using the tracker in an asynchronous way, the tracker clock may run ahead of the messages that arrive, which renders the messages 'out-of-sequence'. Setting the tracker OOSM handling to retrodiction allows the tracker to process them instead of terminating with an error or neglecting the OOSM. Overall, retrodiction improves the tracker accuracy.

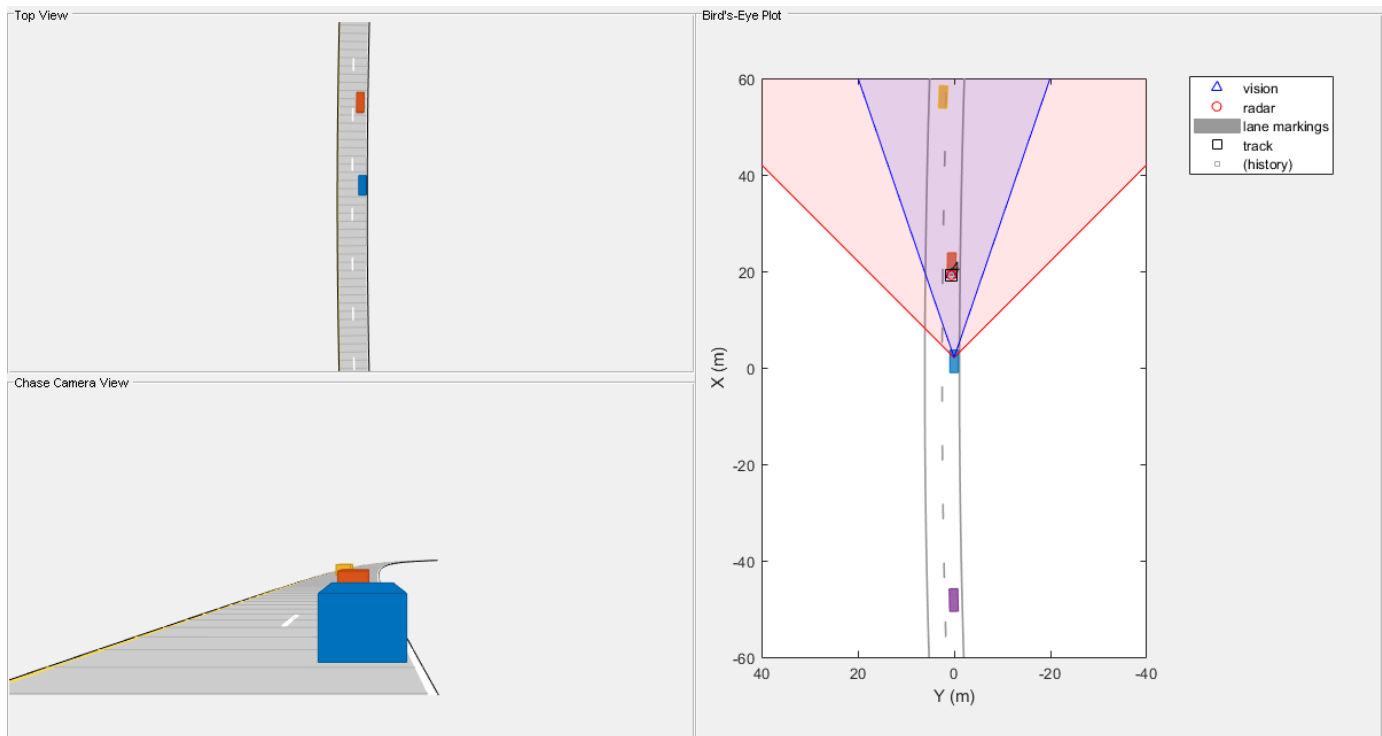
Using retrodiction requires more memory because the tracker must maintain a history of each track. To reduce the memory allocation, you reduce the maximum number of tracks to 20, because there are just a few objects in the scenario. Similarly, you reduce the maximum number of sensors to 2, because only two sensors report to the tracker.

You increase the threshold for assigning detections to tracks from the default 30 to 50 to allow vision and radar detections that may have different offsets on the object to be assigned to the same track. A larger assignment threshold may result in false detections getting assigned to each other and creating false tracks. To reduce the rate of false tracks, you make the confirmation threshold stricter by increasing it from the default 2-out-of-3 to 4-out-of-5 detections.

Run the Model and See the Results

You build and run the model using the command below.

```
sim('AsynchronousTrackingModel.slx');
close_system('AsynchronousTrackingModel.slx');
```



The simulation shows that the tracker tracks the vehicle in front of the ego vehicle after a few steps required for confirmation and maintains the track throughout the scenario. The passing vehicle, in yellow, is tracked only after it enters the field of view of sensors. The vehicle behind the ego vehicle is never detected by any sensor and therefore it is never tracked.

Summary

This example showed you how to use an asynchronous sensor fusion and tracking system. The example showed how to connect sensors with different update rates using an asynchronous tracker and how to trigger the tracker to process sensor data at a different rate from sensors. The tracker uses the retrodiction out-of-sequence measurement handling technique to process sensor data that arrives out of sequence.

Object Tracking and Motion Planning Using Frenet Reference Path

This example shows you how to dynamically replan the motion of an autonomous vehicle based on the estimate of the surrounding environment. You use a Frenet reference path and a joint probabilistic data association (JPDA) tracker to estimate and predict the motion of other vehicles on the highway. Compared to the “Highway Trajectory Planning Using Frenet Reference Path” (Navigation Toolbox) example, you use these estimated trajectories from the multi-object tracker in this example instead of ground truth for motion planning.

Introduction

Dynamic replanning for autonomous vehicles is typically done with a local motion planner. The local motion planner is responsible for generating optimal trajectory based on a global plan and real-time information about the surrounding environment. The global plan for highway trajectory planning can be described as a pre-generated coordinate list of the highway centerline. The surrounding environment can be described mainly in two ways:

- 1 Discrete set of objects in the surrounding environment with defined geometries.
- 2 Discretized grid with estimates about free and occupied regions in the surrounding environment.

In the presence of dynamic obstacles, a local motion planner also requires predictions about the surroundings to assess the validity of planned trajectories. In this example, you represent the surrounding environment using the *discrete set of objects* approach. For an example using discretized grid, refer to the “Motion Planning in Urban Environments Using Dynamic Occupancy Grid Map” on page 6-679 example.

Object State Transition and Measurement Modeling

The object list and their future predictions for motion planning are typically estimated by a multi-object tracker. The multi-object tracker accepts data from sensors and estimates the list of objects. In the tracking community, this list of objects is often termed as *track list*.

In this example, you use radar and camera sensors and estimate the track list using a JPDA multi-object tracker. The first step towards using any multi-object tracker is defining the object state, how the state evolves with time (state transition model) and how the sensor perceives it (measurement model). Common state transition models include constant-velocity model, constant-acceleration model etc. However, in the presence of map information, road network can be integrated into the motion model. In this example, you use a Frenet coordinate system to describe the object state at any given time step, k .

$$x_k = [s_k \dot{s}_k \ d_k \ \dot{d}_k]$$

where s_k and d_k represents the distance of the object along and perpendicular to highway centerline, respectively. You use a constant-speed state transition model to describe the object motion along the highway and a decaying-speed model to describe the motion perpendicular to the highway centerline. This decaying speed model allows you to represent lane change maneuvers by other vehicles on the highway.

$$\begin{bmatrix} s_{k+1} \\ \dot{s}_{k+1} \\ d_{k+1} \\ \dot{d}_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \tau \left(1 - e^{-\frac{\Delta T}{\tau}} \right) \\ 0 & 0 & 0 & e^{-\frac{\Delta T}{\tau}} \end{bmatrix} \begin{bmatrix} s_k \\ \dot{s}_k \\ d_k \\ \dot{d}_k \end{bmatrix} + \begin{bmatrix} \frac{\Delta T^2}{2} & 0 \\ \Delta T & 0 \\ 0 & \frac{\Delta T^2}{2} \\ 0 & \Delta T \end{bmatrix} \begin{bmatrix} w_s \\ w_d \end{bmatrix}$$

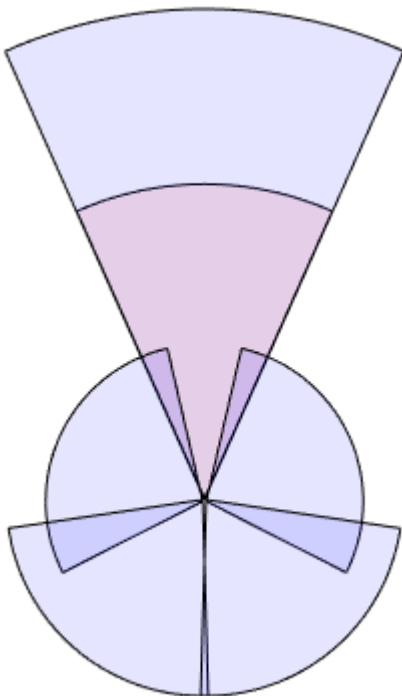
where ΔT is the time difference between steps k and $k + 1$, w_s and w_d are zero-mean Gaussian noise representing unknown acceleration in Frenet coordinates, and τ is a decaying constant.

This choice of coordinate in modeling the object motion allows you to integrate the highway reference path into the multi-object tracking framework. The integration of reference path acts as additional information for the tracker and allows the tracker to improve current state estimates as well as predicted trajectories of the estimated objects. You can obtain measurement model by first transforming the object state into Cartesian position and velocity and then converting them to respective measured quantities such as azimuth and range.

Setup

Scenario and Sensors

The scenario used in this example is created using the Driving Scenario Designer (Automated Driving Toolbox) and then exported to a MATLAB® function. The ego vehicle is mounted with 1 forward-looking radar and 5 cameras providing 360-degree coverage. The radar and cameras are simulated using the `drivingRadarDataGenerator` (Automated Driving Toolbox) and `visionDetectionGenerator` (Automated Driving Toolbox) System objects, respectively.



The entire scenario and sensor setup is defined in the helper function, `helperTrackingAndPlanningScenario`, attached with this example. You define the global plan describing the highway centerline using a `referencePathFrenet` (Navigation Toolbox) object. As multiple algorithms in this example need access to the reference path, you define the `helperGetReferencePath` function, which uses a persistent object that can be accessed by any function.

```
rng(2022); % For reproducible results

% Setup scenario and sensors
[scenario, egoVehicle, sensors] = helperTrackingAndPlanningScenario();
```

Joint Probabilistic Data Association Tracker

You set up a joint probabilistic data association tracker using the `trackerJPDA` System object. You set the `FilterInitializationFcn` property of the tracker to `helperInitRefPathFilter` function. This helper function defines an extended Kalman filter, `trackerJPDA`, used to estimate the state of a single object. Local functions inside the `helperInitRefPathFilter` file define the state transition as well as measurement model for the filter. Further, to predict the tracks at a future time for the motion planner, you use the `predictTracksToTime` function of the tracker.

```
tracker = trackerJPDA('FilterInitializationFcn',@helperInitRefPathFilter,...
    'AssignmentThreshold',[200 inf],...
    'ConfirmationThreshold',[8 10],...
    'DeletionThreshold',[5 5]);
```

Motion Planner

You use a similar highway trajectory motion planner as outlined in the “Highway Trajectory Planning Using Frenet Reference Path” (Navigation Toolbox) example. The motion planner uses a planning horizon of 5 seconds and considers three modes for sampling trajectories for the ego vehicle — cruise control, lead vehicle follow, and basic lane change. The entire process for generating an optimal trajectory is wrapped in the helper function, `helperPlanHighwayTrajectory`.

The helper function accepts an `dynamicCapsuleList` (Navigation Toolbox) object as an input to find non-colliding trajectories. The collision checking is performed in the entire planning horizon at an interval of 0.5 seconds. As the track states vary with time, you update the `dynamicCapsuleList` object in the simulation loop using the `helperUpdateCapsuleList` function, attached with this example.

```
% Collision check time stamps
tHorizon = 5; % seconds
deltaT = 0.5; % seconds
tSteps = deltaT:deltaT:tHorizon;

% Create the dynamicCapsuleList object
capList = dynamicCapsuleList;
capList.MaxNumSteps = numel(tSteps) + 1;

% Specify the ego vehicle geometry
carLen = 4.7;
carWidth = 1.8;
rearAxleRatio = 0.25;
egoID = 1;
[egoID, egoGeom] = egoGeometry(capList,egoID);
```

```

% Inflate to allow uncertainty and safety gaps
egoGeom.Geometry.Length = 2*carLen; % in meters
egoGeom.Geometry.Radius = carWidth/2; % in meters
egoGeom.Geometry.FixedTransform(1,end) = -2*carLen*rearAxleRatio; % in meters
updateEgoGeometry(capList, egoID, egoGeom);

```

Run Simulation

In this section, you advance the simulation, generate sensor data and perform dynamic replanning using estimations about the surroundings. The entire process is divided into 5 main steps:

- 1 You collect simulated sensor data from radar and camera sensors.
- 2 You feed the sensor data to the JPDA tracker to estimate current state of objects.
- 3 You predict the state of objects using the `predictTracksToTime` function.
- 4 You update the object list for the planner and plan a highway trajectory.
- 5 You move the simulated ego vehicle on the planned trajectory.

```

% Create display for visualizing results
display = HelperTrackingAndPlanningDisplay;

```

```

% Initial state of the ego vehicle
refPath = helperGetReferencePath;
egoState = frenet2global(refPath, [0 0 0 0.5*3.6 0 0]);
helperMoveEgoToState(egoVehicle, egoState);

```

```

while advance(scenario)
    % Current time
    time = scenario.SimulationTime;

    % Step 1. Collect data
    detections = helperGenerateDetections(sensors, egoVehicle, time);

    % Step 2. Feed detections to tracker
    tracks = tracker(detections, time);

    % Step 3. Predict tracks in planning horizon
    timesteps = time + tSteps;
    predictedTracks = repmat(tracks, [1 numel(timesteps)+1]);
    for i = 1:numel(timesteps)
        predictedTracks(:,i+1) = predictTracksToTime(tracker, 'confirmed', timesteps(i));
    end

    % Step 4. Update capsule list and plan highway trajectory
    currActorState = helperUpdateCapsuleList(capList, predictedTracks);
    [optimalTrajectory, trajectoryList] = helperPlanHighwayTrajectory(capList, currActorState, egoVehicle);

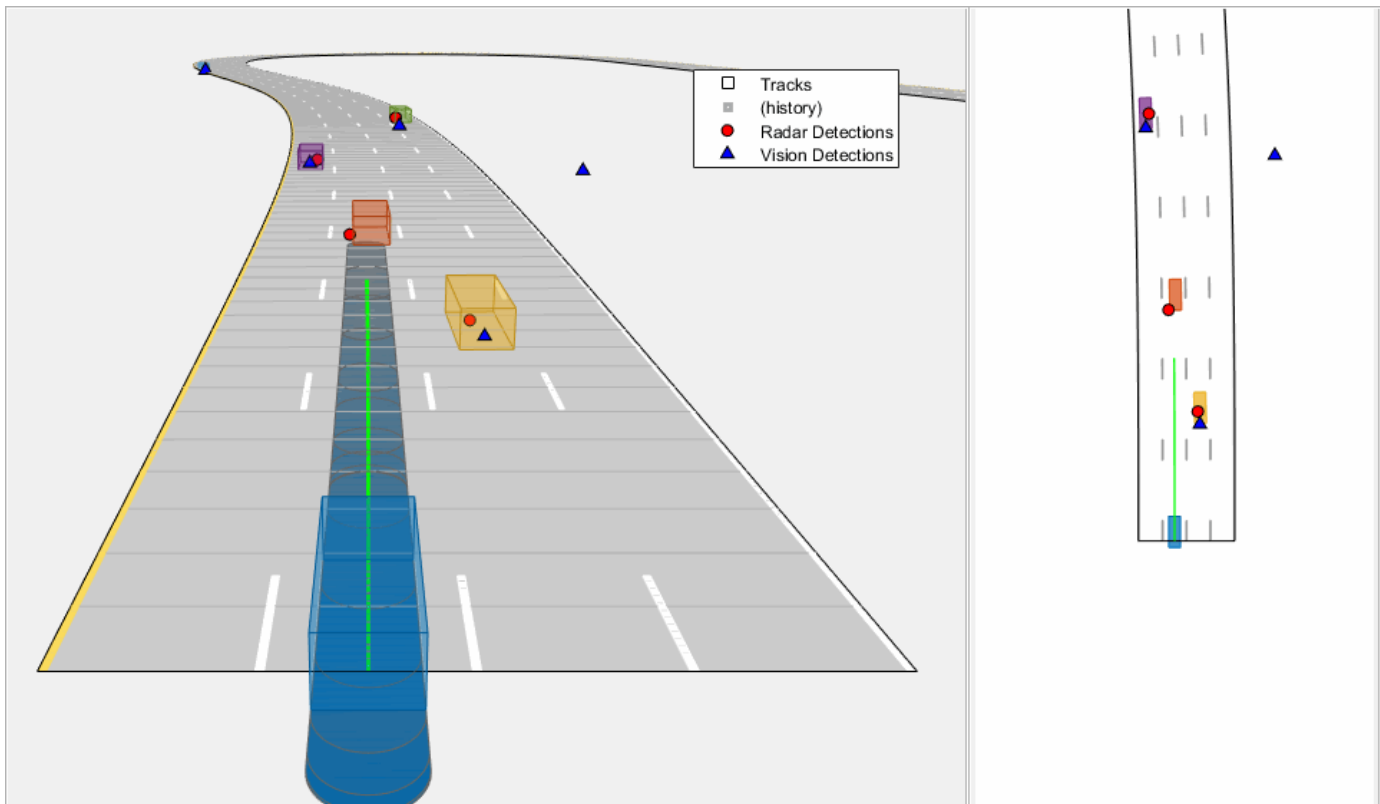
    % Visualize the results
    display(scenario, egoVehicle, sensors, detections, tracks, capList, trajectoryList);

    % Step 5. Move ego on planned trajectory
    egoState = optimalTrajectory(2,:);
    helperMoveEgoToState(egoVehicle, egoState);
end

```

Results

In the animation below, you can observe the planned ego vehicle trajectories highlighted in green color. The animation also shows all other sampled trajectories for the ego vehicle. For these other trajectories, the colliding trajectories are shown in red, unevaluated trajectories are shown in grey, and kinematically-infeasible trajectories are shown in cyan color. Each track is annotated by an ID representing its unique identity. Notice that the ego vehicle successfully maneuvers around obstacles in the scene.

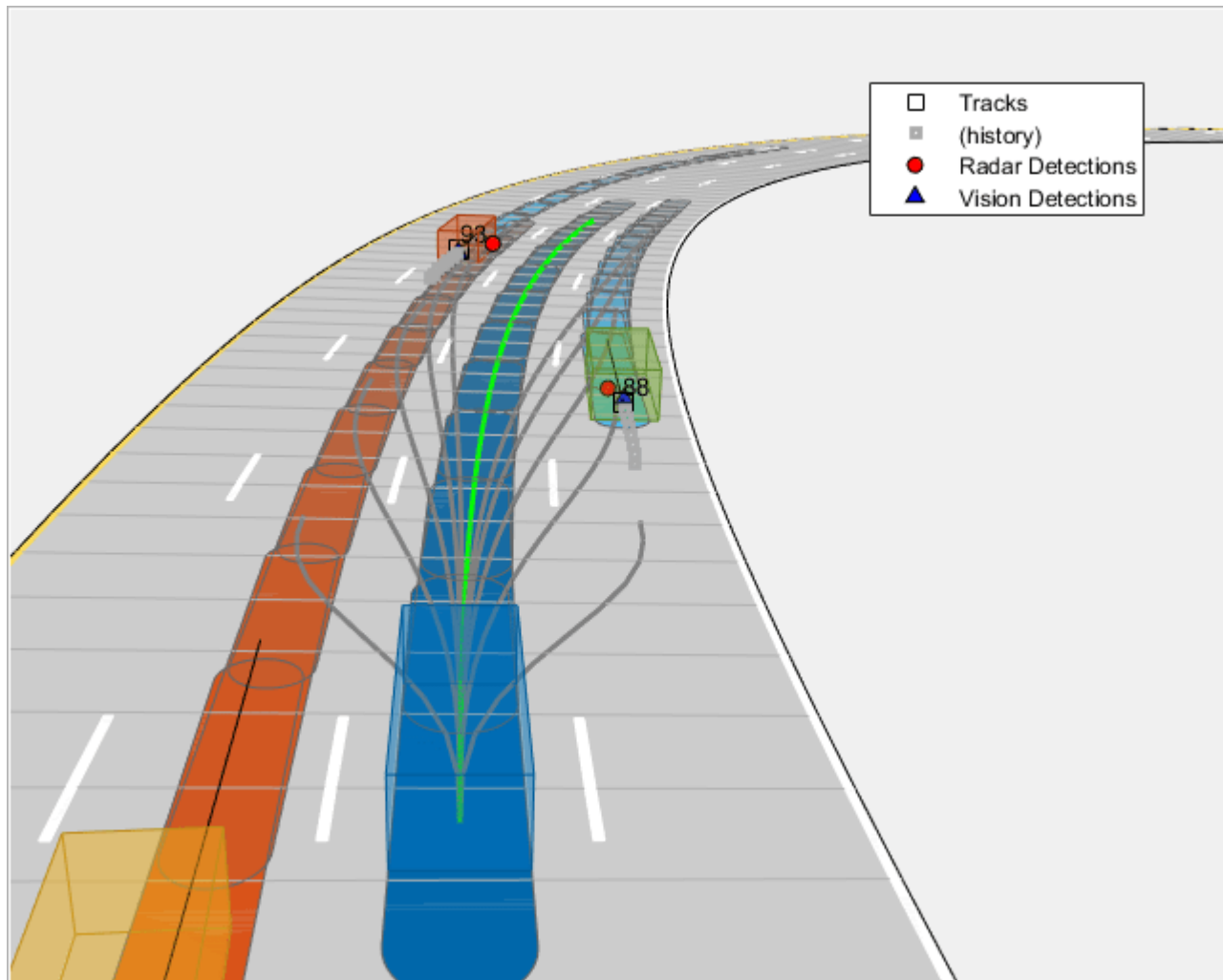


In the following sub-sections, you analyze the estimates from the tracker at certain time steps and understand how it impacts the choices made by the motion planner.

Road-integrated motion prediction

In this section, you learn how the road-integrated motion model allows the tracker to obtain more accurate long-term predictions about the objects on the highway. Shown below is a snapshot from the simulation taken at time = 30 seconds. Notice the trajectory predicted for the green vehicle to the right of the blue ego vehicle. The predicted trajectory follows the lane of the vehicle because the road network information is integrated with the tracker. If instead, you use a constant-velocity model assumption for objects, the predicted trajectory will follow the direction of instantaneous velocity and will be falsely treated as a collision by the motion planner. In this case, the motion planner can possibly generate an unsafe maneuver.

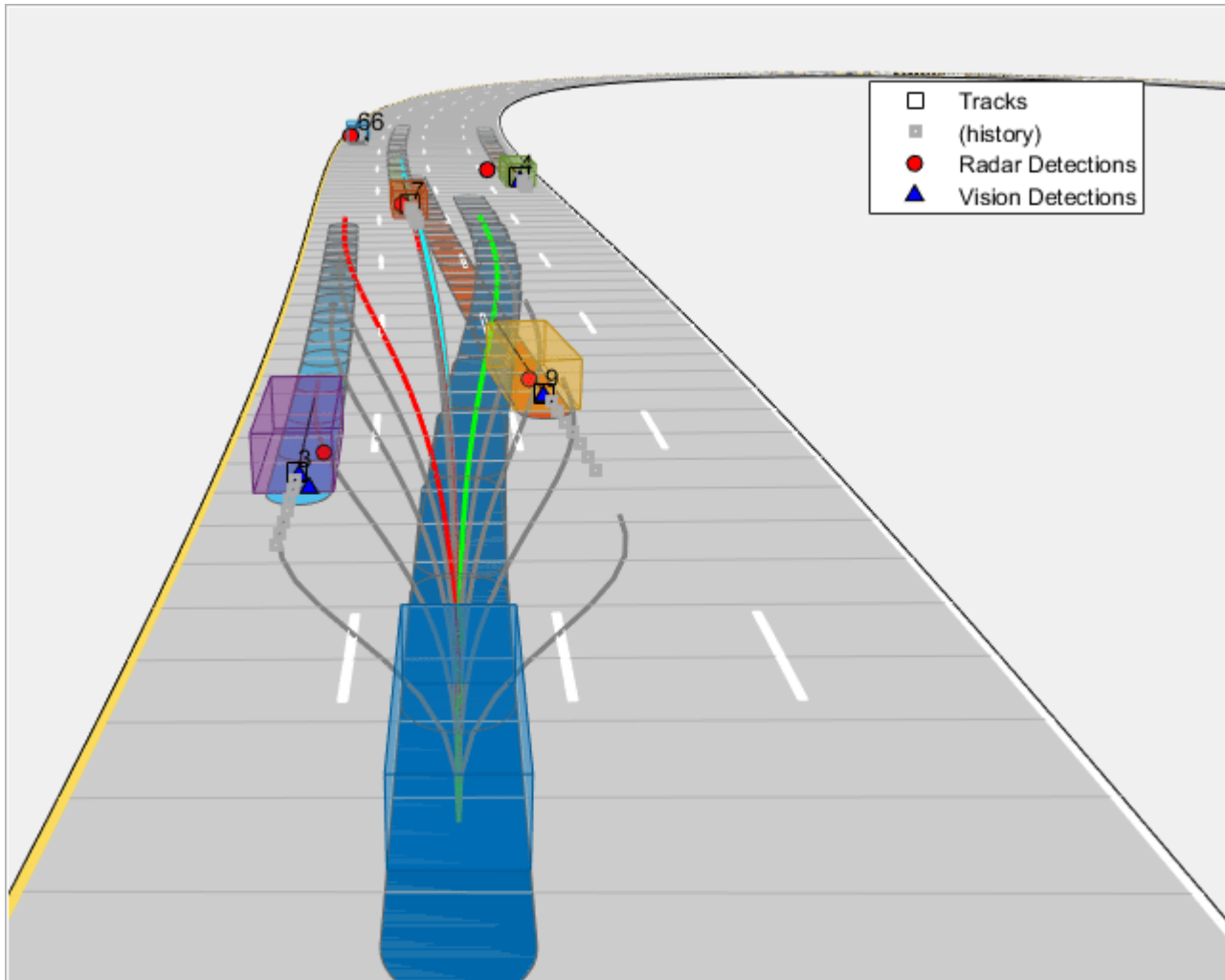
```
showSnaps(display,2,4); % Shows snapshot while publishing
```



Lane change prediction

In the first section, you learned how the lane change maneuvers are captured by using a decaying lateral velocity model of the objects. Now, notice the snapshot taken at time = 17.5 seconds. At this time, the yellow vehicle on the right side of the ego vehicle initiates a lane change and intends to enter the lane of the ego vehicle. Notice that its predicted trajectory captures this maneuver, and the tracker predicts it to be in the same lane as the ego vehicle at the end of planning horizon. This prediction informs the motion planner about a possible collision with this vehicle, thus the planner first proceeds to test feasibility for the ego vehicle to change lane to the left. However, the presence of purple vehicle on the left and its predicted trajectory causes the ego vehicle to make a right lane change. You can also observe these colliding trajectories colored as red in the snapshot below.

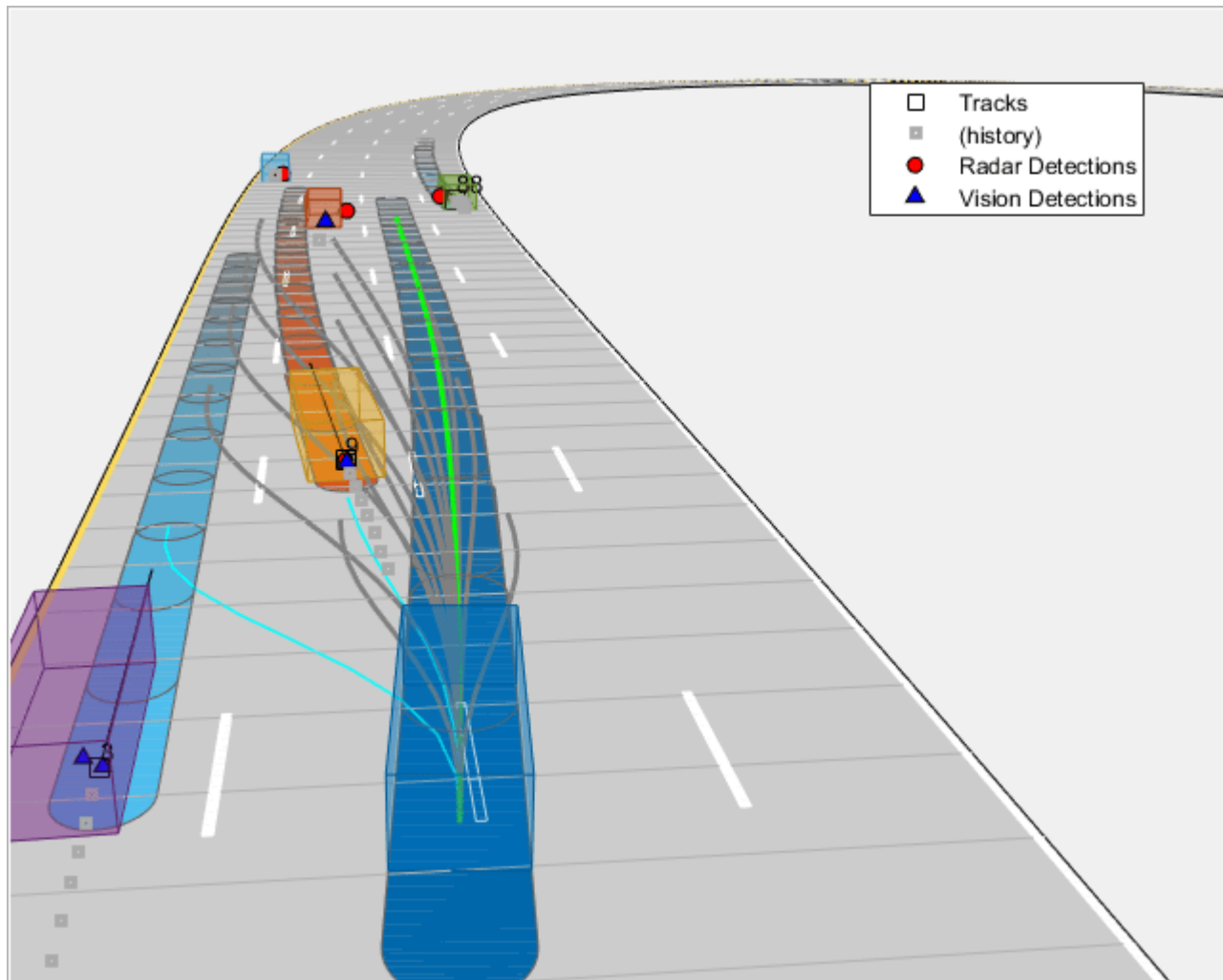
```
showSnaps(display,2,1); % Shows snapshot while publishing
```



Tracker imperfections

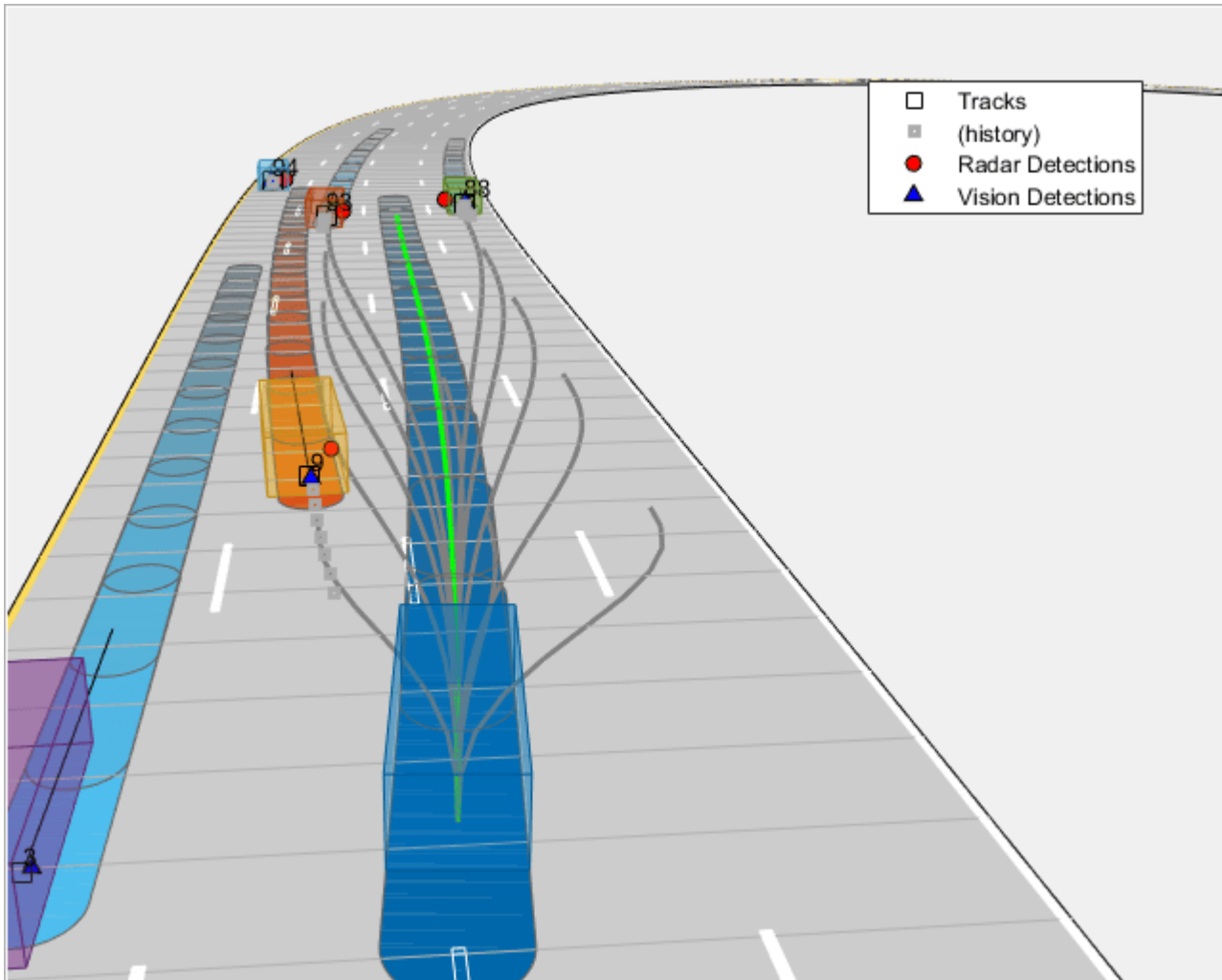
A multi-object tracker may have certain imperfections that can affect motion planning decisions. Specifically, a multi-object tracker can miss objects, report false tracks, or sometimes report redundant tracks. In the snapshot below taken at time = 20 seconds, the tracker drops tracks on two vehicles in front of the ego vehicle due to occlusion. In this particular situation, these missed targets are less likely to influence the decision of the motion planner due to their distance from the ego vehicle.

```
showSnaps(display,2,2); % Shows snapshot while publishing
```



However, as the ego vehicle approaches these vehicles, their influence on the ego vehicle's decision increases. Notice that the tracker is able to establish a track on these vehicles by time = 20.4 seconds, as shown in the snapshot below, thus making the system slightly robust to these imperfections. While configuring a tracking algorithm for motion planning, it is important to consider these imperfections from the tracker and tune the track confirmation and track deletion logics.

```
showSnaps(display,2,3); % Show snapshot while publishing
```



Summary

You learned how to use a joint probabilistic data association tracker to track vehicles using a Frenet reference path with radar and camera sensors. You configured the tracker to use highway map data to provide long term predictions about objects. You also used these long-term predictions to drive a motion planner for planning trajectories on the highway.

Supporting Functions

```
function detections = helperGenerateDetections(sensors, egoVehicle, time)
    detections = cell(0,1);
    for i = 1:numel(sensors)
        thisDetections = sensors{i}(targetPoses(egoVehicle),time);
        detections = [detections;thisDetections]; %#ok<AGROW>
    end

    detections = helperAddEgoVehicleLocalization(detections,egoVehicle);
    detections = helperPreprocessDetections(detections);
end

function detectionsOut = helperAddEgoVehicleLocalization(detectionsIn, egoPose)
```

```

defaultParams = struct('Frame','Rectangular',...
    'OriginPosition',zeros(3,1),...
    'OriginVelocity',zeros(3,1),...
    'Orientation',eye(3),...
    'HasAzimuth',false,...
    'HasElevation',false,...
    'HasRange',false,...
    'HasVelocity',false);

fNames = fieldnames(defaultParams);

detectionsOut = cell(numel(detectionsIn),1);

for i = 1:numel(detectionsIn)
    thisDet = detectionsIn{i};
    if iscell(thisDet.MeasurementParameters)
        measParams = thisDet.MeasurementParameters{1};
    else
        measParams = thisDet.MeasurementParameters(1);
    end

    newParams = struct;
    for k = 1:numel(fNames)
        if isfield(measParams,fNames{k})
            newParams.(fNames{k}) = measParams.(fNames{k});
        else
            newParams.(fNames{k}) = defaultParams.(fNames{k});
        end
    end

    % Add parameters for ego vehicle
    thisDet.MeasurementParameters = [newParams;newParams];
    thisDet.MeasurementParameters(2).Frame = 'Rectangular';
    thisDet.MeasurementParameters(2).OriginPosition = egoPose.Position(:);
    thisDet.MeasurementParameters(2).OriginVelocity = egoPose.Velocity(:);
    thisDet.MeasurementParameters(2).Orientation = rotmat(quaternion([egoPose.Yaw egoPose.Pitch

    % No information from object class and attributes
    thisDet.ObjectClassID = 0;
    thisDet.ObjectAttributes = struct;
    detectionsOut{i} = thisDet;
end

end

function detections = helperPreprocessDetections(detections)
    % This function pre-process the detections from radars and cameras to
    % fit the modeling assumptions used by the tracker

    % 1. It removes velocity information from camera detections. This is
    % because those are filtered estimates and the assumptions from camera
    % may not align with defined prior information for tracker.
    %
    % 2. It fixes the bias for camera sensors that arise due to camera
    % projections for cars just left or right to the ego vehicle.
    %

```



```

% 3. It inflates the measurement noise for range-rate reported by the
% radars to match the range-rate resolution of the sensor
for i = 1:numel(detections)
    if detections{i}.SensorIndex > 1 % Camera
        % Remove velocity
        detections{i}.Measurement = detections{i}.Measurement(1:3);
        detections{i}.MeasurementNoise = blkdiag(detections{i}.MeasurementNoise(1:2,1:2),25)
        detections{i}.MeasurementParameters(1).HasVelocity = false;

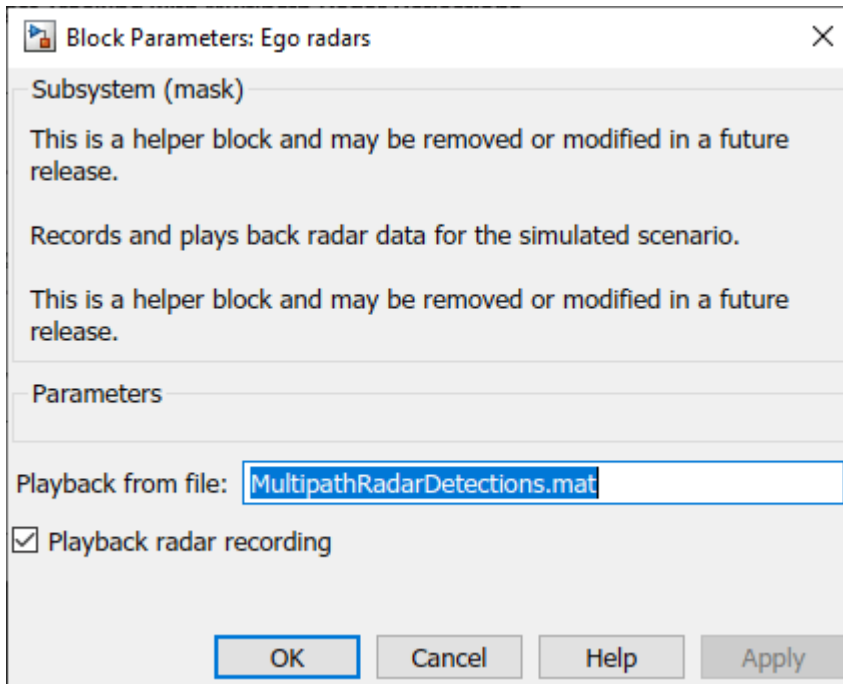
        % Fix bias
        pos = detections{i}.Measurement(1:2);
        if abs(pos(1)) < 5 && abs(pos(2)) < 5
            [az, ~, r] = cart2sph(pos(1),pos(2),0);
            [pos(1),pos(2)] = sph2cart(az, 0, r + 0.7); % Increase range
            detections{i}.Measurement(1:2) = pos;
            detections{i}.MeasurementNoise(2,2) = 0.25;
        end
    else % Radars
        detections{i}.MeasurementNoise(3,3) = 0.5^2/4;
    end
end
end

function helperMoveEgoToState(egoVehicle, egoState)
egoVehicle.Position(1:2) = egoState(1:2);
egoVehicle.Velocity(1:2) = [cos(egoState(3)) sin(egoState(3))]*egoState(5);
egoVehicle.Yaw = egoState(3)*180/pi;
egoVehicle.AngularVelocity(3) = 180/pi*egoState(4)*egoState(5);
end

```


Use the **Ego radars** helper block to play back detections recorded from four radars providing full 360 degree coverage around the ego vehicle. To record a new set of detections, clear the **Playback radar recording** check box.

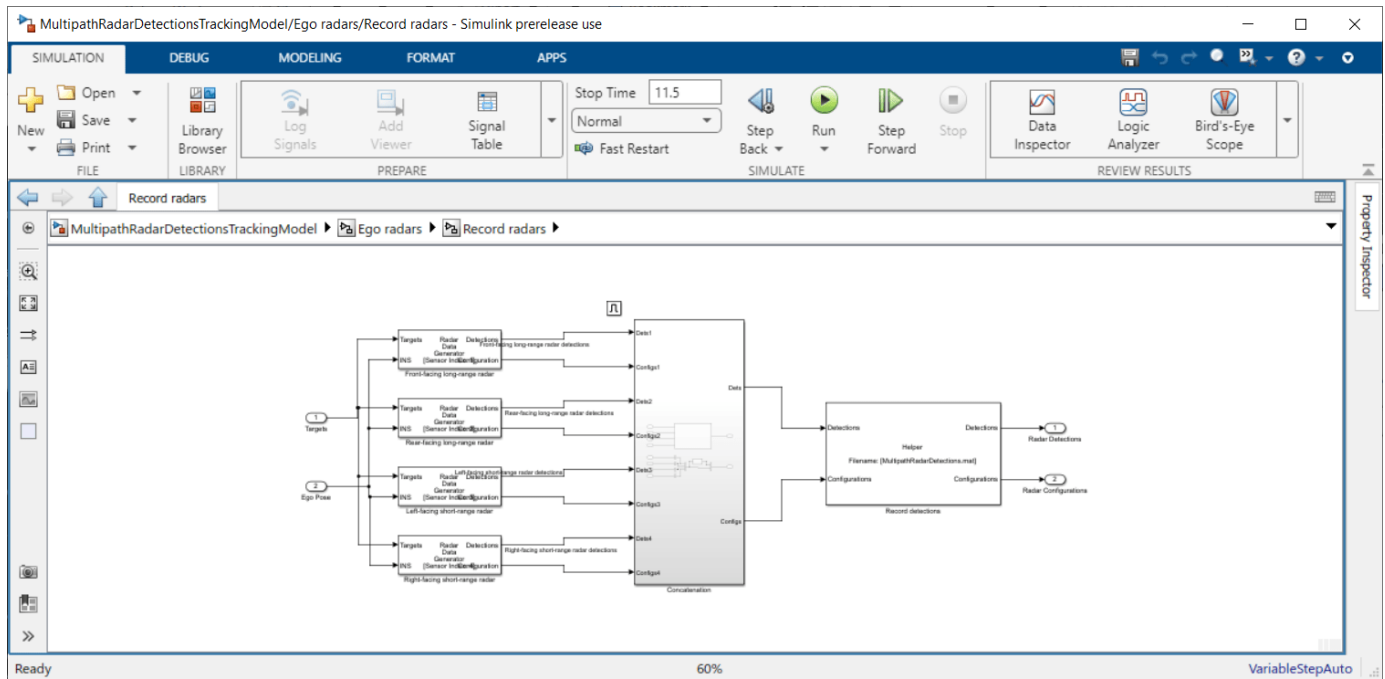
```
open_system('MultipathRadarDetectionsTrackingModel/Ego radars')
```



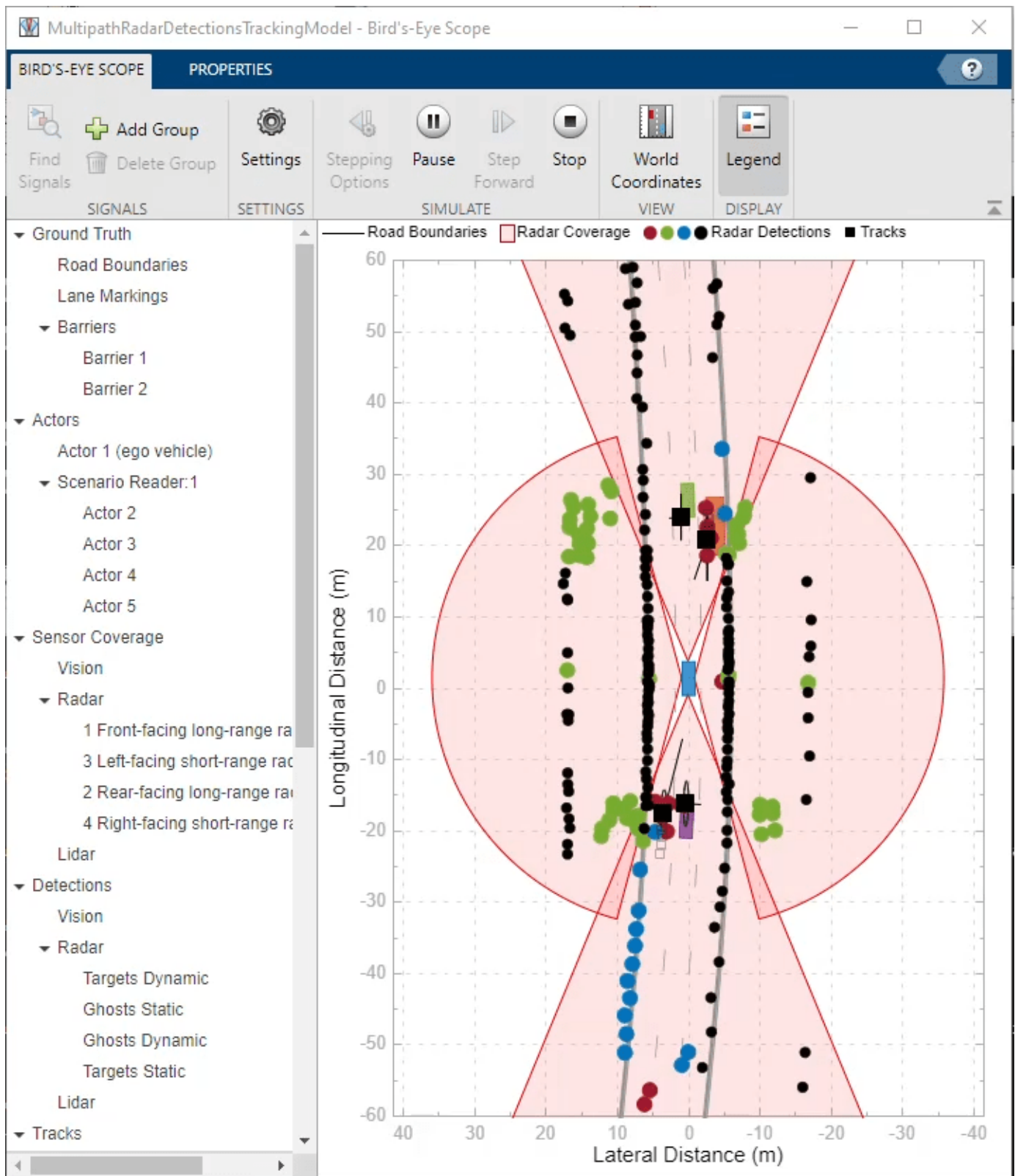
```
close_system('MultipathRadarDetectionsTrackingModel/Ego radars')
```

The four sensor models are configured in the **Record radars** block.

```
open_system('MultipathRadarDetectionsTrackingModel/Ego radars/Record radars')
```



Use Bird's-Eye Scope (Automated Driving Toolbox) to visualize the scenario and sensor coverage in this model.



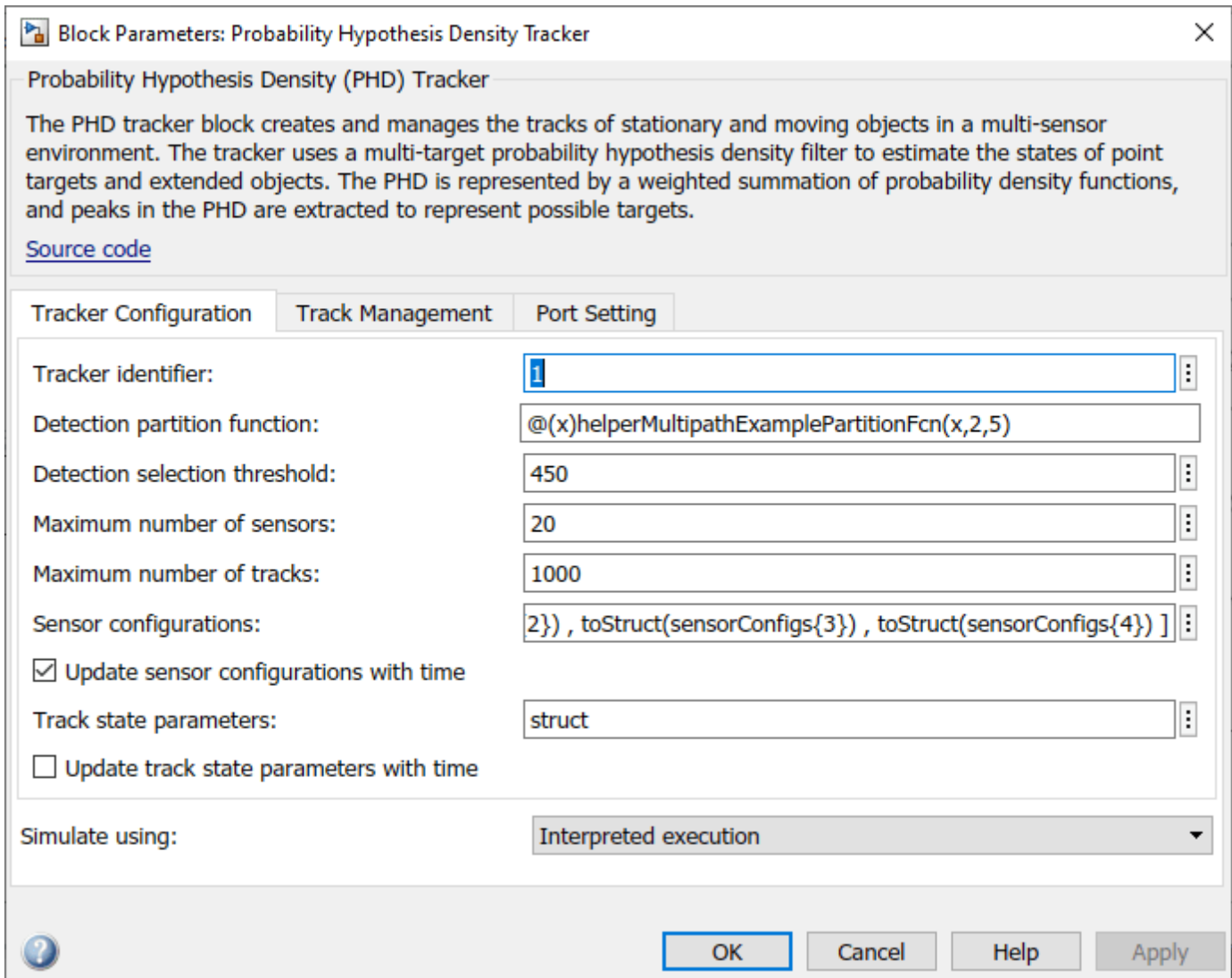
The **Classify detections** helper block classifies the detections generated by the four radars by comparing their measurements to the confirmed tracks from the Probability Hypothesis Density (PHD) Tracker block. The detection classification utilizes the measured radial velocity from the targets to determine if the target generating the detection was static or dynamic [1]. The detections are classified into four categories:

- 1 Dynamic targets — These (red) detections are classified to originate from real dynamic targets in the scene.
- 2 Static ghosts — These (green) detections are classified to originate from dynamic targets but reflected via the static environment.
- 3 Dynamic ghosts — These (blue) detections are classified to originate from dynamic targets but reflected via other dynamic objects.
- 4 Static targets — These (black) detections are classified to originate from the static environment.

Configure GGIW-PHD Extended Object Tracker

Configure the Probability Hypothesis Density (PHD) Tracker block with the same parameters as used by the “Highway Vehicle Tracking with Multipath Radar Reflections” (Radar Toolbox) example. Reuse the `helperMultipathExamplePartitionFcn` function to define the detection partitions used within the tracker.

```
open_system('MultipathRadarDetectionsTrackingModel/Probability Hypothesis Density Tracker')
```

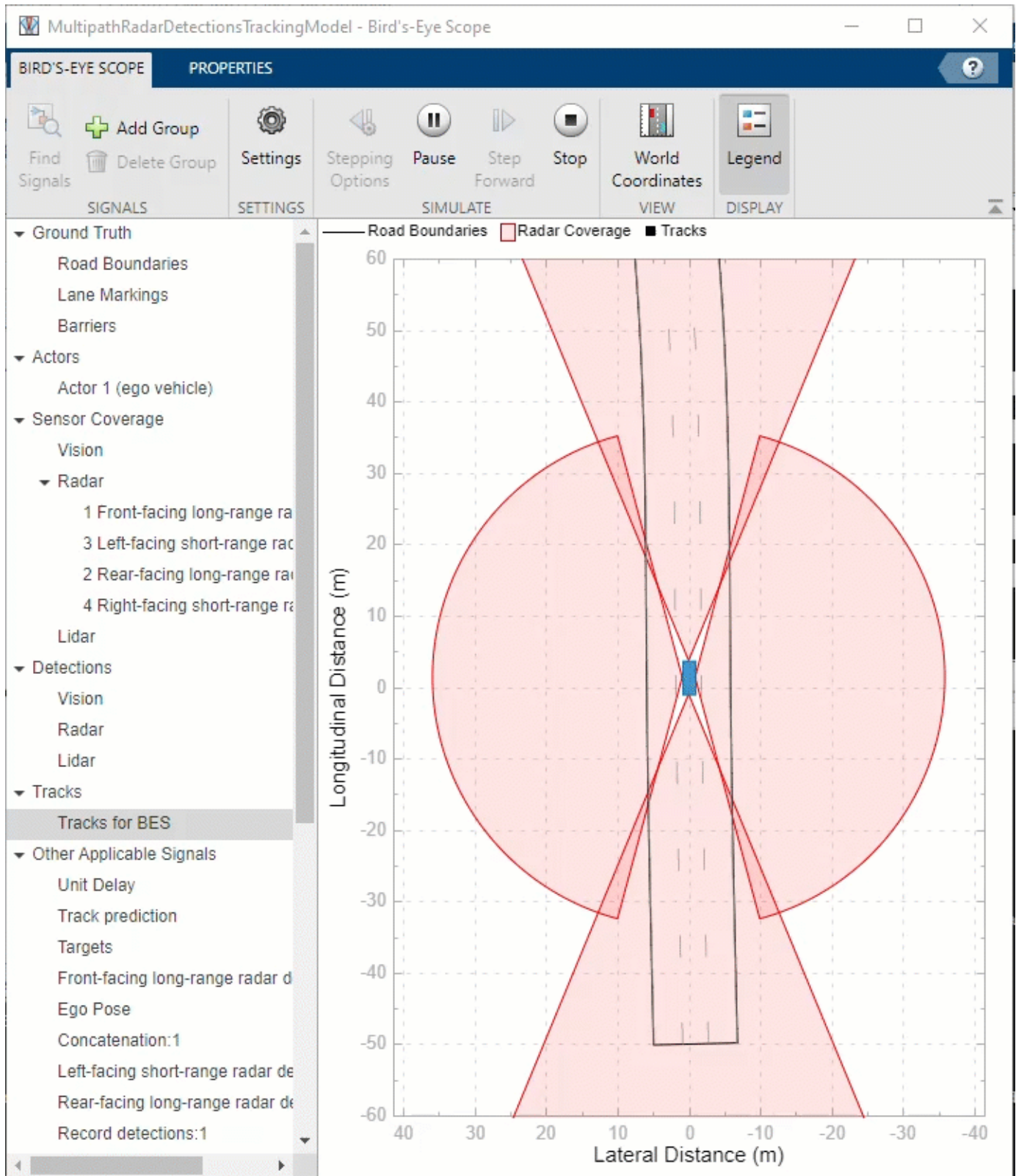


```
close_system('MultipathRadarDetectionsTrackingModel/Probability Hypothesis Density Tracker',0)
```

Run Simulation

Use the following command to play back the recorded detections and generate tracks.

```
simout = sim('MultipathRadarDetectionsTrackingModel')
```



Use `helperSaveSimulationLogs` to save the logged tracks and classified detections for offline analysis.

```
helperSaveSimulationLogs('MultipathRadarDetectionsTrackingModel',simout);
```

Analyze Performance

Load the logged tracks and detections to assess the performance of the tracking algorithm by using the GOSPA metric and its associated components.

```
[confirmedTracks,confusionMatrix] = helperLoadSimulationLogs('MultipathRadarDetectionsTrackingMo
```

Use `trackGOSPAMetric` to calculate GOSPA metrics from the logged tracks.

```
gospaMetric = trackGOSPAMetric('Distance','custom', ...
    'DistanceFcn',@helperGOSPADistance, ...
    'CutoffDistance',35);

% Number of simulated track updates
numSteps = numel(confirmedTracks.Time);

% GOSPA metric
gospa = NaN(4,numSteps);

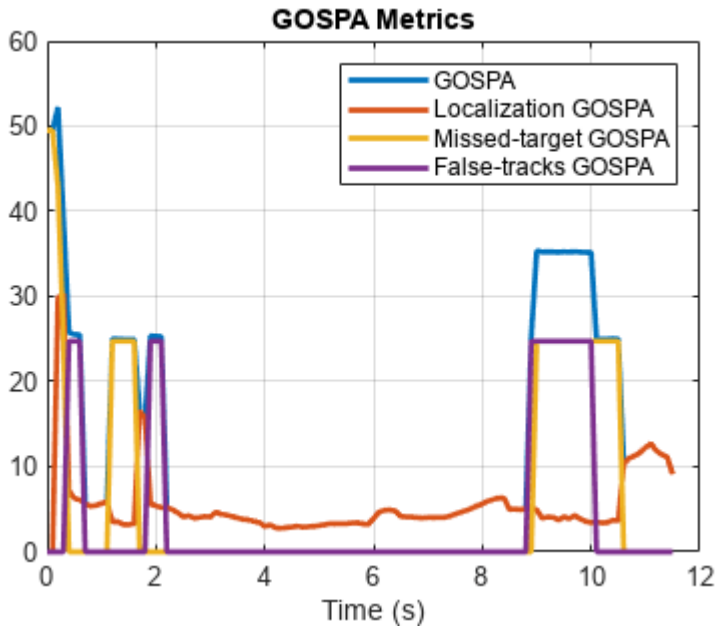
restart(scenario);
groundTruth = scenario.Operators(2:end);
iStep = 1;
tol = seconds(scenario.SampleTime/4);
while scenario.SimulationTime<=seconds(confirmedTracks.Time(end))
    % Select data from time table for current simulation time
    tsim = scenario.SimulationTime;
    wt = withtol(seconds(tsim),tol);

    % Select tracks from time table and compute GOSPA metrics
    theseTracks = confirmedTracks{wt,'Tracks'}{1};
    [gospa(1,iStep),~,~,gospa(2,iStep),gospa(3,iStep),gospa(4,iStep)] = gospaMetric(theseTracks,

    if scenario.IsRunning
        advance(scenario);
    else
        break
    end
    iStep = iStep+1;
end
```

Quantitatively assess the performance of the tracking algorithm by using the GOSPA metric and its associated components. A lower value of the metric denotes better tracking performance. In the following figure, the Missed-target component of the metric remains zero after a few steps in the beginning, representing establishment delay of the tracker. This component shows that no targets were missed by the tracker. The False-tracks component of the metric is zero for most of the simulation, indicating that no false tracks were confirmed by the tracker during those times.

```
% Plot GOSPA metrics
plot(seconds(confirmedTracks.Time),gospa','LineWidth',2);
xlabel('Time (s)');
title('GOSPA Metrics');
grid on;
legend('GOSPA','Localization GOSPA','Missed-target GOSPA','False-tracks GOSPA');
```



Similar to the tracking algorithm, you also quantitatively analyze the performance of the radar detection classification algorithm by using a confusion matrix [2]. The rows shown in the table denote the true classification information of the radar detections and the columns represent the predicted classification information. For example, the second element of the first row defines the percentage of target detections predicted as ghosts from static object reflections.

91% of the target detections are classified correctly. However, a small percentage of the target detections are misclassified as ghosts from dynamic reflections. Also, approximately 4% of ghosts from static object reflections and 22% of ghosts from dynamic object reflections are misclassified as targets and sent to the tracker for processing. A common situation when this occurs in this example is when the detections from two-bounce reflections lie inside the estimated extent of the vehicle. Further, the classification algorithm used in this example is not designed to find false alarms or clutter in the scene. Therefore, the fifth column of the confusion matrix is zero. Due to spatial distribution of the false alarms inside the field of view, the majority of false alarm detections are either classified as reflections from static objects or dynamic objects.

```
% Accumulate confusion matrix over all steps
confMat = shiftdim(reshape([confusionMatrix{:}, 'Confusion Matrix'}], numSteps, 5, 5), 1);
confMat = sum(confMat, 3);

% Number of detections for each target type
numDetections = sum(confMat, 2);

numDetsTable = array2table(numDetections, 'RowNames', {'Targets', 'Ghost (S)', 'Ghost (D)', 'Environment'},
    'VariableNames', {'Number of Detections'});

disp('True Information'); disp(numDetsTable);
```

True Information

	Number of Detections
Targets	1990
Ghost (S)	3242

```

Ghost (D)          848
Environment        27451
Clutter            139

```

```
% Calculate classification percentages
```

```
percentMatrix = confMat./numDetections*100;
```

```
percentMatrixTable = array2table(round(percentMatrix,2), 'RowNames', {'Targets', 'Ghost (S)', 'Ghost (D)', 'Environment', 'Clutter'}, 'VariableNames', {'Targets', 'Ghost (S)', 'Ghost (D)', 'Environment', 'Clutter'});
```

```
disp('True vs Predicted Confusion Matrix (%)'); disp(percentMatrixTable);
```

```

True vs Predicted Confusion Matrix (%)
                Targets    Ghost (S)    Ghost (D)    Environment    Clutter
                _____    _____    _____    _____    _____
Targets          90.9         0.75         7.94         0.4            0
Ghost (S)        3.52         84.86        11.32        0.31           0
Ghost (D)        22.29        0.24         77.48         0              0
Environment      1.53         2.93         3.42         92.12          0
Clutter          19.42        66.19        13.67        0.72           0

```

Summary

In this example, you simulated radar detections due to multipath propagation in an urban highway driving scenario using Simulink. You configured a data processing algorithm to simultaneously filter ghost detections and track vehicles on the highway. You also analyzed the performance of the tracking algorithm and the classification algorithm using the GOSPA metric and confusion matrix.

References

[1] Prophet, Robert, et al. "Instantaneous Ghost Detection Identification in Automotive Scenarios." *2019 IEEE Radar Conference (RadarConf)*. IEEE, 2019.

[2] Kraus, Florian, et al. "Using machine learning to detect ghost images in automotive radar." *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2020.

Lidar and Radar Fusion in Urban Air Mobility Scenario

This example shows how to use multiobject trackers to track various unmanned aerial vehicles (UAVs) in an urban environment. You create a scene using the `uavScenario` object based on building and terrain data available online. You then use lidar and radar sensor models to generate synthetic sensor data. Finally, you use various tracking algorithms to estimate the state of all UAVs in the scene.

UAVs are designed for a wide range of operations. Many applications are set in urban environments, such as drone package delivery, air taxis, and power line inspection. The safety of these operations becomes critical as the number of applications grows, making controlling the urban airspace a challenge.

Create Urban Air Mobility Scenario

In this example, you use the terrain and building data of Boulder, CO. The Digital Terrain Elevation Data (DTED) file is downloaded from the SRTM Void Filled dataset available from the U.S. Geological Survey. The building data in `southboulder.osm` was downloaded from <https://www.openstreetmap.org/>, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), <https://opendatacommons.org/licenses/odbl/>.

```
dtedfile = "n39_w106_3arc_v2.dt1";
buildingfile = "southboulder.osm";
scene = createScenario(dtedfile,buildingfile);
```

Next, add a few UAVs to the scenario.

To model a package delivery operation, define a trajectory leaving from the roof of a building and flying to a different building. The trajectory is composed of three legs. The quadrotor takes off vertically, then flies toward the next delivery destination, and finally lands vertically on the roof.

```
waypointsA = [1895 90 20; 1915 108 35; 1900 115 20];
timeA = [0 25 50];
trajA = waypointTrajectory(waypointsA, "TimeOfArrival", timeA, "ReferenceFrame", "ENU", "AutoBank");
uavA = uavPlatform("UAV", scene, "Trajectory", trajA, "ReferenceFrame", "ENU");
updateMesh(uavA, "quadrotor", {5}, [0.6350 0.0780 0.1840], eye(4));
```

Add another UAV to model an air taxi flying by. Its trajectory is linear and slightly descending. Use the `fixedwing` geometry to model a larger UAV that is suitable for transporting people.

```
waypointsB = [1940 120 50; 1800 50 20];
timeB = [0 41];
trajB = waypointTrajectory(waypointsB, "TimeOfArrival", timeB, "ReferenceFrame", "ENU", "AutoBank");
uavB = uavPlatform("UAV2", scene, "Trajectory", trajB, "ReferenceFrame", "ENU");
updateMesh(uavB, "fixedwing", {10}, [0.6350 0.0780 0.1840], eye(4));
```

Then add a quadrotor with a trajectory following the street path. This represents a UAV inspecting power grid lines for maintenance purposes.

```
waypointsC = [1950 60 35; 1900 60 35; 1890 80 35];
timeC = linspace(0,41,size(waypointsC,1));
trajC = waypointTrajectory(waypointsC, "TimeOfArrival", timeC, "ReferenceFrame", "ENU", "AutoBank");
uavC = uavPlatform("UAV3", scene, "Trajectory", trajC, "ReferenceFrame", "ENU");
updateMesh(uavC, "quadrotor", {5}, [0.6350 0.0780 0.1840], eye(4));
```

Finally, add the ego UAV, a UAV responsible for surveilling the scene and tracking different moving platforms.

```
waypointsD = [1900 140 65; 1910 100 65];
timeD = [0 60];
trajD = waypointTrajectory(waypointsD, "TimeOfArrival", timeD, ...
    "ReferenceFrame", "ENU", "AutoBank", true, "AutoPitch", true);
egoUAV = uavPlatform("EgoVehicle", scene, "Trajectory", trajD, "ReferenceFrame", "ENU");
updateMesh(egoUAV, "quadrotor", {5}, [0 0 1], eye(4));
```

Define UAV Sensor Suite

Mount sensors on the ego vehicle. Use a lidar puck that is commonly used in automotive applications [1]. The puck is a small sensor that can be attached on a quadrotor. Use the following specification for the lidar puck:

- Range resolution: 3 cm
- Maximum range: 100 m
- 360 degrees azimuth span with 0.2° resolution
- 30 degrees elevation span with 2° resolution
- Update rate: 10 Hz
- Mount with a 90° tilt to look down

```
% Mount a lidar on the quadrotor
lidarOrient = [90 90 0];
lidarSensor = uavLidarPointCloudGenerator("MaxRange", 100, ...
    "RangeAccuracy", 0.03, ...
    "ElevationLimits", [-15 15], ...
    "ElevationResolution", 2, ...
    "AzimuthLimits", [-180 180], ...
    "AzimuthResolution", 0.2, ...
    "UpdateRate", 10, ...
    "HasOrganizedOutput", false);
lidar = uavSensor("Lidar", egoUAV, lidarSensor, "MountingLocation", [0 0 -3], "MountingAngles", 1);
```

Next, add a radar using the `radarDataGenerator` System object from the Radar Toolbox. To add this sensor to the UAV platform, you need to define a custom adaptor class. For details on that, see the “Simulate Radar Sensor Mounted On UAV” (UAV Toolbox) example. In this example, you use the `helperRadarAdaptor` class. This class uses the mesh geometry of targets to define cuboid dimensions for the radar model. The mesh is also used to derive a simple RCS signature for each target. Based on the Echodyne EchoFlight UAV radar [2], set the radar configuration as:

- Frequency: 24.45-24.65 GHz
- Field of view: 120° azimuth 80° elevation
- Resolution: 2 deg in azimuth, 6° in elevation
- Full scan rate: 1 Hz
- Sensitivity: 0 dBsm at 200 m

Additionally, configure the radar to output multiple detections per object. Though the radar can output tracks representing point targets, you want to estimate the extent of the target, which is not available with the default track output. Therefore, set the `TargetReportFormat` property to `Detections` so that the radar reports crude detections directly.

```

% Mount a radar on the quadrotor.
radarSensor = radarDataGenerator("no_scanning", "SensorIndex", 1, ...
    "FieldOfView", [120 80], ...
    "UpdateRate", 1, ...
    'MountingAngles', [0 30 0], ...
    "HasElevation", true, ...
    "ElevationResolution", 6, ...
    "AzimuthResolution", 2, ...
    "RangeResolution", 4, ...
    "RangeLimits", [0 200], ...
    'ReferenceRange', 200, ...
    'CenterFrequency', 24.55e9, ...
    'Bandwidth', 200e6, ...
    "TargetReportFormat", "Detections", ...
    "DetectionCoordinates", "Sensor_rectangular", ...
    "HasFalseAlarms", false, ...
    "FalseAlarmRate", 1e-7);

radar = uavSensor("Radar", egoUAV, helperRadarAdaptor(radarSensor));

```

Define Tracking System

Lidar Point Cloud Processing

Lidar sensors return point clouds. To fuse the lidar output, the point cloud must be clustered to extract object detections. Segment out the terrain using the `segmentGroundSMRF` function from Lidar Toolbox. The remaining point cloud is clustered, and a simple threshold is applied to each cluster mean elevation to filter out building detections. Fit each cluster with a cuboid to extract a bounding box detection. The helper class `helperLidarDetector` available in this example has the implementation details.

Lidar cuboid detections are formatted using the `objectDetection` object. The measurement state for these detections is $[x, y, z, L, W, H, q_0, q_1, q_2, q_3]$, where:

- x, y, z are the cuboid center coordinates along the east, north, and up (ENU) axes of the scenario, respectively.
- L, W, H are the length, width, and height of the cuboid, respectively.
- $q = q_0 + q_1 \cdot i + q_2 \cdot j + q_3 \cdot k$ is the quaternion defining the orientation of the cuboid with respect to the ENU axes.

```
lidarDetector = helperLidarDetector(scene)
```

```
lidarDetector =
    helperLidarDetector with properties:
```

```

        MaxWindowRadius: 3
        GridResolution: 1.5000
    SegmentationMinDistance: 5
    MinDetectionsPerCluster: 2
        MinZDistanceCluster: 20
        EgoVehicleRadius: 10

```

Lidar Tracker

Use a point target tracker, `ttrackerJPDA`, to track the lidar bounding box detections. A point tracker assumes that each UAV can generate at most one detection per sensor scan. This assumption is valid

because you have clustered the point cloud into cuboids. To set up a tracker, you need to define the motion model and the measurement model. In this example, you model the dynamics of UAVs using an augmented constant velocity model. The constant velocity model is sufficient to track trajectories consisting of straight flight legs or slowly varying segments. Moreover, assume the orientation of the UAV is constant and assume the dimensions of the UAVs are constant. As a result, the track state and state transition equations are $X = [x, v_x, y, v_y, z, v_z, L, W, H, q_0, q_1, q_2, q_3]$ and

$$X_{k+1} = \begin{bmatrix} 1 & t_s & 0 & . & . & 0 \\ 0 & 1 & 0 & . & . & . \\ . & . & 1 & t_s & . & . \\ . & . & . & 1 & 0 & 0 \\ . & . & . & . & 1 & t_s \\ 0 & . & . & . & 0 & 1 \\ & & 0_3 & & I_3 & 0_{3 \times 4} \\ & & 0_3 & & 0_{4 \times 3} & I_4 \end{bmatrix} X_k + Q_k$$

Here, v_x, v_y, v_z are the cuboid velocity vector coordinates along the scenario ENU axes. Track orientation using a quaternion because of the discontinuity of Euler angles when using tracking filters. t_s , the time interval between updates k and $k+1$, is equal to 0.1 seconds. Lastly, Q_k is the additive process noise that captures the modeling inaccuracy.

The inner transition matrix corresponds to the constant velocity model. Define an augmented state version of `constvel` and `cvmeas` to account for the additional constant states. The details are implemented in the supporting functions `initLidarFilter`, `augmentedConstvel`, `augmentedConstvelJac`, `augmentedCVmeas`, and `augmentedCVmeasJac` at the end of the example.

```
lidarJPDA = trackerJPDA('TrackerIndex',2,...
    'AssignmentThreshold',[70 150],...
    'ClutterDensity',1e-16,...
    'DetectionProbability',0.99,...
    'DeletionThreshold',[10 10],... Delete lidar track if missed for 1 second
    'ConfirmationThreshold',[4 5],...
    'FilterInitializationFcn',@initLidarFilter)
```

```
lidarJPDA =
  trackerJPDA with properties:

    TrackerIndex: 2
    FilterInitializationFcn: @initLidarFilter
    MaxNumEvents: Inf
    EventGenerationFcn: 'jpdaEvents'
    MaxNumTracks: 100
    MaxNumDetections: Inf
    MaxNumSensors: 20
    TimeTolerance: 1.0000e-05

    AssignmentThreshold: [70 150]
    InitializationThreshold: 0
    DetectionProbability: 0.9900
    ClutterDensity: 1.0000e-16

    OOSMHandling: 'Terminate'
```

```

        TrackLogic: 'History'
ConfirmationThreshold: [4 5]
DeletionThreshold: [10 10]
HitMissThreshold: 0.2000

HasCostMatrixInput: false
HasDetectableTrackIDsInput: false
StateParameters: [1x1 struct]

        NumTracks: 0
        NumConfirmedTracks: 0

EnableMemoryManagement: false

```

Radar Tracker

In this example, you assume that the radar returns are preprocessed such that only returns from moving objects are preserved, that is, there are no returns from the ground or the buildings. The radar measurement state is $[x, v_x, y, v_y, z, v_z]$. The radar resolution is fine enough to generate multiple returns per UAV target and its detections should not be fed directly to a point target tracker. There are two possible approaches to track with the high-resolution radar detections. One of the approaches is that you can cluster the detections and augment the state with dimensions and orientation constants as done previously with the lidar cuboids. In the other approach, you can feed the detections to an extended target tracker adopted in this example by using a GGIW-PHD tracker. This tracker estimates the extent of each target using an inverse Wishart distribution, whose expectation is a 3-by-3 positive definite matrix, representing the extent of a target as a 3-D ellipse. This second approach is preferable because you do not have too many detections per object and clustering is less accurate than extended target tracking.

To create a GGIW-PHD tracker, first define the tracking sensor configuration for each sensor reporting to the tracker. In this case, you need to define the configuration for only one radar. When the radar mounting platform is moving, you need to update this configuration with the current radar pose before each tracker step. Next, define a filter initialization function based on the sensor configuration. Finally, construct a `trackerPHD` object and increase the partitioning threshold to capture the dimensions of objects tracked in this example. The implementation details are shown at the end of the example in the supporting function `createRadarTracker`.

```
radarPHD = createRadarTracker(radarSensor, egoUAV)
```

```
radarPHD =
```

```
  trackerPHD with properties:
```

```

        TrackerIndex: 1
SensorConfigurations: {[1x1 trackingSensorConfiguration]}
PartitioningFcn: @(dets)partitionDetections(dets,threshold(1),threshold(2),'Dis
        MaxNumSensors: 20
        MaxNumTracks: 1000
        MaxNumComponents: 1000

AssignmentThreshold: 50
        BirthRate: 1.0000e-03
        DeathRate: 1.0000e-06

ExtractionThreshold: 0.8000

```



```

ConfirmationThreshold: 0.9900
  DeletionThreshold: 0.1000
  MergingThreshold: 50
  LabelingThresholds: [1.0100 0.0100 0]

  StateParameters: [1x1 struct]
HasSensorConfigurationsInput: true
  NumTracks: 0
  NumConfirmedTracks: 0

```

Track Fusion

The final step in creating the tracking system is to define a track fuser object to fuse lidar tracks and radar tracks. Use the 13-dimensional state of lidar tracks as the fused state definition.

```

radarConfig = fuserSourceConfiguration('SourceIndex',1,...
  'IsInitializingCentralTracks',true);

lidarConfig = fuserSourceConfiguration('SourceIndex',2,...
  'IsInitializingCentralTracks',true);

fuser = trackFuser('SourceConfigurations',{radarConfig,lidarConfig},...
  'ProcessNoise',blkdiag(2*eye(6),1*eye(3),0.2*eye(4)),...
  'HasAdditiveProcessNoise',true,...
  'AssignmentThreshold',200,...
  'ConfirmationThreshold',[4 5],...
  'DeletionThreshold',[5 5],...
  'StateFusion','Cross',...
  'StateTransitionFcn',@augmentedConstvel,...
  'StateTransitionJacobianFcn',@augmentedConstvelJac);

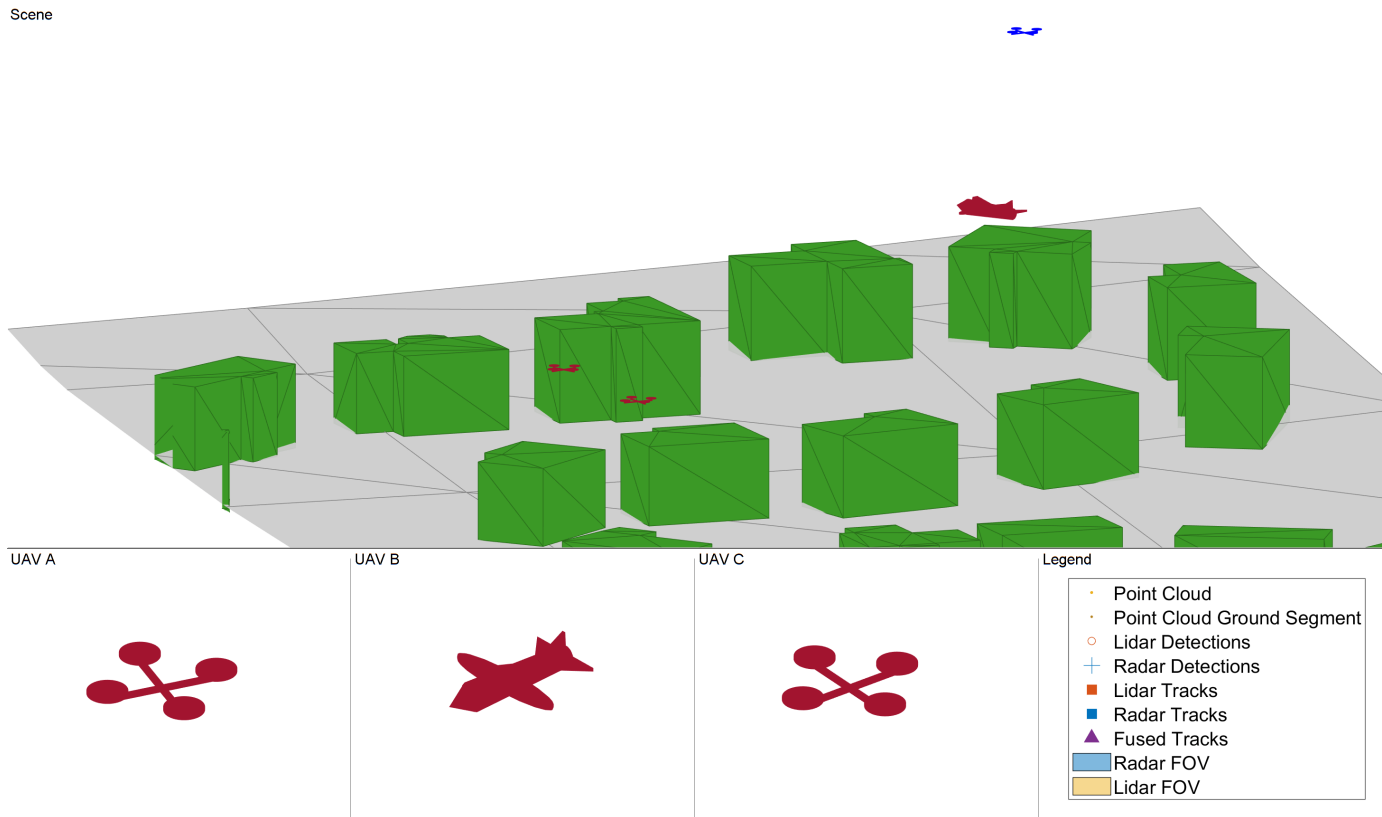
```

Visualization

Use a helper class to visualize the scenario. The helper class in this example utilizes the `uavScenario` visualization capabilities and the `theaterPlot` plotter to represent detection and track information.

The display is divided into five tiles, showing respectively, the overall 3-D scene, three chase cameras for three UAVs, and the legend.

```
viewer = helperUAVDisplay(scene);
```



```
% Radar and lidar coverages for display
[radarcov,lidarcov] = sensorCoverage(radarSensor, lidar);
```

Simulate Scenario

Run the scenario and visualize the results of the tracking system. The true pose of each target as well as the radar, lidar, and fused tracks are saved for offline metric analysis.

```
setup(scene);
s = rng;
rng(2021);

numSteps = scene.StopTime*scene.UpdateRate;
truthlog = cell(1,numSteps);
radarlog = cell(1,numSteps);
lidarlog = cell(1,numSteps);
fusedlog = cell(1,numSteps);
logCount = 0;

while advance(scene)
    time = scene.CurrentTime;
    % Update sensor readings and read data.
    updateSensors(scene);
    egoPose = read(egoUAV);

    % Track with radar
    [radardets, radarTracks, inforadar] = updateRadarTracker(radar,radarPHD, egoPose, time);

    % Track with lidar
```

```

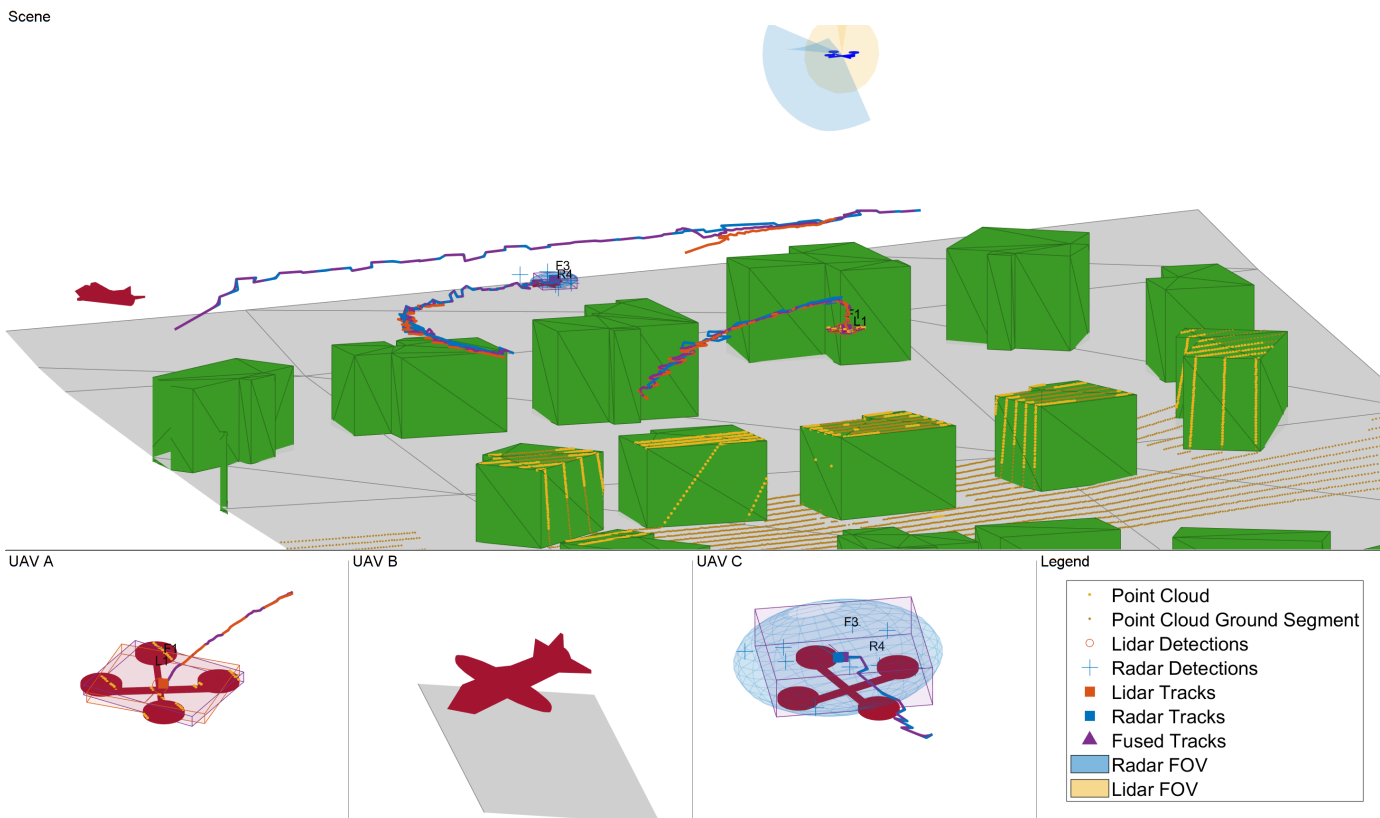
[lidardets, lidarTracks, nonGroundCloud, groundCloud] = updateLidarTracker(lidar,lidarDetecto

% Fuse lidar and radar tracks
rectRadarTracks = formatPHDTracks(radarTracks);
if isLocked(fuser) || ~isempty(radarTracks) || ~isempty(lidarTracks)
    [fusedTracks,~,allfused,info] = fuser([lidarTracks;rectRadarTracks],time);
else
    fusedTracks = objectTrack.empty;
end

% Save log
logCount = logCount + 1;
lidarlog{logCount} = lidarTracks;
radarlog{logCount} = rectRadarTracks;
fusedlog{logCount} = fusedTracks;
truthlog{logCount} = logTargetTruth(scene.Platforms(1:3));

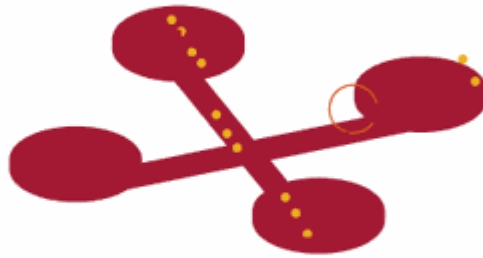
% Update figure
viewer(radarcov, lidarcov, nonGroundCloud, groundCloud, lidardets, radardets, lidarTracks, r
end

```

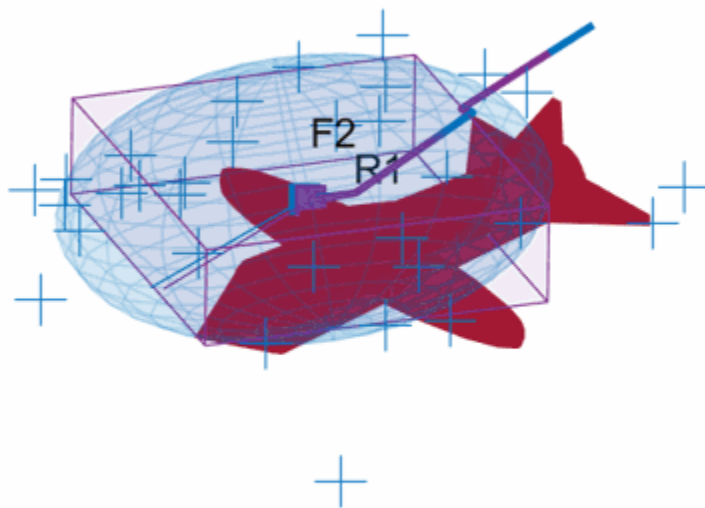


Based on the visualization results, perform an initial qualitative assessment of the tracking performance. The display at the end of the scenario shows that all three UAVs were well tracked by the ego. With the current sensor suite configuration, lidar tracks were only established partially due to the limited coverage of the lidar sensor. The wider field of view of the radar allowed establishing radar tracks more consistently in this scenario.

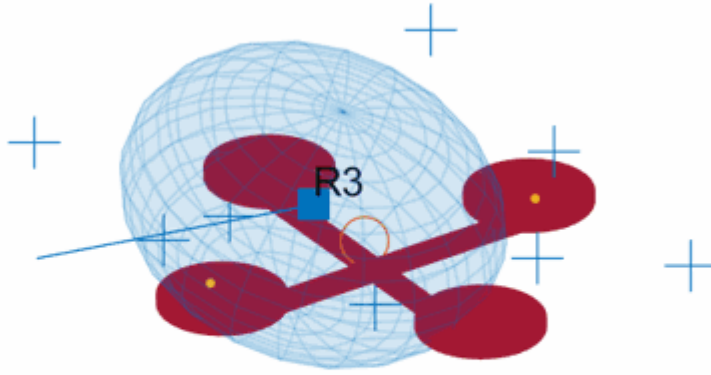
UAV A



UAV B



UAV C



The three animated GIFs above show parts of the chase views. You can see that the quality of lidar tracks (orange box) is affected by the geometry of the scenario. UAV A (left) is illuminated by the lidar (shown in yellow) almost directly from above. This enables the tracker to capture the full extent of the drone over time. However, UAV C (right) is partially seen by the radar which leads to underestimating the size of the drone. Also, the estimated centroid periodically oscillates around the true drone center. The larger fixed-wing UAV (middle) generates many lidar points. Thus, the tracker can detect and track the full extent of the target once it has completely entered the field of view of the lidar. In all three cases, the radar, shown in blue, provides more accurate information of the target extent. As a result, the fused track box (in purple) is more closely capturing the extent of each UAV. However, the radar returns are less accurate in position. Radar tracks show more position bias and poorer orientation estimate.

Tracking Metrics

In this section, you analyze the performance of the tracking system using the OSPA(2) tracking metric. First define the distance function which quantifies the error between track and truth using a scalar value. A lower OSPA value means an overall better performance.

```
ospaR = trackOSPAMetric('Metric','OSPA(2)','Distance','custom','DistanceFcn',@metricDistance);
ospaL = clone(ospaR);
ospaF = clone(ospaR);

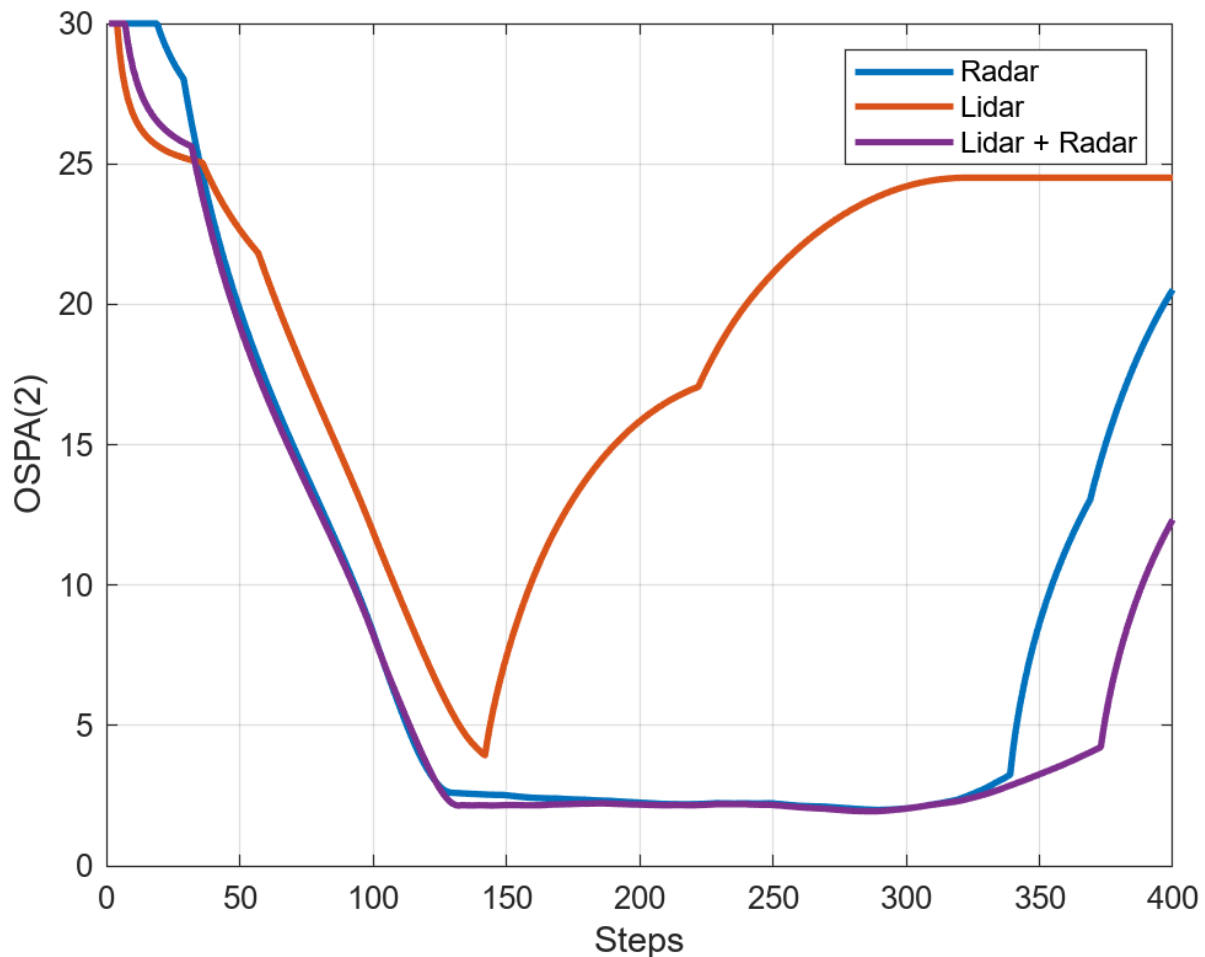
ospaRadar = zeros(1,numSteps);
ospaLidar = zeros(1,numSteps);
ospaFused = zeros(1,numSteps);
```

```

for i=1:numSteps
    truth = truthlog{i};
    ospaRadar(i) = ospaR(radarlog{i},truth);
    ospaLidar(i) = ospaL(lidarlog{i},truth);
    ospaFused(i) = ospaF(fusedlog{i},truth);
end

figure
plot(ospaRadar,'Color',viewer.RadarColor,'LineWidth',2);
hold on
grid on
plot(ospaLidar,'Color',viewer.LidarColor,'LineWidth',2);
plot(ospaFused,'Color',viewer.FusedColor,'LineWidth',2);
legend('Radar','Lidar','Lidar + Radar');
xlabel('Steps')
ylabel('OSPA(2)')

```



Analyze the overall system performance. Each tracker is penalized for not tracking any of the UAVs even if the target UAV is outside of the sensor coverage. This shows improved performance when fusing lidar and radar due to the added surveillance area. This is particularly noticeable at the end of

the simulation where two targets are tracked, one by radar and the other by lidar, but both are tracked by the fuser. Additionally, you can see that the fused OSPA is below the minimum of lidar and radar OSPA, showing the fused track has better quality than each individual track.

```
% clean up
removeCustomTerrain("southboulder");
rng(s);
```

Summary

This example showed you how to model a UAV-borne lidar and radar tracking system and tested it on an urban air mobility scenario. You used the `uavScenario` object to create a realistic urban environment with terrain and buildings. You then generated synthetic sensor data to test a complete tracking system chain, involving point cloud processing, point target and extended target tracking, and track fusion.

Supporting Functions

`createScenario` creates the `uavScenario` using the OpenStreetMap terrain and building mesh data.

```
function scene = createScenario(dtedfile,buildingfile)
```

```
try
    addCustomTerrain("southboulder",dtedfile);
catch
    % custom terrain was already added.
end
```

```
minHeight = 1.6925e+03;
latlonCenter = [39.9786 -105.2882 minHeight];
scene = uavScenario("UpdateRate",10,"StopTime",40,...
    "ReferenceLocation",latlonCenter);
```

```
% Add terrain mesh
sceneXLim = [1800 2000];
sceneYLim = [0 200];
scene.addMesh("terrain", {"southboulder", sceneXLim, sceneYLim},[0 0 0]);
```

```
% Add buildings
scene.addMesh("buildings", {buildingfile, sceneXLim, sceneYLim, "auto"}, [0 0 0]);
```

```
end
```

`createRadarTracker` creates the `ttrackerPHD` tracker to fuse radar detections.

```
function tracker = createRadarTracker(radar, egoUAV)
```

```
% Create sensor configuration for trackerPHD
```

```
fov = radar.FieldOfView;
sensorLimits = [-fov(1)/2 fov(1)/2; -fov(2)/2 fov(2)/2; 0 inf];
sensorResolution = [radar.AzimuthResolution;radar.ElevationResolution; radar.RangeResolution];
Kc = radar.FalseAlarmRate/(radar.AzimuthResolution*radar.RangeResolution*radar.ElevationResolution);
Pd = radar.DetectionProbability;
```

```
sensorPos = radar.MountingLocation(:);
sensorOrient = rotmat(quaternion(radar.MountingAngles, 'eulerd', 'ZYX', 'frame'),'frame');
```

```

% Specify frame info of radar with respect to UAV
sensorTransformParameters(1) = struct('Frame','Spherical',...
    'OriginPosition', sensorPos,...
    'OriginVelocity', zeros(3,1),...% Sensor does not move relative to ego
    'Orientation', sensorOrient,...
    'IsParentToChild',true,...% Frame rotation is supplied as orientation
    'HasElevation',true,...
    'HasVelocity',false);

% Specify frame info of UAV with respect to scene
egoPose = read(egoUAV);
sensorTransformParameters(2) = struct('Frame','Rectangular',...
    'OriginPosition', egoPose(1:3),...
    'OriginVelocity', egoPose(4:6),...
    'Orientation', rotmat(quaternion(egoPose(10:13)),'Frame'),...
    'IsParentToChild',true,...
    'HasElevation',true,...
    'HasVelocity',false);

radarPHDconfig = trackingSensorConfiguration(radar.SensorIndex,...
    'IsValidTime', true,...
    'SensorLimits',sensorLimits,...
    'SensorResolution', sensorResolution,...
    'DetectionProbability',Pd,...
    'ClutterDensity', Kc,...
    'SensorTransformFcn',@cvmeas,...
    'SensorTransformParameters', sensorTransformParameters);

radarPHDconfig.FilterInitializationFcn = @initRadarFilter;

radarPHDconfig.MinDetectionProbability = 0.4;

% Threshold for partitioning
threshold = [3 16];
tracker = trackerPHD('TrackerIndex',1,...
    'HasSensorConfigurationsInput',true,...
    'SensorConfigurations',{radarPHDconfig},...
    'BirthRate',1e-3,...
    'AssignmentThreshold',50,...% Minimum negative log-likelihood of a detection cell to add birth
    'ExtractionThreshold',0.80,...% Weight threshold of a filter component to be declared a track
    'ConfirmationThreshold',0.99,...% Weight threshold of a filter component to be declared a confirmation
    'MergingThreshold',50,...% Threshold to merge components
    'DeletionThreshold',0.1,...% Threshold to delete components
    'LabelingThresholds',[1.01 0.01 0],...% This translates to no track-splitting. Read Labeling
    'PartitioningFcn',@(dets) partitionDetections(dets, threshold(1),threshold(2),'Distance','Euclidean')
end

```

`initRadarFilter` implements the GGIW-PHD filter used by the `trackerPHD` object. This filter is used during a tracker update to initialize new birth components in the density and to initialize new components from detection partitions.

```

function phd = initRadarFilter (detectionPartition)

if nargin == 0

    % Process noise
    sigP = 0.2;
    sigV = 1;

```



```

Q = diag([sigP, sigV, sigP, sigV, sigP, sigV].^2);

phd = ggiwphd(zeros(6,0), repmat(eye(6), [1 1 0]), ...
    'ScaleMatrices', zeros(3,3,0), ...
    'MaxNumComponents', 1000, ...
    'ProcessNoise', Q, ...
    'HasAdditiveProcessNoise', true, ...
    'MeasurementFcn', @cvmeas, ...
    'MeasurementJacobianFcn', @cvmeasjac, ...
    'PositionIndex', [1 3 5], ...
    'ExtentRotationFcn', @(x,dT)eye(3,class(x)), ...
    'HasAdditiveMeasurementNoise', true, ...
    'StateTransitionFcn', @constvel, ...
    'StateTransitionJacobianFcn', @constveljac);

else %margin == 1
    % -----
    % 1) Configure Gaussian mixture
    % 2) Configure Inverse Wishart mixture
    % 3) Configure Gamma mixture
    % -----

    %% 1) Configure Gaussian mixture
    meanDetection = detectionPartition{1};
    n = numel(detectionPartition);

    % Collect all measurements and measurement noises.
    allDets = [detectionPartition{:}];
    zAll = horzcat(allDets.Measurement);
    RAll = cat(3, allDets.MeasurementNoise);

    % Specify mean noise and measurement
    z = mean(zAll, 2);
    R = mean(RAll, 3);
    meanDetection.Measurement = z;
    meanDetection.MeasurementNoise = R;

    % Parse mean detection for position and velocity covariance.
    [posMeas, velMeas, posCov] = matlabshared.tracking.internal.fusion.parseDetectionForInitFcn(meas);

    % Create a constant velocity state and covariance
    states = zeros(6, 1);
    covariances = zeros(6, 6);
    states(1:2:end) = posMeas;
    states(2:2:end) = velMeas;
    covariances(1:2:end, 1:2:end) = posCov;
    covariances(2:2:end, 2:2:end) = 10*eye(3);

    % process noise
    sigP = 0.2;
    sigV = 1;
    Q = diag([sigP, sigV, sigP, sigV, sigP, sigV].^2);

    %% 2) Configure Inverse Wishart mixture parameters
    % The extent is set to the spread of the measurements in positional-space.
    e = zAll - z;
    Z = e*e'/n + R;
    dof = 150;

```

```

% Measurement Jacobian
p = detectionPartition{1}.MeasurementParameters;
H = cvmeasjac(states,p);

Bk = H(:,1:2:end);
Bk2 = eye(3)/Bk;
V = (dof-4)*Bk2*Z*Bk2';

% Configure Gamma mixture parameters such that the standard deviation
% of the number of detections is n/4
alpha = 16; % shape
beta = 16/n; % rate

phd = ggiwphd(...
    ... Gaussian parameters
    states,covariances,...
    'HasAdditiveMeasurementNoise' ,true,...
    'ProcessNoise',Q,...
    'HasAdditiveProcessNoise',true,...
    'MeasurementFcn' , @cvmeas,...
    'MeasurementJacobianFcn' , @cvmeasjac,...
    'StateTransitionFcn' , @constvel,...
    'StateTransitionJacobianFcn' , @constveljac,...
    'PositionIndex' ,[1 3 5],...
    'ExtentRotationFcn' , @(x,dT) eye(3),...
    ... Inverse Wishart parameters
    'DegreesOfFreedom',dof,...
    'ScaleMatrices',V,...
    'TemporalDecay',150,...
    ... Gamma parameters
    'Shapes',alpha,'Rates',beta,...
    'GammaForgettingFactors',1.05);
end
end

```

`formatPHDTracks` formats the elliptical GGIW-PHD tracks into rectangular augmented state tracks for track fusion. `convertExtendedTrack` returns state and state covariance of the augmented rectangular state. The Inverse Wishart random matrix eigen values are used to derive rectangular box dimensions. The eigen vectors provide the orientation quaternion. In this example, you use an arbitrary covariance for radar track dimension and orientation, which is often sufficient for tracking.

```

function trackout = formatPHDTracks(tracksin)
% Convert track struct from ggiwphd to objectTrack with state definition
% [x y z vx vy vz L W H q0 q1 q2 q3]
N = numel(tracksin);
trackout = repmat(objectTrack,N,1);
for i=1:N
    trackout(i) = objectTrack(tracksin(i));
    [state, statecov] = convertExtendedTrack(tracksin(i));
    trackout(i).State = state;
    trackout(i).StateCovariance = statecov;
end
end

function [state, statecov] = convertExtendedTrack(track)
% Augment the state with the extent information

```

```

extent = track.Extent;
[V,D] = eig(extent);
% Choose L > W > H. Use 1.5 sigma as the dimension
[dims, idx] = sort(1.5*sqrt(diag(D)), 'descend');
V = V(:,idx);
q = quaternion(V, 'rotmat', 'frame');
q = q./norm(q);
[q1, q2, q3, q4] = parts(q);
state = [track.State; dims(:); q1 ; q2 ; q3 ; q4 ];
statecov = blkdiag(track.StateCovariance, 4*eye(3), 4*eye(4));

```

end

`updateRadarTracker` updates the radar tracking chain. The function first reads the current radar returns. Then the radar returns are passed to the GGIW-PHD tracker after updating its sensor configuration with the current pose of the ego drone.

```

function [radardets, radarTracks, inforadar] = updateRadarTracker(radar, radarPHD, egoPose, time)
[~,~,radardets, ~, ~] = read(radar); % isUpdated and time outputs are not compatible with this version
inforadar = [];
if mod(time,1) ~= 0
    radardets = {};
end
if mod(time,1) == 0 && (isLocked(radarPHD) || ~isempty(radardets))
    % Update radar sensor configuration for the tracker
    configs = radarPHD.SensorConfigurations;
    configs{1}.SensorTransformParameters(2).OriginPosition = egoPose(1:3);
    configs{1}.SensorTransformParameters(2).OriginVelocity = egoPose(4:6);
    configs{1}.SensorTransformParameters(2).Orientation = rotmat(quaternion(egoPose(10:13)), 'frame');
    [radarTracks,~,~,inforadar] = radarPHD(radardets,configs,time);
elseif isLocked(radarPHD)
    radarTracks = predictTracksToTime(radarPHD, 'confirmed', time);
    radarTracks = arrayfun(@(x) setfield(x, 'UpdateTime', time), radarTracks);
else
    radarTracks = objectTrack.empty;
end
end

```

`updateLidarTracker` updates the lidar tracking chain. The function first reads the current point cloud output from the lidar sensor. Then the point cloud is processed to extract object detections. Finally, these detections are passed to the point target tracker.

```

function [lidardets, lidarTracks, nonGroundCloud, groundCloud] = updateLidarTracker(lidar, lidarPHD, time, ptCloud)
[~, time, ptCloud] = read(lidar);
% lidar is always updated
[lidardets, nonGroundCloud, groundCloud] = lidarDetector(egoPose, ptCloud, time);
if isLocked(lidarJPDA) || ~isempty(lidardets)
    lidarTracks = lidarJPDA(lidardets, time);
else
    lidarTracks = objectTrack.empty;
end
end

```

`initLidarFilter` initializes the filter for the lidar tracker. The initial track state is derived from the detection position measurement. Velocity is set to 0 with a large covariance to allow future detections to be associated to the track. Augmented state motion model, measurement functions, and Jacobians are also defined here.

```

function ekf = initLidarFilter(detection)

% Lidar measurement: [x y z L W H q0 q1 q2 q3]
meas = detection.Measurement;
initState = [meas(1);0;meas(2);0;meas(3);0; meas(4:6);meas(7:10) ];
initStateCovariance = blkdiag(100*eye(6), 100*eye(3), eye(4));

% Process noise standard deviations
sigP = 1;
sigV = 2;
sigD = 0.5; % Dimensions are constant but partially observed
sigQ = 0.5;

Q = diag([sigP, sigV, sigP, sigV, sigP, sigV, sigD, sigD, sigD, sigQ, sigQ, sigQ, sigQ].^2);

ekf = trackingEKF('State',initState,...
    'StateCovariance',initStateCovariance,...
    'ProcessNoise',Q,...
    'StateTransitionFcn',@augmentedConstvel,...
    'StateTransitionJacobianFcn',@augmentedConstvelJac,...
    'MeasurementFcn',@augmentedCVmeas,...
    'MeasurementJacobianFcn',@augmentedCVmeasJac);
end

function stateOut = augmentedConstvel(state, dt)
% Augmented state for constant velocity
stateOut = constvel(state(1:6,:),dt);
stateOut = vertcat(stateOut,state(7:end,:));
% Normalize quaternion in the prediction stage
idx = 10:13;
qparts = stateOut(idx,:);
n = sqrt(sum(qparts.^2));
qparts = qparts./n;
stateOut(idx,qparts(1,:)<0) = -qparts(:,qparts(1,:)<0);
end

function jacobian = augmentedConstvelJac(state,varargin)
jacobian = constveljac(state(1:6,:),varargin{:});
jacobian = blkdiag(jacobian, eye(7));
end

function measurements = augmentedCVmeas(state)
measurements = cvmeas(state(1:6,:));
measurements = [measurements; state(7:9,:); state(10:13,:)];
end

function jacobian = augmentedCVmeasJac(state,varargin)
jacobian = cvmeasjac(state(1:6,:),varargin{:});
jacobian = blkdiag(jacobian, eye(7));
end

sensorCoverage constructs sensor coverage configuration structures for visualization.

function [radarcov,lidarcov] = sensorCoverage(radarSensor, lidar)
radarcov = coverageConfig(radarSensor);
% Scale down coverage to limit visual clutter
radarcov.Range = 10;
lidarSensor = lidar.SensorModel;

```

```
lidarcov = radarcov;
lidarcov.Index = 2;
lidarcov.FieldOfView = [diff(lidarSensor.AzimuthLimits); diff(lidarSensor.ElevationLimits)];
lidarcov.Range = 5;
lidarcov.Orientation = quaternion(lidar.MountingAngles, 'eulerd', 'ZYX', 'frame');
end
```

logTargetTruth logs true pose and dimensions throughout the simulation for performance analysis.

```
function logEntry = logTargetTruth(targets)
n = numel(targets);
targetPoses = repmat(struct('Position', [], 'Velocity', [], 'Dimension', [], 'Orientation', []), 1, n);
uavDimensions = [5 5 0.3 ; 9.8 8.8 2.8; 5 5 0.3];
for i=1:n
    pose = read(targets(i));
    targetPoses(i).Position = pose(1:3);
    targetPoses(i).Velocity = pose(4:6);
    targetPoses(i).Dimension = uavDimensions(i,:);
    targetPoses(i).Orientation = pose(10:13);
    targetPoses(i).PlatformID = i;
end
logEntry = targetPoses;
end
```

metricDistance defines a custom distance for GOSPA. This distance incorporates errors in position, velocity, dimension, and orientation of the tracks.

```
function out = metricDistance(track, truth)
positionIdx = [1 3 5];
velIdx = [2 4 6];
dimIdx = 7:9;
qIdx = 10:13;

trackpos = track.State(positionIdx);
trackvel = track.State(velIdx);
trackdim = track.State(dimIdx);
trackq = quaternion(track.State(qIdx)');

truepos = truth.Position;
truevel = truth.Velocity;
truedim = truth.Dimension;
trueq = quaternion(truth.Orientation);

errpos = truepos(:) - trackpos(:);
errvel = truevel(:) - trackvel(:);
errdim = truedim(:) - trackdim(:);

% Weights expressed as inverse of the desired accuracy
posw = 1/0.2; %m^-1
velw = 1/2; % (m/s) ^-1
dimw = 1/4; % m^-1
orw = 1/20; % deg^-1

distPos = sqrt(errpos'*errpos);
distVel = sqrt(errvel'*errvel);
distDim = sqrt(errdim'*errdim);
distq = rad2deg(dist(trackq, trueq));
```

```
out = (distPos * posw + distVel * velw + distdim * dimw + distq * orw)/(posw + velw + dimw + orw)
end
```

References

- 1 Velodyne Lidar puck: <https://velodynelidar.com/products/puck/>
- 2 Echodyne UAV radar: <https://www.echodyne.com/defense/uav-radar/>

Smooth Trajectory Estimation of trackingIMM Filter

This example shows how to smooth state estimates of a target using the `smooth` object function. Smoothing is a technique to refine previous state estimates using the up-to-date measurements and the state estimate information. In this example, you will learn how to improve previously corrected estimates from an Interacting Multi-Model (IMM) filter by running a backward recursion, which produces smoothed and more accurate state estimates. In the first section, you implement a smooth algorithm to smooth the trajectory of a turning car. In the remainder of this example, you perform smoothing on several highly maneuvering aircraft trajectories, taken from the “Benchmark Trajectories for Multi-Object Tracking” on page 6-259 example.

Fixed Interval Smoothing to Track Turning Car

In this section, you smooth the trajectory estimate of a turning car. First, you use the `helperGenerateCarMeasurements` function to generate the position measurements of the car and the corresponding truth.

```
rng(2021); % For repeatable results

% Generate measurements for a car taking a turn
[measPositionCar, trueStateCar, timeCar] = helperGenerateCarMeasurements();

% Define the initial detection in the format of objectDetection
detection = objectDetection(0, [0;0;0], 'MeasurementNoise', eye(3,3));
```

Using the `initekfirm` initialization function, you create an IMM filter based on the constant velocity, constant acceleration, and constant turn motion models. To setup the filter for backward smoothing, you set the `EnableSmoothing` property of the filter to `true`. Since you want to perform fixed interval smoothing (offline smoothing) on the state estimates, which utilizes all corrected and predicted steps, you set the `MaxNumSmoothingSteps` property to the number of measurement steps.

```
defaultIMMCar = initekfirm(detection);
% Enable Smoothing
defaultIMMCar.EnableSmoothing = true;
defaultIMMCar.MaxNumSmoothingSteps = size(measPositionCar,1);
% Initialize IMM
positionSelector = [1 0 0 0 0 0;0 0 1 0 0 0;0 0 0 0 1 0]; % Select position from state
initialState = positionSelector * measPositionCar(1,:);
initialCovariance = diag([1,1e4,1,1e4,1,1e4]);
initialize(defaultIMMCar, initialState, initialCovariance);
```

To obtain the forward estimates of the car, you run the filter by iteratively predicting and correcting the state estimates.

```
% Initialize the corrected states
numSteps = numel(timeCar);
correctState = zeros(6,numSteps);
correctStateCovariance = zeros(6,6,numSteps);
correctState(:,1) = defaultIMMCar.State;
correctStateCovariance(:,:,1) = defaultIMMCar.StateCovariance;

% Forward tracking with prediction and correction
for i = 2:numSteps
    dt = timeCar(i) - timeCar(i-1);
    predict(defaultIMMCar, dt);
```

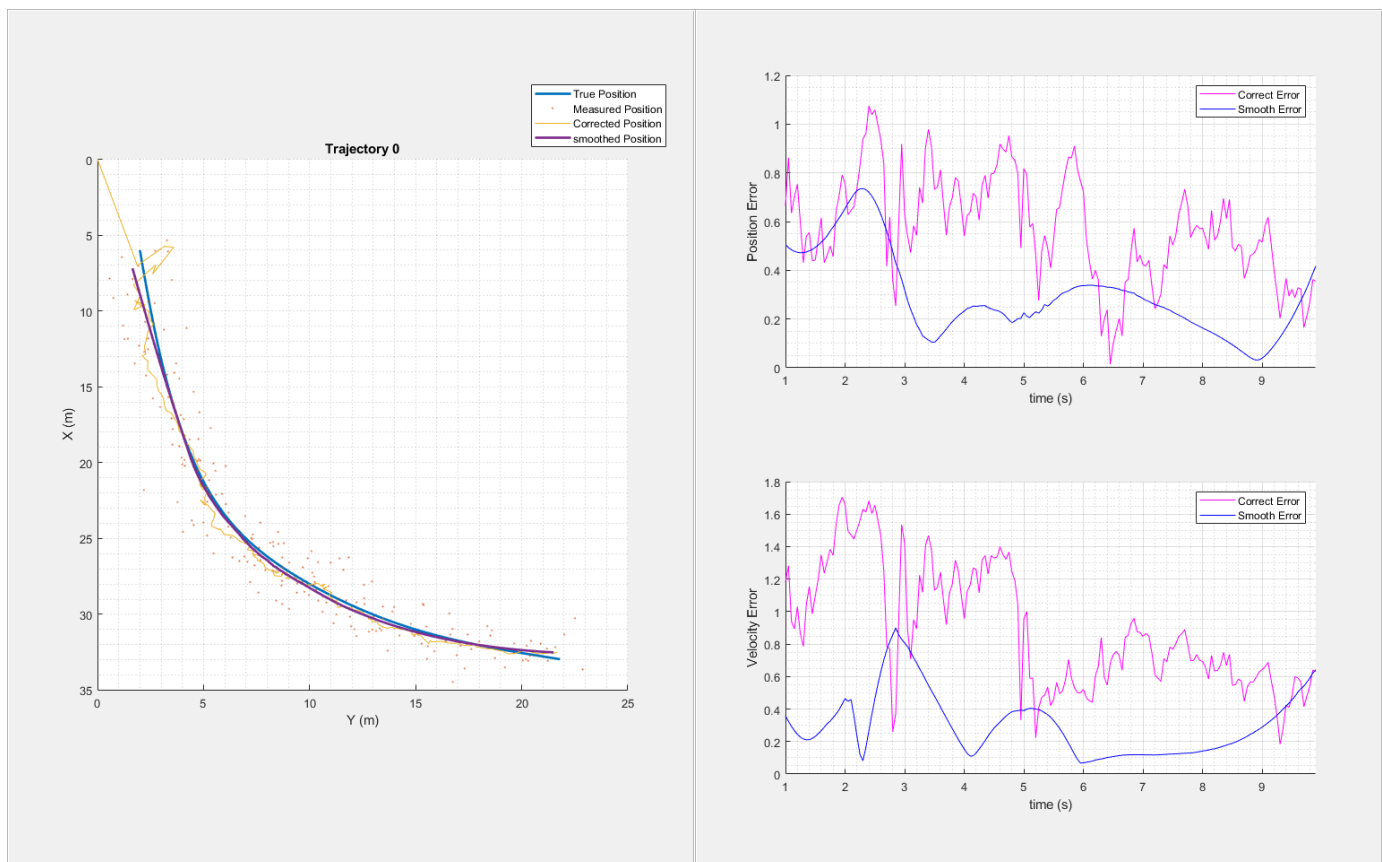
```
[correctState(:,i), correctStateCovariance(:, :,i)] = correct(defaultIMMCar, measPositionCar(:,i));
end
```

To perform the smoothing, simply call the `smooth` object function of the filter. The function returns the smoothed states, state covariance, and model probabilities.

```
[smoothState, smoothStateCovariance, modelProbabilities] = smooth(defaultIMMCar);
```

Next, use the `helperTrajectoryViewer` function to visualize the smooth results and the RMS errors. The results show that using offline smoothing of an IMM filter enables you to reduce the errors in estimates.

```
trajNum = 0;
helperTrajectoryViewer(trajNum,timeCar,correctState, smoothState, trueStateCar, measPositionCar);
```



Smooth Highly Maneuvering Aircraft Trajectories with Default IMM Configuration

In this section, you smooth the trajectories of six highly maneuvering aircraft. The trajectories used in this section are the same as those in the “Benchmark Trajectories for Multi-Object Tracking” on page 6-259 example. In the example, the aircraft acceleration changes as much as 35 m/s^2 during some of the maneuvers. You use the `helperGenerateAircraftTrajMeasurements` function to generate the measurement data and truth.

```
[measPosition, trueState, time] = helperGenerateAircraftTrajMeasurements;
```

Configure an IMM filter similar as the previous section.


```

% Define initial detection
detection = objectDetection(0, [0;0;0], 'MeasurementNoise', eye(3,3));

defaultIMMAircraft = initekfimm(detection);
% Enable Smoothing
defaultIMMAircraft.EnableSmoothing = true;
defaultIMMAircraft.MaxNumSmoothingSteps = size(measPosition,1);

```

Using the helperGenerateSmoothData function, you run the created IMM filter, obtain the corrected state estimates, and generate the smoothed state estimates for the first trajectory.

```

% Only estimate the first trajectory
trajNum = 1;

```

```

% Extract measurement data
measPosTraj = measPosition(:, :, trajNum);

```

```

% Estimate and smooth the trajectory using the helperGenerateSmoothData function
[correctState, correctStateCovariance, smoothState, smoothStateCovariance] = helperGenerateSmooth

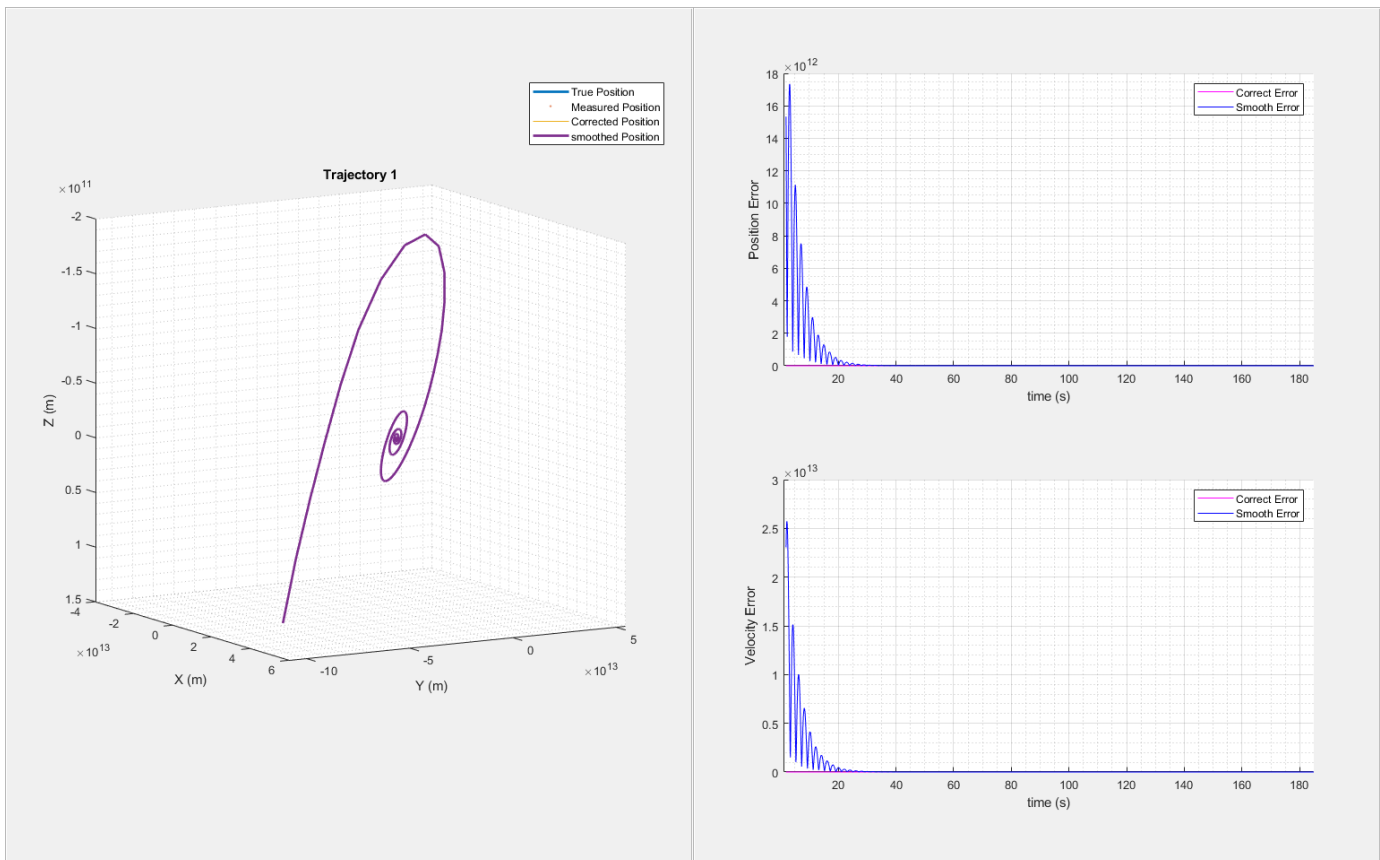
```

Visualize the forward estimation and smoothing results using the helperTrajectoryViewer function.

```

helperTrajectoryViewer(trajNum, time, correctState, smoothState, trueState(:, :, trajNum), measPosTraj

```



As seen from the results, the smoothing process performs poorly and becomes unstable near 120 seconds. This is mainly due to insufficient process noise in the IMM filter. Rapid changes in aircraft

acceleration and sharp turns in short intervals requires higher process noise for the constant acceleration and constant turn motion models of the IMM filter.

Adjusting Process Noise for Better Smoothing Performance

In this section, you adjust the process noise of the filter for better estimation and smoothing results. First, construct a constant velocity, a constant acceleration, and a constant turn-rate model to be used in the IMM filter.

```
constVelocityEKF = initcvekf(detection);
constantAccelerationEKF = initcaekf(detection);
constantTurnEKF = initctekf(detection);
```

To compensate the velocity uncertainty, increase the process noise of the constant velocity model in all three axes based on the estimates of the change in the velocity values of the maneuvering target. Similarly, to compensate the acceleration uncertainty, increase the process noise for the constant acceleration model in all three axes based on the estimate of the change in the acceleration values of the maneuvering target. To compensate the turning rate uncertainty, increase the process noise for the turn rate of the constant-turn model.

```
constantVelocityEKF.ProcessNoise = 36*eye(3,3);
constantAccelerationEKF.ProcessNoise = 100*eye(3,3);
constantTurnEKF.ProcessNoise = 36*eye(4,4);
```

Create an IMM filter using the three defined estimation filters and enable the smoothing capability.

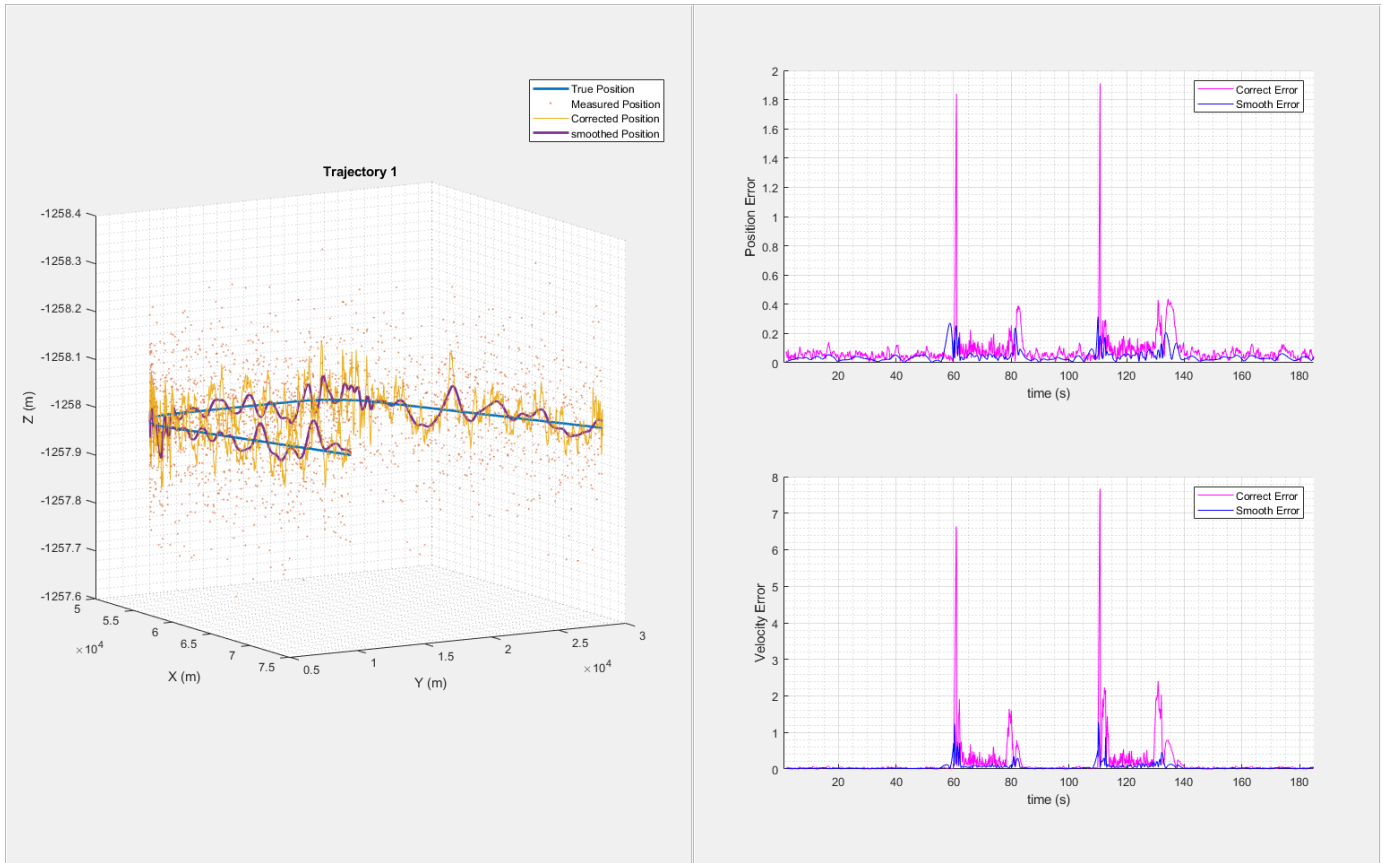
```
filters = {constVelocityEKF;constantAccelerationEKF;constantTurnEKF};
imm = trackingIMM(filters, 'TransitionProbabilities', 0.99);
```

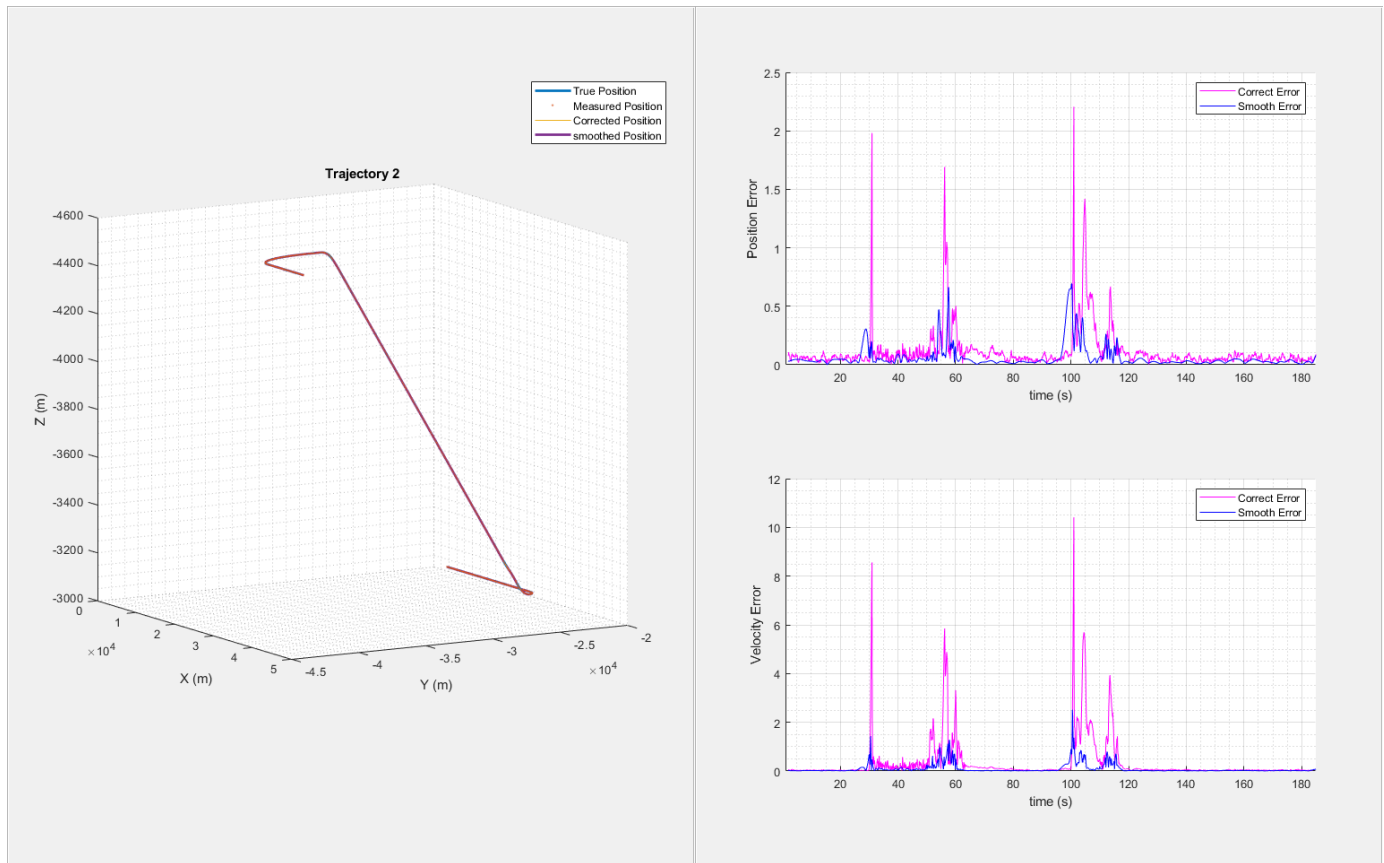
```
imm.EnableSmoothing = true;
imm.MaxNumSmoothingSteps = size(measPosition,1);
```

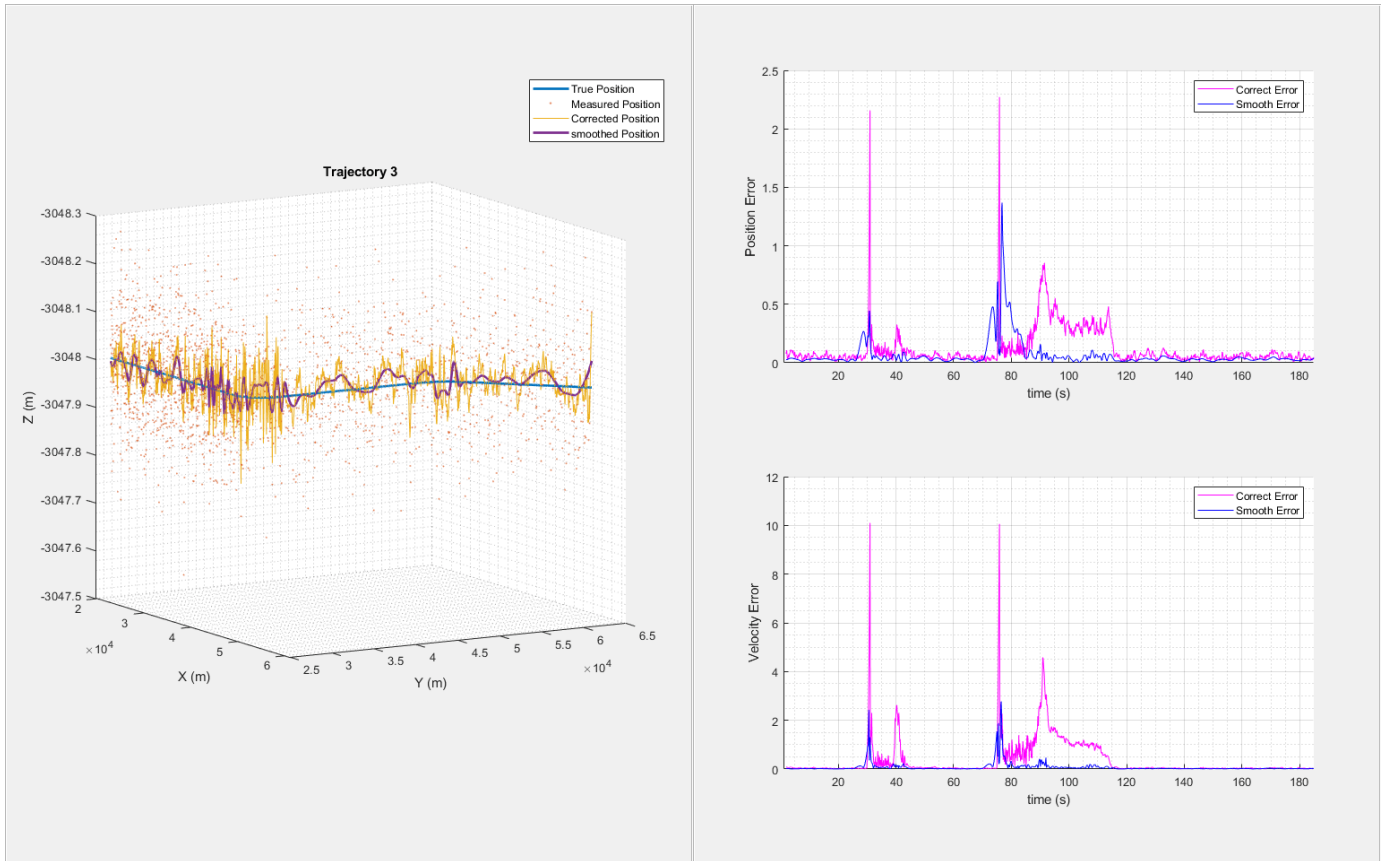
Use the filter to estimate the trajectories of six aircraft one-by-one and obtain the smoothed estimates. Visualize the results for each trajectory.

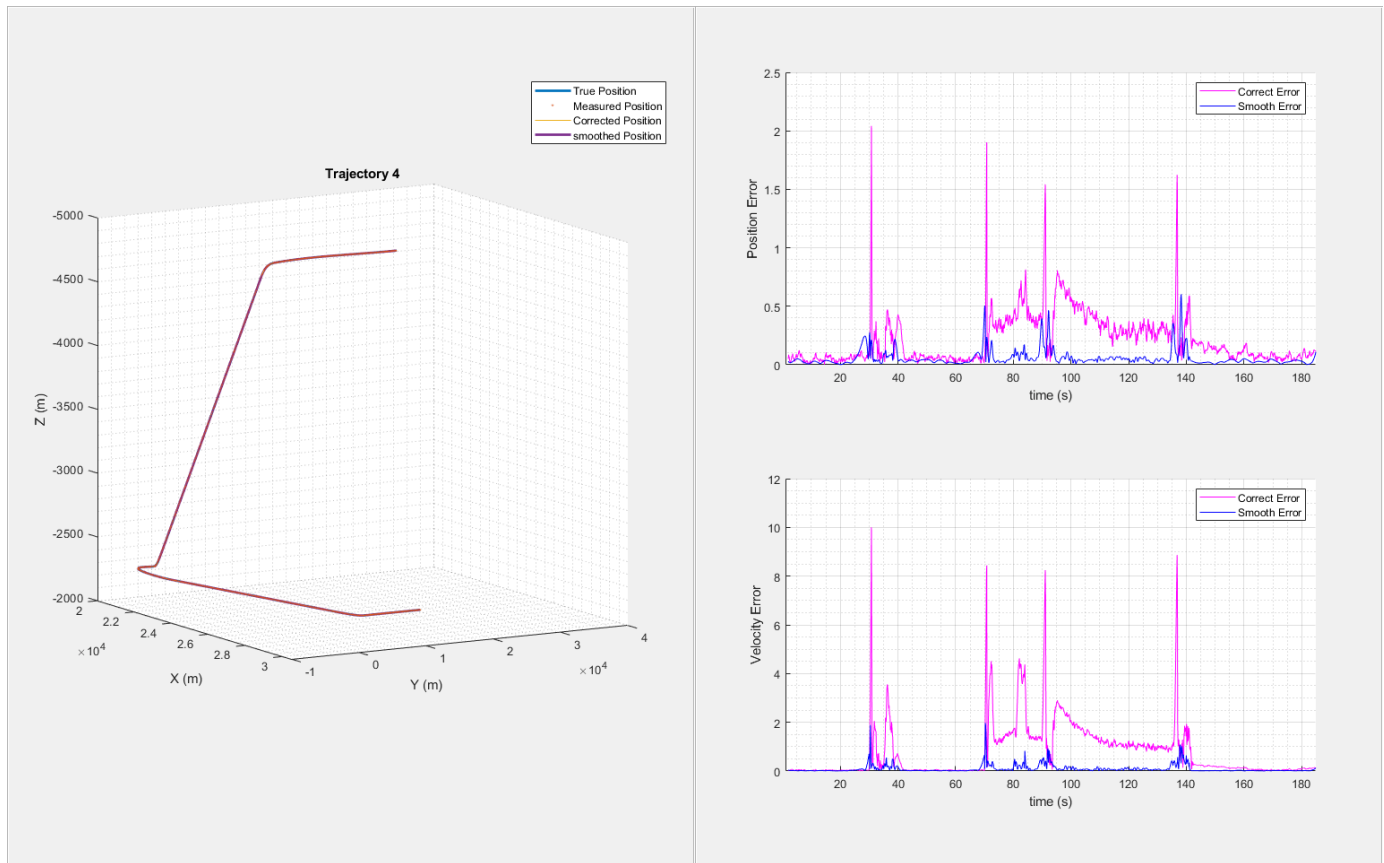
```
for i = 1:6
    trajNum = i;
    measPosTraj = measPosition(:,:,trajNum);
    [correctState, correctStateCovariance, smoothState, smoothStateCovariance] = helperGenerateS

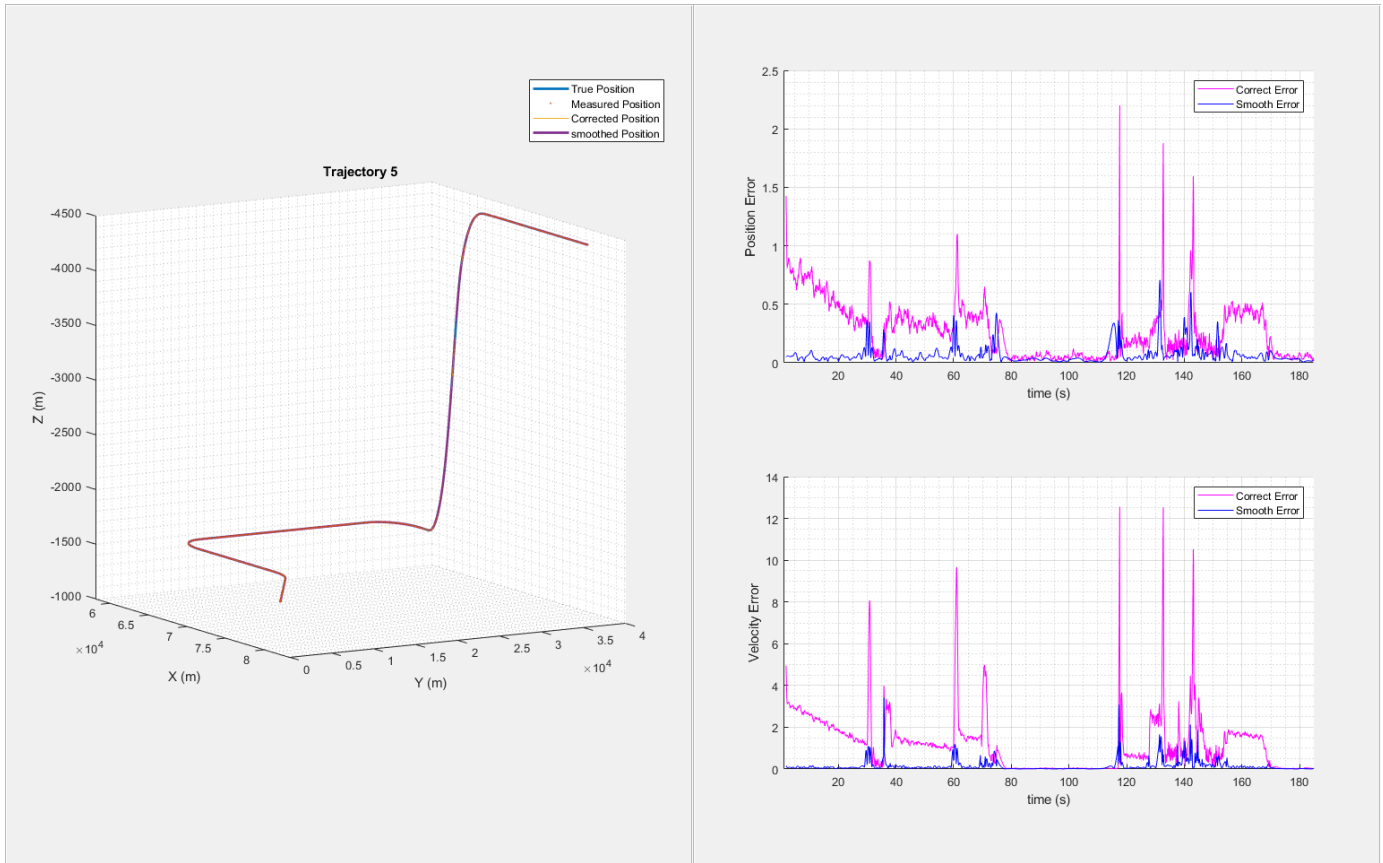
    % Visualize the results for each trajectory
    helperTrajectoryViewer(trajNum, time, correctState, smoothState, trueState(:,:,trajNum), meas
end
```

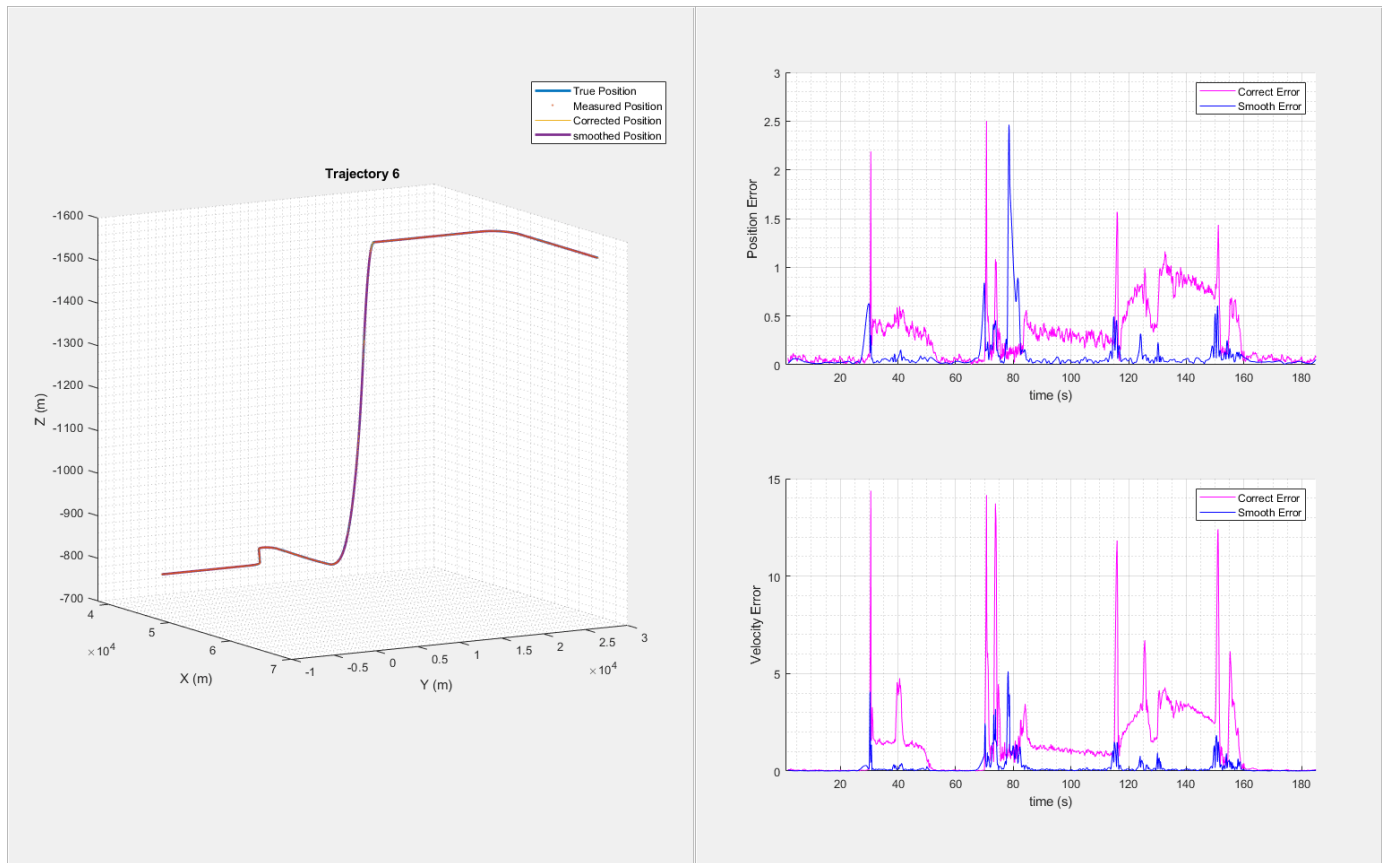












From the results, the use of the adjusted process noise greatly reduces the estimation errors compared with the previous smoothing results without adjusting the process noise.

Summary

In this example, you learned how to smooth filter results to obtain better state estimates. You also learn the importance of adjusting filter noise properly for your specific applications.

Supporting Functions

helperGenerateCarMeasurements

Generate measurement inputs for the trajectory of a turning car

```
function [measPosition, trueState, time] = helperGenerateCarMeasurements()
```

```
% Create a trajectory of a car taking a turn.
```

```
carWayPoints = [6 2 0; 18 4 0; 25 7 0; 28 10 0; 31 15 0; 33 22 0];
```

```
timeSteps = [0 2 4 6 8 10];
```

```
% Scenario Generation.
```

```
scene = trackingScenario('UpdateRate',20);
```

```
plat = platform(scene);
```

```
plat.Trajectory = waypointTrajectory(carWayPoints, timeSteps);
```



```

% Create trajectory using waypoints from the scenario.
numSteps = 0;
truePosition = [];
trueVelocity = [];
trueAcceleration = [];
time = [];
while advance(scene)
    poses = platformPoses(scene);
    t = scene.SimulationTime;
    numSteps = numSteps + 1;
    truePosition(numSteps,1:3) = poses.Position;
    trueVelocity(numSteps,1:3) = poses.Velocity;
    trueAcceleration(numSteps,1:3) = poses.Acceleration;
    time(numSteps) = t;
end

trueState = [truePosition(:,1,:),trueVelocity(:,1,:),truePosition(:,2,:),trueVelocity(:,2,:),trueAcceleration(:,1:3)];

% Add measurement noise to true position.
measNoise = 1* randn(size(truePosition));
measPosition = truePosition + [measNoise(:,1:2), zeros(numSteps,1)];

```

end

helperGenerateAircraftTrajMeasurements

Generate measurement inputs for the trajectories of multiple aircraft

```

function [measPosition, trueState, time] = helperGenerateAircraftTrajMeasurements()

% Load trajectory waypoints and velocities. The file contains tables of waypoints and
% velocities (in units of meters and meters per second) that are used to reconstruct six aircraft
load('benchmarkTrajectoryTables.mat', 'trajTable');

% Scenario generation.
scene = trackingScenario('UpdateRate',10);

% Assign each platform with a trajectory
for n=1:6
    plat = platform(scene);
    traj = trajTable{n};
    plat.Trajectory = waypointTrajectory(traj.Waypoints, traj.Time, 'Velocities', traj.Velocities);
end

% Create trajectory using waypoints from the scenario.
numSteps = 0;
truePosition = [];
trueVelocity = [];
trueAcceleration = [];
time = [];
while advance(scene)
    poses = platformPoses(scene);
    t = scene.SimulationTime;
    numSteps = numSteps + 1;
    position = vertcat(poses.Position);
    velocity = vertcat(poses.Velocity);
    acceleration = vertcat(poses.Acceleration);
    truePosition(numSteps,1:3,1:6) = permute(position,[3 2 1]);
end

```

```

    trueVelocity(numSteps,1:3,1:6) = permute(velocity,[3 2 1]);
    trueAcceleration(numSteps,1:3,1:6) = permute(acceleration,[3 2 1]);
    time(numSteps) = t;
end
trueState = [truePosition(:,1,:),trueVelocity(:,1,:),truePosition(:,2,:),trueVelocity(:,2,:),trueAcceleration(:,1,:),trueAcceleration(:,2,:),trueAcceleration(:,3,)]';

% Define the measurements to be the position and add normal random noise with a standard deviation of 0.1
measNoise = 0.1* randn(size(truePosition));
measPosition = truePosition + measNoise;
end

```

helperGenerateSmoothData

Generate corrected and smoothed states

```

function [correctState, correctStateCovariance, smoothState, smoothStateCovariance] = helperGenerateSmoothData(imm, measPosTraj, time)

% Output correct and smooth states

numSteps = numel(time);

positionSelector = [1 0 0 0 0 0;0 0 1 0 0 0;0 0 0 0 1 0];
initialState = positionSelector' * measPosTraj(1,:);
initialCovariance = diag([1,16e4,1,16e4,1,16e4]); % Velocity is not measured
initialize(imm, initialState, initialCovariance);

correctState = zeros(6,numSteps);
correctStateCovariance = zeros(6,6,numSteps);
correctModelProb = zeros(3,numSteps);
correctState(1,1) = measPosTraj(1,1);
correctState(3,1) = measPosTraj(1,2);
correctState(5,1) = measPosTraj(1,3);
correctStateCovariance(:,:,1) = imm.StateCovariance;
correctModelProb(:,1) = imm.ModelProbabilities;

for i = 2:numSteps
    dt = time(i) - time(i-1);
    predict(imm, dt);
    [correctState(:,i), correctStateCovariance(:,:,i)] = correct(imm, measPosTraj(i,:));
    correctModelProb(:,i) = imm.ModelProbabilities;
end

[smoothState, smoothStateCovariance, smoothModelProb] = smooth(imm);
end

```

helperTrajectoryViewer

Visualize smoothing results and compare RMS error

```

function helperTrajectoryViewer(n, time, correctState, smoothState, trueStateTraj, measPosTraj)

% Create figure and two panels, first panel for position and second panel
% for error visualization

truePosition = trueStateTraj(:,[1 3 5])';
correctPosition = correctState([1 3 5],:);
smoothPosition = smoothState([1 3 5],:);

```

```

correctPosError = correctState([1,3,5],1:end-1) - trueStateTraj(1:end-1,[1,3,5]);
smoothPosError = smoothState([1,3,5],:) - trueStateTraj(1:end-1,[1,3,5]);

correctVelError = correctState([2,4,6],1:end-1) - trueStateTraj(1:end-1,[2,4,6]);
smoothVelError = smoothState([2,4,6],:) - trueStateTraj(1:end-1,[2,4,6]);

numSteps = numel(time);

correctPosRMS = zeros(1,numSteps-1);
smoothPosRMS = zeros(1,numSteps - 1);

correctVelRMS = zeros(1,numSteps-1);
smoothVelRMS = zeros(1,numSteps - 1);

for i = 1:numSteps - 1
    deltaPc = correctPosError(:,i);
    correctPosRMS(:,i) = sqrt((deltaPc'*deltaPc));

    deltaPs = smoothPosError(:,i);
    smoothPosRMS(:,i) = sqrt((deltaPs'*deltaPs));

    deltaVc = correctVelError(:,i);
    correctVelRMS(:,i) = sqrt((deltaVc'*deltaVc));

    deltaVs = smoothVelError(:,i);
    smoothVelRMS(:,i) = sqrt((deltaVs'*deltaVs));
end

hfig = figure('Name',"Trajectory " + n,'NumberTitle','off','Units','normalized','Position',[0.1 0.1 0.9 0.9]);
hLeftPanel = uipanel(hfig,'Position',[0 0 1/2 1]);
hRightPanel = uipanel(hfig,'Position',[1/2 0 1/2 1]);

xMin = 10*floor(min(truePosition(:,1))/10e3)-5;
xMax = 10*ceil(max(truePosition(:,1))/10e3)+5;
yMin = 10*floor(min(truePosition(:,2))/10e3)-5;
yMax = 10*ceil(max(truePosition(:,2))/10e3)+5;
zMin = 10*floor(min(truePosition(:,3))/10e3)-5;
zMax = 10*ceil(max(truePosition(:,3))/10e3)+5;

% Place x-y plot in left panel for plotting true trajectory, corrected
% trajectory, and smoothed trajectory
hAx1 = axes('Parent',hLeftPanel);
axis(hAx1,[xMin xMax yMin yMax zMin zMax]);
plot3(hAx1,truePosition(1,:),truePosition(2,:),truePosition(3,:),'-','lineWidth',2);
hold on
plot3(hAx1,measPosTraj(:,1),measPosTraj(:,2),measPosTraj(:,3)),'.','MarkerSize',3,'lineWidth',2);
plot3(hAx1,correctPosition(1,:),correctPosition(2,:),correctPosition(3,:),'-','lineWidth',.05);
plot3(hAx1,smoothPosition(1,:),smoothPosition(2,:),smoothPosition(3,:),'-','lineWidth',2);

title("Trajectory " + n);
xlabel('X (m)');
ylabel('Y (m)');
zlabel('Z (m)');
lObj = legend('True Position', 'Measured Position', 'Corrected Position', 'smoothed Position','Location','best');
axis(hAx1,'square');
grid(hAx1,'minor');

```

```
% Switch view to X-Y if Z 0
viewSwitch = mean(truePosition(3,:));
if viewSwitch == 0
    view(90,90);
else
    view(60,10);
end
% Set legend position
legendPos = lObj.Position;
set(lObj, 'Position', [legendPos(1)*1.1 legendPos(2) legendPos(3) legendPos(4)])
set(gca, 'ZDir', 'reverse');

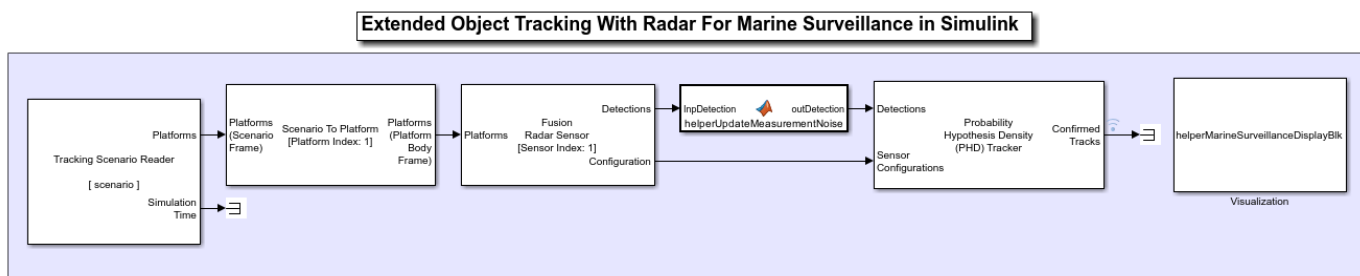
% Position RMS plot
hAx2 = subplot(2,1,1, 'Parent', hRightPanel);
line(hAx2, time(20:end-1), correctPosRMS(20:end), 'color', 'm');
hold on;
line(hAx2, time(20:end-1), smoothPosRMS(20:end), 'color', 'b');
grid(hAx2, 'on');
grid(hAx2, 'minor');
xlim([1 inf]);
xlabel('time (s)');
ylabel('Position Error');
legend('Correct Error', 'Smooth Error');

% Velocity RMS plot
hAx3 = subplot(2,1,2, 'Parent', hRightPanel);
line(hAx3, time(20:end-1), correctVelRMS(20:end), 'color', 'm');
hold on;
line(hAx3, time(20:end-1), smoothVelRMS(20:end), 'color', 'b');
grid(hAx3, 'on');
grid(hAx3, 'minor');
xlim([1 inf]);
xlabel('time (s)');
ylabel('Velocity Error');
legend('Correct Error', 'Smooth Error');
end
```

Extended Object Tracking with Radar for Marine Surveillance in Simulink

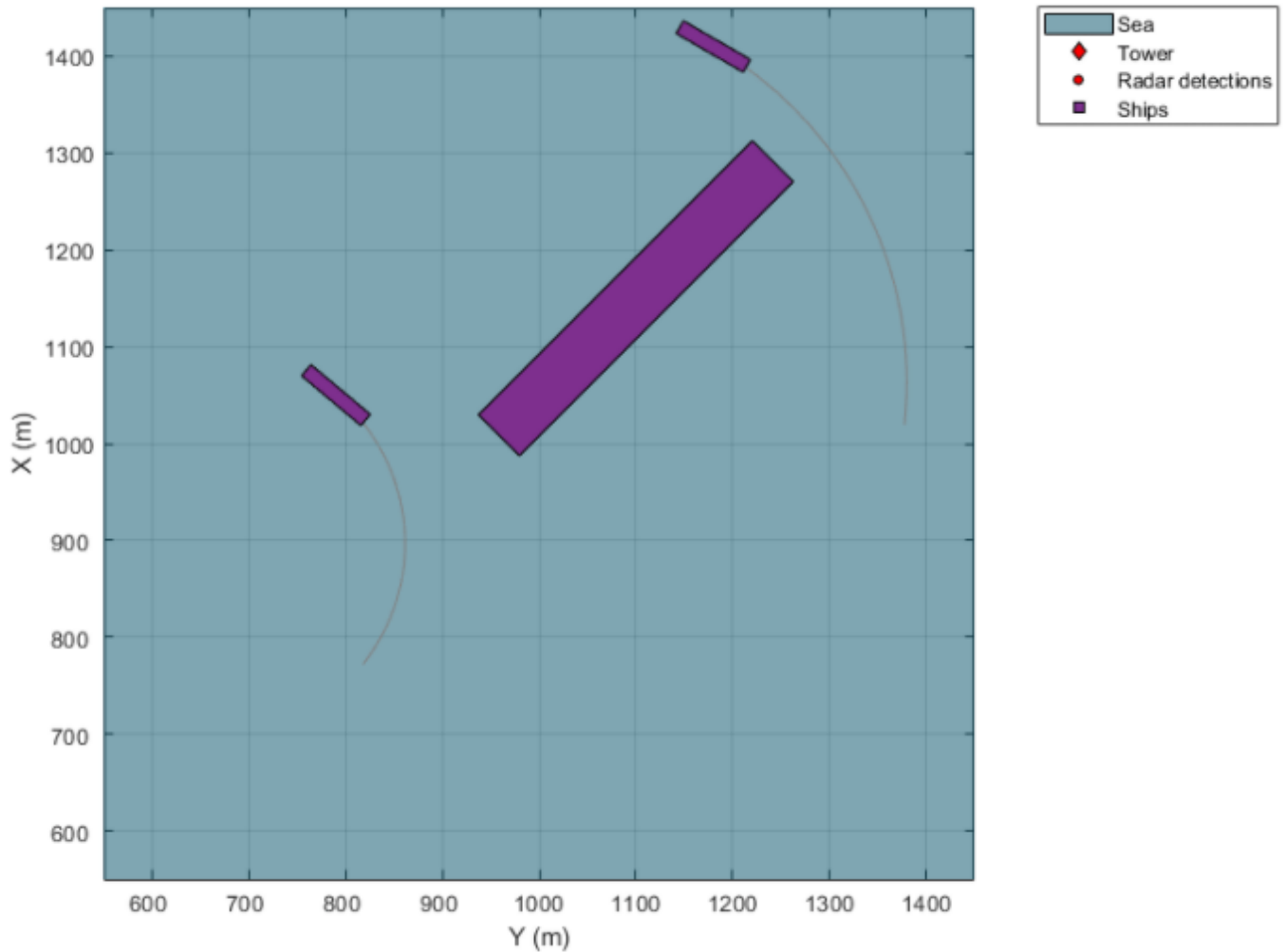
This example shows how to model a marine scenario, simulate detections from a marine surveillance radar, and configure a multi-target probability hypothesis density (PHD) tracker to estimate the location and size of simulated ships using the radar detections in Simulink®. Simulink provides a powerful environment for the modeling and simulation of dynamic systems and processes. This example closely follows the “Extended Object Tracking With Radar For Marine Surveillance” on page 6-370 MATLAB® example.

Overview of the model



Scenario Simulation

The scenario is created by using the `trackingScenario` object. The scenario has three target platforms and one tower platform. The radar is mounted on the tower platform 20 meters above sea level observing two boats maneuvering in the vicinity of a large ship. The radar stares into the harbor, surveying a 30-degree azimuth sector. The two boats are turning at 20 and 30 knots, whereas the large ship is traveling with a constant heading at 10 knots. See helper file `getSimulationData` for more details on the scenario.



You use the Tracking Scenario Reader block to read the scenario object from the MATLAB workspace and generate platform poses in the model. The block outputs platform poses as `Simulink.Bus` (`Simulink`) objects. You use the Fusion Radar Sensor block to model the radar sensor and configure the block with the following specifications for the marine surveillance radar:

- Total angular field of view: 30 deg azimuth, 10 deg elevation
- Azimuth resolution: 2 deg
- Range resolution: 5 m

Parameters	Measurements	Target profiles
Resolution settings		
Azimuth resolution (deg):	<input type="text" value="2"/>	⋮
Range resolution (m):	<input type="text" value="5"/>	⋮
Bias settings		
Azimuth bias fraction:	<input type="text" value="0.1"/>	⋮
Range bias fraction:	<input type="text" value="0.05"/>	⋮
Detector settings		
Total angular field of view [AZ, EL] (deg):	<input type="text" value="[30 10]"/>	<small>[30,10]</small> ⋮
Range limits [MIN, MAX] (m):	<input type="text" value="[0, 100e3]"/>	<small>[0,100000]</small> ⋮
Detection probability:	<input type="text" value="0.9"/>	⋮
False alarm rate:	<input type="text" value="1e-6"/>	<small>1e-06</small> ⋮
Reference target range (m):	<input type="text" value="5e3"/>	<small>5000</small> ⋮
Reference target RCS (dBsm):	<input type="text" value="0"/>	⋮
Center frequency (Hz):	<input type="text" value="300e6"/>	<small>300000000</small> ⋮
Random Number Generator Settings		
Random number generation:	<input type="text" value="Specify seed"/>	⋮
Initial seed:	<input type="text" value="2022"/>	⋮

The Tracking Scenario Reader block generates platform poses in scenario frame and the Fusion Radar Sensor block requires poses in radar tower frame. You use the Scenario To Platform block to transform platform poses from scenario frame to radar tower frame before passing them to the radar block. You configure the block to transform platform poses with respect to the tower platform by setting the reference platform index parameter to 1.

You use the `helperUpdateMeasurementNoise` function block to increase the measurement noise reported by the Fusion Radar Sensor block, since the reported measurement noise in the detections is too small.

Multi-Target GGIW-PHD Tracker

Pass the generated detections to the multi-target Probability Hypothesis Density (PHD) Tracker block. Configure the tracker with a Gamma Gaussian Inverse Wishart filter, which estimates the ship dimensions and orientations as ellipsoids. The tracker allows multiple detections from each ship to be associated to a single track and use them to estimate the ship extent. 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.

Probability Hypothesis Density (PHD) Tracker

The PHD tracker block creates and manages the tracks of stationary and moving objects in a multi-sensor environment. The tracker uses a multi-target probability hypothesis density filter to estimate the states of point targets and extended objects. The PHD is represented by a weighted summation of probability density functions, and peaks in the PHD are extracted to represent possible targets.

[Source code](#)

Tracker Configuration	Track Management	Port Setting
Tracker identifier:	<input type="text" value="0"/>	⋮
Detection partition function:	<input type="text" value="@(<math>x</math>)partitionDetections(x,1.5,6)"/>	
Detection selection threshold:	<input type="text" value="25"/>	⋮
Maximum number of sensors:	<input type="text" value="20"/>	⋮
Maximum number of tracks:	<input type="text" value="1000"/>	⋮
Sensor configurations:	<input type="text" value="sensorConfig"/>	⋮
<input checked="" type="checkbox"/> Update sensor configurations with time		
Track state parameters:	<input type="text" value="struct"/>	⋮
<input type="checkbox"/> Update track state parameters with time		
Simulate using:	<input type="text" value="Interpreted execution"/>	

Probability Hypothesis Density (PHD) Tracker

The PHD tracker block creates and manages the tracks of stationary and moving objects in a multi-sensor environment. The tracker uses a multi-target probability hypothesis density filter to estimate the states of point targets and extended objects. The PHD is represented by a weighted summation of probability density functions, and peaks in the PHD are extracted to represent possible targets.

[Source code](#)

Tracker Configuration	Track Management	Port Setting
Birth rate of new targets:	<input type="text" value="1e-5"/>	⋮
Death rate of components:	<input type="text" value="1e-6"/>	⋮
Threshold for initializing tentative track:	<input type="text" value="0.75"/>	⋮
Threshold for track confirmation:	<input type="text" value="0.8"/>	⋮
Threshold for track deletion:	<input type="text" value="1e-6"/>	⋮
Threshold for components merging:	<input type="text" value="25"/>	⋮
Thresholds for label management:	<input type="text" value="[1.1, 1, 0.8]"/>	⋮

The first step toward configuring the PHD tracker is to define the sensor configuration. You obtain the configuration from the sensor block by using the `trackingSensorConfiguration` object. See helper file `getSimulationData` for more details. The tracker uses the `ggiwfilterInitFcn` supporting function to initialize a constant turn-rate Gamma Gaussian Inverse Wishart (GGIW) PHD filter. `ggiwfilterInitFcn` 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.

You configure the tracker block to update the sensor configuration with time and specify a detection partition function as `partitionDetections` with 2 and 6 as the lower and upper bounds for distance threshold. You also configure the tracker block with following parameters:

- Birth rate of new targets: 1e-5
- Threshold for initializing tentative track: 0.75
- Threshold for track deletion: 1e-6

The track states of the three ships report the estimated size of each ship using a 3D positional extent matrix. Take the eigen decomposition of the covariance matrices to compute the estimated length, width, and height for each ship.

This table summarizes the estimated and actual dimensions of the three ships.

TrackID	Length	Actual_Length	Width	Actual_Width	Height	Actual_Height
1	98.7172	80	17.5076	15	16.4814	5
2	473.5169	400	55.4217	60	9.1532	15
3	100.9145	80	18.9855	15	17.5708	5

The tracker is able to differentiate between the sizes of the large and smaller ships by estimating the shape of each ship as an ellipse. In the simulation, however, the true shape of each ship is modeled using a cuboid. This mismatch between the shape assumption made by the tracker and the shapes of the modeled ships results in overestimates of 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. As a result, the height estimates reported by the tracker are inaccurate.

Visualization

Visualization

A helper block to visualize scenario.

[Source code](#)

Parameters

IsSea:

Minimum and maximum distance in the x-axis:

Minimum and maximum distance in the y-axis:

Minimum and maximum distance in the z-axis:

Filename for movie to be written by writeMovie:

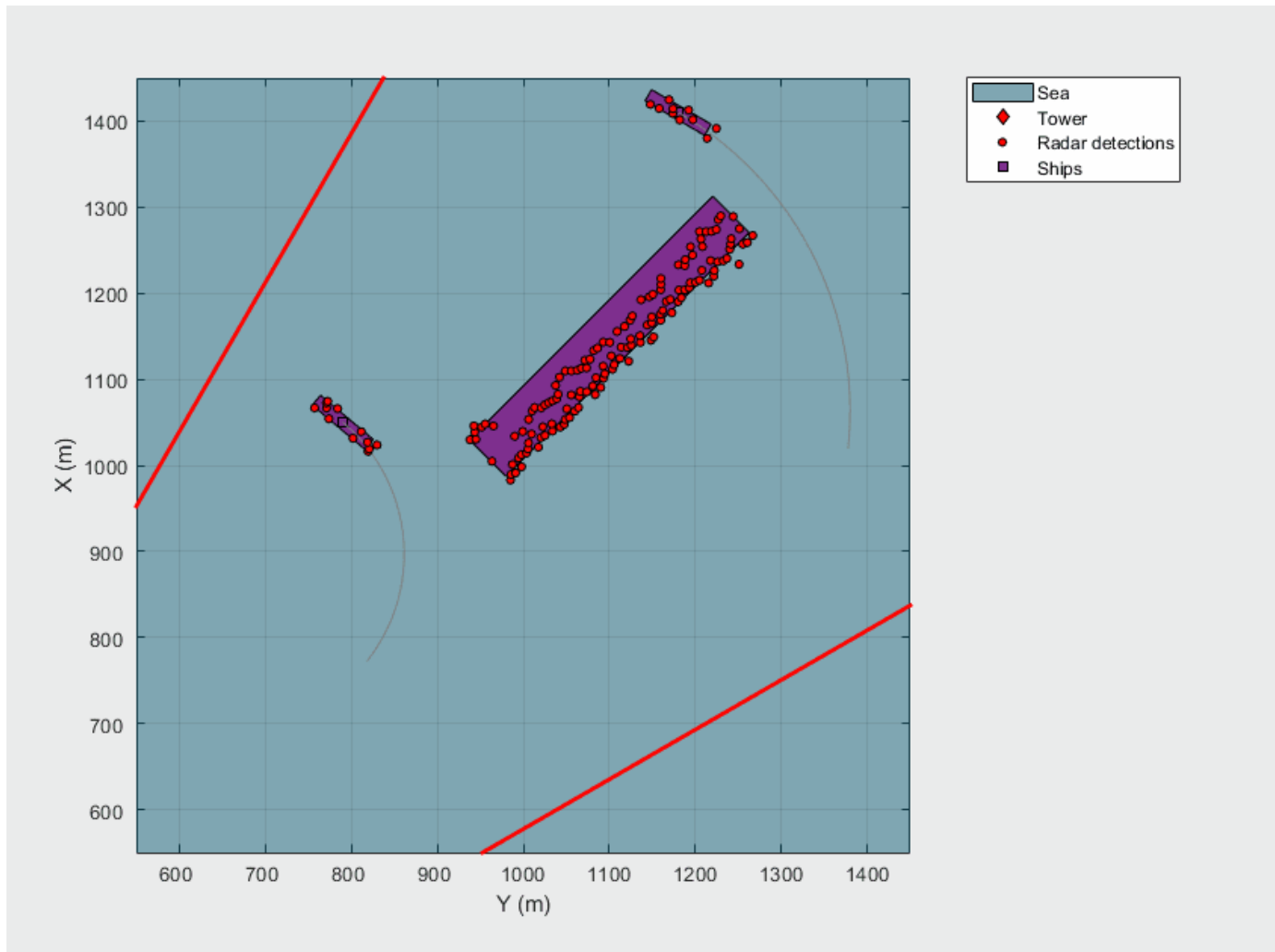
DistanceUnits:

Select x and y from the state vector:

Select vx and vy from the state vector:

Record a GIF animation of visualization

The visualization used for this example is defined using a MATLAB System (Simulink) block, `helperMarineSurveillanceDisplayBlk` attached with this example.



The animation shows the radar detections as red dots, the estimated track locations as yellow squares annotated with the track ID, and the estimated extents as yellow ellipses. The radar tower is located at the origin, with the coordinates of (0,0), which is not shown in the figure. The radar's field of view is represented by the two red lines crossing the top and bottom of the figure. All the ships lie within the radar's field of view. Because the sizes of the ships are much larger than the radar's range and azimuth resolution, there are many detections made along the ship surfaces visible to the radar.

Summary

This example shows how to configure the Fusion Radar Sensor block and the multi-target Probability Hypothesis Density (PHD) tracker block in Simulink. It also shows how to track and visualize extended objects in a marine scenario in Simulink.

How to Simulate Out-of-Sequence Measurements

This example shows how to simulate out-of-sequence measurements (OOSMs). OOSMs occur in multisensor systems when detections from some sensors arrive at the tracker after the tracker has processed detections of the same or earlier time from other sensors. Several reasons, mostly sensor processing lag and sensor-to-tracker connection delays, can cause this problem. For more about OOSMs, see “Introduction to Out-of-Sequence Measurement Handling” on page 4-18

Handling OOSMs is a key requirement from the trackers. There are several ways to handle OOSMs, including neglecting them, erroring out, or including them using a technique like retrodiction. You have to make a choice based on the multisensor system, its update rate, the reasons of OOSMs, and the data contained in the OOSMs. Whichever OOSM technique you choose, it is critical to test and verify that the tracker behaves correctly when OOSMs are encountered. Thus, it is necessary to simulate OOSMs under various conditions of the multisensor system.

In this example, you learn how to simulate OOSMs using the `objectDetectionDelay` object, which serves as a buffer that holds measurements in the format of `objectDetection` objects and simulates time delay before they reach the tracker.

Simulate Constant Delay from All Sensors

First, define a default `objectDetectionDelay` object.

```
allDelayer = objectDetectionDelay;
disp(allDelayer)

objectDetectionDelay with properties:
    SensorIndices: "All"
    Capacity: Inf
    DelaySource: 'Property'
    DelayDistribution: 'Constant'
    DelayParameters: 1
```

In this setting, the `allDelayer` object adds a constant time delay of 1 second to all sensors. Create an `objectDetection` object.

```
timeDelays = [];
detection = objectDetection(0, [100;10;1]);
disp(detection);

objectDetection with properties:
    Time: 0
    Measurement: [3x1 double]
    MeasurementNoise: [3x3 double]
    SensorIndex: 1
    ObjectClassID: 0
    ObjectClassParameters: []
    MeasurementParameters: {}
    ObjectAttributes: {}
```

Delay the detection and observe that the output is empty, because the current time is 0 seconds, and the detection should be delivered from the delay object at time = 1 second.

```
outDet = allDelayer({detection}, 0)
```

```
outDet =
```

```
    0x1 empty cell array
```

Call the delay object again with time = 0.99 seconds, right before time = 1 second. Observe that the output is still empty. You can use an empty cell to pass no new detections to the delay object.

```
outDet = allDelayer({}, 0.99); % There are still no detections in the output
```

Now, call the delay object with a time of 1 second and observe the delayed detection in the output.

```
currentTime = 1;
outDet = allDelayer({}, currentTime);
timeDelays(end+1) = currentTime - outDet{1}.Time;

% Display that this is the detection added at time = 0
disp(outDet{1})
```

```
objectDetection with properties:
```

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

You can control the delay time by setting the DelayParameters property.

```
allDelayer.DelayParameters = 2;
detection = objectDetection(1.5,[100;10;1]);
disp(detection);
```

```
objectDetection with properties:
```

```
        Time: 1.5000
    Measurement: [3x1 double]
MeasurementNoise: [3x3 double]
    SensorIndex: 1
    ObjectClassID: 0
ObjectClassParameters: []
MeasurementParameters: {}
    ObjectAttributes: {}
```

The delay object will only output the detection when the it is stepped beyond time = 3.5. You can verify that using the DeliveryTime field of the information structure.

```
[~, ~, info] = allDelayer({detection}, 3);
disp(info)
```

```
DetectionTime: 1.5000
        Delay: 2
    DeliveryTime: 3.5000
```

Now add a new detection to the buffer:

```
detection = objectDetection(3.2,[-100;-10;-1]);  
[~,~,info] = allDelayer({detection}, 3.2);  
disp(info)
```

```
DetectionTime: [1.5000 3.2000]  
Delay: [2 2]  
DeliveryTime: [3.5000 5.2000]
```

As shown in the following code, the delay object outputs each detection at their corresponding delivery time.

```
% Get the detection from time 1.5  
currentTime = 3.5;  
outDet = allDelayer({}, currentTime);  
timeDelays(end+1) = currentTime - outDet{1}.Time;  
disp(outDet{1})
```

```
objectDetection with properties:
```

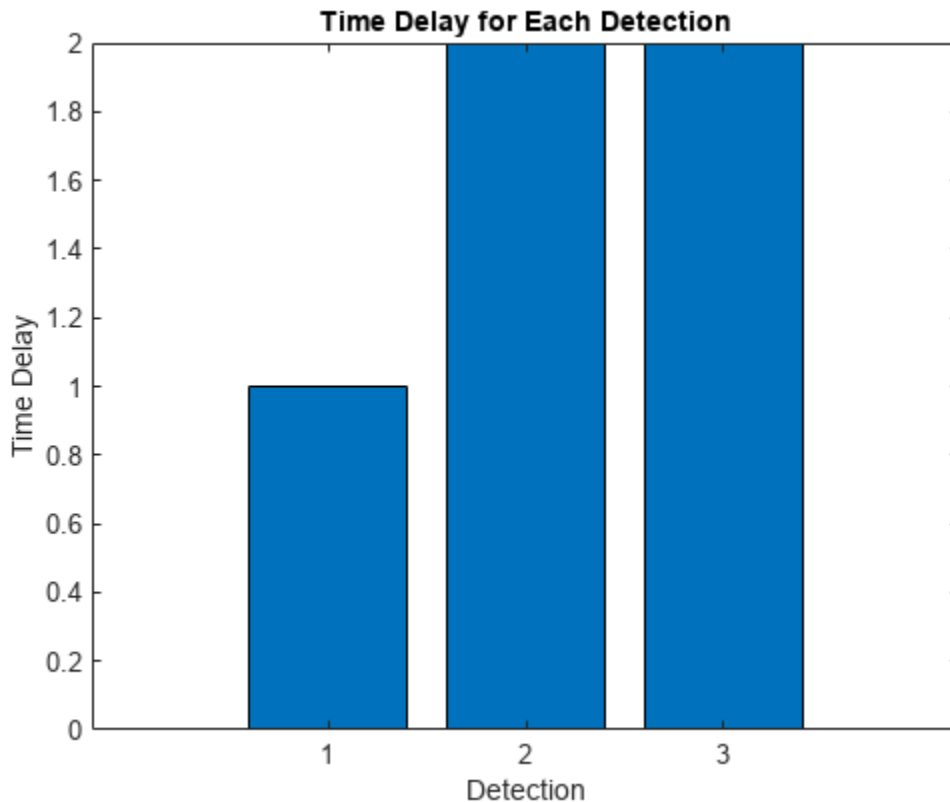
```
Time: 1.5000  
Measurement: [3x1 double]  
MeasurementNoise: [3x3 double]  
SensorIndex: 1  
ObjectClassID: 0  
ObjectClassParameters: []  
MeasurementParameters: {}  
ObjectAttributes: {}
```

```
% Get the detection from time 3.2  
currentTime = 5.2;  
outDet = allDelayer({}, currentTime);  
timeDelays(end+1) = currentTime - outDet{1}.Time;  
disp(outDet{1})
```

```
objectDetection with properties:
```

```
Time: 3.2000  
Measurement: [3x1 double]  
MeasurementNoise: [3x3 double]  
SensorIndex: 1  
ObjectClassID: 0  
ObjectClassParameters: []  
MeasurementParameters: {}  
ObjectAttributes: {}
```

```
bar(timeDelays);  
title("Time Delay for Each Detection")  
xlabel("Detection");  
ylabel("Time Delay");
```



Simulate Constant Delay from Specific Sensors

In multisensor systems the time delay for each sensor is usually different, because of the sensor-specific processing time or the sensor-to-tracker communication lag. To simulate these differences, you can define the delay object to work on detections from specific sensors.

% Sensor 1 is delayed by 1 second.

```
sensor1Delayer = objectDetectionDelay(SensorIndices = 1, DelayParameters = 1);
sensor1TimeDelays = [];
```

% Sensors 2 and 3 are delayed by 2.5 seconds.

```
sensor23Delayer = objectDetectionDelay(SensorIndices = [2 3], DelayParameters = 2.5);
sensor23TimeDelays = [];
sensor4TimeDelays = [];
```

The object does not delay detections from any sensor that is not listed in the `SensorIndices` property. Therefore, you can use both detection delay objects in series to get the delayed objects. In this block of code, notice that the fourth detection, from sensor 4, is not delayed.

```
allDetections = {...
    objectDetection(0, [10;10;10], SensorIndex = 1); ...
    objectDetection(0, [20;20;20], SensorIndex = 2); ...
    objectDetection(0, [30;30;30], SensorIndex = 3); ...
    objectDetection(0, [40;40;40], SensorIndex = 4)};
out1 = sensor1Delayer(allDetections, 0);
out2 = sensor23Delayer(out1, 0);
sensor4TimeDelays(end+1) = allDetections{4}.Time - out2{1}.Time;
disp(out2{:})
```

objectDetection with properties:

```

        Time: 0
        Measurement: [3x1 double]
        MeasurementNoise: [3x3 double]
        SensorIndex: 4
        ObjectClassID: 0
        ObjectClassParameters: []
        MeasurementParameters: {}
        ObjectAttributes: {}

```

You use the following loop to step the delay objects and receive the detections after their corresponding delays. The detection from sensor 1 should be released at $t = 1$ second and the detections from sensors 2 and 3 should be released at $t = 2.5$ seconds.

```

for time = 0.5:0.5:3
    out1 = sensor1Delayer({}, time);
    out2 = sensor23Delayer(out1, time);
    if isempty(out2)
        disp("The time is: " + time + " and there are no delivered detections.")
    else
        disp("The time is: " + time + " and delivered detections after both delay objects are:")
        out2{:}
        for i = 1:numel(out2)
            if out2{i}.SensorIndex == 1
                sensor1TimeDelays(end+1) = time - out2{i}.Time;
            else
                sensor23TimeDelays(end+1) = time - out2{i}.Time;
            end
        end
    end
end
end
end

```

The time is: 0.5 and there are no delivered detections.

The time is: 1 and delivered detections after both delay objects are:

```

ans =
    objectDetection with properties:

```

```

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

```

The time is: 1.5 and there are no delivered detections.

The time is: 2 and there are no delivered detections.

The time is: 2.5 and delivered detections after both delay objects are:

```

ans =
    objectDetection with properties:

```

```

        Time: 0

```

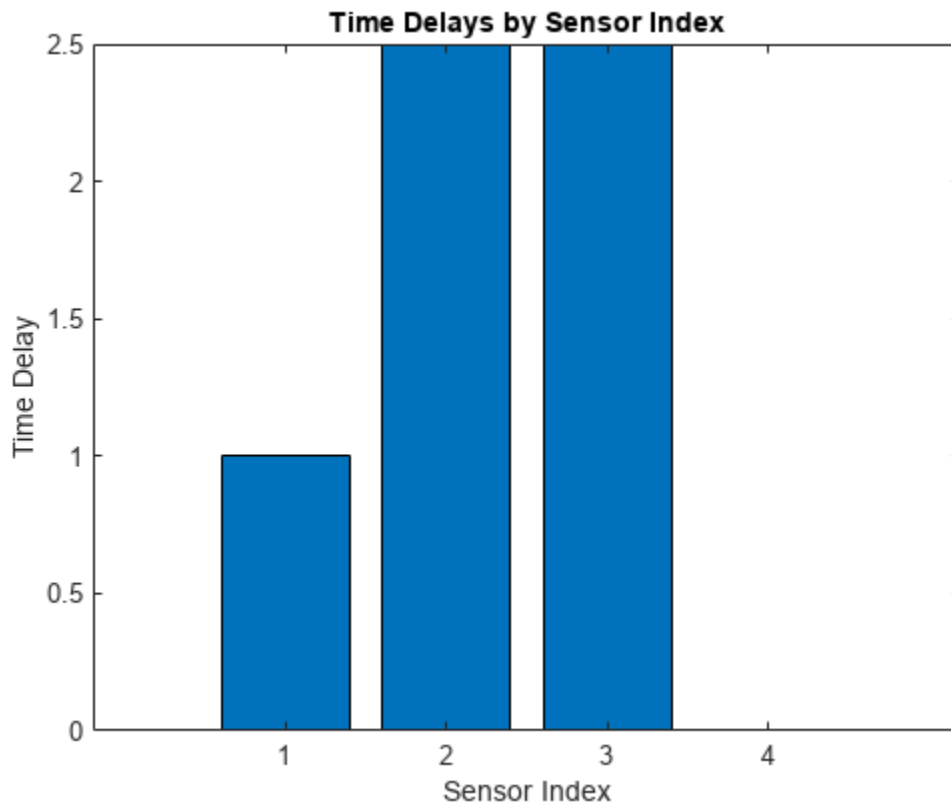


```
Measurement: [3x1 double]
MeasurementNoise: [3x3 double]
SensorIndex: 2
ObjectClassID: 0
ObjectClassParameters: []
MeasurementParameters: {}
ObjectAttributes: {}
```

```
ans =
  objectDetection with properties:
      Time: 0
      Measurement: [3x1 double]
      MeasurementNoise: [3x3 double]
      SensorIndex: 3
      ObjectClassID: 0
      ObjectClassParameters: []
      MeasurementParameters: {}
      ObjectAttributes: {}
```

The time is: 3 and there are no delivered detections.

```
bar([mean(sensor1TimeDelays), mean(sensor23TimeDelays), mean(sensor23TimeDelays), mean(sensor4TimeDelays)]);
title("Time Delays by Sensor Index");
xlabel("Sensor Index");
ylabel("Time Delay");
```



Simulate Random Delay

In some cases, for example random network delays, the lag from the moment the sensor creates the measurement until it arrives at the tracker is random. You can use the `objectDetectionDelay` object to simulate random time delays by setting the `DelayDistribution` property to "Uniform" or "Normal". You can further specify the delay distribution by setting the `DelayParameters` property. The code below shows how to specify the `objectDetectionDelay` object to simulate random time delays that are normally distributed with a mean of 0.5 seconds and a standard deviation of 0.1 seconds.

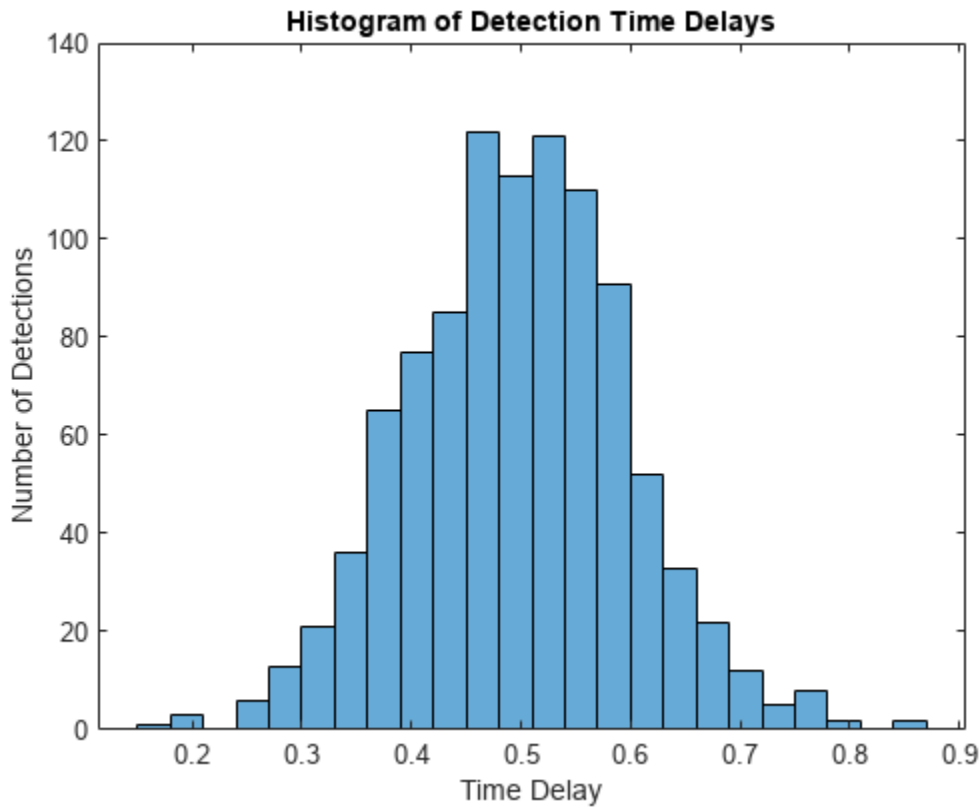
```
uniformDelayer = objectDetectionDelay(SensorIndices = 1, DelayDistribution = "Normal", DelayParameters = [0.5 0.1]);
timeDelays = [];
disp(uniformDelayer)
```

```
objectDetectionDelay with properties:
```

```
    SensorIndices: 1
      Capacity: Inf
    DelaySource: 'Property'
DelayDistribution: 'Normal'
  DelayParameters: [0.5000 0.1000]
```

Use the uniform delay in a loop and observe the time delays using the info output. In the first second, there will be an input detection to the `uniformDelayer` object and after that there will be no input.

```
for time = 0:1e-3:2
    if time < 1
        inDet = {objectDetection(time, [1;1;1], SensorIndex = 1)};
    else
        inDet = {};
    end
    outDet = uniformDelayer(inDet, time);
    for i = 1:numel(outDet)
        timeDelays(end+1) = time - outDet{i}.Time;
    end
end
histogram(timeDelays)
title("Histogram of Detection Time Delays")
xlabel("Time Delay")
ylabel("Number of Detections");
```



Use Time Delay Input

To have the maximum flexibility of adding time delay to OOSMs, you set the `DelaySource` property to "Input" and use the third input of the delay object to add a time delay. You can delay all the detections with the same time delay or each one with a different time delay. The delay is applied only to detections with `SensorIndex` property value matching one of the `SensorIndices`.

```
inputDelayer = objectDetectionDelay(DelaySource = "Input");
disp(inputDelayer);
```

objectDetectionDelay with properties:

```
SensorIndices: "All"
Capacity: Inf
DelaySource: 'Input'
```

The following code shows how to delay a detection by 2 seconds.

```
time = 0;
timeDelay = 2;
det1 = objectDetection(time, [1;1;1]);
[~,~,info] = inputDelayer(det1, time, timeDelay);
disp(info)
```

```
DetectionTime: 0
Delay: 2
DeliveryTime: 2
```

You delay multiple detections by 3 seconds

```
time = 1;
timeDelay = 3;
detections = [objectDetection(time, [1;1;1]), objectDetection(time, [2;2;2])];
[~,~,info] = inputDelayer(detections, time, timeDelay);
disp(info)
```

```
DetectionTime: [0 1 1]
              Delay: [2 3 3]
              DeliveryTime: [2 4 4]
```

Then, you delay three additional detections by different time delays.

```
time = 2;
timeDelays = [0.5;1;2];
detections = [objectDetection(time, [1;1;1]), objectDetection(time, [2;2;2]), objectDetection(time, [3;3;3]), objectDetection(time, [4;4;4])];
[outDet,~,info] = inputDelayer(detections, time, timeDelays);
disp("After two seconds the first detection is delivered")
```

After two seconds the first detection is delivered

```
disp(outDet)
```

```
objectDetection with properties:
```

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

```
disp("The rest of the detections are delayed and their time delays are listed in the info:")
```

The rest of the detections are delayed and their time delays are listed in the info:

```
disp(info)
```

```
DetectionTime: [1 1 2 2 2]
              Delay: [3 3 0.5000 1 2]
              DeliveryTime: [4 4 2.5000 3 4]
```

Use Delay in Simulated Scenario

So far, you have learned how to simulate OOSMs using the `objectDetectionDelay` object with various ways to control the time delay. In this section, you integrate the `objectDetectionDelay` object into a scenario that includes target and sensor simulation. Create a tracking scenario and the visualization using the `createScenario` and `createVisualization` helper functions, respectively.

```
scenario = createScenario();
[tp1, tp2] = createVisualization(scenario);
```

You define the `objectDetectionDelay` object to only delay detections from sensor 1. In this case, you simulate the time delay as a combination of two delays: the sensor always lags by 1 or 2 simulation steps and the network adds a lag that is normally distributed with zero mean and a standard deviation of 0.03. To combine the two delays, you define the `DelaySource` as "Input".

```
delayer = objectDetectionDelay(SensorIndices = 1, DelaySource = "Input");
```

Run the following code and observe that the detections from sensor 1 (in red) always lag behind the object. You can also compare the detections with time delay (top) and without time delay (bottom).

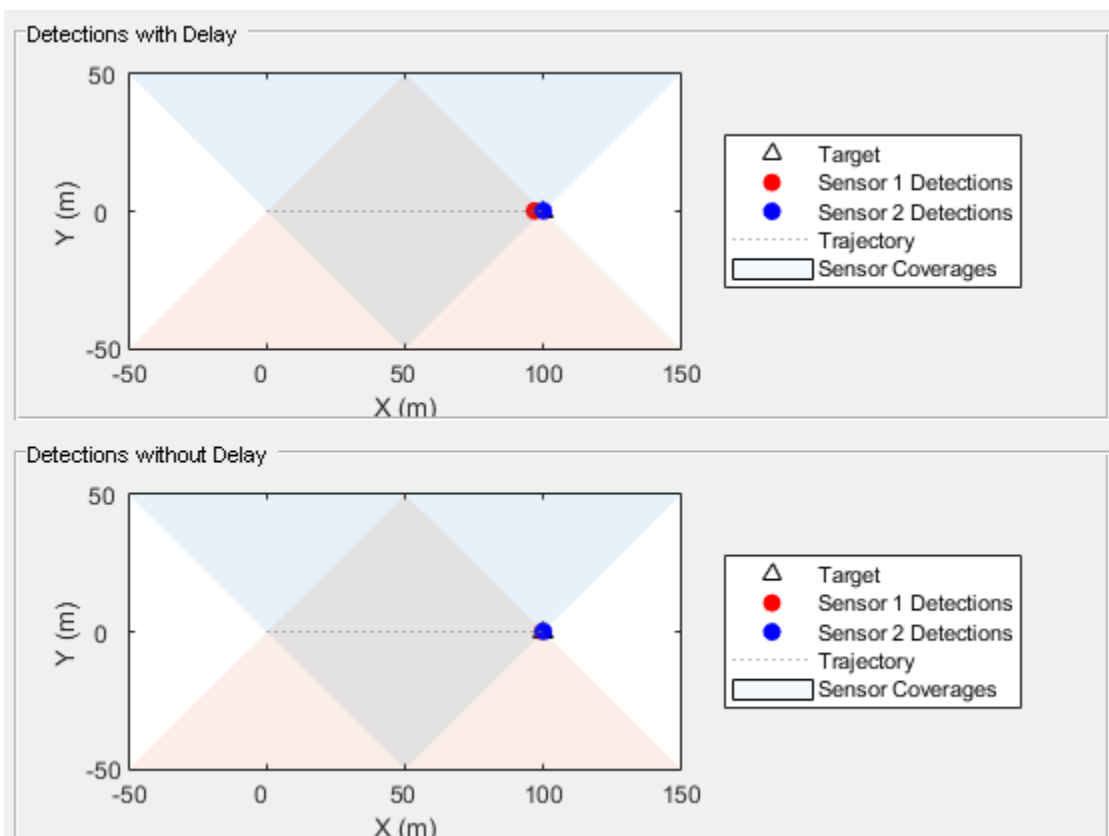
```
while advance(scenario)
    simTime = scenario.SimulationTime;

    % Simulate all the detections from the scenario.
    simDets = detect(scenario);

    % Add a time delay of 1 or 2 simulation steps plus random delay.
    timeDelay = randi(2,1) / scenario.UpdateRate + 0.03 * randn;

    % Apply time delay to the detections.
    delayedDets = delayer(simDets, simTime, timeDelay);

    % Update visualization.
    updateVisualization(tp1, scenario, simDets);
    updateVisualization(tp2, scenario, delayedDets);
end
```



Summary

In this example, you used the `objectDetectionDelay` object to simulate out-of-sequence measurements (OOSM). You learned how to control which sensors are affected by the time delay, how to define various random distributions to the delay, and how to use the time delay input argument.

You can use the `objectDetectionDelay` object within a scenario simulation or with recorded data to simulate detections delays.

Supporting Functions

createScenario create a tracking scenario that has one platform with a waypoint trajectory and two radar sensors.

```
function scenario = createScenario
scenario = trackingScenario(UpdateRate = 10);
% Define a target
platform(scenario, Trajectory = waypointTrajectory([0 0 0; 100 0 0], [0;10]));

% Define a radar with SensorIndex = 1 mounted on one platform
radar1 = fusionRadarSensor(1, "No scanning", MountingAngles = [90 0 0], FieldOfView = [90 5], ...
    RangeResolution = 1, UpdateRate = 10, RangeLimits = [0 200], ...
    DetectionCoordinates = "Scenario", HasINS = true, HasFalseAlarms = false);
r1p = platform(scenario, Sensors = radar1, Position = [50 -50 0]);
r1p.Signatures{1} = rcsSignature(Pattern = -1e5); % To avoid detection by other radar

% Define a radar with SensorIndex = 2 mounted on another platform
radar2 = fusionRadarSensor(2, "No scanning", MountingAngles = [-90 0 0], FieldOfView = [90 5], ...
    RangeResolution = 1, UpdateRate = 10, RangeLimits = [0 200], ...
    DetectionCoordinates = "Scenario", HasINS = true, HasFalseAlarms = false);
r2p = platform(scenario, Sensors = radar2, Position = [50 50 0]);
r2p.Signatures{1} = rcsSignature(Pattern = -1e5); % To avoid detection by other radar
end
```

createVisualization create visualization for the tracking scenario

```
function [tp1, tp2] = createVisualization(scenario)
f = figure;
p1 = uipanel(f, Title = "Detections without Delay", Position = [0.01 0.01 0.98 0.48]);
p2 = uipanel(f, Title = "Detections with Delay", Position = [0.01 0.51 0.98 0.48]);
a1 = axes(p1);
a2 = axes(p2);
legend(a1, Location = "eastoutside", Orientation = "vertical");
legend(a2, Location = "eastoutside", Orientation = "vertical");
tp1 = theaterPlot(Parent = a1, XLimits = [-50 150], YLimits = [-50 50], ZLimits = [-50 50]);
tp2 = theaterPlot(Parent = a2, XLimits = [-50 150], YLimits = [-50 50], ZLimits = [-50 50]);
platformPlotter(tp1, DisplayName = "Target", Tag = "Target");
platformPlotter(tp2, DisplayName = "Target", Tag = "Target");
detectionPlotter(tp1, DisplayName = "Sensor 1 Detections", Tag = "Sensor1", MarkerEdgeColor = "r");
detectionPlotter(tp1, DisplayName = "Sensor 2 Detections", Tag = "Sensor2", MarkerEdgeColor = "b");
detectionPlotter(tp2, DisplayName = "Sensor 1 Detections", Tag = "Sensor1", MarkerEdgeColor = "r");
detectionPlotter(tp2, DisplayName = "Sensor 2 Detections", Tag = "Sensor2", MarkerEdgeColor = "b");
trp = trajectoryPlotter(tp1, DisplayName = "Trajectory");
traj = scenario.Platforms{1}.Trajectory;
sampledTimes = traj.TimeOfArrival(1):0.1:traj.TimeOfArrival(end);
pposition = lookupPose(traj,sampledTimes);
plotTrajectory(trp,{pposition});
trp = trajectoryPlotter(tp2, DisplayName = "Trajectory");
traj = scenario.Platforms{1}.Trajectory;
sampledTimes = traj.TimeOfArrival(1):0.1:traj.TimeOfArrival(end);
pposition = lookupPose(traj,sampledTimes);
plotTrajectory(trp,{pposition});
cvp = coveragePlotter(tp1, DisplayName = "Sensor Coverages", Alpha = [0.05 0.05]);
plotCoverage(cvp, coverageConfig(scenario));
```

```
cvp = coveragePlotter(tp2, DisplayName = "Sensor Coverages", Alpha = [0.05 0.05]);  
plotCoverage(cvp, coverageConfig(scenario));  
end
```

updateVisualization updates the visualization

```
function updateVisualization(tp, scenario, dets)  
tarp = findPlotter(tp, Tag = "Target");  
detp1 = findPlotter(tp, Tag = "Sensor1");  
detp2 = findPlotter(tp, Tag = "Sensor2");  
plotPlatform(tarp, scenario.Platforms{1}.Position);  
dets = [dets{:}];  
if ~isempty(dets)  
    d1 = dets([dets.SensorIndex] == 1);  
    plotDetection(detp1, detectionPositions(d1));  
    d2 = dets([dets.SensorIndex] == 2);  
    plotDetection(detp2, detectionPositions(d2));  
end  
end
```

detectionPositions extracts positions from the detections

```
function pos = detectionPositions(detections)  
if ~isempty(detections)  
    pos = [detections.Measurement]';  
else  
    pos = zeros(0,3);  
end  
end
```

Processor-in-the-Loop Verification of JPDA Tracker for Automotive Applications

This example shows you how to generate embedded code for a `trackerJPDA` (JPDA) tracker and verify it using processor-in-the-loop (PIL) simulations on a STM32 Nucleo board with 1 MB RAM and 2 MB flash memory. In this example, you configure the JPDA tracker to process detections from a camera and a radar sensor mounted in front of an ego vehicle in highway scenarios. For PIL simulations, you use simulated detections to verify the tracking and computational performance of the generated code.

Setup Tracking Algorithm for Embedded Code Generation

Balanced between computing requirements and tracking performance, the JPDA tracker is a suitable choice for embedded systems. At every step, the JPDA tracker splits the detections-to-tracks data association problem into multiple clusters per sensor. Each cluster contains a set of detections and tracks that can be assigned to each other after gating. The exact separation of detections and tracks into clusters, the size of each cluster, and number of feasible data association events per cluster is typically determined by run-time inputs and is not known at compilation time. For more information about the JPDA tracking algorithm, refer to the “Algorithms” section of `trackerJPDA`.

When generating embedded code from trackers for safety critical applications such as highway lane following, dynamic memory allocation is typically discouraged. This means that the amount of memory allocated to the tracker must be known at compilation time. Further, the generated code must fit the memory offered by the embedded device. To efficiently manage the memory footprint of a tracker without dynamic memory allocation, you must specify certain bounds on the tracker. These bounds are typically defined using prior knowledge about the targeted application. To bound the number of feasible events per cluster, you use a K-best JPDA tracker by specifying a finite value for the `MaxNumEvents` property. This allows the tracker to use a maximum of K data association events per cluster without enumerating over all feasible events. You use the `MaxNumDetectionsPerCluster` and `MaxNumTracksPerCluster` properties to bound the size of the cluster. For highway driving scenarios, the cluster size can be bounded by using prior knowledge about the maximum number of closely-spaced vehicles. You choose an appropriate value for `AssignmentThreshold` property for gating the detection-to-track association. A large `AssignmentThreshold` value can cause the gate size to be much larger than desired, which can result in the formation of large clusters. To avoid large clusters, you set the `ClusterViolationHandling` property to 'Terminate', which causes the tracker to error out if the cluster sizes are violated. You set the `MaxNumDetections` and `MaxNumDetectionsPerSensor` properties using the information from the simulated or actual sensor. In this example, the radar outputs a maximum of 36 object-level detections and the camera outputs a maximum of 10 object-level detections.

Finally, embedded code generation using Embedded® Coder™ requires MATLAB® code to be written in the form of a function. This function is typically referred to as an entry-point function. To rewrite tracking algorithm as a function, you define the tracker inside the entry-point function using as a persistent variable to preserve its state between function calls. For this example, the tracking algorithm is wrapped in the entry-point function, `trackingAlgorithm` shown below and attached with this example.

```
type trackingAlgorithm.m

function tracks = trackingAlgorithm(detections, time)
```



```

% Define the tracker as a persistent variable
persistent tracker

% Initialize the tracker on first call using isempty
if isempty(tracker)
    tracker = trackerJPDA(FilterInitializationFcn=@helperInitFVSFFilter,...
        MaxNumEvents=5,...% 5 best events per cluster
        MaxNumSensors=2,...% Only 2 sensors feed data to tracker
        MaxNumTracks=36,...% should be at least MaxNumDetectionsPerSensor
        MaxNumDetections=46,...% maximum number of detections from all sensors
        ClutterDensity=1e-9,...% False alarm rate per unit measurement volume
        AssignmentThreshold=50,...% Threshold for gating assignments
        ConfirmationThreshold=[5 6],...% Confirm a track with 5 hits out of 6
        DeletionThreshold=[4 5],...% Delete a track with 4 misses out of 5
        HitMissThreshold=0.5,...% Probability of assignment resulting in hit/miss
        EnableMemoryManagement=true,...% Enable memory management for reducing footprint
        MaxNumDetectionsPerCluster=5,...% Maximum detections per cluster
        MaxNumTracksPerCluster=5,...% Maximum tracks per cluster
        MaxNumDetectionsPerSensor=36,...% Maximum detections per cluster
        ClusterViolationHandling='Terminate'...% Error if cluster sizes are violated
    );
end

% Update the tracker every step using current detections and time stamp
tracks = tracker(detections, time);

end

```

Setup the Test Bench

To test the tracking algorithm, you use the `drivingScenario` (Automated Driving Toolbox) object to simulate a highway driving scenario. You use the `drivingRadarDataGenerator` (Automated Driving Toolbox) and `visionDetectionGenerator` (Automated Driving Toolbox) objects to simulate detections from radar and camera sensor respectively. The scenarios and sensor configurations used in this example are similar to the one shown in the “Forward Vehicle Sensor Fusion” (Automated Driving Toolbox) example and are applicable for automotive applications such as “Highway Lane Following” (Automated Driving Toolbox). The process of scenario and sensor model generation is wrapped in the helper function, `helperCreateFVSFPILScenario`, attached with this example. This function accepts the name of the scenario as an input. For compatible scenario names, see the [Explore Other Scenarios](#) on page 6-895 section at the bottom of this example.

The target board used in this example supports floating point operations in both single and double precision. To reduce the memory footprint of the tracker, you use single-precision inputs to the tracker. Using single-precision inputs to the tracker allows it to use strict single-precision arithmetic in the generated code. To cast detections to the single-precision, you use the `helperCastDetections` function attached with this example. You can configure the tracking algorithm to use double-precision inputs by changing the `dataType` variable to 'double'.

You evaluate the performance of the tracking algorithm using the `trackGOSPAMetric` (GOSPA) metric. The GOSPA metric uses the available ground truth from the scenario simulation and captures the accuracy of a tracking algorithm as a scalar distance per step. This feature of the metric also makes it an attractive method to assess the equivalency of a tracking algorithm during PIL simulation. In this example, you verify that the generated code on the target hardware produces the same results by comparing the GOSPA values from MATLAB simulation and PIL simulation.

```
% Create scenario.
scenarioName = 'scenario_FVSF_01_Curve_FourVehicles';
[scenario, egoVehicle, radar, camera] = helperCreateFVSFPILScenario(scenarioName);
```

```
% Create GOSPA metric object
gospaObj = trackGOSPAMetric(Distance='posabserr');
```

Next, you run the test bench on this particular scenario by running the tracker in MATLAB environment to ensure that the test bench and tracking algorithm produce expected results. You also capture the GOSPA metric during the MATLAB execution.

```
% Capture the GOSPA metric
gospa = zeros(0,1);
```

```
% Create display
scope = HelperJPDATrackerPILDisplay;
```

```
% Clear persistent variable before every run
clear trackingAlgorithm;
```

```
% Choose data type
dataType = 'single';
```

```
while advance(scenario)
    % Get current simulation time
    time = cast(scenario.SimulationTime, dataType);

    % Collect detections from both radar and camera sensors
    detections = helperCollectDetections(egoVehicle, radar, camera, time);

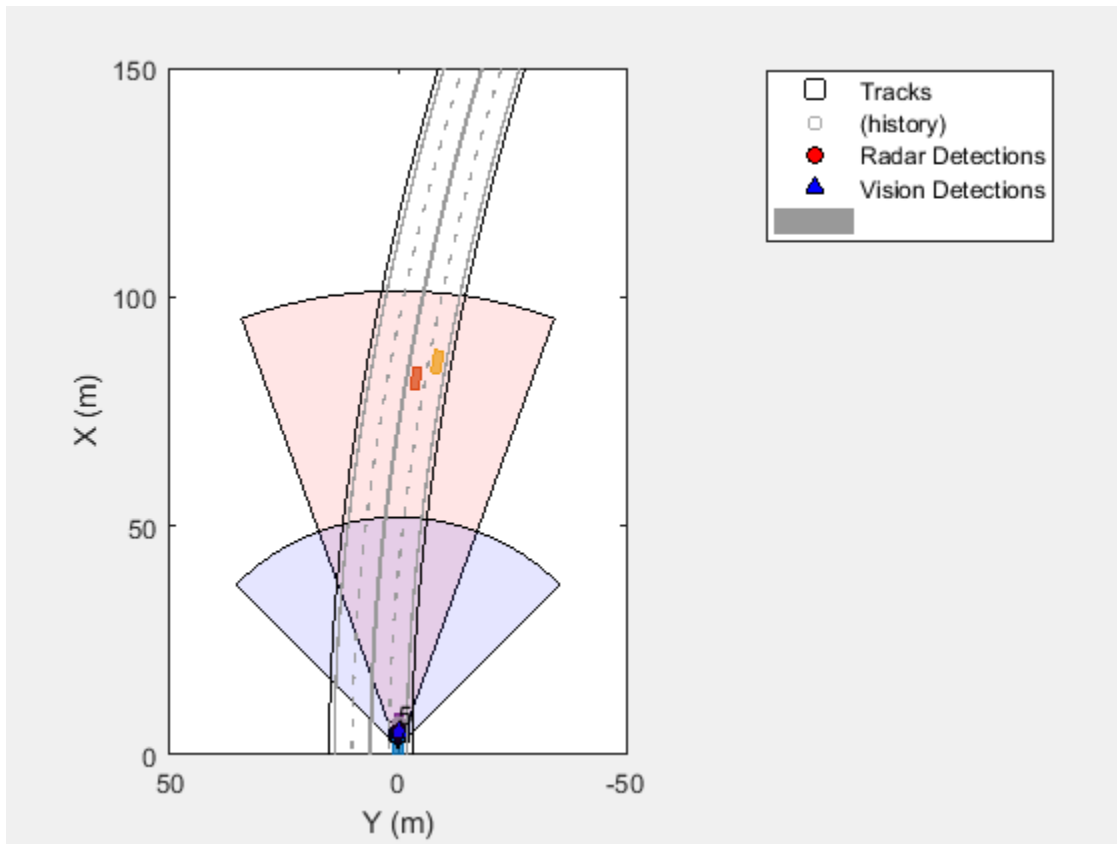
    % Cast detections to single precision
    detections = helperCastDetections(detections, dataType);

    % Feed detections to the tracking algorithm
    tracks = trackingAlgorithm(detections, time);

    % Find detectable targets for truth
    truths = helperFilterWithinCoverage(egoVehicle, radar, camera);

    % Calculate the GOSPA metric
    gospa(end+1,1) = gospaObj(tracks, truths); %#ok<SAGROW>

    % Visualize the results
    scope(scenario, egoVehicle, {radar;camera}, detections, tracks);
end
```



To conveniently to re-run this test bench during PIL simulations, you also wrap the test bench in a separate function, `helperJPDATrackerPILTestBench`. This function can be called with the following syntax:

```
gospa = helperJPDATrackerPILTestBench(scenarioName, trackingAlgorithmName, dataType); % No visual
gospa = helperJPDATrackerPILTestBench(scenarioName, trackingAlgorithmName, dataType, true); % En
```

Generate Code for PIL

In this section, you generate standalone C code for the tracking algorithm as a static library. You further verify the code by running PIL simulations on a STM32 Nucleo H743ZI2 target board. This target board has an ARM®-Cortex® M7 CPU with 1 MB of RAM and 2 MB of Flash memory. For more information regarding PIL simulations on this board, refer to the “Processor-in-the-Loop Verification of MATLAB Functions Using STMicroelectronics Nucleo Boards” (Simulink Coder Support Package for STMicroelectronics Nucleo Boards) example. Although this example discusses about PIL simulation on the Nucleo target hardware, this approach can be used on any supported hardware. See the “Embedded Coder Supported Hardware” (Embedded Coder) page for more information about supported hardware boards.

To generate code for the tracking algorithm, you must define the types of the inputs to the entry-point function. An easy way to define these inputs is by using the `codegen` (MATLAB Coder) argument. You use the detections captured during the MATLAB execution to define the input types for the entry point function. Note that the data type of the inputs cannot be changed after code generation. Therefore, if the embedded code is generated with single-precision measurements, the test bench must provide single-precision measurements as inputs to the tracking algorithm. As the number of detections change between each call to the tracking algorithm, you define the detection input type as

a variable-sized cell array with a maximum of 46 elements using the `coder.typeof` (MATLAB Coder) function. You also define the input type of time input using the correct data type as defined in the Setup the Test Bench on page 6-887 section.

```
sampleDetection = detections{1};
detectionsInput = coder.typeof({sampleDetection},[46 1],[1 0]);
timeInput = cast(0,dataType);
```

You define the code generation configuration for PIL verification by creating a `coder.EmbeddedCodeConfig` (MATLAB Coder) object. You define the `VerificationMode` as 'PIL' and specify certain hardware properties on the configuration. To profile the generated code on the target hardware, you also set the `CodeExecutionProfiling` property to `true`.

```
cfg = coder.config('lib','ecoder',true); % Creates a coder.EmbeddedCodeConfig object
cfg.VerificationMode = 'PIL'; % Enable PIL for verification
cfg.DynamicMemoryAllocation = 'off'; % Turn-off dynamic memory allocation
cfg.Toolchain = 'GNU Tools for ARM Embedded Processors'; % Specify toolchain
cfg.Hardware = coder.hardware('STM32 Nucleo H743ZI2'); % Specify hardware board
cfg.StackUsageMax = 512; % (Bytes) Limit stack usage
cfg.Hardware.PILCOMPort = 'COM3'; % Specify the port for connecting with hardware board
cfg.CodeExecutionProfiling = true; % Enable code execution profiling
```

Generate code using the `codegen` function. This function produces a MEX file named `trackingAlgorithm_pil` in the current working directory. This MEX file provides a wrapper to send inputs from MATLAB environment to the target hardware and collect outputs from the target hardware back to MATLAB.

```
codegen('trackingAlgorithm.m','-args',{detectionsInput,timeInput}),'-config',cfg);
### Connectivity configuration for function 'trackingAlgorithm': 'STM32 Microcontroller'
### COM port: COM3
### Baud rate: 115200
Code generation successful.
```

PIL Simulation and Results

In this section, you use the MEX generated from the previous section to run PIL simulations using the target hardware. To reuse the test bench created in the Setup the Test Bench on page 6-887 section, you specify the tracking algorithm name as `trackingAlgorithm_pil`

```
trackingAlgorithmName = 'trackingAlgorithm_pil';
gospaPIL = helperJPDATrackerPILTestBench(scenarioName, trackingAlgorithmName, dataType);
### Starting application: 'codegen\lib\trackingAlgorithm\pil\trackingAlgorithm.elf'
    To terminate execution: clear trackingAlgorithm_pil
### Downloading executable to the hardware on Drive: S:
H:\MATLAB\Examples\driving_fusion_nucleo-ex84625170\codegen\lib\trackingAlgorithm\pil\..\..\..\
1 File(s) copied
    Execution profiling data is available for viewing. Open Simulation Data Inspector.
    Execution profiling report available after termination.
```

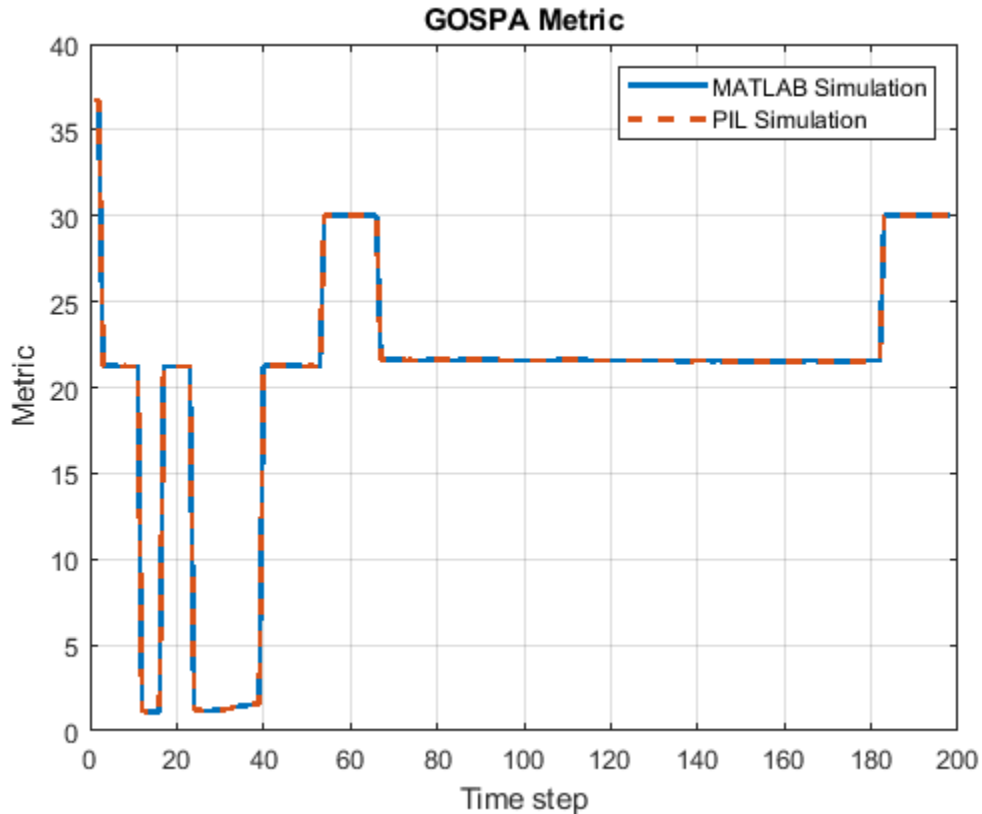
The plots below show the GOSPA metric captured during MATLAB run and during PIL simulation. Note that the GOSPA metrics captured during both runs are same, which assures that the generated code running on target hardware produces the same results as MATLAB.

```
figure;
plot(gospa,'LineWidth',2);
hold on;
```

```

plot(gospaPIL, 'LineWidth',2, 'LineStyle', '--');
legend('MATLAB Simulation', 'PIL Simulation');
title("GOSPA Metric");
xlabel('Time step');
ylabel('Metric');
grid on;

```



In addition to tracking performance, you also use the profiling results captured by the PIL simulation to check computational performance of the tracking algorithm on the target hardware. The plots below show the run-time performance of the tracking algorithm on the target hardware. Note that the tracker is able to run at a rate faster than 100 Hz, assuring the capability of real-time computation on this particular board.

```
clear(trackingAlgorithmName); % Results available after PIL ends
```

```
Execution profiling report: report(getCoderExecutionProfile('trackingAlgorithm'))
```

```
% Plot execution profile
```

```
executionProfile = getCoderExecutionProfile('trackingAlgorithm')
```

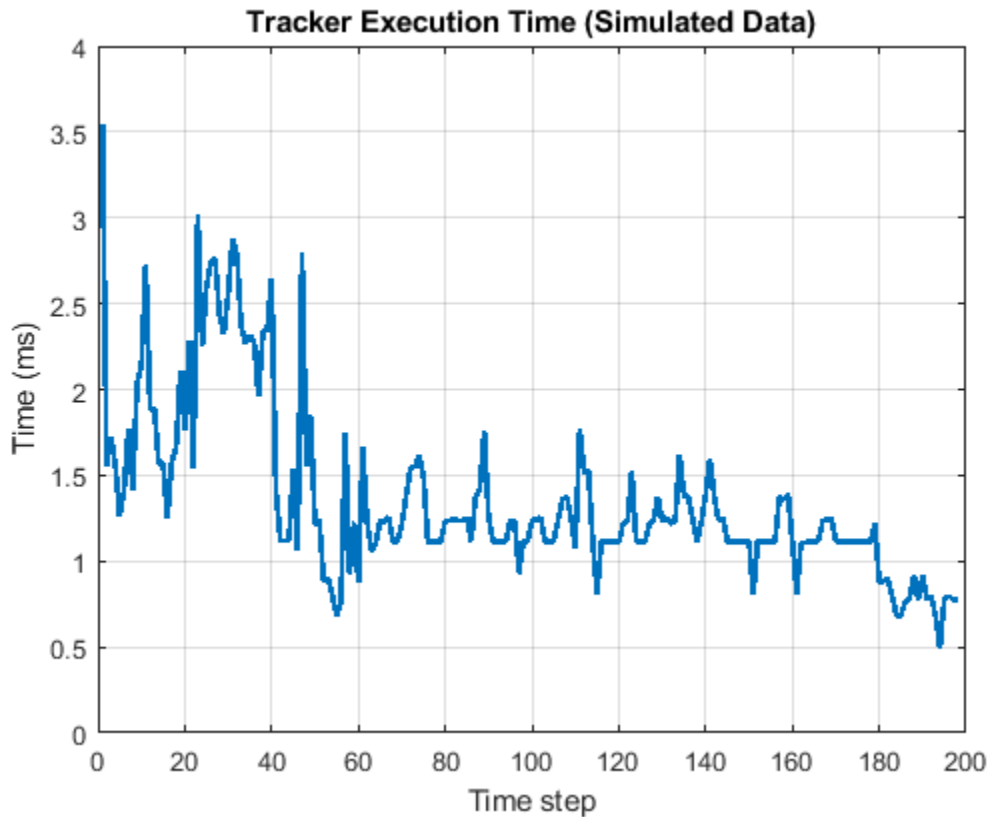
Code execution profiling data for trackingAlgorithm. To open a report, enter the command `report(executionProfile)`.

```
figure;
stepSection = executionProfile.Sections(2);
execTime = stepSection.ExecutionTimeInSeconds;
```

```

plot(1e3*execTime,'LineWidth',2);
title('Tracker Execution Time (Simulated Data)');
xlabel('Time step');
ylabel('Time (ms)');
grid on;

```



Real-Time Performance Verification on Recorded Data

In the previous sections, you verified the tracking and computational performance of the tracking algorithm on the Nucleo target hardware. The scenario simulation allows you to define a variety of situations and verify the performance of the tracker in such situations. However, it is also critical to verify the performance of the tracker on a real data set. This ensures that the tracking algorithm can bear the challenges and complexity of real-world situations.

In this section, you verify the computational performance of the tracker using recorded data from camera and radar on a highway scenario. The radar used in this recording is a multimode radar, which provides a wide coverage at mid-range and a narrow but high-resolution coverage at long range. In addition to providing detections from target objects, the radar also outputs detections from the road infrastructure, making the tracking algorithm susceptible to many false tracks. You filter out the infrastructure detections using the helper function, `helperFilterStaticDetections`. This helper function uses the recorded speed, yaw-rate of the ego vehicle, as well as doppler (range-rate) information from the radar to filter out detections from static objects in the environment.

```

videoFile = '05_highway_lanechange_25s.mp4';
sensorFile = '05_highway_lanechange_25s_sensor.mat';

```

```
% Load the data
```

```

recording = load(sensorFile);
numSteps = numel(recording.radar);

% Visualize the scenario using a camera recording
videoReader = VideoReader(videoFile);

% Initialize display
scope = HelperJPDATrackerPILDisplay('UseRecordedData',true);

% Timer at 20 Hz
time = cast(0,dataType);
timeStep = cast(0.05,dataType);

% Reinitialize tracker
clear(trackingAlgorithmName);

for currentStep = 1:numSteps
    % Update time
    time = time + timeStep;

    % Collect detections from recording
    [radarTotalDetections, visionDetections, laneData, imuData] = helperCollectDetectionsFromRecording(recording,currentStep);

    % Radar detections from the targets and clutter
    [radarDetections, staticDetections] = helperFilterStaticDetections(radarTotalDetections, imuData);

    % Concatenate detections
    detections = [radarDetections;visionDetections];

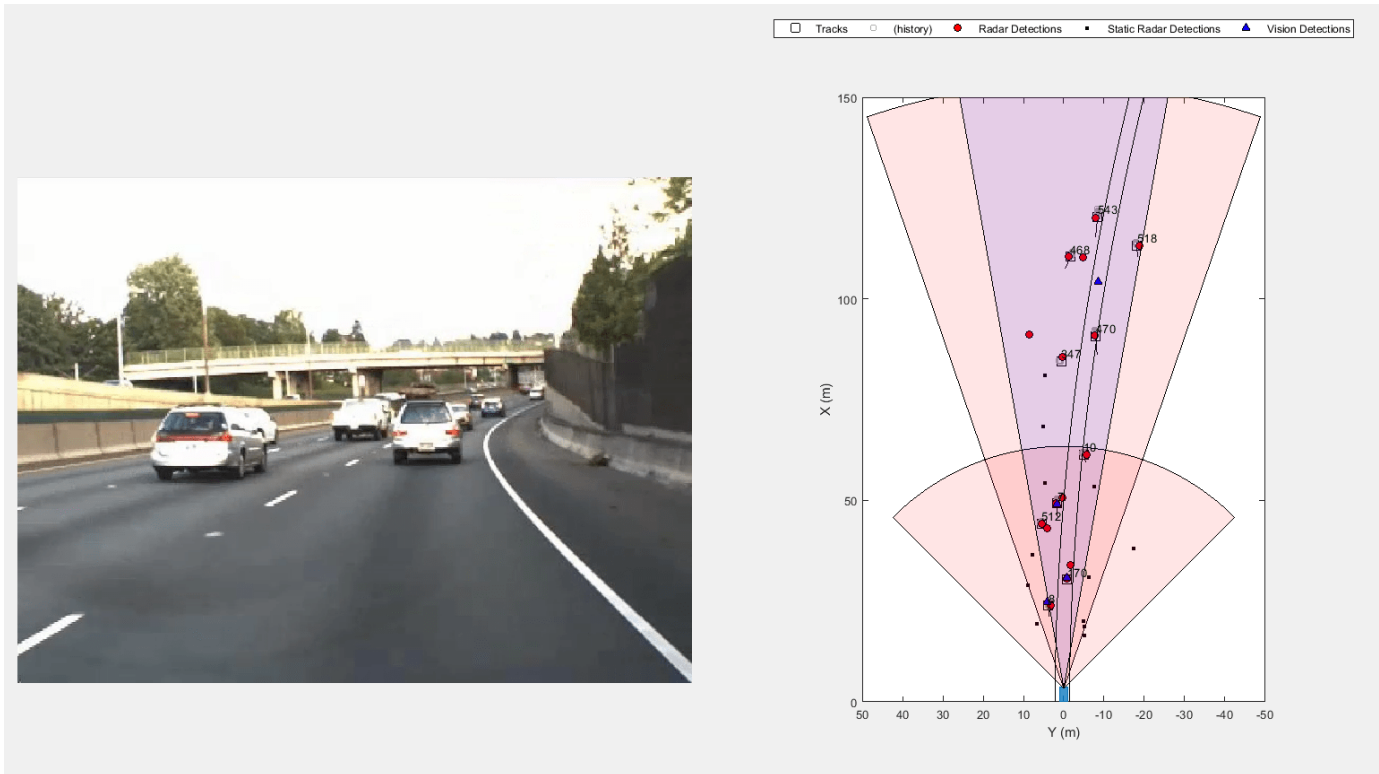
    detections = helperCastDetections(detections,dataType);

    % Run the tracker on hardware
    tracks = feval(trackingAlgorithmName,detections, time); %#ok<FVAL>

    % Visualize
    vidImage = readFrame(videoReader);
    scope(vidImage, laneData, detections, staticDetections, tracks);
end

### Connectivity configuration for function 'trackingAlgorithm': 'STM32 Microcontroller'
### COM port: COM3
### Baud rate: 115200
### Starting application: 'codegen\lib\trackingAlgorithm\pil\trackingAlgorithm.elf'
    To terminate execution: clear trackingAlgorithm_pil
### Downloading executable to the hardware on Drive: S:
H:\MATLAB\Examples\driving_fusion_nucleo-ex84625170\codegen\lib\trackingAlgorithm\pil\..\..\..\
1 File(s) copied
    Execution profiling data is available for viewing. Open Simulation Data Inspector.
    Execution profiling report available after termination.

```



```
clear(trackingAlgorithmName); % Results available after PIL ends
```

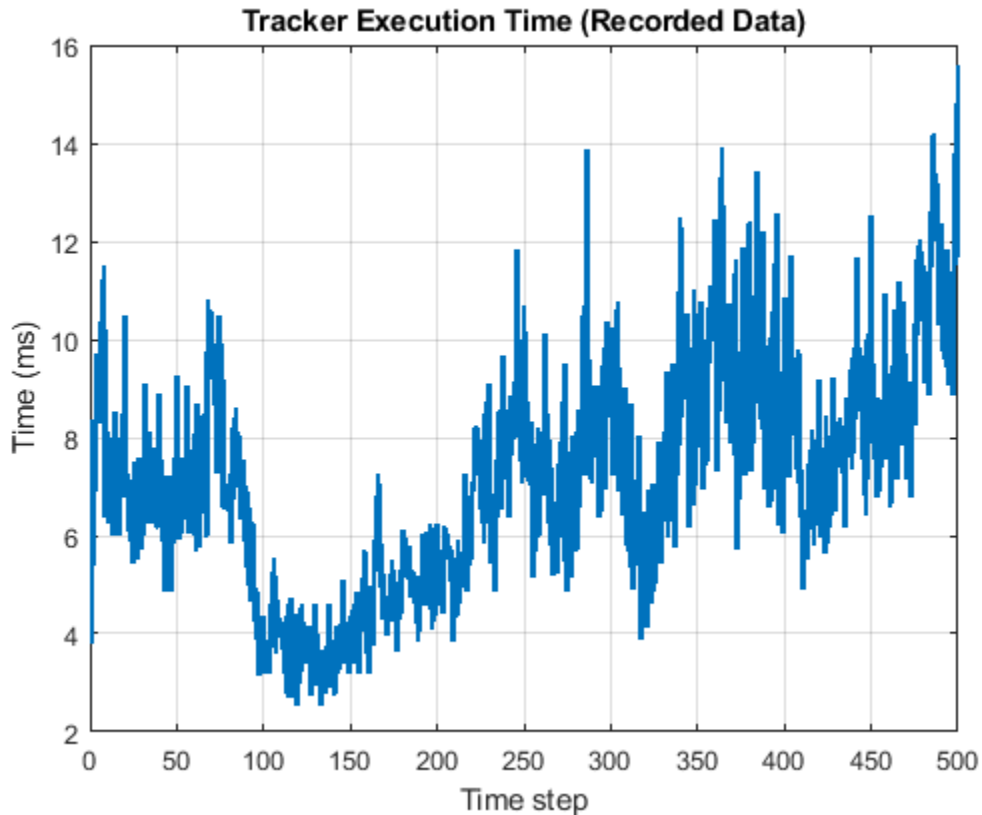
```
    Execution profiling report: report(getCoderExecutionProfile('trackingAlgorithm'))
```

```
% Plot execution profile
```

```
executionProfile = getCoderExecutionProfile('trackingAlgorithm')
```

Code execution profiling data for trackingAlgorithm. To open a report, enter the command `report(executionProfile)`.

```
figure;
stepSection = executionProfile.Sections(2);
execTime = stepSection.ExecutionTimeInSeconds;
plot(1e3*execTime,'LineWidth',2);
title('Tracker Execution Time (Recorded Data)');
xlabel('Time step');
ylabel('Time (ms)');
grid on;
```

Note that the tracker is able to track the targets within the field of view of the sensors and is able to run faster than 60 Hz on this particular hardware board. This verifies the real-time tracking capability of the algorithm in denser traffic scenarios captured in the recording.

Explore Other Scenarios

It is important to assess the performance of the tracking algorithm under different scenarios. You can use the simulation environment in this example to explore other scenarios, compatible with the test bench defined by `helperJPDATrackerPILTestBench`. Here are five compatible scenarios that you can use by specifying the `scenarioName` input as one of the following:

- 'scenario_FVSF_01_Curve_FourVehicles'
- 'scenario_FVSF_02_Straight_FourVehicles'
- 'scenario_FVSF_03_Curve_SixVehicle'
- 'scenario_FVSF_04_Straight_FourVehicles'
- 'scenario_FVSF_05_Straight_TwoVehicles'

Summary

In this example, you learned how to generate code from a tracking algorithm for PIL simulations. You verified the generated code on a STM32 Nucleo board using simulated data as well as recorded data from highway driving scenarios. You further assessed the computational performance and real-time capability of the tracking algorithm in such scenarios on the chosen target hardware.

Track Objects with Wrapping Azimuth Angles and Ambiguous Range and Range Rate Measurements

This example shows how to track objects when measurements wrap in angle, range, or range rate.

Track with Angular Measurement Wrapping

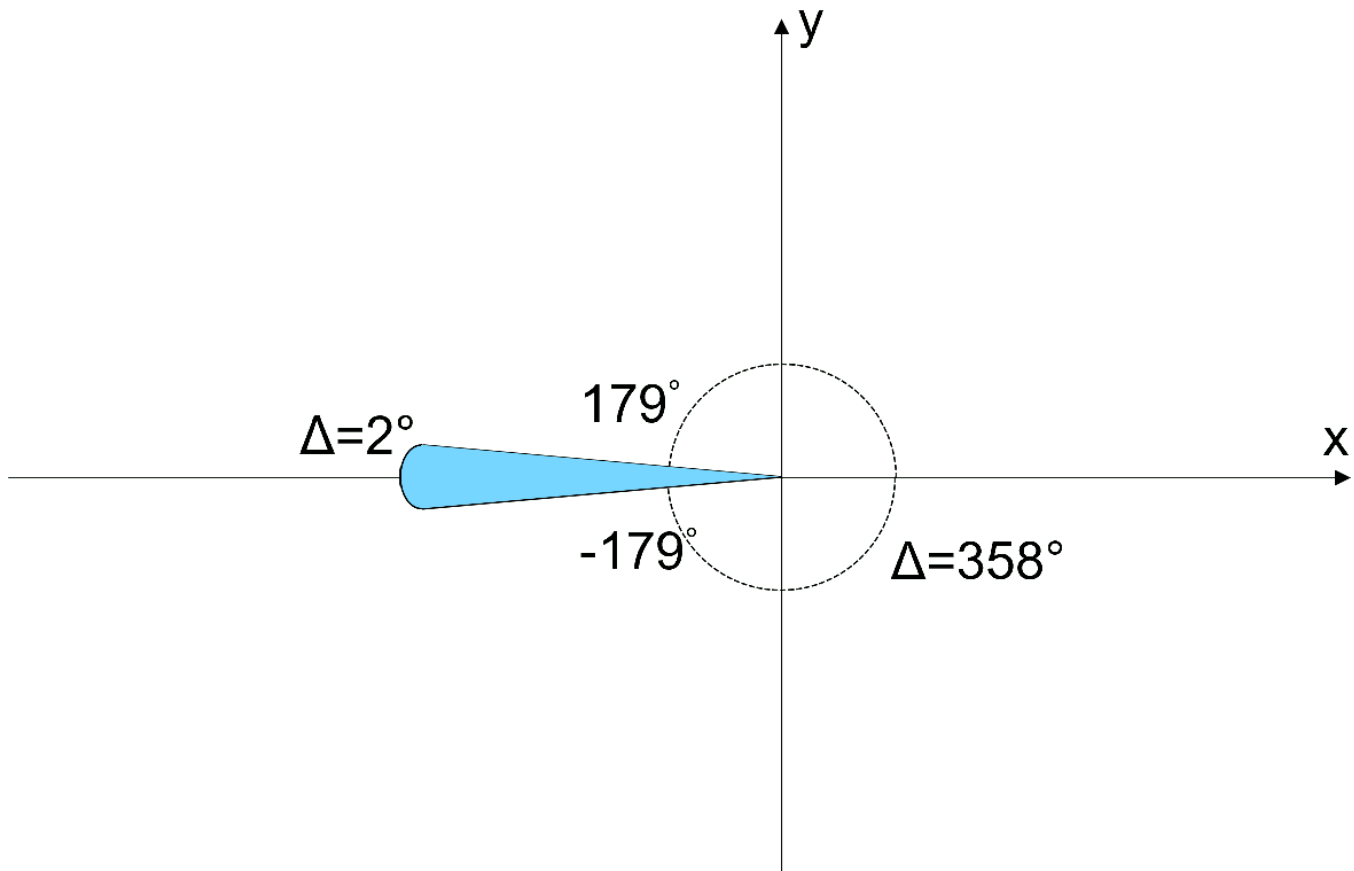
When detecting an object, most sensors use a spherical coordinate frame and report the position of the object in azimuth, elevation, and range of the object relative to the sensor. The sensor reports the angular measurements within certain bounds. For example, azimuth angles are often reported within a range of -180 degrees to 180 degrees or 0 degrees to 360 degrees.

While the sensor reports the measurement in spherical coordinates, tracking is usually done in a rectangular frame. To be able to handle the nonlinear transformations required to convert the state from a rectangular frame to a measurement in a spherical frame, a nonlinear filter is required. Some examples of nonlinear filters are the extended Kalman filter and the unscented Kalman filter. For a simple example, consider a filter that maintains a two-dimensional state with positions defined as x and y in the rectangular frame and a sensor that measures position in the two-dimensional plane as azimuth, ϕ , and range, r . The conversion from the rectangular state to the measurement, assuming the sensor is located at the origin of the x - y plane is:

$$\phi = \tan^{-1} \frac{y}{x}$$

$$r = \sqrt{x^2 + y^2}$$

For azimuth angles reported in the range of -180 degrees to 180 degrees, if the azimuth angle of the object is close to -180 degrees, the sensor may report an azimuth measurement that wraps around to 180 degrees. These measurements are often referred to as *wrapping* or *circular*.



To see that, consider the following results:

```
% Object is just above the negative x axis
disp(atan2d(1e-5, -100));
```

```
180.0000
```

```
% Object is just below the negative x axis
disp(atan2d(-1e-5, -100));
```

```
-180.0000
```

In the correction step, the Kalman filter uses the residual between the measurement reported by the sensor, z , and the measurement expected from the filter state, $h(x)$, to correct the state estimate:

$$y = z - z_{exp} = z - h(x)$$

If an object is located near the wrapping boundary, even if the measurement is relatively accurate, the values of z and $h(x)$ may be on different sides of the bound, leading to a large residual value. This, in turn, causes the filter to correct the state by a large amount, and may lead the filter to diverge from the accurate state.

The following code uses the `azrmeas` function, a supporting function attached at the end of this example, to demonstrate filter divergence due to wrapping measurements.

```

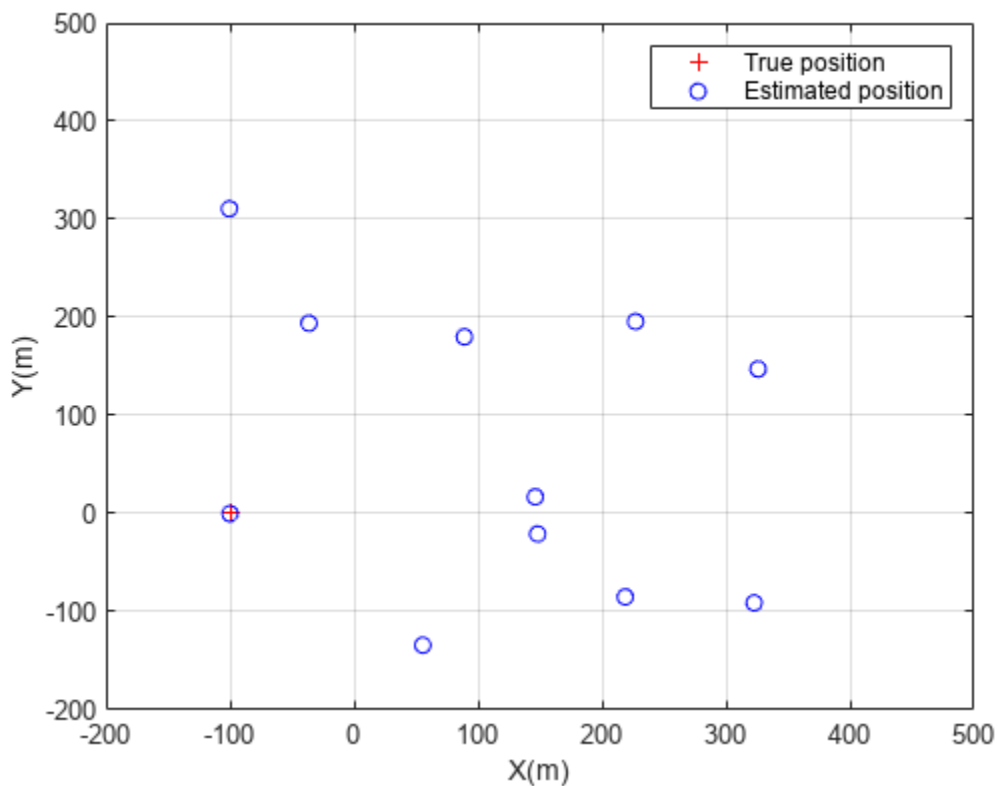
% Initial position
x = -100;
y = 0;
ekf = trackingEKF(StateTransitionFcn = @constvel, MeasurementFcn = @azrmeas, State = [x;0;y;0]);

% Create a figure
figure(1)
plot(x,y,"r+");
hold on;
grid on;

% Plot the filter estimate on the figure
plot(ekf.State(1), ekf.State(3), "bo");

runFilter(ekf, x, y);

```



From the figure above, the estimated positions diverge significantly from the accurate object position at $x = -100$ and $y = 0$.

The `trackingEKF` object provides an ability to handle measurement wrapping by setting the `HasMeasurementWrapping` property to true. With this specification, the filter uses the second output from the measurement function to get the measurement bounds and wrap the residual within these bounds as proposed in [1]. To support the filter ability to handle measurement wrapping, the function `azrmeas` has a second output that returns the measurement bounds.

```

% Initial position
x = -100;

```

```

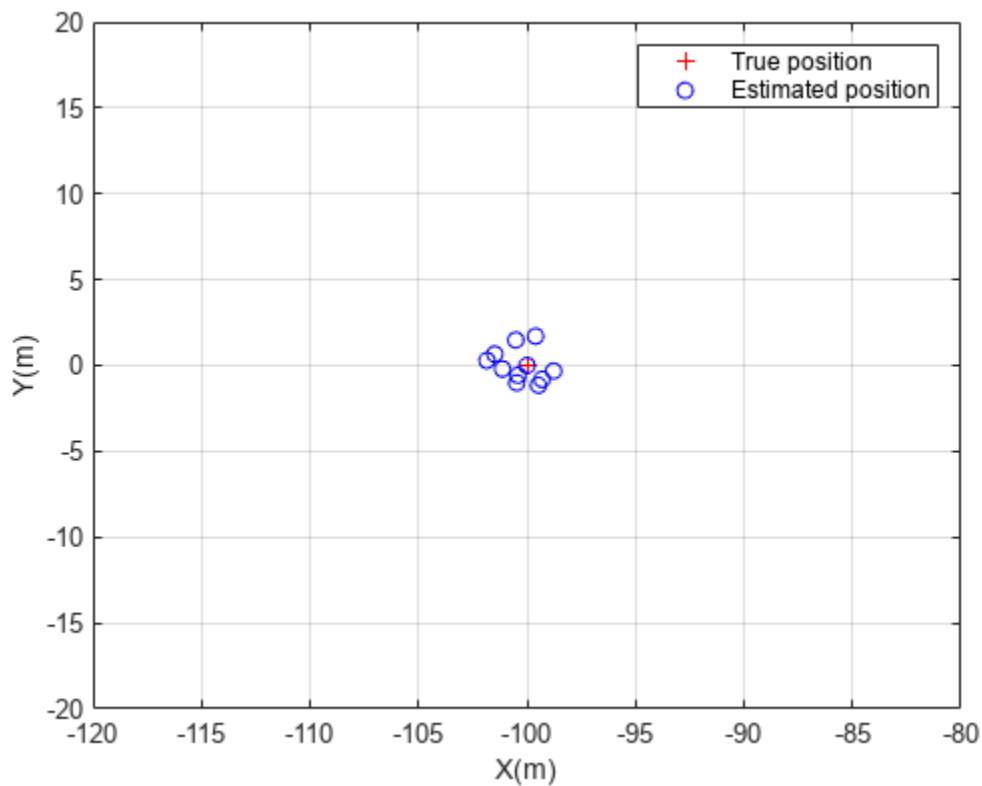
y = 0;
ekf = trackingEKF(StateTransitionFcn = @constvel, MeasurementFcn = @azrmeas, State = [x;0;y;0], I

% Create a figure
figure(2)
plot(x,y,"r+");
hold on;
grid on;

% Plot the filter estimate on the figure
plot(ekf.State(1), ekf.State(3), "bo");

runFilter(ekf, x, y);

```



You use the supporting function, `azrmeas`, to obtain the foregoing results. You can also use the built-in measurement and filter initialization functions to handle measurement wrapping and get the same results. The following code block shows the equivalent workflow to use built-in functions.

```

% Initialize the filter with a detection in spherical coordinates
mp = struct(Frame = "Spherical", HasElevation = false, HasVelocity = false);
detection = objectDetection(0, [-180;100], MeasurementParameters = mp);
ekf = initcvkf(detection);

% Create a figure
figure(3)
plot(x,y,"r+");
hold on;

```

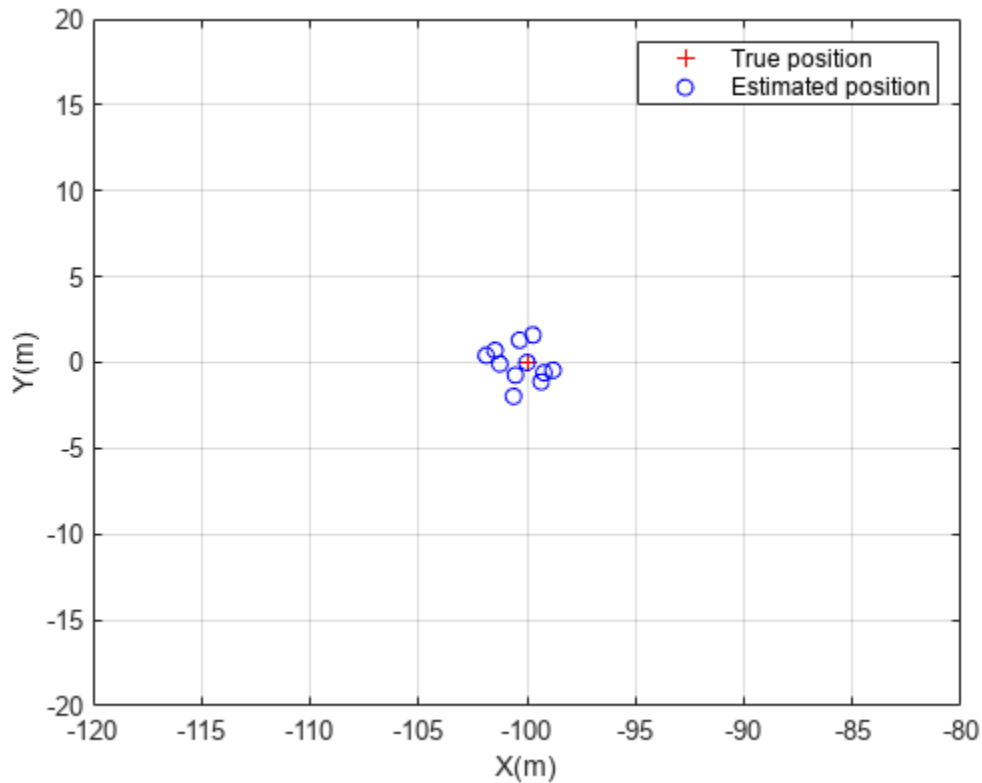
```

grid on;

% Plot the filter estimate on the figure
plot(ekf.State(1), ekf.State(3), "bo");

runFilter(ekf, x, y, mp);

```



Radar Range and Range Rate Ambiguities

So far, the measurement wrapping was due to the way format that the sensor reports angle but had nothing to do with how the sensor processes its measurement. This section focuses on a different source of measurement wrapping in radars: the process the radar measures range and range rate of an object.

To measure range, the radar emits a pulse of radio frequency waves, which propagates at the speed of light, c . After the pulse is reflected from the object, it travels back to the radar. The radar measures the time difference between the arrival time and the pulse emission time, τ . For a monostatic radar, in which the radar pulse travels to the object and back to the radar, the distance that the pulse traveled is twice the range to the object.

$$2r = c\tau$$

The radar repeats the pulses at a certain frequency, called the pulse repetition frequency (PRF). To be able to unambiguously determine which pulse corresponds to the return from the object, the return must arrive at the radar before the emission of the next pulse. The maximum distance that the radar can unambiguously determine using the returned pulse is called the maximum unambiguous range, defined as:

$$r_{max} = \frac{cT}{2} = \frac{c}{2PRF}$$

From the expression, to be able to unambiguously measure larger ranges, the radar needs to have a low PRF.

In addition, to measure the object velocity along the range axis from the radar, known as range rate, the radar uses the Doppler effect. The Doppler effect shifts the frequency observed by the radar when the target moves in a direction along the vector to the radar. This shift must be within the radar pulse repetition frequency. Therefore, the range rate measurement is unambiguous in the interval:

$$-\frac{cPRF}{4f} \leq rr \leq \frac{cPRF}{4f}$$

where, f is the frequency of the radio waves. From the expression, to be able to unambiguously measure high range rates, the PRF must be high.

Obviously, there is a contradiction between the PRF needed for long range measurements, which must be low, and the PRF needed for high velocity measurements, which must be high. In many cases, a compromise is made by choosing a medium PRF, which is selected based on the required maximum range and the maximum expected object speed. However, if the object is either located farther from the radar or moves faster than expected, the radar reports an ambiguous range or an ambiguous range rate according to the following:

$$r_{reported} = \text{mod}(r_{true}, r_{max})$$

$$rr_{reported} = \text{mod}(rr_{true}, rr_{max})$$

For example, if a radar has a maximum unambiguous range of 100 km, but the target is located at a range of 110 km from the radar, the radar reports a range of 10 km.

A few techniques can be used to improve the maximum unambiguous range and the maximum unambiguous range rate using signal processing techniques. For example, radar engineers sometimes use the `crt` (Phased Array System Toolbox) if the radar uses multiple PRFs.

To summarize, while angle wrapping is caused by the bounding of angle measurements to a certain range, range and range rate wrapping are caused by the radar inherent mode of operation. Even when signal processing techniques are used to improve the range and range rate ambiguities, some ambiguities still happen and must be handled by the estimation filter and the multi-object tracking algorithms.

Estimate Object Position with Range or Range Rate Ambiguities

In this section, you use an extended Kalman filter to estimate the position of an object that is moving away from the radar and crosses over the maximum unambiguous range. You must first define a measurement function that reports the range bounds, in this case the function `azrmeas100`, attached as a supporting function at the end of this example. This function is similar to the function `azrmeas` used in the previous sections except that the maximum range is bounded to 100.

```
% Initial position
r = 70; % Range from radar, in m
rr = 10; % Range rate, the object moves away from the radar at 10 m/s
az = 10; % The azimuth angle to the object is 10 degrees
[x,y] = pol2cart(deg2rad(az),r);
ekf = trackingEKF(StateTransitionFcn = @constvel, MeasurementFcn = @azrmeas100, State = [x;0;y;0];
```

```
% Create a figure
figure(4)
th = plot(x,y,"r+");
hold on;
axis equal
axis([0 220 -120 120])
xlabel("X(m)");
ylabel("Y(m)");

% Plot the reported position
rh = plot(x,y,"kx");

% Plot the estimated position
eh = plot(ekf.State(1), ekf.State(3), "bo");
grid on;

% Add the maximum unambiguous range circle
rectangle(Position = [-100 -100 200 200], Curvature = [1 1], LineStyle = "--", EdgeColor = "r");
text(75, 80, "Maximum" + newline + "Unambiguous" + newline + "Range");

hold off
legend("True position", "Reported position", "Estimated position", Location = "southeast");

% Use the same seed
rng(0,"twister");

dt = 0.2;
for time = dt:dt:8
    predict(ekf,1);

    % Move the object
    r = r + rr * dt;
    [x,y] = pol2cart(deg2rad(az),r);
    set(th, XData = x, YData = y)

    % Measurement with some noise
    z = [az;r] + 0.1 * randn(2,1);

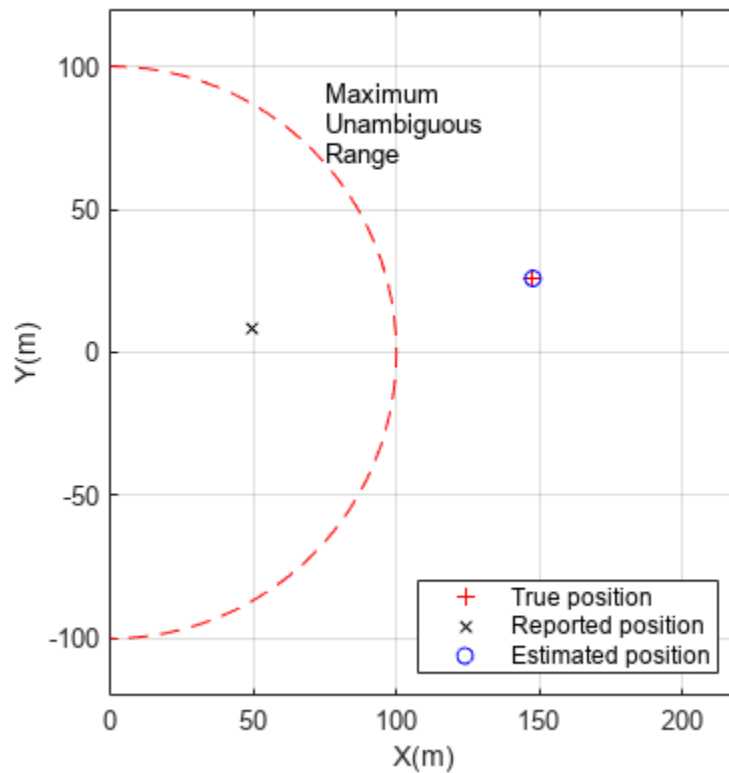
    % Wrap the azimuth measurement in the range -180 to 180 and the radius
    % in the range 0 to 100
    z(1) = mod(z(1)+180, 360) - 180;
    z(2) = mod(z(2),100);

    % Plot the reported position
    [x,y] = pol2cart(deg2rad(z(1)),z(2));
    set(rh, XData = x, YData = y);

    % Correct the filter with the measurement
    correct(ekf,z);

    % Plot the filter estimate on the figure
    set(eh, XData = ekf.State(1), YData = ekf.State(3));

drawnow
end
```

You can observe that the filter was able to estimate the position of the object even when the object exceeds the maximum unambiguous range.

Multi-Object Tracking in the Presence of Angular and Range Ambiguities

When tracking multiple objects, the tracker has to assign the measurement to the correct object even if the reported measurement was wrapped, in addition to estimating the dynamics of each object. This section shows how to use multiobject trackers to track three objects, where each object causes the radar to report measurements that wrap in range, range rate, or azimuth.

The following code creates a radar that covers 360 degrees around the radar without scanning. The radar has a maximum unambiguous range of 100 meters and has a maximum unambiguous range rate of 12 meters per second. You mount the radar on a platform.

```
scenario = trackingScenario(UpdateRate = 0);
radar = fusionRadarSensor( ...
    SensorIndex = 1, ...
    ScanMode = "No scanning", ...
    UpdateRate = 10, ...
    FieldOfView = [360;10], ...
    HasElevation = false, ...
    HasRangeRate = true, ...
    HasRangeAmbiguities = true, ...
    HasRangeRateAmbiguities = true, ...
    AzimuthResolution = 0.1, ...
    ReferenceRange = 100, ...
    RangeResolution = 1, ...
```

```

    RangeRateResolution = 1, ...
    HasFalseAlarms = false, ...
    HasINS = true, ...
    DetectionCoordinates = "Sensor Spherical",...
    MaxUnambiguousRange = 100,...
    MaxUnambiguousRadialSpeed = 12);
platform(scenario, sensors = radar);

```

You define the trajectories of the three objects. The first object begins its motion about half the distance to the radar maximum unambiguous range, continues moving away, and eventually crosses the maximum unambiguous range. The second object begins its motion with a relatively low speed but then accelerates to a range rate that is above the maximum unambiguous range rate of the radar. The third object begins its motion just above the negative x axis and moves down, crossing the x axis where the azimuth measurement wraps. In practice, some objects may give rise to measurements with all three ambiguities.

```

traj1 = waypointTrajectory([50 10 0; 130 20 0], [0;10]); % Across the range ambiguity
platform(scenario, Trajectory = traj1);
traj2 = waypointTrajectory([60 -20 0; 40 0 0; -60 60 0], [0;3;10]); % Faster than the range rate
platform(scenario, Trajectory = traj2);
traj3 = waypointTrajectory([-70 10 0; -70 -10 0], [0;10]); % Across the azimuth wrapping
platform(scenario, Trajectory = traj3);

```

You visualize the scenario using the `createVisualization` supporting function.

```
[plp, detp, trkp] = createVisualization(scenario);
```

You use the `trackerGNN` object with a constant velocity extended Kalman filter to track the objects. The built-in `initcvekf` initialization function uses the constant velocity measurement function, `cvmeas`, which supports azimuth wrapping, but does not support range or range rate wrapping, because they are radar specific. Therefore, you modify the filter initialization function and add a measurement function that supports range and range rate wrapping. You can see the details of the `initBoundedCVEKF` and `boundedCVMeas` functions in the Supporting Functions section.

```

tracker = trackerGNN(FilterInitializationFcn = @initBoundedCVEKF);
positionSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 1 0];

```

Run the simulation and track the objects.

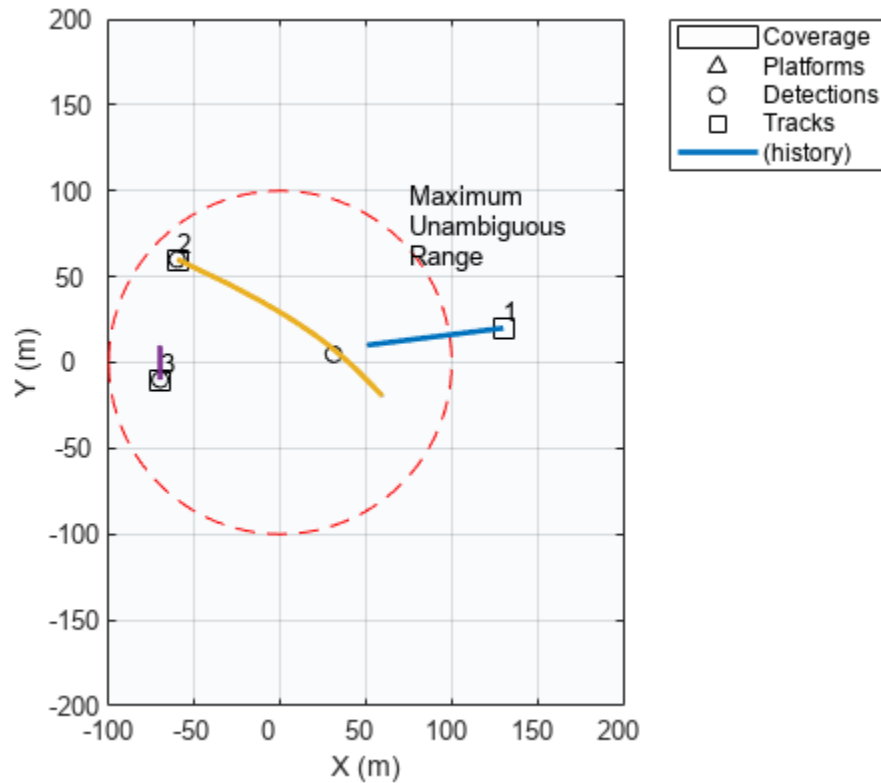
```

while advance(scenario)
    % Simulation time
    time = scenario.SimulationTime;

    % Collect all the detections from the scenario and plot them
    detections = detect(scenario);
    detPositions = getDetectionPositions(detections);
    plotDetection(detp, detPositions);

    % Update the tracker and plot the tracks
    tracks = tracker(detections, time);
    [pos,cov] = getTrackPositions(tracks,positionSelector);
    trIDs = string([tracks.TrackID]);
    plotTrack(trkp, pos, cov, trIDs);
end

```



You observe that the tracker is able to track all three objects even though the measurements reported by the radar are wrapping in azimuth, range, and range rate.

Summary

This example shows how to track with wrapped measurements. Sensors often report angular measurements that wrap due to the angle circular nature, which is common to all sensors. In addition, due to the way radars operate, their measurements can have ambiguity in range and in range rate, which is specific to each radar. You saw how to define nonlinear filters that can handle wrapping measurements. The nonlinear filters require a measurement function that provides the measurement bounds. For angular measurement wrapping, the built-in measurement functions, such as `cameas`, `cvmeas`, `ctmeas`, `singermeas`, and `cvmeasmsc` provide the default azimuth and elevation bounds. To handle measurement wrapping in range and range rate, you must define the radar specific bounds, as shown in this example. Finally, this example showed how the multi-object trackers, when configured with tracking filters and measurement functions that report the measurement bounds, can track objects even in the presence of wrapping measurements.

References

[1] Crouse David Frederic, "Cubature/unscented/sigma point Kalman filtering with angular measurement models," *2015 18th International Conference on Information Fusion (Fusion)*, Washington DC, 2015, pp. 1550-1557.

Supporting Functions

`azrmeas`

This function provides the azimuth and range measurement without bounds.

```
function [z, bounds] = azrmeas(state)
x = state(1);
y = state(3);
az = atan2d(y,x);
r = sqrt(x^2 + y^2);
z = [az;r];
bounds = [-180 180; -inf inf];
end
```

runFilter

The function runs the filter for wrapped angular measurement cases.

```
function runFilter(filter, x, y, mp)
% Use a constant random number generator seed
rng(0, "v4");

for step = 1:10
    predict(filter,1);

    % Measurement with some noise
    z = azrmeas([x;0;y;0]) + randn(2,1);

    % Wrap the azimuth measurement in the range -180 to 180
    z(1) = mod(z(1)+180, 360) - 180;

    % Correct the filter with the measurement
    if nargin < 4
        correct(filter,z);
    else
        correct(filter, z, mp);
    end

    % Plot the filter estimate on the figure
    plot(filter.State(1), filter.State(3), "bo");
end
hold off
legend("True position", "Estimated position")
xlabel("X(m)");
ylabel("Y(m)");

if filter.HasMeasurementWrapping
    axis([-120 -80 -20 20]);
else
    axis([-200 500 -200 500]);
end
end
snapnow
end
```

azrmeas100

This function provides the azimuth and range measurements. Range is bounded between 0 and 100.

```
function [z, bounds] = azrmeas100(state)
x = state(1);
y = state(3);
```

```

az = atan2d(y,x);
r = mod(sqrt(x^2 + y^2), 100);
z = [az;r];
bounds = [-180 180; 0 100];
end

```

boundedCVMeas

The function is similar to `cvmeas` with range measurement bounded between 0 and 100 and range rate bounded between -12 and 12.

```

function [z, bounds] = boundedCVMeas(state, varargin)
% Use the regular cvmeas to get the measurement and bounds
mp = varargin{1};
[z, bounds] = cvmeas(state, mp);

% Add the range ambiguity and range rate ambiguity to the bounds
if mp(1).HasRange
    rangeIndex = mp(1).HasAzimuth + mp(1).HasElevation + mp(1).HasRange;
    bounds(rangeIndex, :) = [0 100];
    if mp(1).HasVelocity % Range rate can only be reported if range is reported
        bounds(rangeIndex + 1, :) = [-12 12];
    end
end
end

```

initBoundedCVEKF

Filter initialization function for a constant velocity extended Kalman filter that uses the `boundedCVMeas` measurement function.

```

function ekf = initBoundedCVEKF(detection)
% Use the regular initcvekf to get the filter
ekf = initcvekf(detection);
ekf.MeasurementFcn = @boundedCVMeas;
end

```

createVisualization

A helper function to create the visualization used in this example. The function returns the handles to the platform plotter, detection plotter, and track plotter.

```

function [plp, detp, trkp] = createVisualization(scenario)
tp = theaterPlot(Xlimits = [-100 200], Ylimits = [-200 200], Zlimits = [-50 50]);
cvp = coveragePlotter(tp, DisplayName = "Coverage", Tag = "Radar", Alpha = [0.01 0.1]);
plotCoverage(cvp, coverageConfig(scenario));
plp = platformPlotter(tp, DisplayName = "Platforms", Tag = "Platforms");
trajp = trajectoryPlotter(tp);
positions = cell(numel(scenario.Platforms)-1,1);
for i = 2:numel(scenario.Platforms)
    traj = scenario.Platforms{i}.Trajectory;
    ts = (traj.TimeOfArrival(1):(traj.TimeOfArrival(end)-traj.TimeOfArrival(1))/100:traj.TimeOfArrival(end));
    positions{i-1} = lookupPose(traj,ts);
end
plotTrajectory(trajp,positions);
detp = detectionPlotter(tp, DisplayName = "Detections", Tag = "Detections");
trkp = trackPlotter(tp, DisplayName = "Tracks", Tag = "Tracks", ConnectHistory = "on", ColorizeH

```

```
% Add the maximum unambiguous range circle
rectangle(Position = [-100 -100 200 200], Curvature = [1 1], LineStyle = "--", EdgeColor = "r");
text(75, 80, "Maximum" + newline + "Unambiguous" + newline + "Range");
grid on
end
```

getDetectionPositions

A helper function to return the detection positions for measurements in spherical coordinates relative to a sensor at the origin.

```
function detPositions = getDetectionPositions(detections)
numDets = numel(detections);
detPositions = zeros(numDets,3);
for i = 1:numDets
    mp = detections{i}.MeasurementParameters(1);
    if mp.HasAzimuth
        az = deg2rad(detections{i}.Measurement(1));
    else
        az = 0;
    end
    if mp.HasElevation
        el = deg2rad(detections{i}.Measurement(1 + mp.HasAzimuth));
    else
        el = 0;
    end
    if mp.HasRange
        r = detections{i}.Measurement(1 + mp.HasAzimuth + mp.HasElevation);
    else
        r = 0;
    end
    [x, y, z] = sph2cart(az, el, r);
    detPositions(i,:) = [x, y, z];
end
end
```

Export trackingArchitecture to Simulink

This example shows how to define a trackingArchitecture object and export it to Simulink.

Introduction

In this example, you create an architecture that includes multiple detection-level multi-object trackers and a track fuser. You create an architecture using the trackingArchitecture object in MATLAB and, export it to a Subsystem (Simulink) in a Simulink model using the exportToSimulink object function of the trackingArchitecture object. You connect the architecture to a tracking scenario, execute the tracking architecture, and evaluate the tracking performance using the trackOSPAMetric block in Simulink.

Create architecture and export it to Simulink

You create a tracking architecture that has two trackers and a track-to-track fuser. The first tracker accepts detections from a single sensor. The second tracker accepts detections from two sensors. The architecture also has a track fuser to fuse tracks from both trackers and output the fused tracks. You implement the first and second trackers using the trackerGNN and trackerJPDA objects respectively. You implement the track-to-track fuser using the trackFuser object.

```
% Define tracking architecture.
ta = trackingArchitecture(ArchitectureName="TrackingSystem");

% Create a sensor-level tracker.
sensorTracker = trackerGNN(TrackerIndex=1, ...
    FilterInitializationFcn=@initcvekf, ...
    AssignmentThreshold=50, ...
    ConfirmationThreshold=[2 3], ...
    DeletionThreshold=[5,5]);
addTracker(ta,sensorTracker,SensorIndices=1,ToOutput=false,Name="Sensor Tracker");

% Create another tracker that accepts detections from two sensors.
tracker = trackerJPDA(TrackerIndex=2, ...
    AssignmentThreshold=50, ...
    ConfirmationThreshold=[2 3], ...
    TrackLogic="History");
addTracker(ta,tracker,SensorIndices=[2,3],ToOutput=false,Name="Tracker");

% Define the track fuser.
fuser = trackFuser(FuserIndex=3, ... % Identify fuser.
    MaxNumSources=2, ...
    StateFusion="Cross"); % Fuser accepts tracks from 2 sources.
addTrackFuser(ta,fuser,Name="Track Fuser");
```

Display the summary of the architecture.

```
summary(ta)
```

```
ans=3x4 table
```

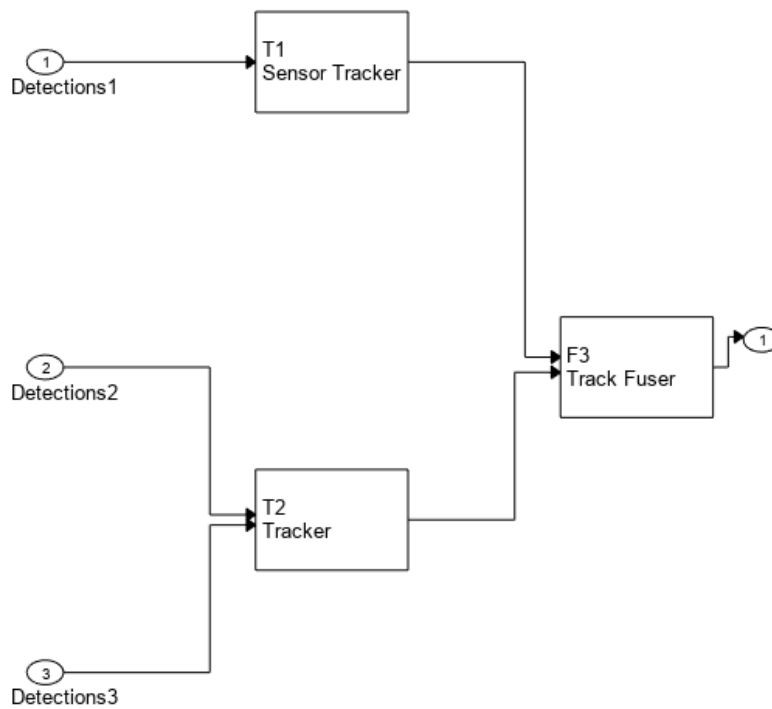
System	ArchitectureInputs	FuserInputs	ArchitectureOutput
{'T1:Sensor Tracker'}	{'1' }	{'Not applicable'}	{0x0 double}
{'T2:Tracker' }	{'2 3' }	{'Not applicable'}	{0x0 double}

```
{'F3:Track Fuser' } {0x0 char} {'1 2' } {[ 1]}
```

Visualize the architecture in a MATLAB figure.

```
show(ta);  
f =(gcf);  
f.Position = [680 153 915 825];
```

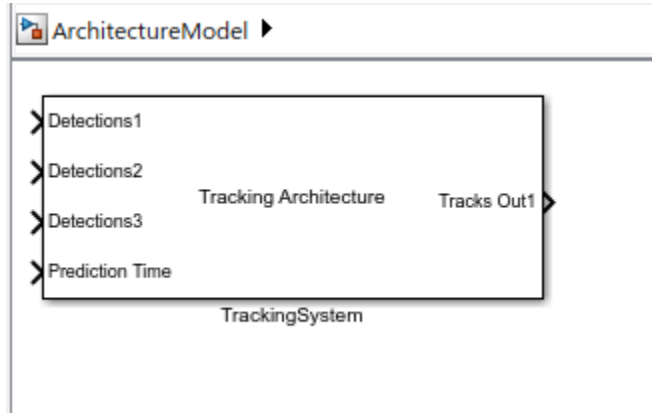
Tracking Architecture: TrackingSystem



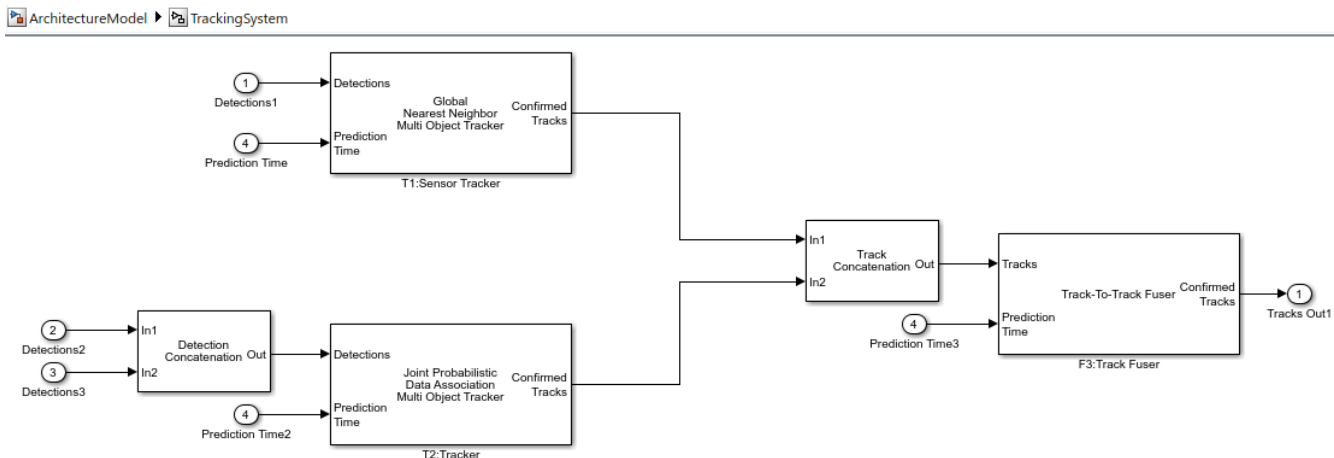
Export the architecture to Simulink.

```
model = exportToSimulink(ta);
```


The `exportToSimulink` object function exports `trackingArchitecture` as a Subsystem (Simulink) in Simulink. The subsystem has input ports for each architecture input and prediction time, as well as an output port for the architecture output.

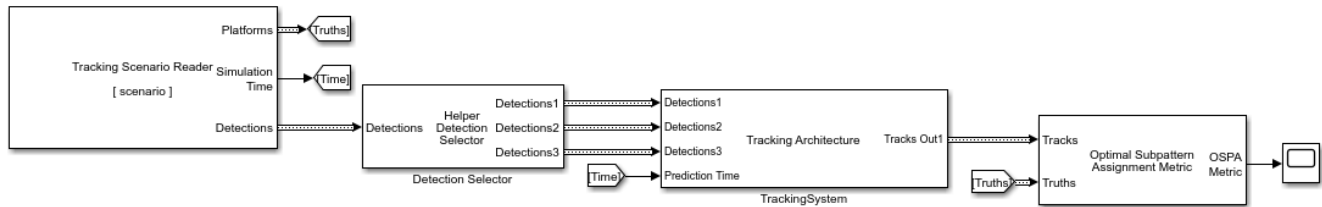


You can open the exported subsystem by double-clicking it and visualize the contained blocks. For each tracker or track fuser in the architecture, the subsystem contains an equivalent block. The block parameters are also configured equivalently as the properties of the tracker or track fuser object. Notice that the subsystem contains two additional blocks: the Detection Concatenation block and the Track Concatenation block. A Detection Concatenation block is required when a tracker has more than one input source. Similarly, a Track Concatenation block is required when a track fuser has more than one input source. Once exported, the model becomes independent from the `trackingArchitecture` object.



You can connect inputs to the tracking architecture subsystem and simulate the model to run the architecture. You use the `helperSetupArchitectureModel` function to connect the inputs to the architecture and configure the rest of the model.

```
helperSetupArchitectureModel(ta,model)
```

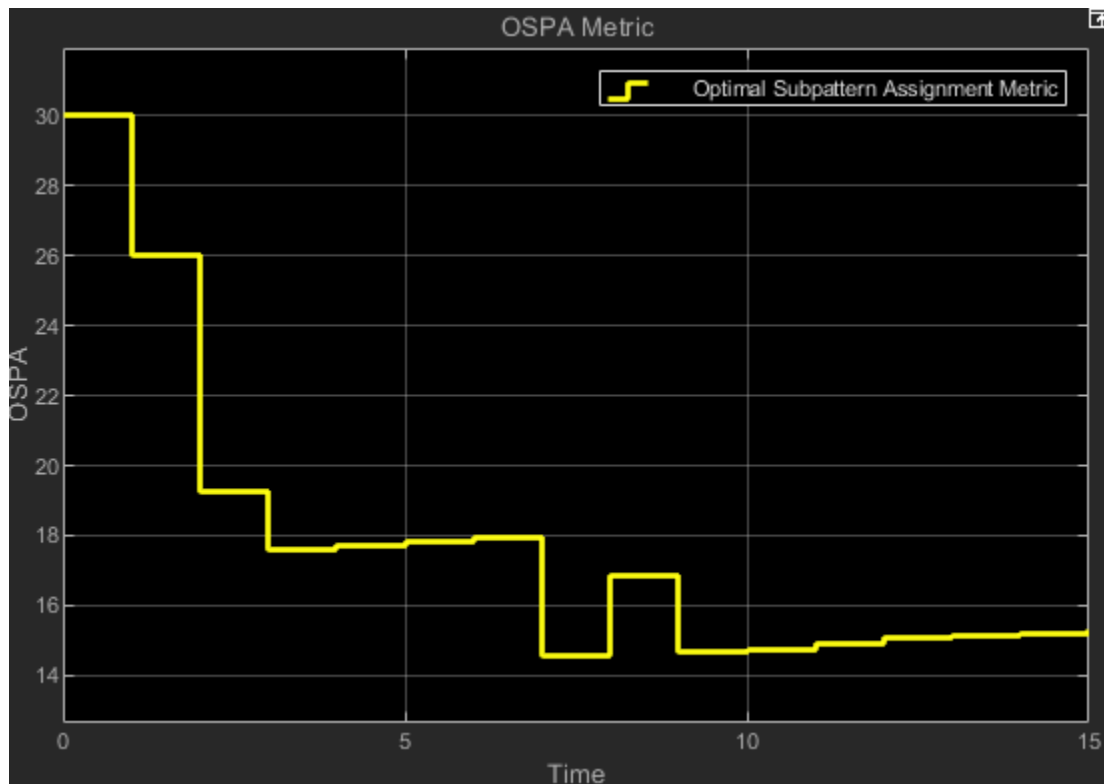


In the configured model, the Tracking Scenario Reader block reads a `trackingScenario` object from the MATLAB workspace and generates simulation data in Simulink.

Note that the block generates detections from all the sensors present in the scenario in a single “Explore Simulink Bus Capabilities” (Simulink), while the architecture subsystem requires a detection bus input from each sensor separately. You use a helper block, Helper Detection Selector, to separate detections from each sensor into a separate output bus. The helper block is implemented using the MATLAB System (Simulink) block, and is defined in the `HelperDetectionSelector` class, saved in the example folder. The architecture subsystem accepts detections and prediction time as inputs and generates tracks as output.

You use the Optimal Subpattern Assignment Metric block to evaluate the performance of the tracking architecture. The OSPA metric evaluates the performance of a tracking system through a scalar cost value, obtained by combining different error components. A lower OSPA value means better tracking. See `trackOSPAMetric` for more details. You use the Scope (Simulink) block to visualize the OSPA metric results.

```
% Simulate the model
sim(model);
```



In the figure above, notice that the OSPA metric goes down after a few steps. The initial value of OSPA metric is higher because of establishment delay for each track.

```
close_system(model,0);
```

Summary

In this example you learned how to define a tracking architecture and export it to a Simulink model. You also learned how to evaluate the performance of an architecture using the OSPA metric in Simulink.

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

This example shows how to define architectures for a tracking system-of-systems in MATLAB and export them to a Simulink model. You compare various tracking system designs that includes multiple detection-level multi-object trackers and track fusers in Simulink. You use Simulink Variant systems to realize different architecture solutions for your system. This example closely follows the “Define and Test Tracking Architectures for System-of-Systems” on page 6-745 MATLAB example.

Introduction

The “Simulate and Track En-Route Aircraft in Earth-Centered Scenarios” on page 6-564 example shows how to track aircraft using multiple long-range radars and fuse data from Automatic Dependent Surveillance Broadcast (ADS-B) transponders to get more accurate air situation picture. In the example, a tracker fuses radar detections centrally before these tracks are fused with tracks from the ADS-B reports.

The architecture described above is only one possible architecture and you may want to explore other tracking architectures. There are many factors to consider when designing a tracking architecture:

- 1 Sensor outputs: Some radars output detections while other radars may track objects internally and only provide sensor tracks as outputs.
- 2 Communication networks: Reporting tracks instead of detections can reduce the amount of data required to be transmitted over a communication network. Additionally, latency, physical distance, and other limitations may require a sensor to report tracks.
- 3 Computational resources: Processing all the detections in a single central tracker usually requires more memory and computation resources than processing data in a distributed fashion at each tracker.

However, there are also reasons to select centralized architectures over decentralized architectures. First, centralized tracking systems can be more accurate, because all the available data is processed in one place and there are no constraints on the data being processed. In addition, the architecture is much simpler as there is only one tracker.

Centralized Air Surveillance Architecture

As a baseline, you define an architecture that follows the centralized tracking system described in the “Simulate and Track En-Route Aircraft in Earth-Centered Scenarios” on page 6-564 example. You first define the same centralized `trackerGNN` processing all the radar detections as the example. Note that the radars update every 12 seconds and the ADS-B receiver updates every second. To accommodate the different update rates, you define the tracker inside a `helperScheduledTracker` object. The `helperScheduledTracker` class inherits from the `fusion.trackingArchitecture.Tracker` interface class and implements an `exportToSimulink` method to export the customized tracker as an equivalent block in Simulink.

```
gnn = trackerGNN( ...
    TrackerIndex=2, ...
    FilterInitializationFcn=@initfilter, ...
    ConfirmationThreshold=[3 5], ...
    DeletionThreshold=[5 5], ...
    AssignmentThreshold=[1000 Inf]);
```

```
tracker = helperScheduledTracker(gnn, 12);
```

You define the tracker and the ADS-B as two sources fused by a `trackFuser`. Note that the `SourceIndex` values for the two `fuserSourceConfiguration` objects, specified in the `SourceConfigurations` property of the `trackFuser` object, must match the `TrackerIndex` value of the tracker and the `SourceIndex` value of the ADS-B track respectively. Additionally, the `FuserIndex` value, which must be unique is set it to 3.

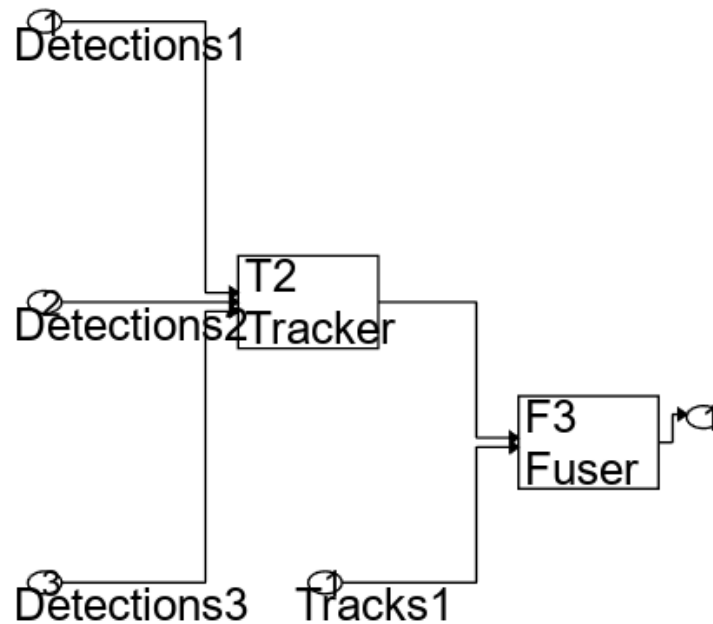
```
sources = {...
    fuserSourceConfiguration(SourceIndex=2) ... % Tracker
    fuserSourceConfiguration(SourceIndex=1);... % ADS-B
};

fuser = trackFuser( ...
    FuserIndex=3, ...
    MaxNumSources=2, ...
    SourceConfigurations=sources, ...
    AssignmentThreshold=[1000 Inf], ...
    StateFusion="Intersection", ...
    StateFusionParameters="trace", ...
    ProcessNoise=10*eye(3));
```

Next, you define the `trackingArchitecture` object. You add the tracker to the architecture, and, sensors 1, 2, and 3 all report to the tracker. You also add the track fuser to the architecture. The `trackingArchitecture` object directs fuser sources to the fuser based on `SourceConfigurations` property of the fuser. You use the `show` object function to display the architecture in a figure.

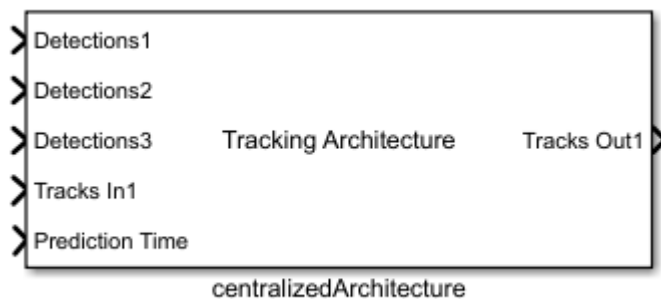
```
centralizedArchitecture = trackingArchitecture;
addTracker(centralizedArchitecture, tracker, Name="Tracker", SensorIndices=[1 2 3], ToOutput=false);
addTrackFuser(centralizedArchitecture, fuser, Name="Fuser", ToOutput=true);
show(centralizedArchitecture);
```

Tracking Architecture: centralizedArchitecture

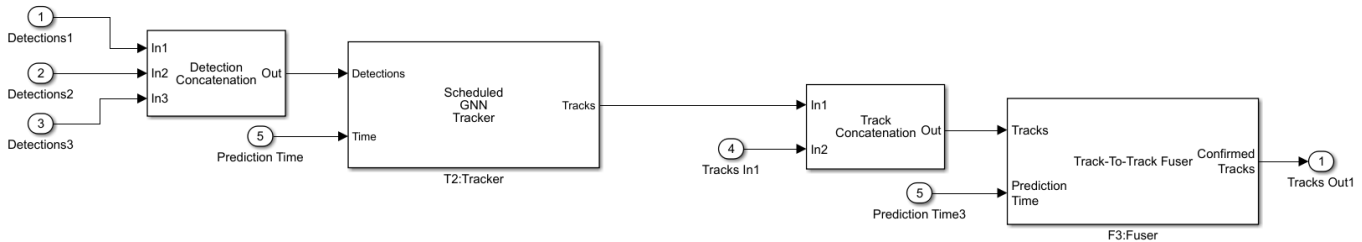


Once you have verified the architecture in MATLAB, you use the `exportToSimulink` object function to export it to a Simulink model.

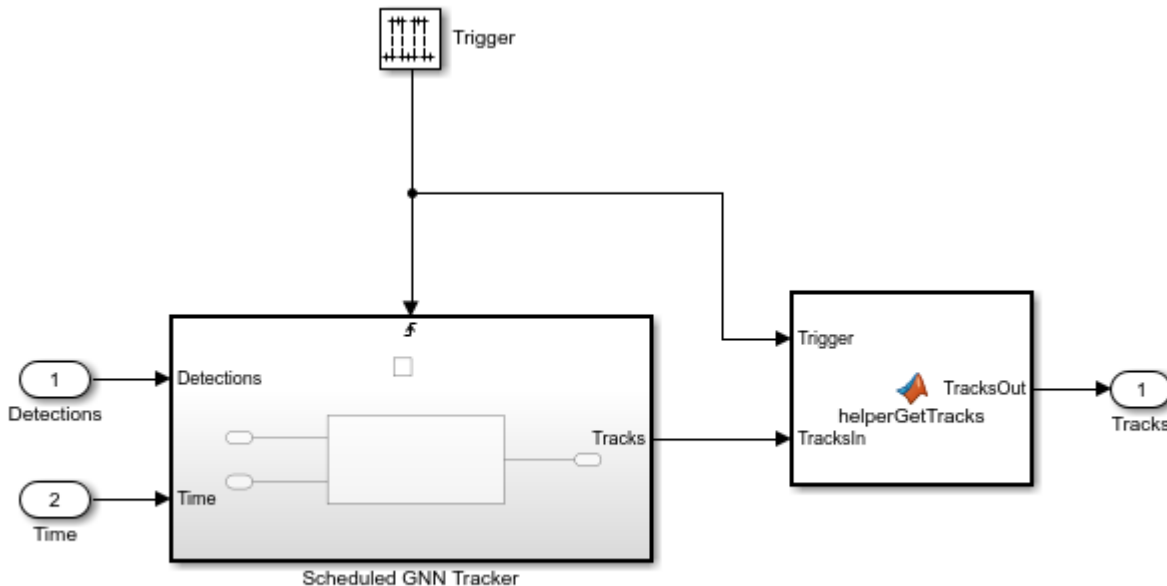
```
model = exportToSimulink(centralizedArchitecture);
```



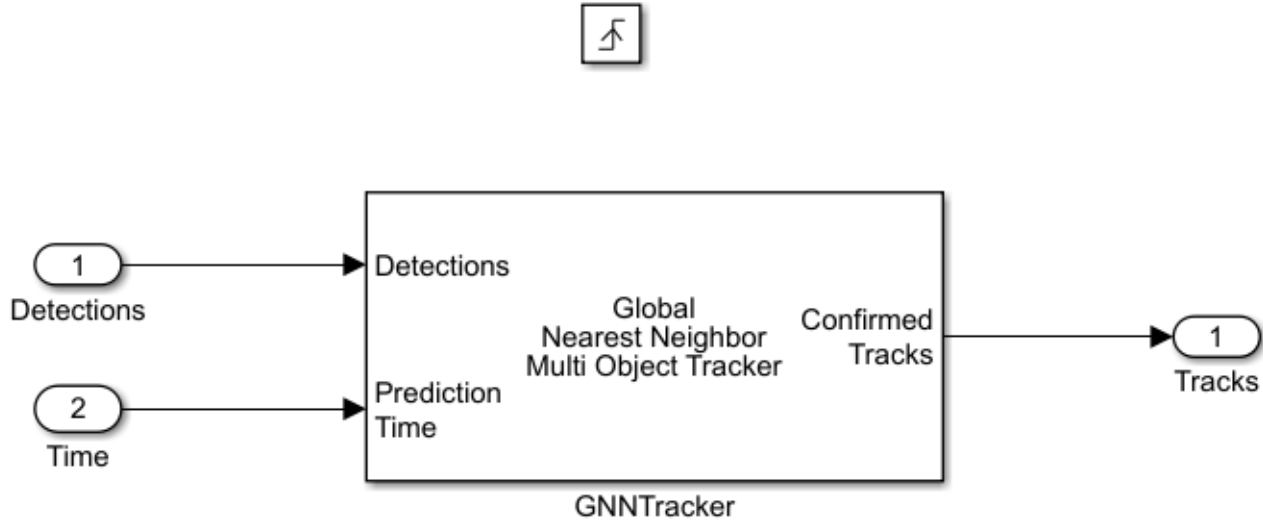
The architecture is exported as a Subsystem (Simulink) which contains blocks that build up the architecture. For each tracker or track fuser object in the MATLAB architecture, an equivalent block is added in the Simulink model. Detection Concatenation and Track Concatenation blocks are used when there are more than one input source to a tracker block and a track-to-track fuser block, respectively.



In the architecture, the GNN tracker is customized to accommodate the different update rates of sensors and ADS-B reports. The `helperScheduledTracker` class implements `exportToSimulink` function to add a customized block in the model. In the above diagram, the T2:Tracker block is a customized GNN tracker block. The tracker block updates every 12 seconds, while the rest of the model updates every 1 second. The tracker block is subsystem block which expands into the diagram below, where a "Using Triggered Subsystems" (Simulink) block is used to update the underlying tracker block whenever a trigger signal is received. The Pulse Generator (Simulink) block is used to generate a trigger signal every 12 seconds. Implemented using MATLAB Function (Simulink) block, the `helperGetTracks` block acts as a switch for the track output. When the tracker is updated, it outputs the generated tracks. Otherwise, it outputs an empty track bus. See the `exportToSimulink` method in the `helperScheduledTracker` class for more details.



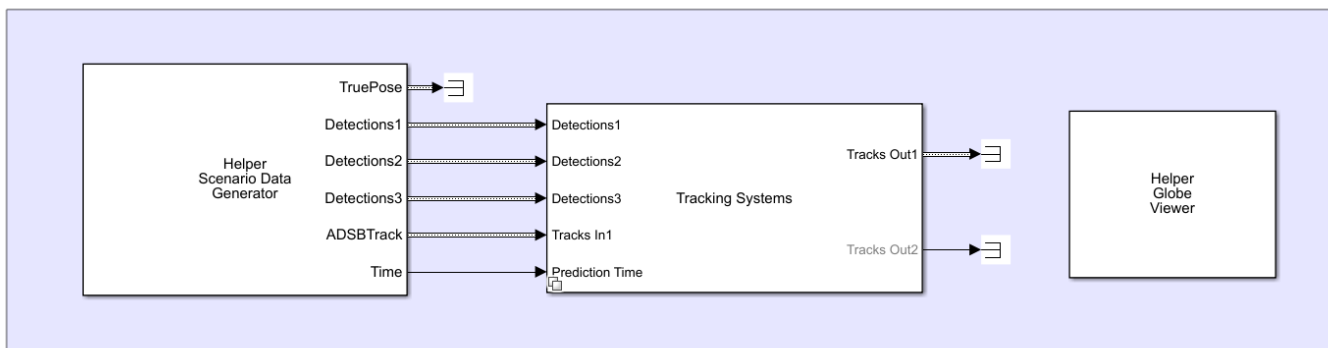
The Scheduled GNN Tracker block expands into the underlying Global Nearest Neighbor Multi Object Tracker block.



You connect the architecture with inputs and simulate the model to get the output tracks from the architecture. In this example you use a preconfigured model which contains configured inputs and scenario visualization blocks.

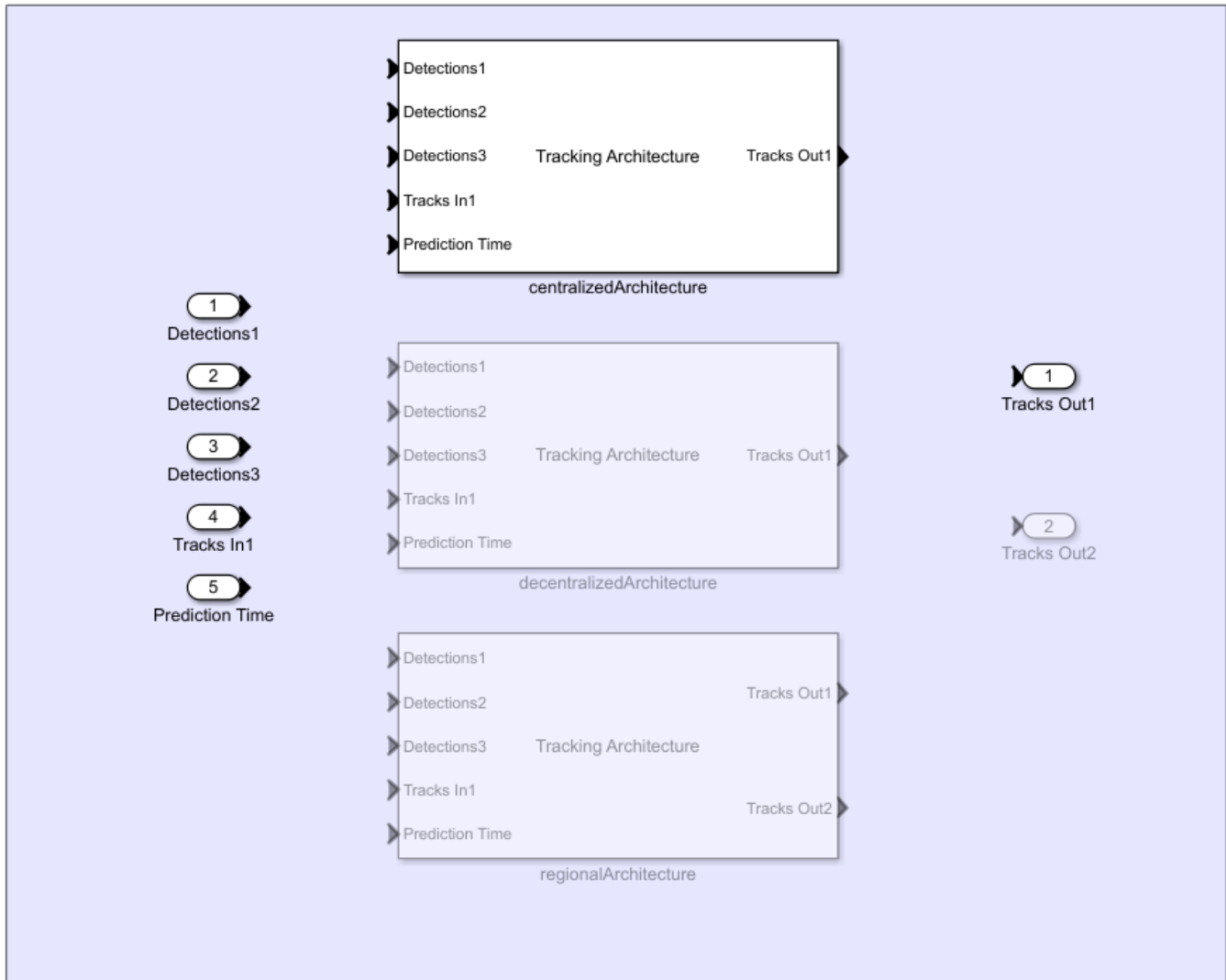
```
close_system(model,0);
open_system('TestTrackingArchitecturesInSimulink');
```

Test Tracking Architectures in Simulink



You use the data recorded from the “Simulate and Track En-Route Aircraft in Earth-Centered Scenarios” on page 6-564 example. The Helper Scenario Data Generator block generates aircraft position, detections from three sensors, ADS-B tracks, and prediction time. Implemented using the MATLAB System (Simulink) block, the code for the block is defined in the `HelperScenarioDataGenerator` class. The Helper Globe Viewer block visualizes the scenario using the `trackingGlobeViewer` object. Implemented using the MATLAB System block, the code

for the block is defined in the `HelperGlobeVisualization` class. The Tracking Systems block is a Variant subsystem block that includes one subsystem for each tracking architecture.

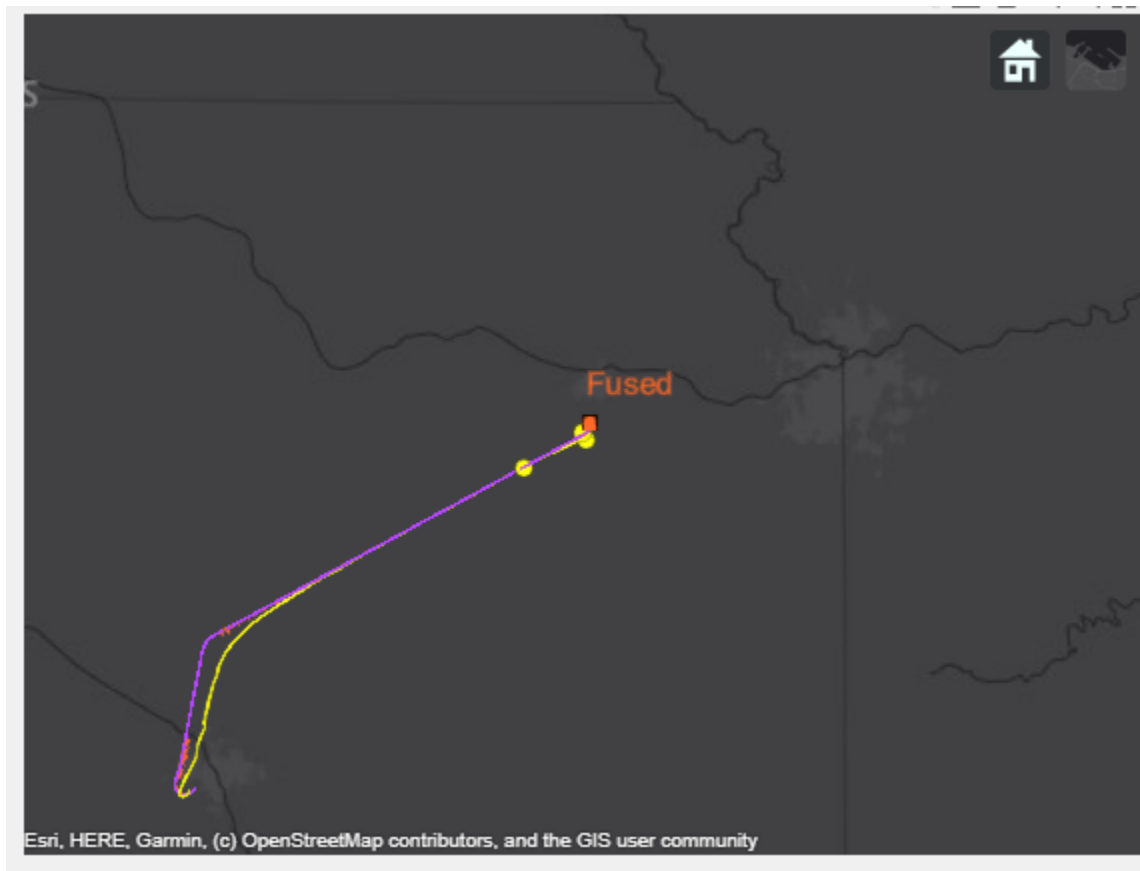


You can select the specific tracking architecture by changing the value of the workspace variable `ARCH`. The default value of `ARCH` is 1, corresponding to the centralized tracking architecture.

ARCH Tracking Architecture

1	Centralized
2	Decentralized
3	Regional

```
ARCH = 1;
sim('TestTrackingArchitecturesInSimulink');
```



In the above snapshot, the aircraft is detected by multiple radars, and is within the ADS-B communication range. The radar tracker establishes a track corresponding to the aircraft. The radar track is fused with the ADS-B track, which improves the accuracy of the fused track.

Decentralized Architecture

You want to explore how the system behaves when each radar reports sensor-level tracks and a track fuser fuses those tracks with the ADS-B tracks.

Define the `trackingArchitecture` object.

```
decentralizedArchitecture = trackingArchitecture;
```

Next, you define a tracker for each radar. There are three radar sensors in the original example, and thus you define three trackers. You define each tracker in a similar way as the centralized case. Since the `SourceIndex` of the ADS-B tracks is 1, you define the `TrackerIndex` of the three trackers as 2, 3, and 4, respectively. Meanwhile, the `SensorIndex` for the detections from the three radars are 1, 2, and 3, respectively.

```
sensorTracker = trackerGNN(...
    TrackerIndex=2, ...
    FilterInitializationFcn=@initfilter, ...
    ConfirmationThreshold=[3 5], ...
    DeletionThreshold=[5 5], ...
    AssignmentThreshold=[1000 Inf]);
```

```

% Add one tracker to each radar.
for i = 2:4
    tracker = helperScheduledTracker(clone(sensorTracker),12);
    tracker.TrackerIndex = i; % Specify each radar tracker with a different index.
    addTracker(decentralizedArchitecture,tracker,Name=strcat('Radar',num2str(i-1)),SensorIndices=
end

```

You add a trackFuser that fuses tracks from four sources, the three radar trackers and the ADS-B, and export the architecture into a Simulink model.

```

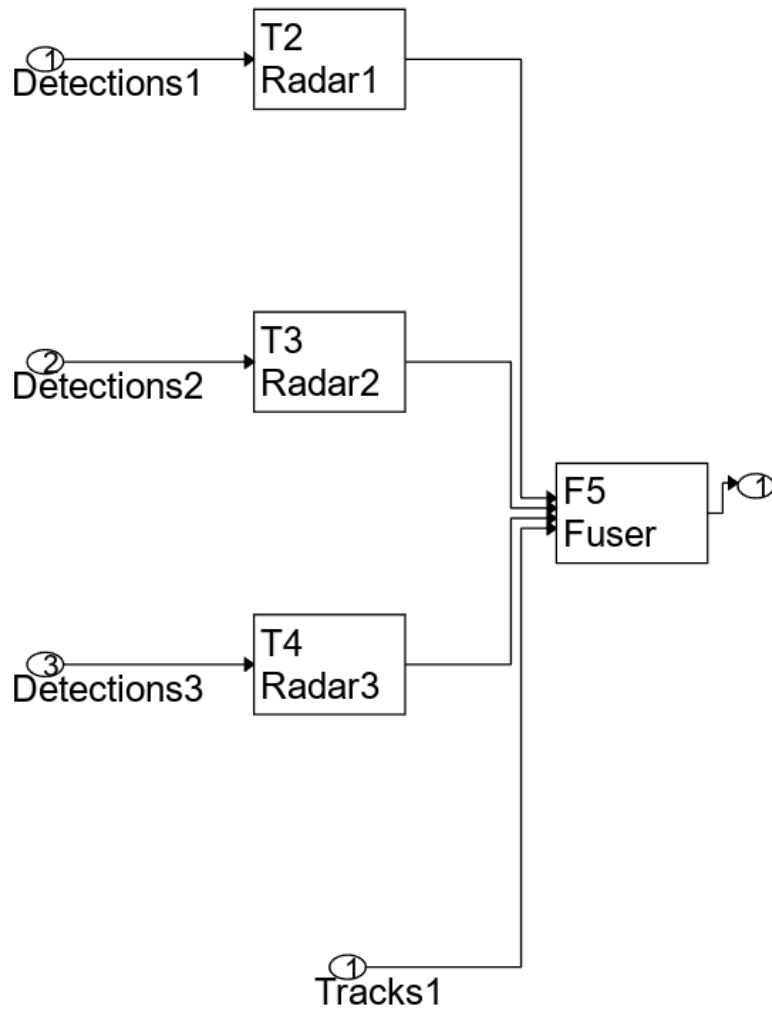
% Define the sources.
sources = {...
    fuserSourceConfiguration(2);... % Tracker for radar 1
    fuserSourceConfiguration(3);... % Tracker for radar 2
    fuserSourceConfiguration(4);... % Tracker for radar 3
    fuserSourceConfiguration(1);... % ADS-B
};

% Add the fuser.
fuser = trackFuser( ...
    FuserIndex=5, ...
    MaxNumSources=4, ...
    SourceConfigurations=sources, ...
    AssignmentThreshold=[1000 Inf],...
    StateFusion="Intersection", ...
    StateFusionParameters="trace", ...
    ProcessNoise=10*eye(3));
addTrackFuser(decentralizedArchitecture, fuser, Name="Fuser");

% Show the tracking architecture in a figure.
ax2 = show(decentralizedArchitecture);
ax2.Parent.Position = [680 456 484 522];

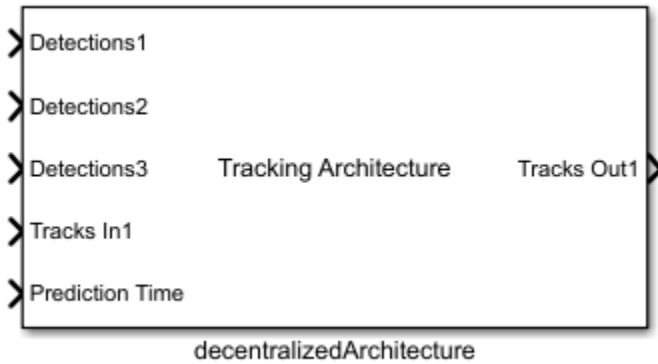
```

Tracking Architecture: decentralizedArchitecture

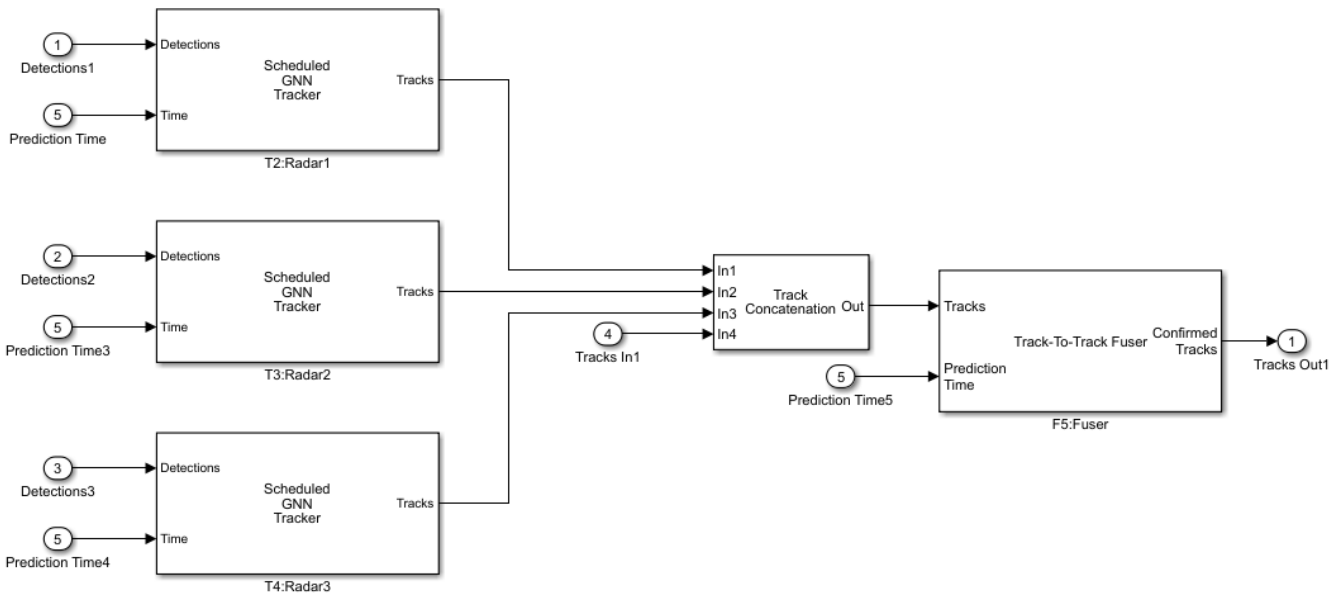


Export the architecture into a Simulink model.

```
model = exportToSimulink(decentralizedArchitecture);
```

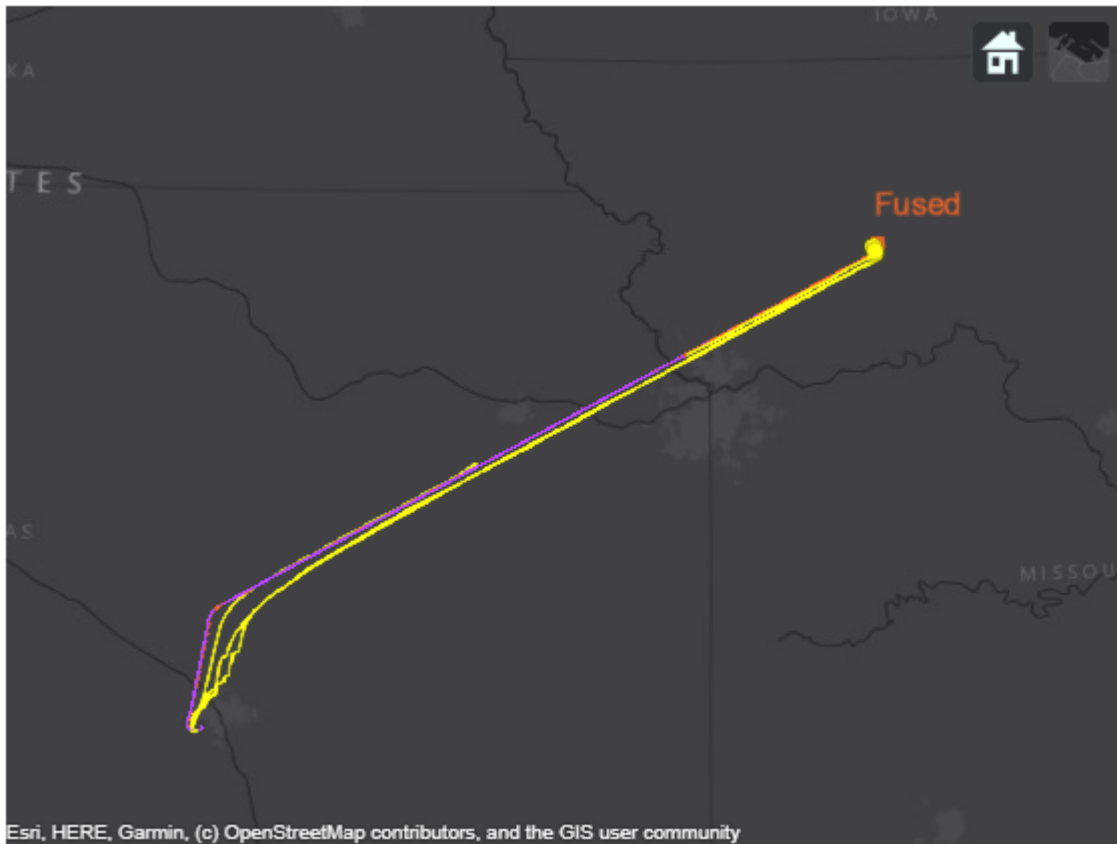


The decentralized architecture subsystem when expanded, shows following block diagram:



You set the workspace variable ARCH to 2, which enables, the decentralized architecture in the preconfigured model.

```
close_system(model,0);
ARCH = 2; %#ok<*NASGU>
sim('TestTrackingArchitecturesInSimulink');
```



In the above snapshot, each radar tracker establishes a track corresponding to the aircraft. Radar closest to the aircraft estimates the track position more accurately. Fusing the radar reported tracks with the ADS-B track improves the accuracy of the fused track.

Regional Air Surveillance Architecture

Another possible configuration that falls between the centralized and decentralized architectures above is a regional surveillance architecture. In many cases, an air traffic control center may only need to track aircraft that are close to it. By using these regional control centers, you can construct a regional tracking architecture.

You define the `trackingArchitecture` object and define trackers for each radar, similar to the decentralized architecture.

```
regionalArchitecture = trackingArchitecture;
for i = 2:4
    tracker = helperScheduledTracker(clone(sensorTracker),12);
    tracker.TrackerIndex = i; % Specify each radar with a different index.
    addTracker(regionalArchitecture,tracker,Name=strcat('Radar',num2str(i-1)),SensorIndices=i-1,
end
```

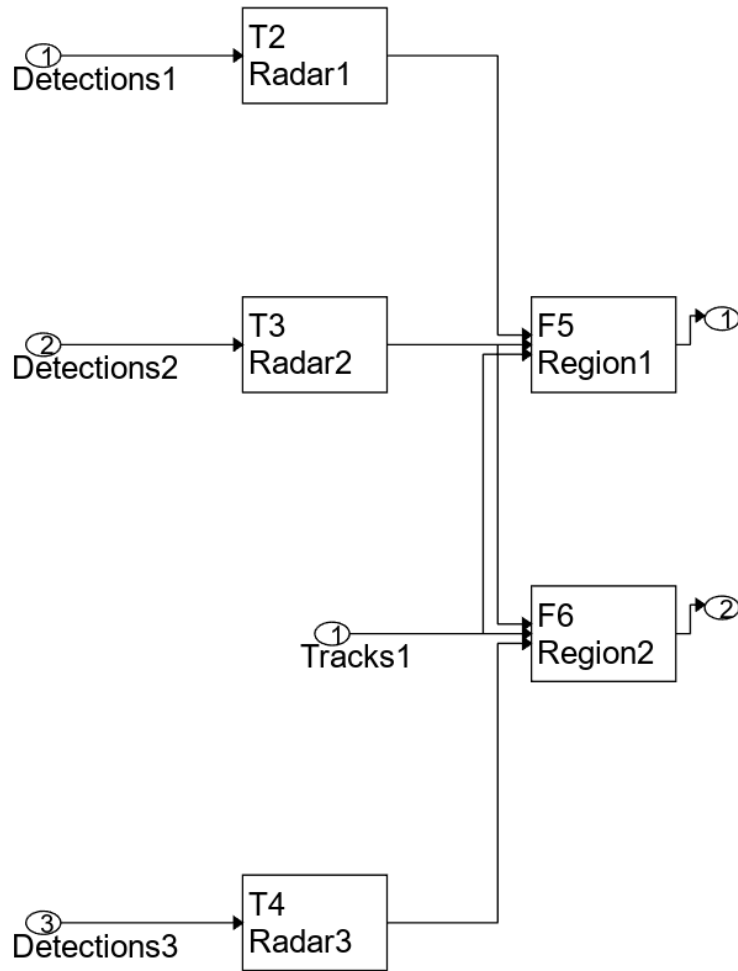
You define two regions and attach two radar sensors to each regional fuser. Additionally, each regional fuser only fuses ADS-B tracks if the reporting aircraft is close enough.

```
region1Fuser = trackFuser( ...
    FuserIndex=5, ...
```

```
    MaxNumSources=3, ...
    SourceConfigurations=sources([1,2,4]), ... % Two radars and ADS-B
    AssignmentThreshold=[1000 Inf], ...
    StateFusion="Intersection", ...
    StateFusionParameters="trace", ...
    ProcessNoise=10*eye(3));
region2Fuser = trackFuser( ...
    FuserIndex=6, ...
    MaxNumSources=3, ...
    SourceConfigurations=sources([4,2,3]), ... % ADS-B and Two radars
    AssignmentThreshold=[1000 Inf],...
    StateFusion="Intersection", ...
    StateFusionParameters="trace", ...
    ProcessNoise=10*eye(3));
addTrackFuser(regionalArchitecture, region1Fuser, Name="Region1");
addTrackFuser(regionalArchitecture, region2Fuser, Name="Region2");

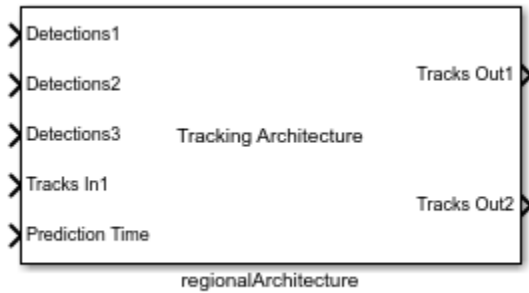
% Show the tracking architecture in a figure.
clf reset;
ax3 = show(regionalArchitecture);
ax3.Parent.Position = [680 403 558 575];
```

Tracking Architecture: regionalArchitecture

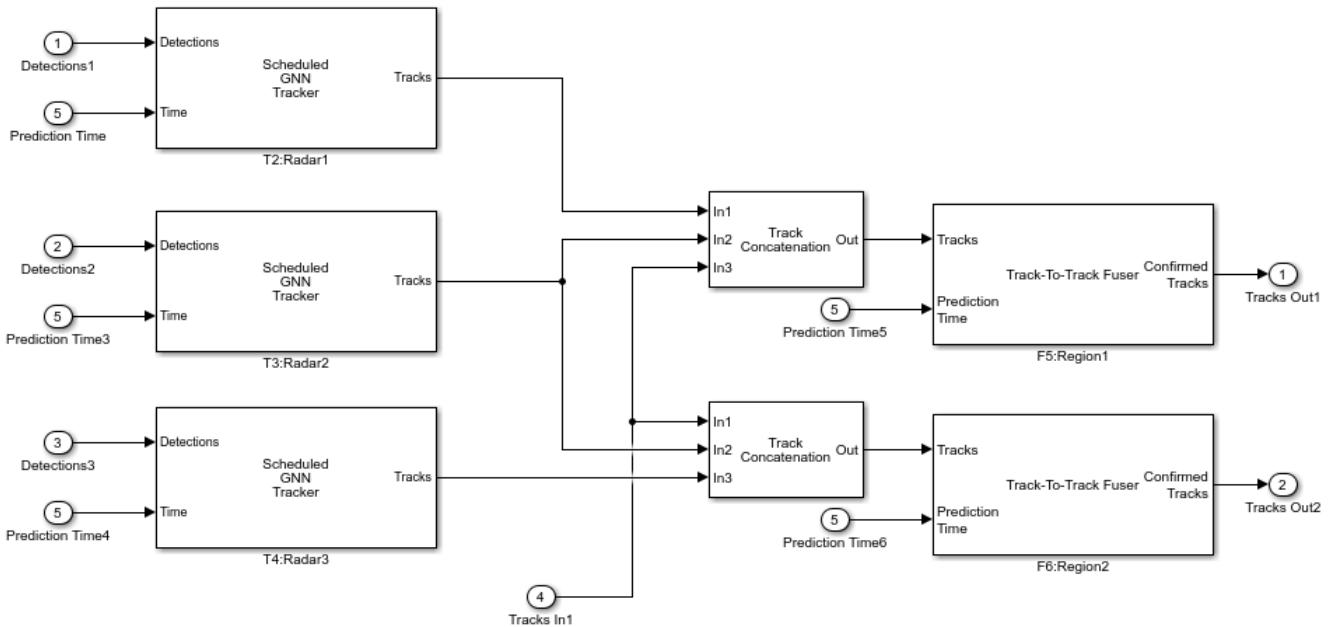


Export the architecture into a Simulink model.

```
model = exportToSimulink(regionalArchitecture);
```

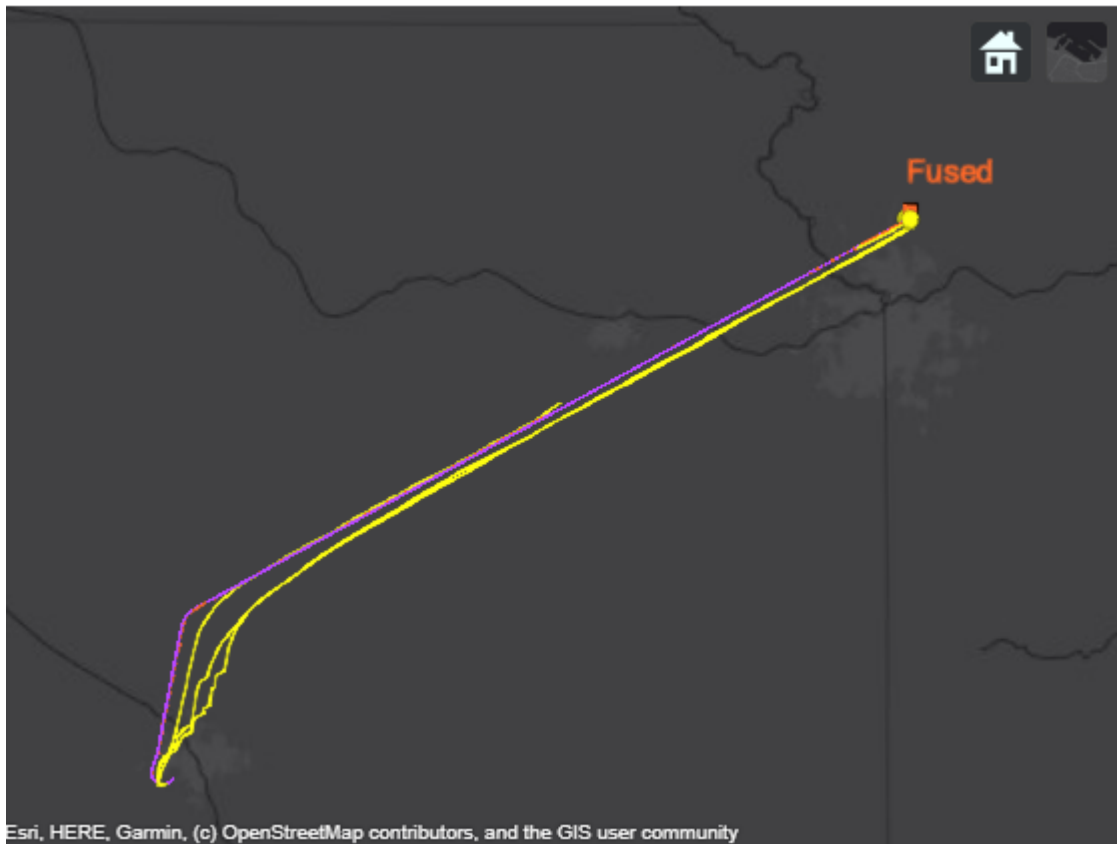



The regional architecture subsystem when expanded, shows following block diagram:



You set the workspace variable ARCH to 3, which enables the regional architecture in the preconfigured model. In the regional architecture you have 2 architecture track outputs for the same object.

```
close_system(model,0);
ARCH = 3;
sim('TestTrackingArchitecturesInSimulink');
```



In the above snapshot, the radar reported tracks are fused with the ADS-B tracks regionally and you see two fused tracks corresponding to the aircraft.

```
close_system('TestTrackingArchitecturesInSimulink',0);
```

Summary

In this example, you learned how to define different tracking architectures and how to export them to Simulink. You also learned how to use variant systems to realize different tracking system designs.

Simulate and Track Targets with Terrain Occlusions

This example shows you how to model a surveillance scenario in a mountainous region where terrain can occlude both ground and aerial vehicles from the surveillance radar. You define a tracking scenario with geo-referenced terrain data from a Digital Terrain Elevation Data (DTED) file, create trajectories following terrain, simulate the scenario, and track targets with a multi-object tracker.

Create Scenario and Add Terrain

First create an Earth-centered tracking scenario, then add terrain data using the scenario object function `groundSurface`. You can specify the terrain as a matrix of height values, or like in this example, as a DTED file. The DTED used in this scenario spans latitude between 39 and 40 degrees North and longitude between 105 and 106 degrees West. This corresponds to a mountainous region in Colorado, USA.

```
scene = trackingScenario(IsEarthCentered=true, UpdateRate=1);
% Add terrain
dtedfile = "n39_w106_3arc_v2.dt1";
groundSurface(scene, Terrain=dtedfile);
```

All surfaces added to the scenario are managed by the `SurfaceManager` object.

```
scene.SurfaceManager

ans =
  SurfaceManager with properties:

    UseOcclusion: 1
    Surfaces: [1x1 fusion.scenario.GroundSurface]

scene.SurfaceManager.Surfaces(1)

ans =
  GroundSurface with properties:

    Terrain: "n39_w106_3arc_v2.dt1"
    ReferenceHeight: 0
    Boundary: [2x2 double]
```

Next, add three platforms to the scenario. The first platform is a drone flying at a constant altitude of 20 meters above the ground. Use the following strategy to define a trajectory following the terrain:

- Define the latitude and longitude components of the trajectory using a `geoTrajectory` object. Set the altitude to 0 meters for this first step.
- Sample positions along the trajectories using an adapted sample size. There is a tradeoff between precision and computation time when sampling.
- Query the terrain height for each sample using the `SurfaceManager` object.
- Build the final trajectory with the `geoTrajectory` object using the computed samples and height values.

```
% Add a platform on the ground
llacoords = [39.7894, -105.6284, 0; ...
```

```

    39.8012, -105.6251, 0;...
    39.80449, -105.6420, 0];
groundTraj = geoTrajectory(llacoords,[0;120;240]);

numTrajectorySamples = 40;
toa = linspace(0,240, numTrajectorySamples);
pos = lookupPose(groundTraj, toa);
flightAltitude = 20; % 20 meters
pos(:,3) = height(scene.SurfaceManager,pos(:,[1 2])) + flightAltitude;

% Build the final trajectory and add platform
groundTraj = geoTrajectory(pos,toa);
drone = platform(scene,Trajectory=groundTraj);

```

The second platform is a ground vehicle that follows a mountain pass road. Coordinates and time values are saved in the `roadCoordinates.mat` file.

```

% Add a vehicle following the road
load roadCoordinates.mat
roadTraj = geoTrajectory(roadlla(10:end-30,:), 1.3*toas(10:end-30));
roadPlat = platform(scene,Trajectory=roadTraj);

```

The third platform is a radar tower. The tower is located on top of a mountain and the ground and aerial surveillance radar, mounted on it, stares at the ground and sky towards the Southeast. Set the `HasElevation` property to true to report elevation, which is important when tracking over terrain where elevation often varies.

```

% Add a tower
towerLatLon = [39.8057, -105.6246];
towerGroundHeight = height(scene.SurfaceManager,towerLatLon');
tower = platform(scene,Position=[ towerLatLon, towerGroundHeight]);
tower.Sensors = fusionRadarSensor(ScanMode="No scanning", ...
    UpdateRate=1/2,...
    SensorIndex=2,...
    FieldOfView=[180;35],...
    HasElevation=true,...
    ElevationResolution=1,...
    RangeResolution=5,...
    RangeLimits=[0 2000],...
    ReferenceRange=2000,...
    FalseAlarmRate=1e-7,...
    HasFalseAlarms=true,...
    MountingAngles=[200 -15 0],...
    MountingLocation=[0 0 -25],...
    HasINS=true, DetectionCoordinates='scenario',...
    HasNoise=true);

```

Define Tracking System

Use a multi-object tracker to track the drone and the ground vehicle. Use the default constant velocity motion model, which works reasonably well for tracking targets that move slowly. Increase slightly the “AssignmentThreshold” property to account for large measurement noise when the radar is detecting at long ranges.

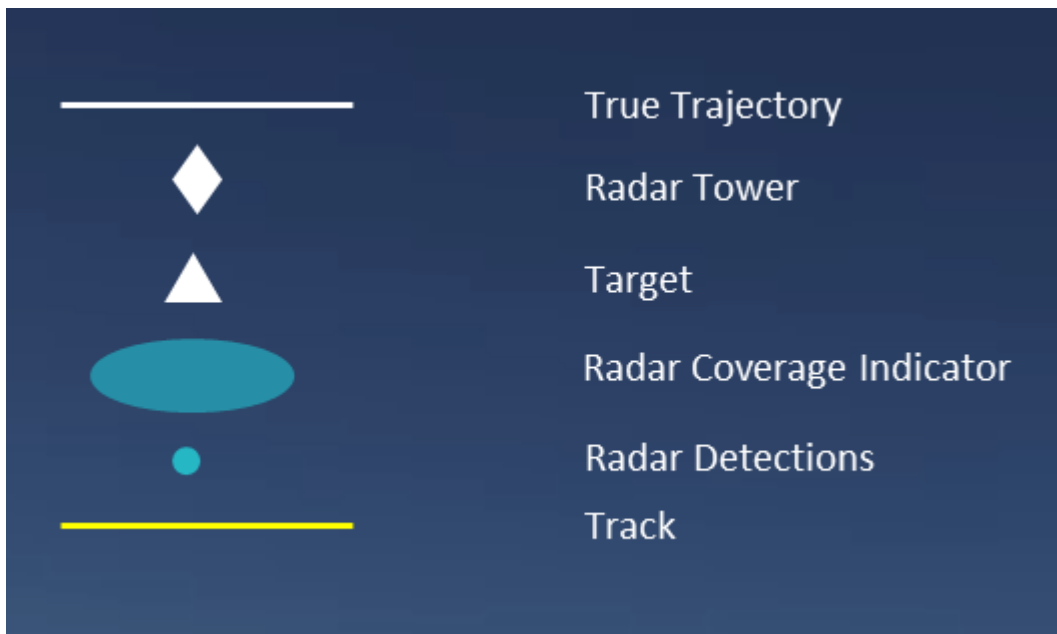
```

towerTracker = trackerGNN(AssignmentThreshold=50);

```

Create Visualization and Simulate the Scenario

Visualize the scene using a `trackingGlobeViewer` object. By default, the viewer object does not display terrain. First, use the `addCustomTerrain` function to add the DTED file. Then specify the `Terrain` property as the DTED file name to visualize the terrain in the viewer. The `addCustomTerrain` function saves the DTED to a location and the function should be used only once. Also, specify the `CoverageRangeScale` property to scale down the radar coverage and reduce visual clutter. This, however, causes that the coverage displayed in the viewer no longer reflects the actual range of the radar. Record and visualize the occlusion history by using the `occlusion` object function of the `SurfaceManager` object. The legend used in the viewer is shown below.



```
try
    addCustomTerrain("TrackingOverTerrainExample",dtedfile);
catch
    disp("Custom Terrain was already added")
end
% Create and setup the trackingGlobeViewer object
f=uifigure;
globe = trackingGlobeViewer(f,Terrain="TrackingOverTerrainExample",CoverageRangeScale=0.1,TrackH:
campos(globe, [39.7861 -105.6287 3265]);
camorient(globe, [19.35 -23.18 0]);
plotPlatform(globe, tower, Marker='d');
plotCoverage(globe, coverageConfig(scene), "ECEF", Alpha=0.2);

% Initialize some variables for the simulation
towerRadarLLA = tower.Position + [0 0 25];
droneOcc = [];
groundOcc = [];
droneTracks = objectTrack.empty;
towerTracks = objectTrack.empty;

% Metric
ospa = trackOSPAMetric(Metric="OSPA",Distance="posabserr");
```

```

ospalog = [];

% Set random seed for repeatable results
s = rng(2022);

while advance(scene)
    time = scene.SimulationTime;

    % Generate detections
    [towerDets,~, config] = detect(tower, time);

    % Update tower tracker
    if config.IsValidTime && (isLocked(towerTracker) || ~isempty(towerDets))
        towerTracks = towerTracker(towerDets, time);
    elseif isLocked(towerTracker)
        towerTracks = predictTracksToTime(towerTracker, 'confirmed',time);
    end

    % Update globe
    plotPlatform(globe,[drone roadPlat],TrajectoryMode="Full",LineWidth=1);
    plotDetection(globe, towerDets, "ECEF");
    plotTrack(globe, towerTracks, "ECEF", LabelStyle="ATC",LineWidth=2);

    % Move camera
    moveGlobeCamera(globe, time);

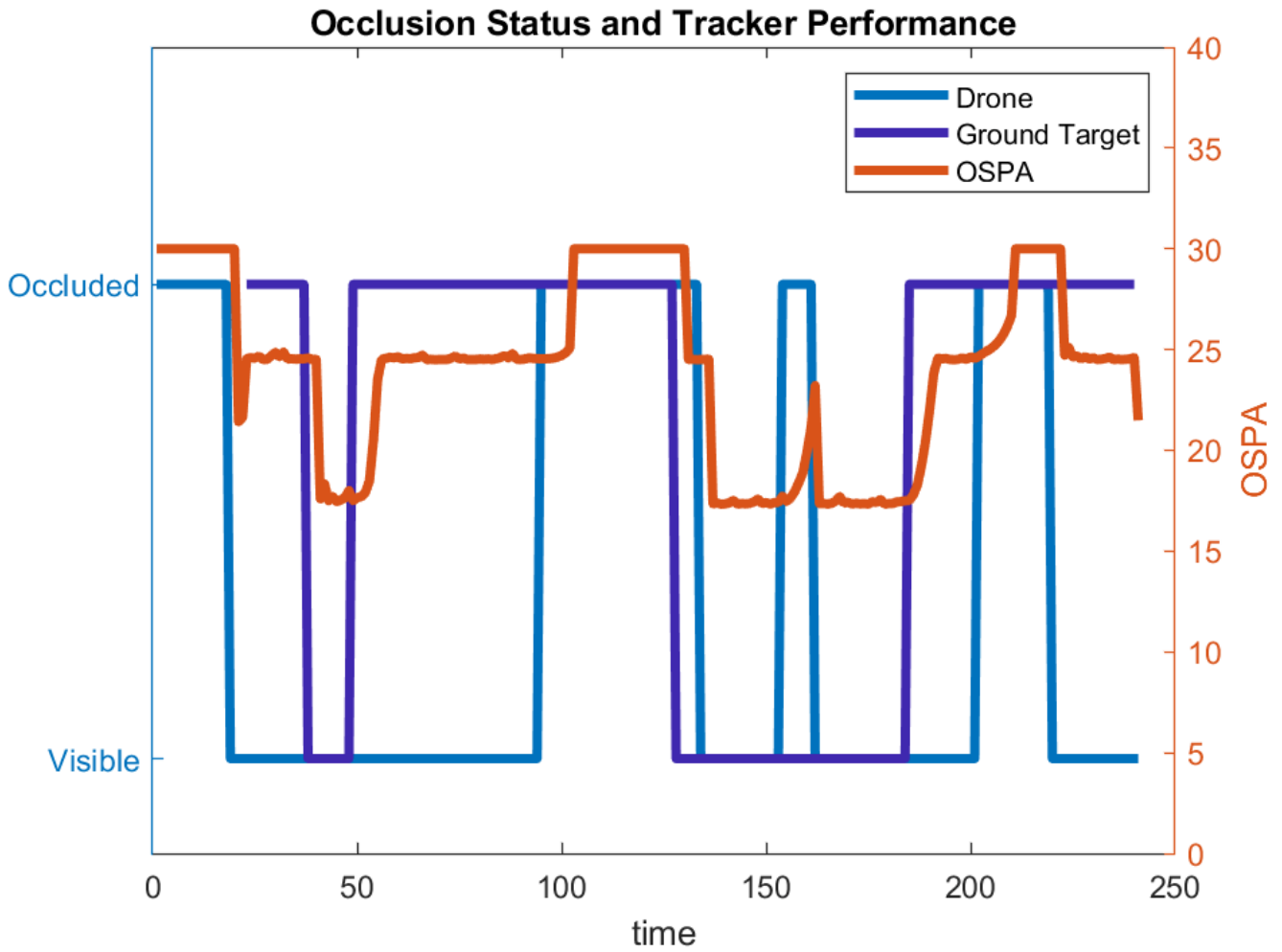
    % Compute metric and save occlusion state for plots
    ospalog(end+1) = ospa(towerTracks,platformPoses(scene)); %#ok<SAGROW>
    drone0cc(end+1) = occlusion(scene.SurfaceManager,towerRadarLLA,drone.Position); %#ok<SAGROW>
    ground0cc(end+1) = occlusion(scene.SurfaceManager,towerRadarLLA,roadPlat.Position); %#ok<SAGROW>

    drawnow;
end

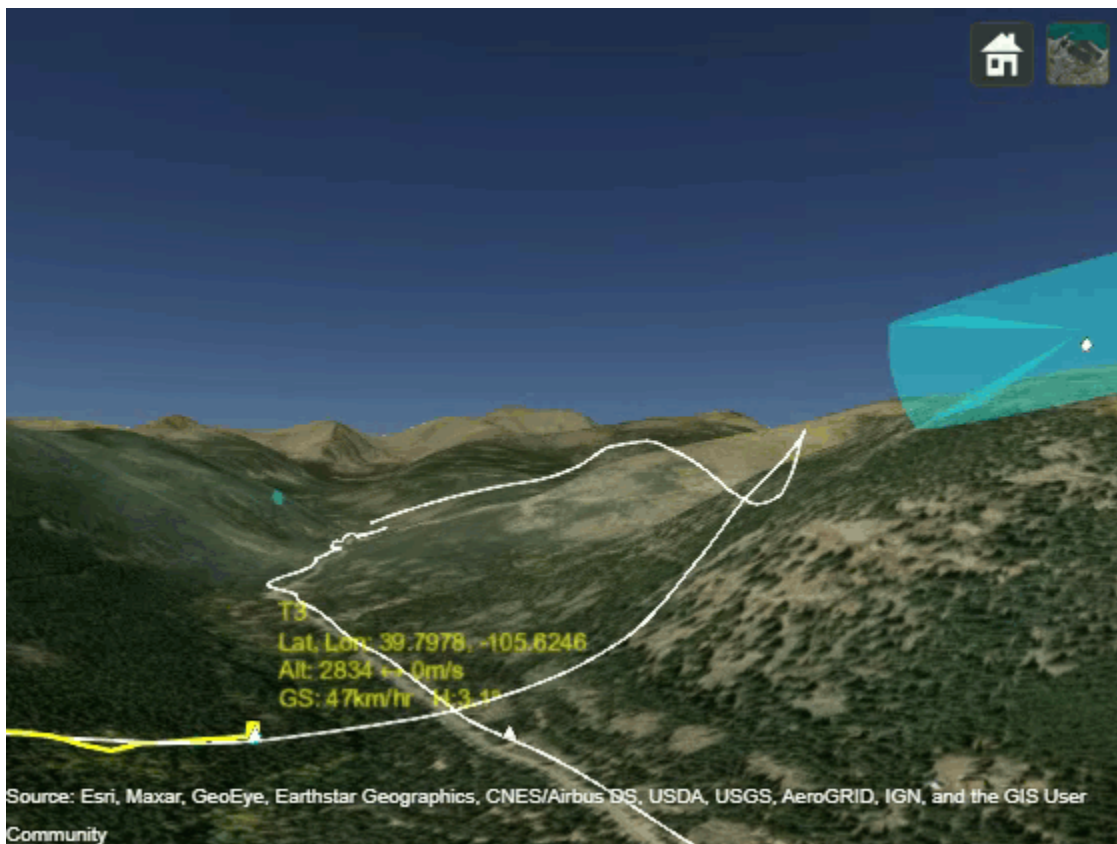
```

In the figure below, the occlusion plot on the left axis helps analyze the scenario by showing the occlusion status of the target over time. This provides the information to understand why radar detections are not available as well as why tracks are dropped during some periods. The OSPA metric plot on the right axis gives a quantitative assessment of the tracker performance and shows how tracking performance correlates with the occlusion status.

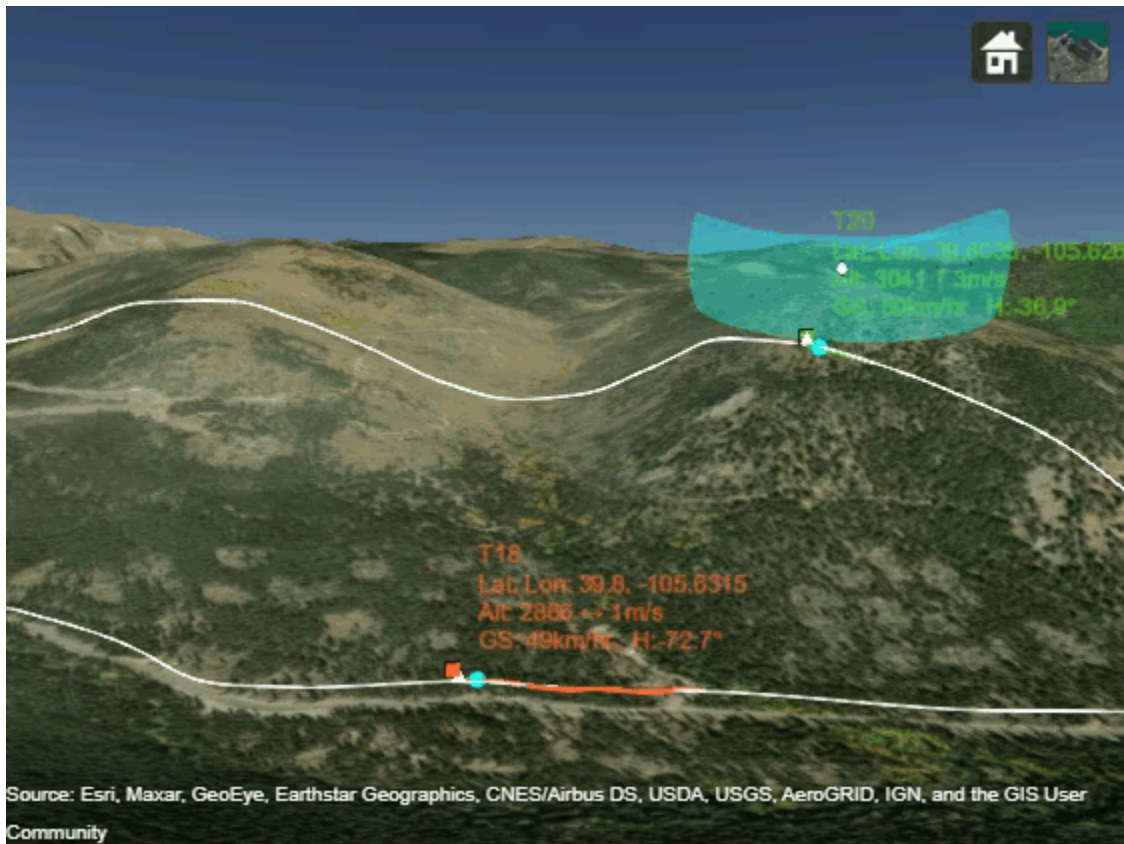
```
plotOcclusionAndMetric(drone0cc, ground0cc, ospalog);
```



The two GIFs below provide analysis of two specific periods in the simulation.



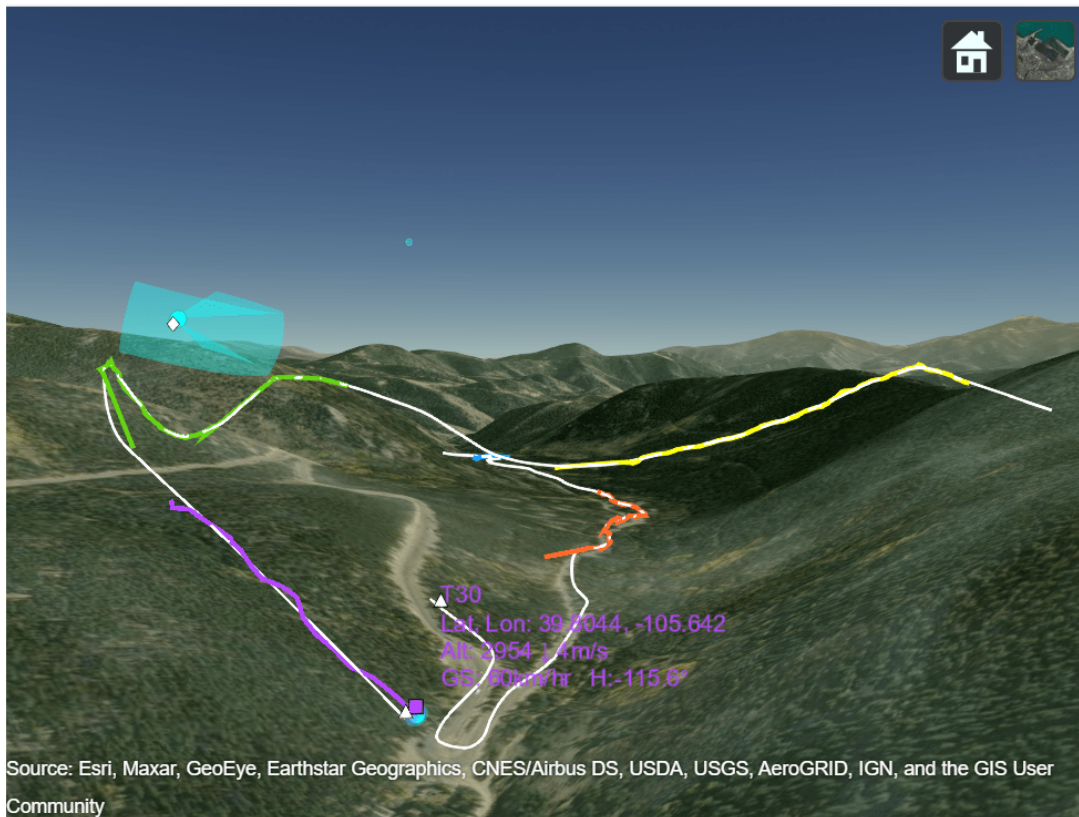
The first GIF above shows a segment of the simulation around 100 seconds, which corresponds to the time that the drone enters the occluded area at the foot of the radar-located mountain. Notice that the track is coasted, and its associated uncertainty grows due to the lack of observations until it is deleted. The previous OSPA plot slowly increases during coasting before jumping to the threshold value upon track deletion. In the meantime, the ground vehicle, travelling on the road, is occluded and therefore undetected.



The second GIF shows a later period of the simulation around 155 seconds. Both drone and ground vehicle are not occluded, as shown in the occlusion state plot, except for a moment when the drone passes by the saddle point in between mountains. The track is briefly coasted and recovered with the next available radar detections. This period of the simulation is also noticeable on the occlusion and OSPA plot. The performance starts to degrade (OSPA value increases) after the occlusion status switches but recovers immediately when the status switches again.

Last, take a snapshot of the globe viewer at the end of the simulation to get the full picture of the scenario.

```
campos(globe, [39.805195 -105.64668 3123]);
camorient(globe, [110 -5 0]);
drawnow;
figure;
snapshot(globe);
```



The true trajectories of the ground target and the drone are displayed in white whereas tracks are represented by colored lines. The three segments of the drone track are shown in yellow, green, and purple, respectively. The ground vehicle tracks are shown in blue and orange. In this particular scenario, the long occlusion time makes it difficult for the tracker to maintain a unique track ID for each target.

Clean Up

Reset random generator seed and remove the added DTED file.

```
rng(s);

% Remove custom terrain
if invalid(f)
    close(f);
end
removeCustomTerrain("TrackingOverTerrainExample");
```

Conclusion

In this example you learned how to include terrain data from DTED files in a tracking scenario and how to use the `SurfaceManager` property of the `trackingScenario` to query height and occlusion

information over the terrain. This allows to create trajectories that follow the terrain, such as the trajectory of a vehicle following a mountain pass road or a drone flying at constant altitude above ground. You also simulated radar detections that accounts for terrain occlusion and used a simple tracking system to track the targets. For targets that are difficult to recognize after a long time of terrain occlusion, alternate techniques relying on appearance or radar signal features could be used to re-identify lost vehicles.

Supporting functions

`plotOcclusionAndMetric` generates the plots of occlusion status between the tower and each platform as well as the track OSPA metric.

```
function plotOcclusionAndMetric(droneOcc, groundOcc, ospalog)
figure;
timevals = 1:numel(groundOcc);

yyaxis right
plot(timevals,ospalog, DisplayName="OSPA",LineWidth=3)
ylim([0 40]);
ylabel("OSPA");

yyaxis left
plot(timevals,droneOcc,DisplayName="Drone",LineWidth=3);
hold on
plot(timevals,groundOcc,Color=[0.2464 0.1569 0.6847],...
      DisplayName="Ground Target",...
      LineStyle='-',LineWidth=3);
yticklabels({'Visible','Occluded'});
yticks([0 1]);
ylim([-0.2 1.5]);

title("Occlusion Status and Tracker Performance");
xlabel("time")
legend();
drawnow
end
```

`moveGlobeCamera` moves the camera on the globe at predefined times.

```
function moveGlobeCamera(globe, time)
if time == 50
% Move camera down the pass
campos(globe, [39.7998 -105.60964 3016]);
camorient(globe, [270 1 0]);

elseif time == 90
% Move camera all the way down to observe occlusion
campos(globe, [39.7963 -105.6188 2936]);
camorient(globe, [306 6 0]);

elseif time == 120
% Move camera up the road and raise view angle
campos(globe, [39.7953 -105.6323 3061]);
camorient(globe, [14 -6 0]);

elseif time == 165
% Move camera to last section of the road
campos(globe, [39.79876 -105.6428 3166]);
camorient(globe, [35 -9 0]);
```

end
end

Tuning Kalman Filter to Improve State Estimation

This example shows how to tune process noise and measurement noise of a constant velocity Kalman filter.

Motion Model

A Kalman filter estimates the state of a physical object by processing a set of noisy measurements and compares the measurements with a motion model. As an idealized representation of the true motion of the object, the motion model is expressed as a function of time and a set of variables, called the state. The filter usually saves the state in a form of a vector, called a state vector.

This example explores one of the simplest motion models: a constant velocity motion model in two dimensions. A constant velocity motion model assumes that the object moves with a nearly constant velocity. The state vector consists of four parameters that represent the position and velocity in the x- and y- dimensions.

There are other popular motion models, for example a constant turn that assumes the object moves mostly along a circular arc with a constant speed. More complicated models, such as constant acceleration, help when an object moves for a significant duration of time according to the model. Nonetheless, a very simple motion model like the constant velocity model can be used successfully to track an object that gradually changes its direction or speed over time. A Kalman filter achieves this flexibility by providing an additional parameter called process noise.

Process Noise

In reality, objects do not exactly follow a particular motion model. Therefore, when a Kalman filter estimates the motion of an object, it must account for unknown deviations from the motion model. The term 'process noise' is used to describe the amount of deviation, or uncertainty, of the true motion of the object from the chosen motion model. Without process noise, a Kalman filter with a constant velocity motion model fits a single straight line to all the measurements. With process noise, a Kalman filter can give newer measurements greater weight than older measurements, allowing for a change in direction or speed. While 'noise' and 'uncertainty' are terms that connote the idea of a chaotic deviation from the model, process noise can also allow for small, predictable, changes to the true motion of an object that would otherwise require considerably more complex motion models.

This example shows a car moving along a curving road with a relatively constant speed profile and uses a constant velocity motion model with a small amount of process noise to account for the minor deviations due to changes in steering and speed. Process noise allows the filter to account for small changes in speed or direction without estimating how fast the car is accelerating or turning. If you examine the trajectory of a car over a short duration of time, you may observe that it goes nearly straight. It is only over larger durations of time that you can observe the change in the direction of motion of the car.

Process noise has an inherent tradeoff. A low process noise may cause the filter to ignore rapid deviations from the true trajectory and instead favor the motion model. This limits the amount of deviation from the motion model that the true trajectory can have at any particular time. A high process noise admits greater local deviations from the motion model but makes the filter too sensitive to noisy measurements.

Measurement Noise

Kalman filters also model "measurement noise" which helps inform the filter how much it should weight the new measurements versus the current motion model. Specifying a high measurement

noise indicates that the measurements are inaccurate and causes the filter to favor the existing motion model and react more slowly to deviations from the motion model.

The ratio between the process noise and the measurement noise determines whether the filter follows closer to the motion model, if the process noise is smaller, or closer to the measurements, if the measurement noise is smaller.

Training and Test Trajectories

A simple approach to tune a Kalman filter is to use measurements with a known ground truth and adjusting the measurement and process noise of the filter.

A constant velocity filter tuned to follow an object that has a steady speed and turns very slowly over a long distance, may not work as well when estimating an object that slows down or turns quickly. Therefore, it is important to tune the filter for the entire range of motion types you expect it to filter. It is also important to consider the expected amount of measurement noise. If you tune the filter using low-noise measurements, the filter may track changes in the motion model better. However, if you use the same tuned filter to track an object measured in a higher noise environment, the resulting track may be unduly influenced by outliers.

The following code generates a training trajectory that models the path of a vehicle travelling at 30 m/s on a highway. It also generates a test trajectory for a vehicle that has a similar speed and follows a highway of similar curvature variation.

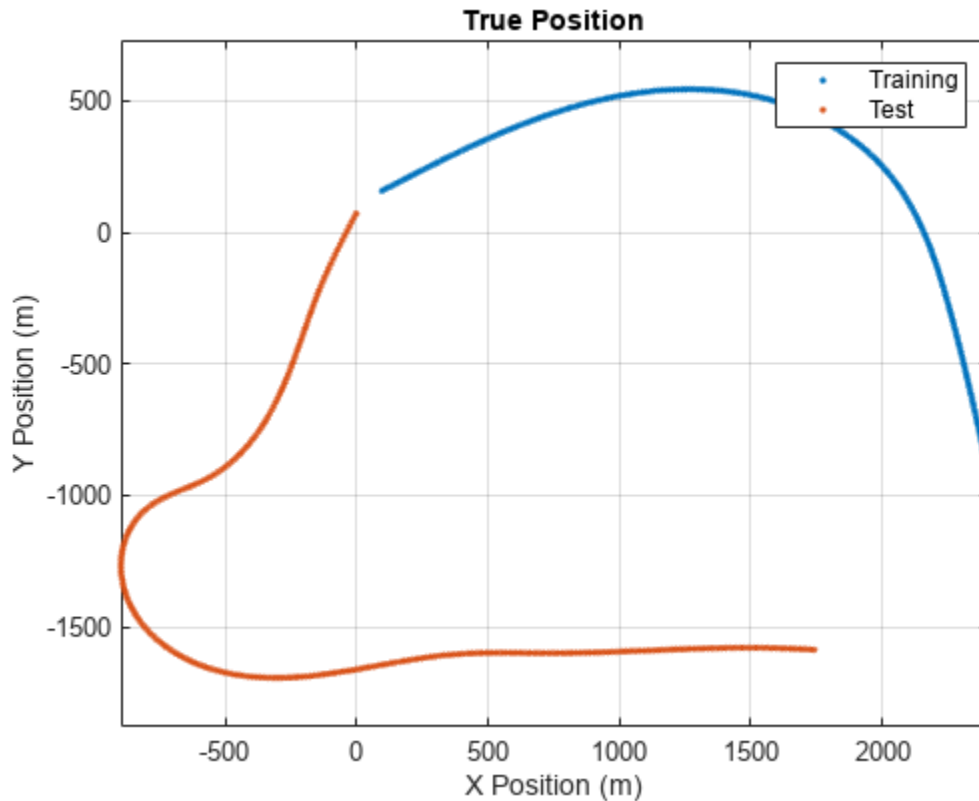
```
% Specify the training trajectory
trajectoryTrain = waypointTrajectory( ...
    [96.4 159.2 0; 2047 197 0; 2245 -248 0; 2407 -927 0], ...
    [0; 71; 87; 110], ...
    'GroundSpeed', [30; 30; 30; 30], ...
    'SampleRate', 2);

dtTrain = 1/trajectoryTrain.SampleRate;
timeTrain = (0:dtTrain:trajectoryTrain.TimeOfArrival(end));
[posTrain, ~, velTrain] = lookupPose(trajectoryTrain,timeTrain);

% Specify the test trajectory
trajectoryTest = waypointTrajectory( ...
    [-2.3 72 0; -137 -204 0; -572 -937 0; -804 -1053 0; -887 -1349 0; ...
    -674 -1608 0; 368 -1604 0; 730 -1599 0; 1633 -1581 0; 1742 -1586 0], ...
    [0; 8; 34; 42; 53; 64; 97; 107; 133; 136], ...
    'GroundSpeed', [35; 35; 34; 30; 30; 30; 35; 35; 35; 35], ...
    'SampleRate', 2);

dtTest = 1/trajectoryTest.SampleRate;
timeTest = (0:dtTest:trajectoryTest.TimeOfArrival(end));
[posTest, ~, velTest] = lookupPose(trajectoryTest,timeTest);

% Plot the trajectories
figure
plot(posTrain(:,1),posTrain(:,2),'.', ...
    posTest(:,1), posTest(:,2),'');
axis equal;
grid on;
legend('Training','Test');
xlabel('X Position (m)');
ylabel('Y Position (m)');
title('True Position')
```



Setting up the Kalman Filter

As stated above, create a Kalman Filter to use a two-dimensional motion model.

```
KF = trackingKF('MotionModel','2D Constant Velocity')
```

KF =

trackingKF with properties:

```

        State: [4x1 double]
    StateCovariance: [4x4 double]

    MotionModel: '2D Constant Velocity'
    ProcessNoise: [2x2 double]

    MeasurementModel: [2x4 double]
    MeasurementNoise: [2x2 double]

    MaxNumOOSMSteps: 0

    EnableSmoothing: 0

```

Conventions

This example contains position-only measurements of the vehicles. The constant velocity motion model, '2D Constant Velocity', has a state vector of the form: $[p_x; v_x; p_y; v_y]$, where p_x

and `py` are the positions in the x- and y-directions, respectively; and `vx` and `vy` are the velocities in the x- and y-directions, respectively.

```
trueStateTrain = [posTrain(:,1), velTrain(:,1), posTrain(:,2), velTrain(:,2)]';
trueStateTest = [posTest(:,1), velTest(:,1), posTest(:,2), velTest(:,2)]';
```

Since the truth is known for both the training and test data, you can directly simulate measurements. You obtain the position measurement by pre-multiplying the state vector by the `MeasurementModel` property of the filter. The `MeasurementModel` property, specified as the matrix, $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$, corresponds to position-only measurements. You also add measurement noise to these position measurements.

```
s = rng;
rng(2021);
posSelector = KF.MeasurementModel; % Position from state
rmsSensorNoise = 5; % RMS deviation of sensor data noise [m]

% Training Data - Normal Sensor. Position Only
truePosTrain = posSelector * trueStateTrain;
measPosTrain = truePosTrain + rmsSensorNoise * randn(size(truePosTrain))/sqrt(2);

% Test Data - Normal Sensor. Position Only
truePosTest = posSelector * trueStateTest;
measPosTest = truePosTest + rmsSensorNoise * randn(size(truePosTest))/sqrt(2);
```

Now the filter can be constructed and tuned using the training data and evaluated using the test data.

Choosing Initial Conditions

A simple way to initialize the state vector is to use the first position measurement and approximate the velocity using the first two measurements.

```
initStateTrain([1 3]) = measPosTrain(:,1);
initStateTrain([2 4]) = (measPosTrain(:,2) - measPosTrain(:,1))./dtTrain;
initStateTest([1 3]) = measPosTest(:,1);
initStateTest([2 4]) = (measPosTest(:,2) - measPosTest(:,1))./dtTest;
```

The state covariance matrix should also be initialized to account for the uncertainty in measurement. To start, initialize just the main diagonal via `diag` $[\sigma_{Px} \ \sigma_{Vx} \ \sigma_{Py} \ \sigma_{Vz}]$ and adjust the position terms to correspond to the noise of the sensor. The velocity terms have higher noise since they are based on *two* measurements of position, not one.

```
initStateCov = diag([1 2 1 2]*rmsSensorNoise.^2);
```

Choosing Process Noise

Process noise can be estimated via the expected deviation from constant velocity using a mean squared step change in velocity at each time step. Using the scalar form for process noise ensures that the components in the x- or y- directions are treated equally.

```
Qinit = var(vecnorm(diff(velTrain)./dtTrain,2,1))
Qinit = 15.2404
```

For many sensors, measurement noise is a known quantity often measured with an RMS value. Initialize the covariance to the square of this value.


```
Rinit = rmsSensorNoise.^2
```

```
Rinit = 25
```

Now you can set the process noise and measurement noise on the filter object.

```
KF.ProcessNoise = Qinit;
KF.MeasurementNoise = Rinit;
```

Initial Results

Training Set

The helper function, `evaluateFilter` sets up the filter and computes the RMS error of the predicted position against the true position of the training set:

```
errorTunedTrain = evaluateFilter(KF, initStateTrain, initStateCov, posSelector, dtTrain, measPosTrain);
```

```
errorTunedTrain = 2.9624
```

Compare the filtered position error against the uncorrected raw measurements of the training set:

```
rms(vecnorm(measPosTrain - truePosTrain,2,1))
```

```
ans = 4.7319
```

It is clear that the filter obtained a much better position estimate.

Test Set

In the test trajectory, the vehicle turns a little bit more and has a few acceleration changes. Therefore, it is expected that the filter estimate may not be as good as that of the training trajectory.

```
errorTunedTest = evaluateFilter(KF, initStateTest, initStateCov, posSelector, dtTest, measPosTest);
```

```
errorTunedTest = 3.7157
```

Nevertheless, it still is better than just using the raw measurements alone.

```
rms(vecnorm(measPosTest - truePosTest,2,1))
```

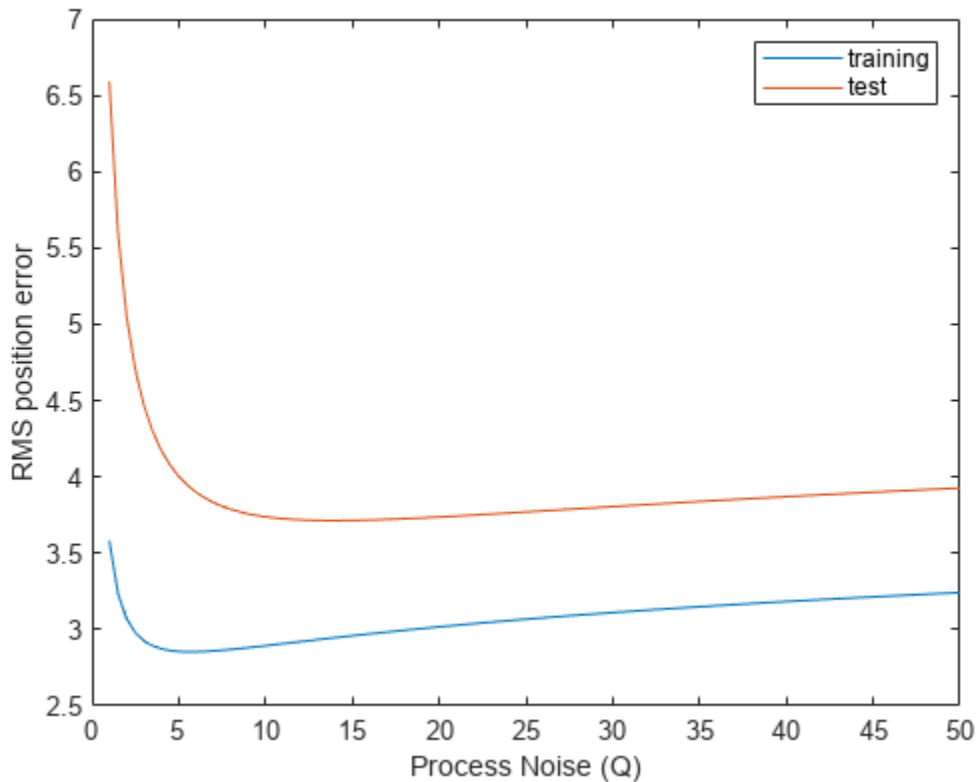
```
ans = 5.2553
```

Tuning the Filter

You can sweep through a few process noise values and determine a desirable value for both the training and test cases.

```
nSweep = 100;
Qsweep = linspace(1,50,nSweep);
errorTunedTrain = zeros(1,nSweep);
errorTunedTest = zeros(1,nSweep);
for i = 1:nSweep
    KF.ProcessNoise = Qsweep(i);
    errorTunedTrain(i) = evaluateFilter(KF, initStateTrain, initStateCov, posSelector, dtTrain, measPosTrain);
    errorTunedTest(i) = evaluateFilter(KF, initStateTest, initStateCov, posSelector, dtTest, measPosTest);
end
plot(Qsweep,errorTunedTrain,'-',Qsweep,errorTunedTest,'-');
legend("training","test")
```

```
xlabel("Process Noise (Q)")
ylabel("RMS position error");
```



As seen in the above plot and in the code below, the training set has a local minimum when setting Q to 5.4 for the training set, and 13 for the test set.

```
[minErrorTunedTrain, iMinTrain] = min(errorTunedTrain);
[minErrorTunedTest, iMinTest] = min(errorTunedTest);
minErrorTunedTrain
```

```
minErrorTunedTrain = 2.8540
```

```
Qsweep(iMinTrain)
```

```
ans = 5.4545
```

```
minErrorTunedTest
```

```
minErrorTunedTest = 3.7141
```

```
Qsweep(iMinTest)
```

```
ans = 13.8687
```

Variations in the optimal point are expected and are due to the differences in the trajectories. With a small amount of speed differences and more turns, the test set is expected to have a larger predicted error. A common way to mitigate differences between training and test data is to use a Monte Carlo simulation involving many training trajectories like the test trajectories.

Automated Tuning

Sometimes measurement parameters such as measurement noise may not be known. In some cases, it is helpful to consider the problem as an optimization problem where you seek to minimize the RMS distance error of the training set over the set of input parameters Q and R.

If the measurement noise is unknown, you can initialize it by comparing the measurements against the true states.

```
n = length(timeTrain);
measErr = measPosTrain - posSelector * trueStateTrain;
sumR = norm(measErr);
Rinit = sum(vecnorm(measErr,2).^2)./(n+1)

Rinit = 22.2895
```

After initializing the measurement noise, you then use an optimization solver to perform the tuning.

In this case, use the `fminunc` function, which finds a local minimum of a function of an unconstrained parameter vector.

Parameter Vector Construction

Since `fminunc` works by iteratively changing a parameter vector, you use the `constructParameterVector` helper function to map the process and measurement covariances into a single vector. See the Supporting Functions section in the end for more details.

```
initialParams = constructParameterVector(Qinit, Rinit);
```

Optimization Function Construction

To run the optimization solver, construct the function that evaluates the cost based on these parameters. Create a function, `measureRMSError`, which runs the EKF and evaluates the root mean squared error of the filtered state against the ground truth. The function uses the noise parameters, initial conditions, measured positions, and ground truth as inputs. For more information, see the Supporting Functions section.

Since the `fminunc` function requires a function that just uses a single parameter vector, the minimization function is wrapped inside an anonymous function that captures the EKF, the training measurement, the truth data, and other variables needed to run the tracker:

```
func = @(noiseParams) measureRMSError(noiseParams, KF, initStateTrain, initStateCov, posSelector
```

Finding the Parameters

Now that all the arguments to `fminunc` are properly initialized, you can find the optimal parameters.

```
optimalParams = fminunc(func,initialParams);
```

```
Local minimum possible.
```

```
fminunc stopped because it cannot decrease the objective function
along the current search direction.
```

Deconstructing the Parameter Vector

Since the two parameters are in a vector form, convert them to the process noise and measurement noise covariance matrices using the `extractNoiseParameters` function:

```
[QautoTuned,RautoTuned] = extractNoiseParameters(optimalParams)
QautoTuned = 0.5671
RautoTuned = 2.5108
```

Notice that the two values differ significantly from their "true" values. This is because it really is the ratio between Q and R that matters, not their actual values.

Evaluating Results

Now that you have the optimized covariance matrices that minimize the residual prediction error, you can initialize the EKF with them and evaluate the results in the same manner as before:

```
KF.ProcessNoise = QautoTuned;
KF.MeasurementNoise = RautoTuned;
autoTunedRMSErrorTrain = evaluateFilter(KF, initStateTrain, initStateCov, posSelector, dtTrain, measPos, truePos);
autoTunedRMSErrorTrain = 2.8525

autoTunedRMSErrorTest = evaluateFilter(KF, initStateTest, initStateCov, posSelector, dtTest, measPos, truePos);
autoTunedRMSErrorTest = 3.9313
```

Summary

In this example, you learned how to tune process noise and measurement noise of a constant velocity Kalman filter using ground truth and noisy measurements. You also learned how to use the optimization solver, `fminunc`, to find optimal values for the process and measurement noise parameters.

Supporting Functions

Evaluating the Kalman Filter

The `evaluateFilter` function evaluates the distance and Euclidean error of a Kalman filter with the given initial conditions, measurements, and ground truth data. The root-mean-square Euclidean error measures how far away the typical measurement is from the training data.

```
function tunedRMSE = evaluateFilter(KF, initState, initStateCov, posSelector, dt, measPos, truePos)
    initialize(KF, initState, initStateCov);
    estPosTuned = zeros(2,size(measPos,2));
    magPosError = zeros(1,size(measPos,2));

    for i = 2:size(measPos,2)
        predict(KF, dt);
        x = correct(KF, measPos(:,i));
        estPosTuned(:,i) = posSelector * x(:);
        magPosError(i) = norm(estPosTuned(:,i) - truePos(:,i));
    end
    tunedRMSE = rms(magPosError(10:end));
end
```

Minimization Functions

Parameter Vector to Noise Matrices Conversions

The `constructParameterVector` function converts the noise covariances into a two-element column vector, where the first element of the vector is the square root of the process noise and the second element is the square root of the measurement noise.

```
function vector = constructParameterVector(Q,R)
vector = sqrt([Q;R]);
end
```

The `constructParameterMatrices` function converts the two-element parameter vector, `v`, back to the covariance matrices. The first element is used to construct the process noise. The second element is used to construct the measurement noise. Squaring these numbers ensures that the noise values are always positive.

```
function [Q,R] = extractNoiseParameters(vector)
Q = vector(1).^2;
R = vector(2).^2;
end
```

Minimizing Residual Prediction Error

The `measureRMSError` function takes the noise parameters, the initial conditions, the measured positions, runs the Kalman filter and evaluates the root mean squared error.

```
function rmse = measureRMSError(noiseParams, KF, initState, initCov, posSelector, dt, measPos, t)
[Qtest, Rtest] = extractNoiseParameters(noiseParams);

KF.ProcessNoise = Qtest;
KF.MeasurementNoise = Rtest;

rmse = evaluateFilter(KF, initState, initCov, posSelector, dt, measPos, truePos);
end
```

Object Tracking Using Time Difference of Arrival (TDOA)

This example shows how to track objects using time difference of arrival (TDOA). This example introduces the challenges of localization with TDOA measurements as well as algorithms and techniques that can be used for tracking single and multiple objects with TDOA techniques.

Introduction

TDOA positioning is a passive technique to localize and track emitting objects by exploiting the difference of signal arrival times at multiple, spatially-separated receivers. Given a signal emission time (t_e) from the object and the propagation speed (c) in the medium, the time-of-arrival (TOA) of the signal at 2 receivers located at ranges r_1 and r_2 respectively from the object can be denoted as:

$$t_1 = t_e + \frac{r_1}{c}$$

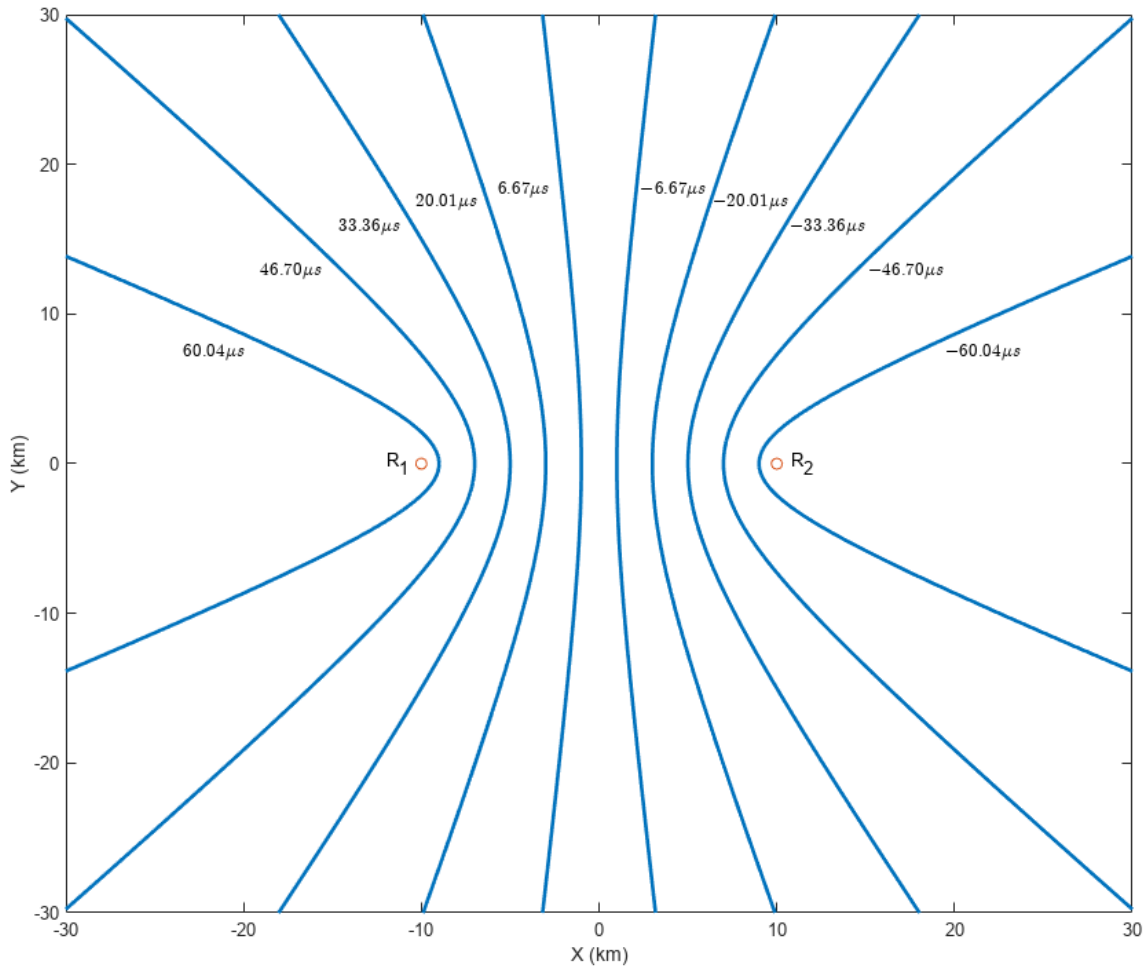
$$t_2 = t_e + \frac{r_2}{c}$$

As the true signal emission time is typically not known, a difference between t_1 and t_2 can be used to obtain some information about the location of an object. Specifically, given the location of each receiver, the time-difference measurement corresponds to a difference in ranges from the two receivers to the unknown object.

$$\sqrt{(x_t - x_1)^2 + (y_t - y_1)^2 + (z_t - z_1)^2} - \sqrt{(x_t - x_2)^2 + (y_t - y_2)^2 + (z_t - z_2)^2} = c(t_1 - t_2)$$

where $[x_t \ y_t \ z_t]$ is the unknown object location, $t_1 - t_2$ is the TDOA measurement, and $[x_1 \ y_1 \ z_1]$, $[x_2 \ y_2 \ z_2]$ are the locations of two receivers. This non-linear relationship between TDOA and object location represents a hyperbola in 2-D or a hyperboloid in 3-D coordinates with two receivers at the foci. The image below shows the hyperbolae of an object for different TDOA measurements ($c =$ speed of light). These curves are typically called *constant TDOA curves*. Notice that using the sign of a TDOA measurement, you can constrain the object location to a single branch of the hyperbola.

```
HelperTDOATrackingExampleDisplay.plotConstantTDOACurves;
```



TDOA Calculation

There are two primary methods used to calculate the TDOA measurement from the signal of an object. In the first method, each receiver measures the absolute time instant of signal arrival (time-of-arrival or TOA) as defined by t_i above. A difference in time-of-arrivals between two receivers is then calculated to obtain the TDOA measurement. This method is applicable when certain signal attributes are known a-priori and the leading edge of the signal can be detected by the receiver.

$$\text{TDOA}_{12} = \text{TOA}_2 - \text{TOA}_1$$

In the second method, each receiver records and transmits the received signal to a shared processing hub. A cross-correlation between signals received from two receivers is calculated at the processing hub. The TDOA is estimated to be the delay which maximizes the cross-correlation between the two signals.

$$\text{TDOA}_{12} = \underset{t \in [-T_{\max}, T_{\max}]}{\operatorname{argmax}} ([S_1 \star S_2](t))$$

where $[S_1 \star S_2](t)$ represents the cross-correlation between the signals at receivers as a function of time delay, t . T_{\max} is the maximum possible absolute delay and can be calculated as $\frac{D}{c}$, where D is the distance between the receivers.

Both methods of TDOA calculation require synchronized clocks at the receivers. In practical applications, you typically achieve this using the Global Positioning System (GPS) clock.

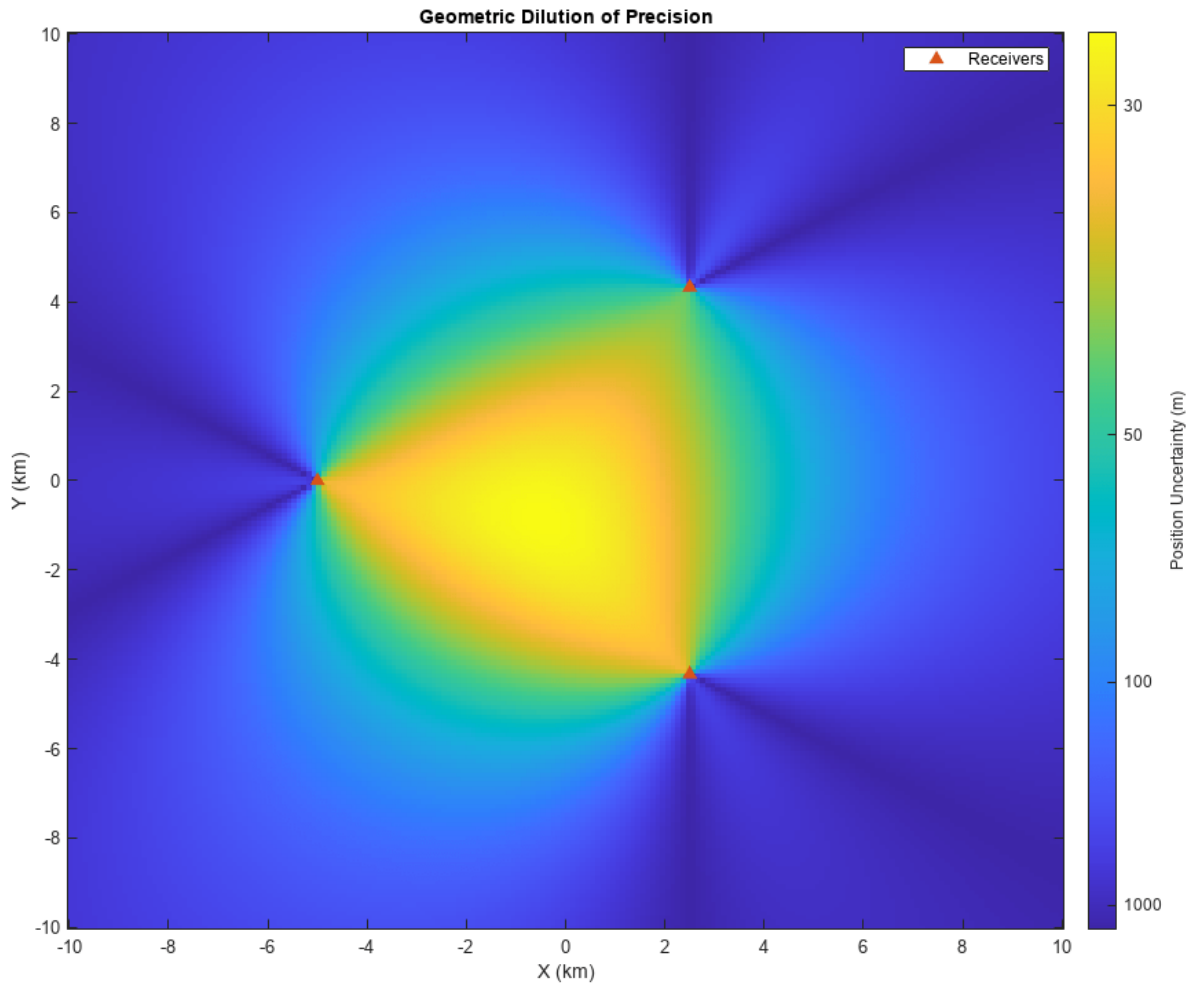
TDOA Localization

As the TDOA between two receivers localizes an object to a hyperbola or hyperboloid, it is not possible to observe the full state of the object by using only two stationary receivers. For 2-D localization, at least 3 spatially-separated receivers are required to estimate the object state. Similarly, for 3-D tracking, at least 4 spatially-separated receivers are required.

With N ($N \geq 2$) receivers, a total of $N\binom{N-1}{2}$ TDOA measurements from an object can be obtained by calculating the time difference of arrival using each combination of receiver. However, out of these measurements, only $N - 1$ measurements are independent and the rest of the TDOA measurements can be formulated as a linear combination of these independent measurements. Given these $N - 1$ measurements from the object, an estimation algorithm is typically used to locate the position of the object. In this example, you use the spherical intersection algorithm [1] to find the position and uncertainty in position estimate. The spherical intersection algorithm assumes that all $N - 1$ TDOAs are calculated with respect to the same receiver, sometimes referred to as the *reference receiver*.

The accuracy or uncertainty in the estimated position obtained from localization depends on the geometry of the network. At regions, where small errors in TDOA measurements results in large errors in estimated position, the accuracy of the estimated position is lower. This effect is called *geometric dilution of precision (GDOP)*. The image below shows the GDOP heat map of the network geometry for a 2-D scenario with 3 receivers. Note that the accuracy is high within the envelope generated by the network and is lowest directly behind the receivers along the receiver-to-receiver line of sights.

```
HelperTDOATrackingExampleDisplay.plotGDOPAccuracyMap;
```

Tracking a Single Emitter

In this section, you simulate TDOA measurements from a single object and use them to localize and track the object in the absence of any false alarms and missed detections.

You define a 2-D scenario using the helper function, `helperCreateSingleTargetTDOAScenario`. This function returns a `trackingScenarioRecording` object representing a single object moving at a constant velocity. This object is observed by three stationary, spatially-separated receivers. The helper function also returns a matrix of two columns representing the pairs formed by the receivers. Each row represents the `PlatformID` of the platforms that form the TDOA measurement.

```
% Reproducible results
rng(2021);
```

```
% Setup scenario with 3 receivers
numReceivers = 3;
[scenario, rxPairs] = helperCreateSingleTargetTDOAScenario(numReceivers);
```

You then use the helper function, `helperSimulateTDOA`, to simulate TDOA measurements from the objects. The helper function allows you to specify the measurement accuracy in the TDOA. You use a measurement accuracy of 100 nanoseconds to represent timing inaccuracy in the clocks as well as TDOA estimation. The emission speed and units of the reported time are speed of light in vacuum and nanoseconds respectively, specified using the `helperGetGlobalParameters` function. The `helperSimulateTDOA` function outputs TDOA detections as a cell array of `objectDetection` objects.

```
tdoaDets = helperSimulateTDOA(scenario, rxPairs, measNoise); % Specify measurement noise variance
```

Each TDOA detection is represented using the `objectDetection` according to the following format:

Property	Description
Time	Detection timestamp in seconds
Measurement	Time-difference-of-arrival (TDOA) in nanoseconds
MeasurementNoise	Uncertainty in TDOA in nanoseconds ²
SensorIndex	Unique identifier of a receiver pair
MeasurementParameters	A 2-element struct array with field <code>OriginPosition</code> . The first element describes the position of first receiver and the second element describes the position of reference receiver. These positions are defined in a global Cartesian frame.

Next, you run the scenario and generate TDOA detections from the object. You use the helper function, `helperTDOA2Pos`, to estimate the position and position covariance of the object at every step using the spherical intersection algorithm [1]. You further filter the position estimate of the object using a global nearest neighbor (GNN) tracker configured using the `trackerGNN` System object™ with a constant-velocity linear Kalman filter. To account for the high speed of the object, this constant velocity Kalman filter is configured using the helper function, `helperInitHighSpeedKF`.

```
% Define accuracy in measurements
measNoise = (100)^2; % nanoseconds^2

% Display object
display = HelperTDOATrackingExampleDisplay(XLimits=[-1e4 1e4],...
                                           YLimits=[-1e4 1e4],...
                                           LogAccuracy=true,...
                                           Title="Single Object Tracking");

% Create a GNN tracker
tracker = trackerGNN(FilterInitializationFcn=@helperInitHighSpeedKF,...
                    AssignmentThreshold=100);

while advance(scenario)
    % Elapsed time
    time = scenario.SimulationTime;

    % Simulate TDOA detections without false alarms and missed detections
    tdoaDets = helperSimulateTDOA(scenario, rxPairs, measNoise);

    % Get estimated position and position covariance as objectDetection
```

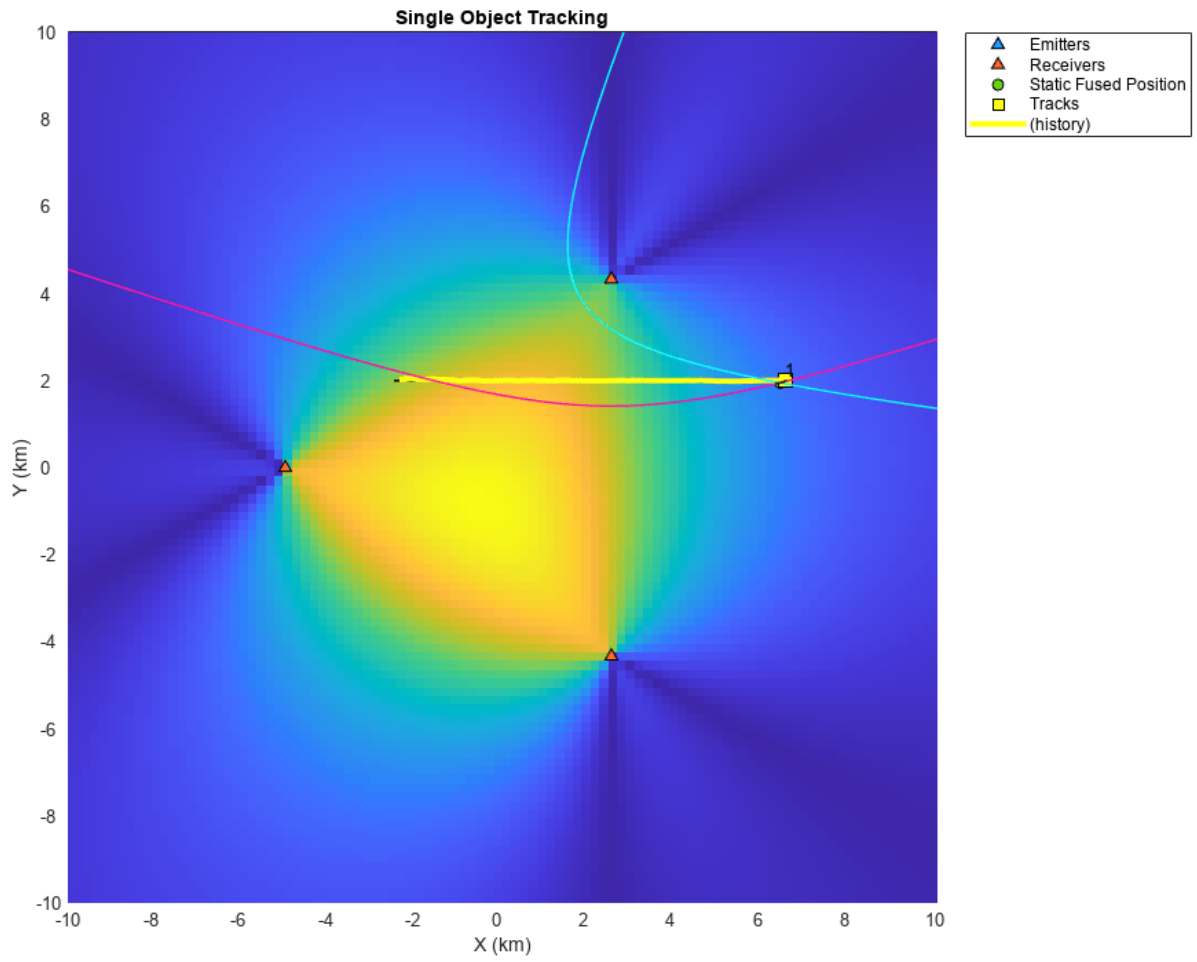
```
% objects
posDet = helperTDOA2Pos(tdoaDets,true);

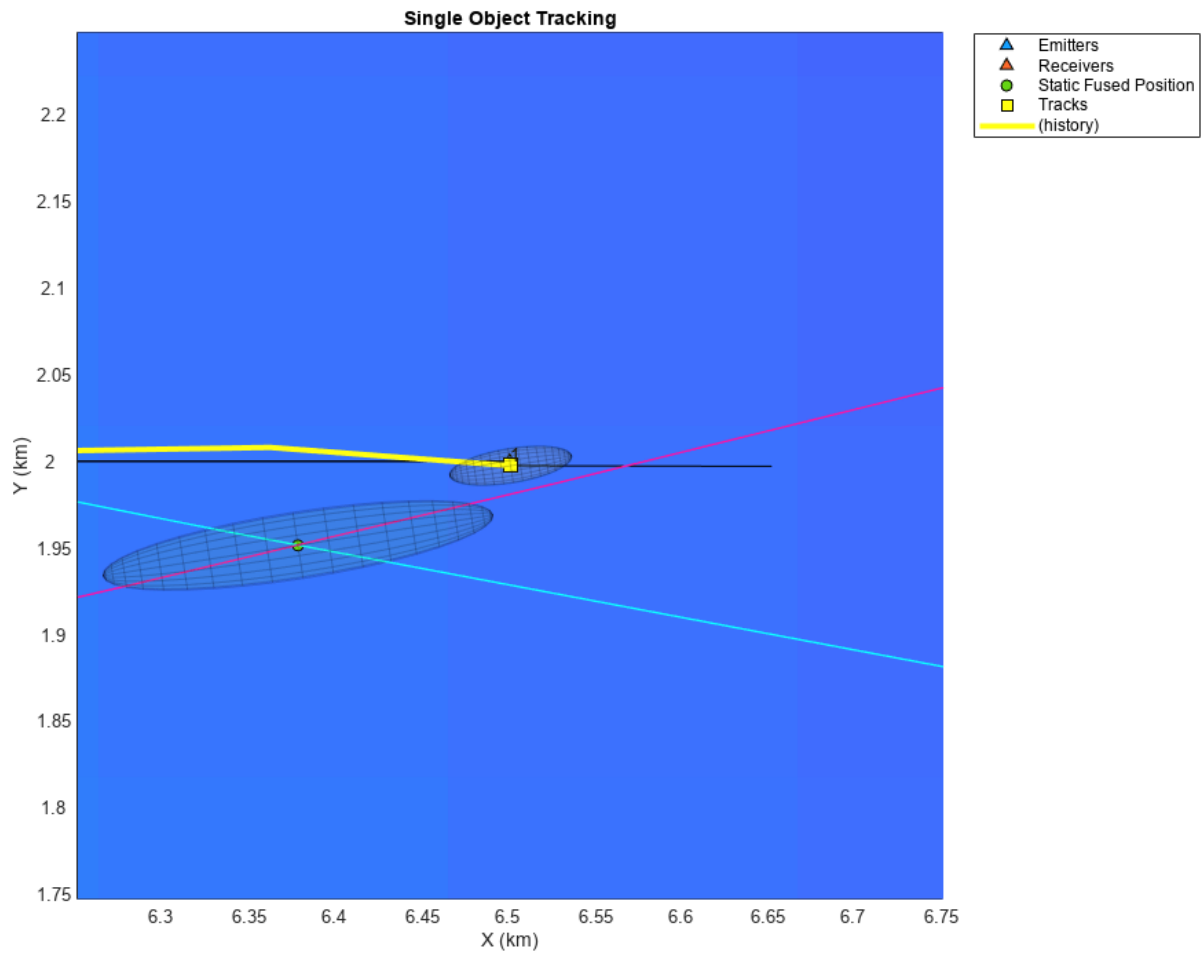
% Update the tracker with position detections
tracks = tracker(posDet, time);

% Display results
display(scenario, rxPairs, tdoaDets, {posDet}, tracks);
end
```

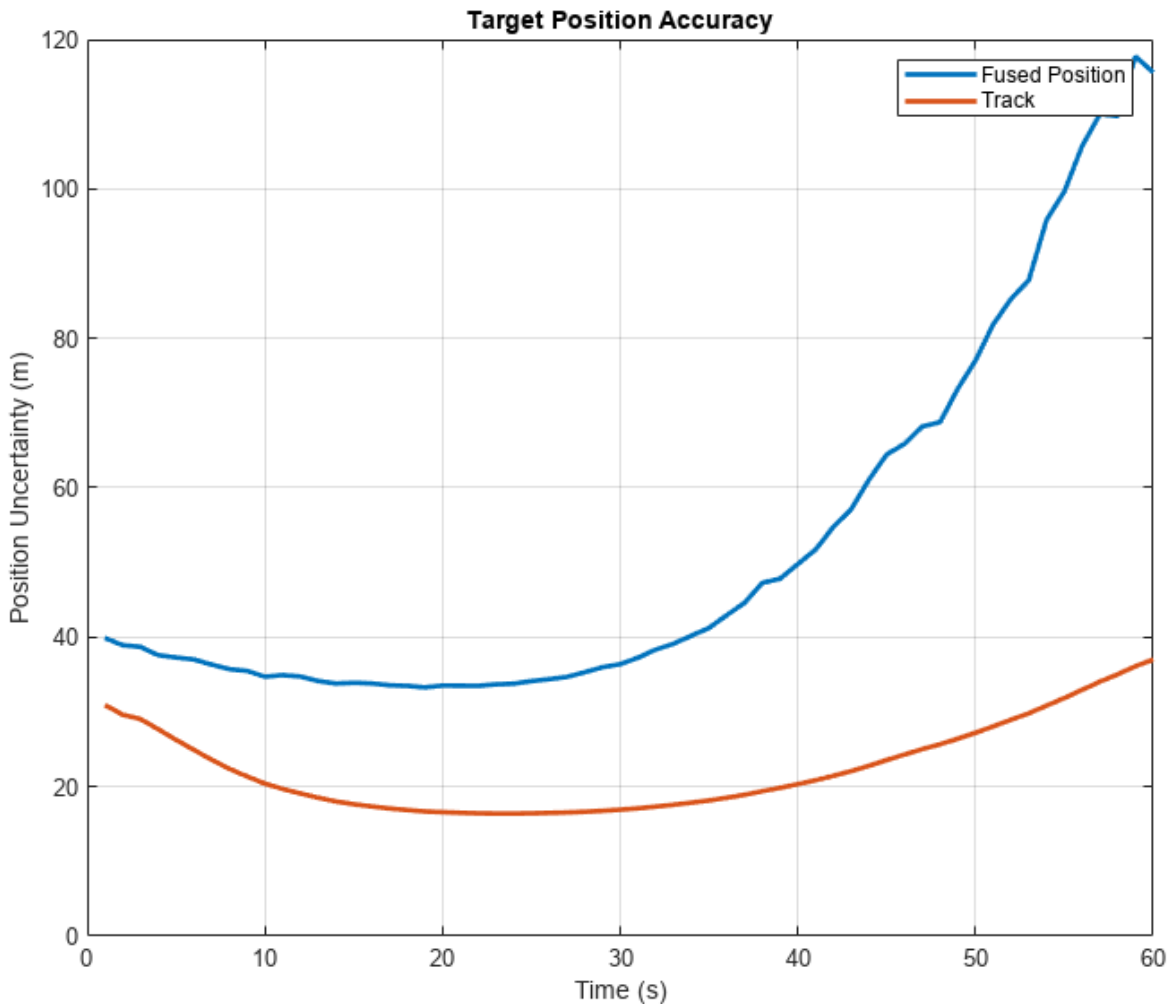
Notice that the tracker is able to maintain a track on the object. The the estimated position of the object lies close to the intersection of hyperbolic curves created by each receiver pair. Also notice in the zoomed plot below that the filtered estimate of the object has less uncertainty as compared to the fused estimate at every step. As the object moves towards positive X axis, the error in estimated position increases due to geometric dilution of precision. Notice that the tracker is able to provide an improved estimate of the object by filtering the position estimate of the object using the Kalman filter with constant velocity motion model.

```
zoomOnTrack(display,tracks(1).TrackID);
```





```
plotDetectionVsTrackAccuracy(display);
```



Tracking Multiple Emitters with Known IDs

In the previous section, you learned how to use TDOA measurements from a single object to generate positional measurement of the object. In practical situations, the scenario may consist of multiple objects.

The main challenge in multi-object TDOA tracking is the estimation of TDOA for each target. In the presence of multiple objects, each receiver receives signals from multiple objects. In the first method for TDOA calculation, each receiver records multiple time-of-arrival measurements. The unknown data association between time-of-arrival measurements reported by each receiver results in many possible TDOA combinations. In the second method of TDOA calculation, the signal cross-correlation function between receiver signals results in multiple peaks corresponding to true objects as well as false alarms. In certain applications, this unknown data association between receivers can be easily solved at the signal level. For example, if the signal encoding is known a-priori, such as in wireless communications signals like LTE, 5G signals or aviation communication signals like ADSB signals, the receiver is typically able to calculate both the time-of-arrival and the unique identity of the object.

Similarly, if the objects are separated in their carrier frequencies, a separate TDOA calculation can be added for each object in their own frequency bands.

In this section, you assume that signal-level data association is performed at the receiver. This helps the processing hub to form TDOA measurements per identified object without ambiguity. To simulate TDOA measurements with known emitter identification, you provide a fourth input argument to the helper function, `helperSimulateTDOA`. The function outputs the unique identity of the emitter as the `objectClassID` property of the `objectDetection` object. You use this object identity, shared between TDOAs from multiple receiver-pairs, to fuse detections from each object separately and obtain their respective positions as well as uncertainties. Then, you use these fused measurements to track these objects using the GNN tracker.

```
% Create the multiple object scenario
numReceivers = 3;
numObjects = 4;
[scenario, rxPairs] = helperCreateMultiTargetTDOAScenario(numReceivers, numObjects);

% Reset display
release(display);
display.LogAccuracy = false;
display.Title = 'Tracking Multiple Emitters with Known IDs';

% Release tracker
release(tracker);

while advance(scenario)
    % Elapsed time
    time = scenario.SimulationTime;

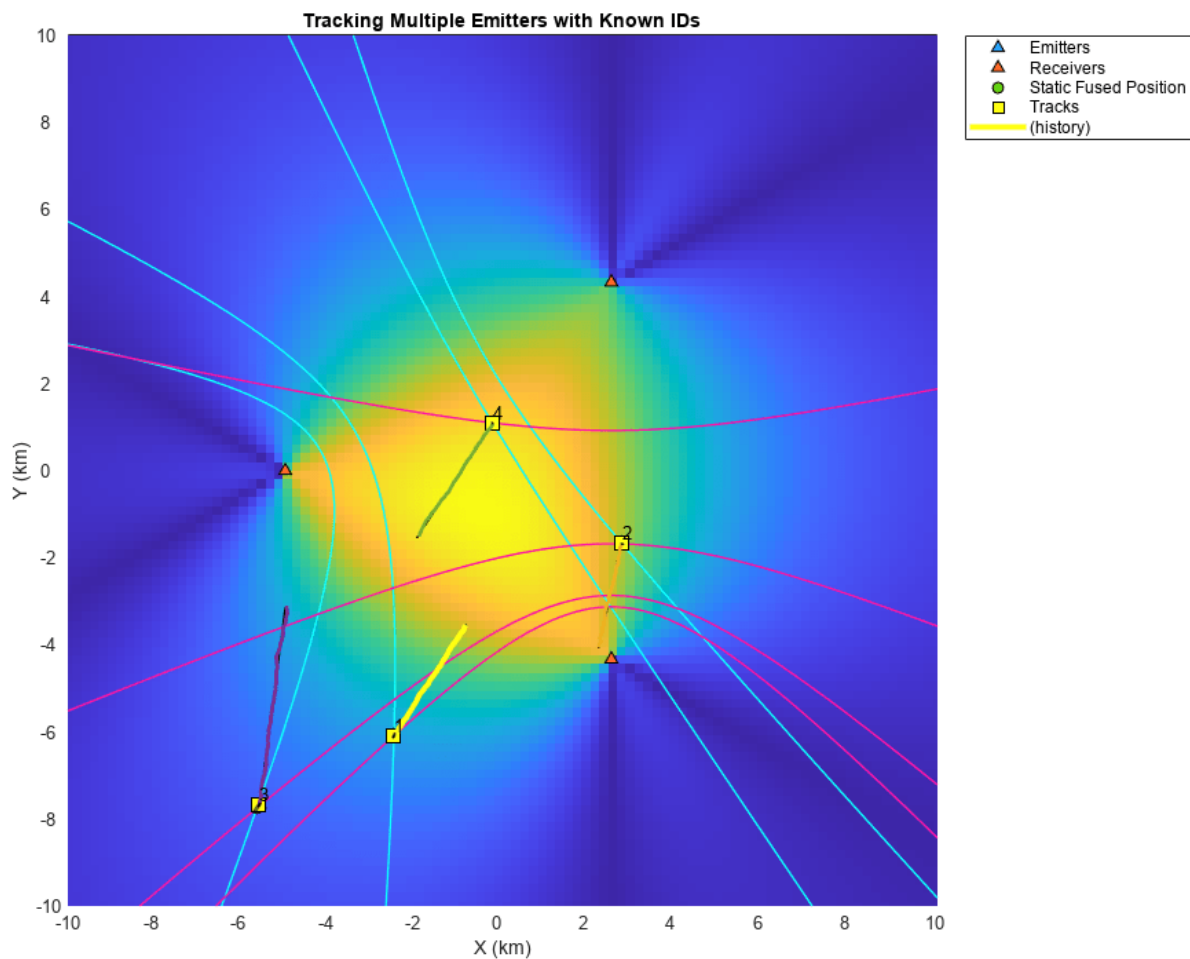
    % Simulate classified TDOA detections
    classifiedTDOADets = helperSimulateTDOA(scenario, rxPairs, measNoise, true);

    % Find unique object identities
    tgtIDs = cellfun(@(x)x.ObjectClassID,classifiedTDOADets);
    uniqueIDs = unique(tgtIDs);

    % Estimate the position and covariance of each emitter as objectDetection
    posDets = cell(numel(uniqueIDs),1);
    for i = 1:numel(uniqueIDs)
        thisEmitterTDOADets = classifiedTDOADets(tgtIDs == uniqueIDs(i));
        posDets{i} = helperTDOA2Pos(thisEmitterTDOADets, true);
    end

    % Update tracker
    tracks = tracker(posDets, time);

    % Update display
    display(scenario, rxPairs, classifiedTDOADets, posDets, tracks);
end
```



Note that the tracker is able to maintain a track on all 4 objects when TDOA-to-TDOA association is assumed to be known. When the data association is provided by the receivers, the multi-object TDOA estimation is simplified to generating multiple TDOA detections with known single-object associations. The multi-object tracking problem is similarly simplified because there are no false TDOA detections.

Tracking Multiple Emitters with Unknown IDs

In this section, you assume that data association between receiver data is not known a-priori i.e., receivers cannot identify objects. In the absence of this data association, each intersection between blue and maroon hyperbolic curves shown in the plot from the Tracking Multiple Emitters with Known IDs on page 6-957 section could be a possible object location. Therefore, when data association is not available from the receivers, associating time-of-arrival (TOA) or TDOA detections from multiple receivers is prone to detections from *ghost* objects. To separate the ghost detections from true object detections, you must add more receivers to reduce the ambiguity. In this section, you will use a static fusion algorithm [2] to determine the unknown data association and estimate position measurements from each object. You will use this static fusion algorithm for systems that transmit

multiple time-of-arrival measurements to the processing hub as well as systems that transmit recorded signals to the hub and calculates TDOA before processing them for object tracking.

Tracking with Time-of-Arrival (TOA) Measurements

In this section, you configure a system in which each receiver transmits multiple time of arrival (TOA) measurements to the central processing hub. To reduce ambiguity in data association and to reduce the number of potential ghost objects, you use 4 stationary receivers to localize and track the objects.

To simulate time-of-arrival measurements from the objects, you use the helper function, `helperSimulateTOA`, to simulate time-of-arrival measurements from multiple objects. These time-of-arrival measurements record the exact time instant in a global reference clock at which the signal was received by the receivers. To simulate exact time instants, the helper function uses the scenario time as the global clock time and assumes that the objects emit signals at the time instant of scenario. Note that the algorithm used to track with these measurements is not tied to this assumption as the exact emission time from the object is not known to the receivers.

You specify the `nFalse` input as 1 to simulate one false time-of-arrival measurement from each receiver. You also specify the `Pd` input, which defines the probability of detecting a true time-of-arrival measurement by each receiver, as 0.95.

```
toaDets = helperSimulateTOA(scenario, receiverIDs, measNoise, nFalse, Pd);
% receiverIDs - PlatformID of all receivers
% measNoise - Accuracy in TOA measurements (ns^2)
% nFalse - number of false alarms per receiver
% Pd - Detection probability for receivers
```

Here is the format of time-of-arrival (TOA) detections.

Property	Description
Time	Detection timestamp in seconds
Measurement	Time-of-arrival (TOA) in nanoseconds
MeasurementNoise	Uncertainty in TOA in nanoseconds ²
SensorIndex	Unique identifier of the receiver
MeasurementParameters	A scalar struct with field <code>OriginPosition</code> representing the position of the receiver in a global Cartesian frame

To fuse multiple time-of-arrival measurements from each receiver, you use the static fusion algorithm by configuring a `staticDetectionFuser` object. The `staticDetectionFuser` object determines the best data association between time of arrival measurements and provides fused detections from possible objects. The static fusion algorithm requires two sub routines or functions. The first function (`MeasurementFusionFcn`) allows the algorithm to estimate a fused measurement from object, given a set of time-of-arrival measurements assumed to originate from the same object. The second function (`MeasurementFcn`) allows the algorithm to obtain the expected time-of-arrival measurement from the fused measurement. Note that to define a measurement function to obtain time-of-arrival measurement, the fused measurement must contain an estimate of the signal emission time from the object. Therefore, you define the fused measurement as $[x_t \ y_t \ z_t \ t_e]$, where t_e is the time instant at which the object emitted the signal.

You use the helper function, `helperTOA2Pos`, to estimate fused measurement - position and emission time - of the object. You also use the helper function, `helperMeasureTOA`, to define the measurement model that calculates time-of-arrival measurement from the fused measurement. The fusion algorithm outputs fused detections as a cell array of `objectDetection` objects. Each element of the cell array defines an `objectDetection` object containing measurement from a potential object as its position and emission time. You can speed-up this static fusion algorithm by specifying the `UseParallel` property to `true`. Specifying `UseParallel` as `true` requires the Parallel Computing Toolbox™.

```
% Create scenario
[scenario, ~, receiverIDs] = helperCreateMultiTargetTDOAScenario(4, 4);

% Specify stastics of TOA simulation
measNoise = 1e4; % 100 ns per receiver
nFalse = 1; % 1 false alarm per receiver per step
Pd = 0.95; % Detection probability of true signal

% Release display
release(display);
display.Title = 'Tracking Using TOA Measurements with Unknown IDs';

% Release tracker
release(tracker);
tracker.ConfirmationThreshold = [4 6];

% Use Parallel Computing Toolbox if available
useParallel = false;

% Define fuser.
toaFuser = staticDetectionFuser(MaxNumSensors=4,...
    MeasurementFormat='custom',...
    MeasurementFcn=@helperMeasureTOA,...
    MeasurementFusionFcn=@helperTOA2Pos,...
    FalseAlarmRate=1e-8,...
    DetectionProbability=Pd,...
    UseParallel=useParallel);

while advance(scenario)
    % Current time
    time = scenario.SimulationTime;

    % Simulate TOA detections with false alarms and missed detections
    toaDets = helperSimulateTOA(scenario, receiverIDs, measNoise, Pd, nFalse);

    % Fuse TOA detections to estimate position amd emission time of
    % unidentified and unknown number of emitters.
    [posDets, info] = toaFuser(toaDets);

    % Remove emission time before feeding detections to the tracker as it
    % is not tracked in this example.
    for i = 1:numel(posDets)
        posDets{i}.Measurement = posDets{i}.Measurement(1:3);
        posDets{i}.MeasurementNoise = posDets{i}.MeasurementNoise(1:3,1:3);
    end

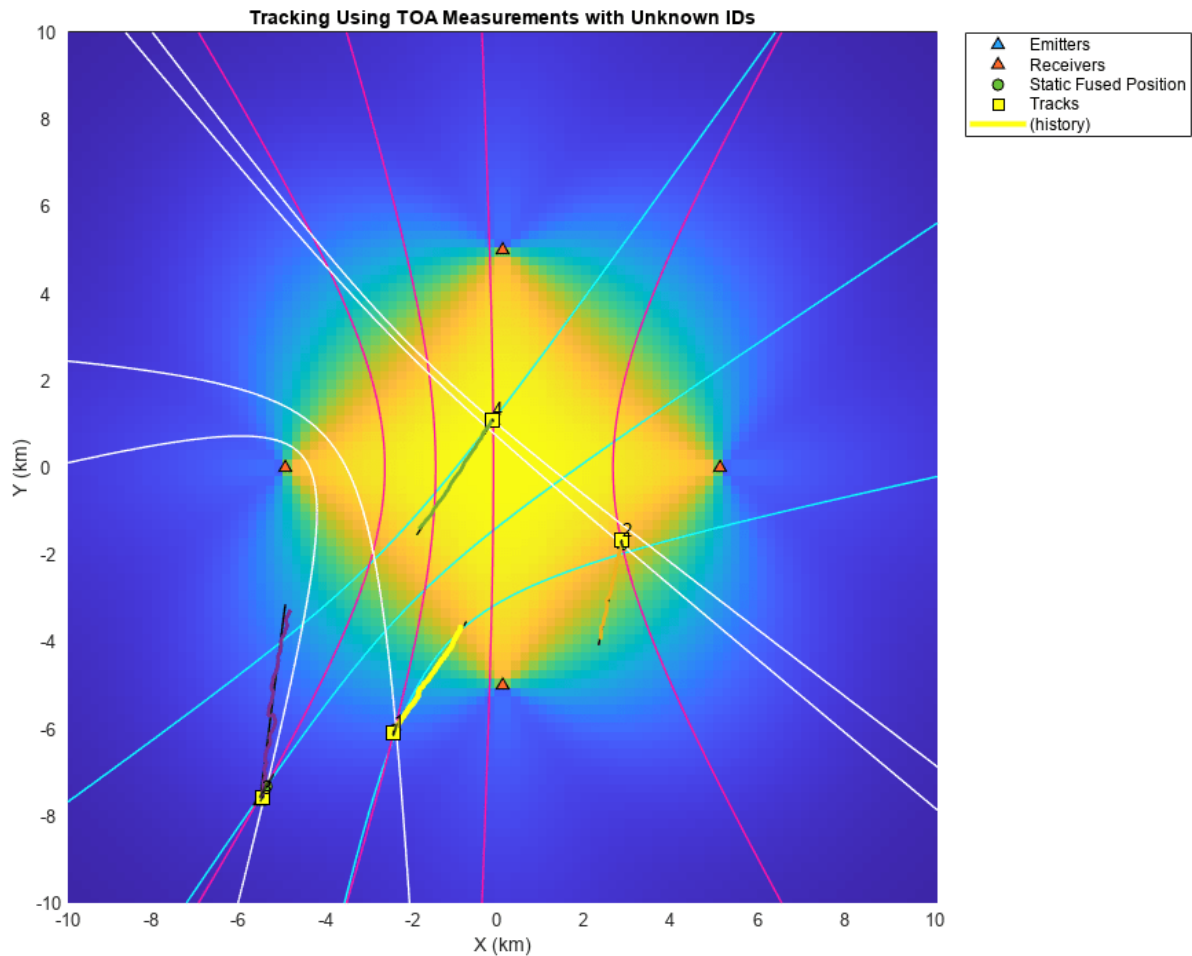
    % Update tracker. The target emission is assumed at timestamp, time.
    % The timestamp of the detection's is therefore time plus signal
```

```

% propagation time. The tracker timestamp must be greater than
% detection time. Use time + 0.05 here. In real-world, this is
% absolute timestamp at which tracker was updated.
tracks = tracker(posDets, time + 0.05);

% Update display
% Form TDOA measurements from assigned TOAs for visualization
tdoaDets = helperFormulateTDOAs(toaDets,info.Assignments);
display(scenario, receiverIDs, tdoaDets, posDets, tracks);
end

```



Notice that the fuser algorithm is able to estimate the position of the object accurately most of the time. However, due to the presence of false alarms and missed measurements, the fusion algorithm may pick a wrong data association at some steps. This can lead to detections from ghost intersections. The tracker assists the static fusion algorithm by discarding these wrong associations as false alarms.

Tracking with Time-difference-of-Arrival (TDOA) Measurements

In this section, you configure the tracking algorithm for a system where TDOAs are formed from the receiver-pairs from multiple objects without any emitter identification. The calculation of TDOA from receiver pairs in a multi-object scenario may generate a few false alarms due to false peaks in the cross-correlation function. To simulate the TDOA measurements from such system, you increase the number of false alarms per receiver pair to 2.

The static fusion algorithm is configured similar to the previous section by using the `staticDetectionFuser` object. In contrast to the time-of-arrival fusion, here you use the measurement fusion function, `helperFuseTDOA`, to fuse multiple TDOAs into a fused measurement. You also use the helper function, `helperMeasureTDOA`, to define the transformation from fused measurement to TDOA measurement. As TDOA measurements do not contain or need information about the true signal emission time, you define the fused measurement as $[x_t \ y_t \ z_t]$. You set the `MaxNumSensors` to 3 as four receivers form three TDOA pairs.

```
% Create scenario
[scenario, rxPairs] = helperCreateMultiTargetTDOAScenario(4, 4);

% Measurement statistics
measNoise = 1e4; % 100 ns per receiver
nFalse = 2; % 2 false alarms per receiver pair
Pd = 0.95; % Detection probability per receiver pair

% Release display
release(display);
display.Title = 'Tracking Using TDOA Measurements with Unknown IDs';

% Release tracker
release(tracker);

% Define fuser.
tdoaFuser = staticDetectionFuser(MaxNumSensors=3,...
    MeasurementFormat='custom',...
    MeasurementFcn=@helperMeasureTDOA,...
    MeasurementFusionFcn=@helperTDOA2Pos,...
    DetectionProbability=Pd,...
    FalseAlarmRate=1e-8,...
    UseParallel=useParallel);

while advance(scenario)
    % Current elapsed time
    time = scenario.SimulationTime;

    % Simulate TDOA detections with false alarms and missed detections
    tdoaDets = helperSimulateTDOA(scenario, rxPairs, measNoise, false, Pd, nFalse);

    % Fuse TDOA detections to estimate position detections of unidentified
    % and unknown number of emitters
    posDets = tdoaFuser(tdoaDets);

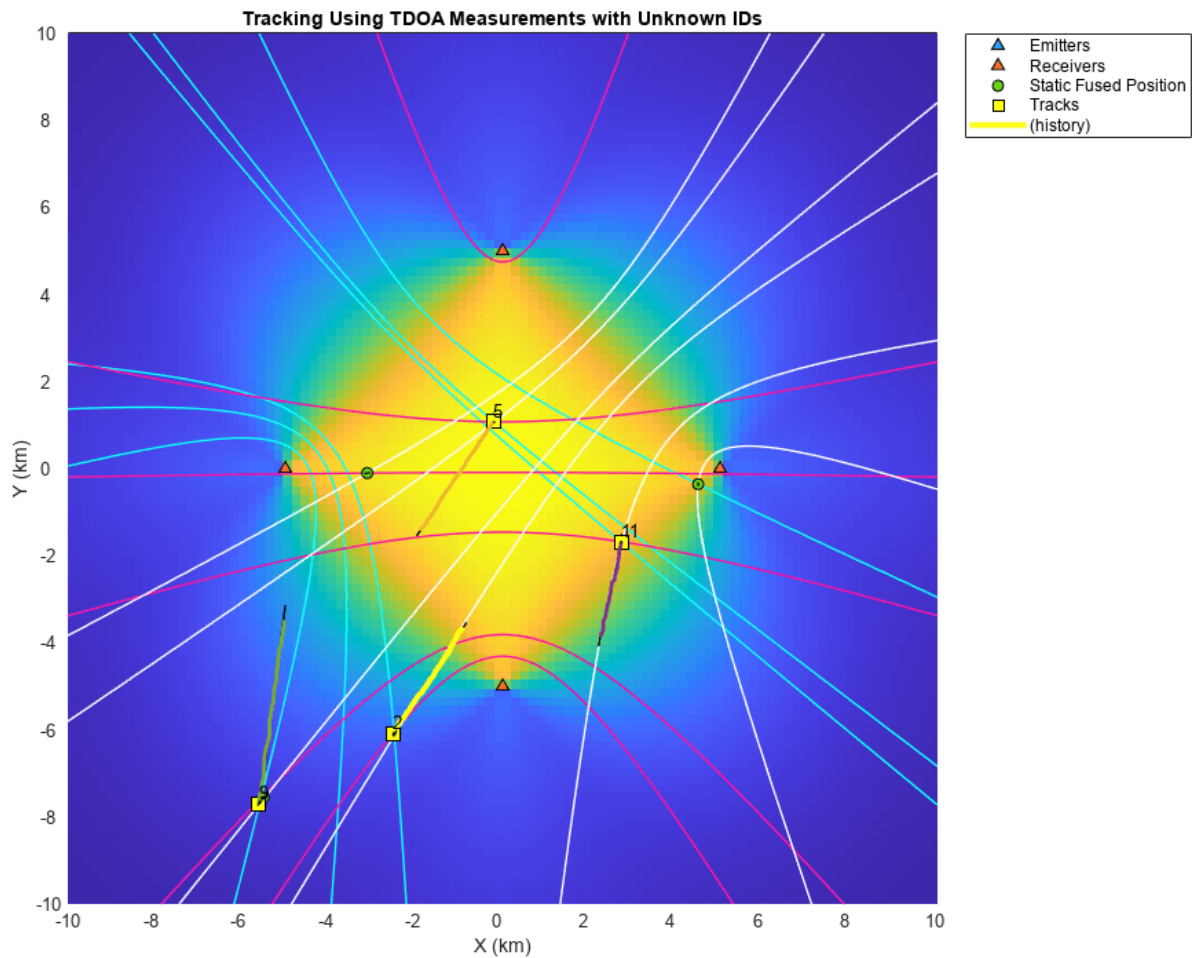
    % Update tracker with position detections
    if ~isempty(posDets)
        tracks = tracker(posDets, time);
    end

    % Update display
```

```

    display(scenario, receiverIDs, tdoaDets, posDets, tracks);
end

```



With the defined measurement statistics in this example, the tracker maintains tracks on all the objects based on this network geometry. The geometry of the problem, the measurement accuracy, as well as the number of false alarms all have major impacts on the data association accuracy of the static fusion algorithm for both the TOA and TDOA systems with unknown data association. For the scenario used in this example, the static fusion algorithm was able to report true detections at sufficient time instants to maintain a track on true objects.

Summary

In this example, you learned how to track single object as well as multiple objects using TDOA measurements. You learned about the challenges associated with multi-object tracking without emitter identification from the receivers and used a static fusion algorithm to compute data association at the measurement level.

References

[1] Smith, Julius, and Jonathan Abel. "Closed-form least-squares source location estimation from range-difference measurements." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 35.12 (1987): 1661-1669.

[2] Sathyan, T., A. Sinha, and T. Kirubarajan. "Passive geolocation and tracking of an unknown number of emitters." *IEEE Transactions on Aerospace and Electronic Systems* 42.2 (2006): 740-750.

Supporting Functions

This section defines a few supporting functions used in this example. The complete list of helper functions can be found in the current working directory.

helperTDOA2Pos

```
function varargout = helperTDOA2Pos(tdoaDets, reportDetection)
% This function uses the spherical intersection algorithm to find the
% object position from the TDOA detections assumed to be from the same
% object.
%
% This function assumes that all TDOAs are measured with respect to the
% same reference sensor.
%
% [pos, posCov] = helperTDOA2Pos(tdoaDets) returns the estimated position
% and position uncertainty covariance.
%
% posDetection = helperTDOA2Pos(tdoaDets, true) returns the estimate
% position and uncertainty covariance as an objectDetection object.

if nargin < 2
    reportDetection = false;
end

% Collect scaling information
params = helperGetGlobalParameters;
emissionSpeed = params.EmissionSpeed;
timeScale = params.TimeScale;

% Location of the reference receiver
referenceLoc = tdoaDets{1}.MeasurementParameters(2).OriginPosition(:);

% Formulate the problem. See [1] for more details
d = zeros(numel(tdoaDets),1);
delta = zeros(numel(tdoaDets),1);
S = zeros(numel(tdoaDets),3);
for i = 1:numel(tdoaDets)
    receiverLoc = tdoaDets{i}.MeasurementParameters(1).OriginPosition(:);
    d(i) = tdoaDets{i}.Measurement*emissionSpeed/timeScale;
    delta(i) = norm(receiverLoc - referenceLoc)^2 - d(i)^2;
    S(i,:) = receiverLoc - referenceLoc;
end

% Pseudo-inverse of S
Swstar = pinv(S);

% Assemble the quadratic range equation
STS = (Swstar'*Swstar);
```

```

a = 4 - 4*d'*STS*d;
b = 4*d'*STS*delta;
c = -delta'*STS*delta;

Rs = zeros(2,1);
% Imaginary solution, return a location outside coverage
if b^2 < 4*a*c
    varargout{1} = 1e10*ones(3,1);
    varargout{2} = 1e10*eye(3);
    return;
end

% Two range values
Rs(1) = (-b + sqrt(b^2 - 4*a*c))/(2*a);
Rs(2) = (-b - sqrt(b^2 - 4*a*c))/(2*a);

% If one is negative, use the positive solution
if prod(Rs) < 0
    Rs = Rs(Rs > 0);
    pos = 1/2*Swstar*(delta - 2*Rs(1)*d) + referenceLoc;
else % Use range which minimize the error
    xs1 = 1/2*Swstar*(delta - 2*Rs(1)*d);
    xs2 = 1/2*Swstar*(delta - 2*Rs(2)*d);
    e1 = norm(delta - 2*Rs(1)*d - 2*S*xs1);
    e2 = norm(delta - 2*Rs(2)*d - 2*S*xs2);
    if e1 > e2
        pos = xs2 + referenceLoc;
    else
        pos = xs1 + referenceLoc;
    end
end

% If required, compute the uncertainty in the position
if nargout > 1 || reportDetection
    posCov = helperCalcPositionCovariance(pos,tdoaDets,timeScale,emissionSpeed);
end

if reportDetection
    varargout{1} = objectDetection(tdoaDets{1}.Time,pos,'MeasurementNoise',posCov);
else
    varargout{1} = pos;
    if nargout > 1
        varargout{2} = posCov;
    end
end

end

function measCov = helperCalcPositionCovariance(pos,thisDetections,timeScale,emissionSpeed)
n = numel(thisDetections);
% Complete Jacobian from position to N TDOAs
H = zeros(n,3);
% Covariance of all TDOAs
S = zeros(n,n);
for i = 1:n
    e1 = pos - thisDetections{i}.MeasurementParameters(1).OriginPosition(:);
    e2 = pos - thisDetections{i}.MeasurementParameters(2).OriginPosition(:);
    Htdoar1 = (e1'/norm(e1))*timeScale/emissionSpeed;

```

```

    Htdoar2 = (e2'/norm(e2))*timeScale/emissionSpeed;
    H(i,:) = Htdoar1 - Htdoar2;
    S(i,i) = thisDetections{i}.MeasurementNoise;
end
Pinv = H'/S*H;
% Z is not measured, use 1 as the covariance
if Pinv(3,3) < eps
    Pinv(3,3) = 1;
end

% Insufficient information in TDOA
if rank(Pinv) >= 3
    measCov = eye(3)/Pinv;
else
    measCov = inf(3);
end

% Replace inf with large number
measCov(~isfinite(measCov)) = 100;

% Return a true symmetric, positive definite matrix for covariance.
measCov = (measCov + measCov')/2;
end

```

helperInitHighSpeedKF

```

function filter = helperInitHighSpeedKF(detection)
% This function initializes a constant velocity Kalman filter and sets a
% higher initial state covariance on velocity components to account for
% high speed of the object vehicles.
filter = initcvkf(detection);
filter.StateCovariance(2:2:end,2:2:end) = 500*eye(3);
end

```

helperTOA2Pos

```

function [posTime, posTimeCov] = helperTOA2Pos(toaDetections)
% This function computes the position and emission time of a target given
% its time-of-arrival detections from multiple receivers.
%
% Convert TOAs to TDOAs for using spherical intersection
[tdoaDetections, isValid] = helperTOA2TDOADetections(toaDetections);

% At least 3 TOAs (2 TDOAs) required. Some TOA pairings can lead to an
% invalid TDOA (> maximum TDOA). In those situations, discard the tuple by
% using an arbitrary position with a large covariance.
if numel(tdoaDetections) < 2 || any(~isValid)
    posTime = 1e10*ones(4,1);
    posTimeCov = 1e10*eye(4);
else
    % Get position estimate using TDOA fusion.
    % Get time estimate using calculated position.
    if nargin > 1 % Only calculate covariance when requested to save time
        [pos, posCov] = helperTDOA2Pos(tdoaDetections);
        [time, timeCov] = helperCalcEmissionTime(toaDetections, pos, posCov);
        posTime = [pos;time];
        posTimeCov = blkdiag(posCov,timeCov);
    end
end

```



```

else
    pos = helperTDOA2Pos(tdoaDetections);
    time = helperCalcEmissionTime(toaDetections, pos);
    posTime = [pos;time];
end
end
end
end

```

helperCalcEmissionTime

```

function [time, timeCov] = helperCalcEmissionTime(toaDetections, pos, posCov)
% This function calculates the emission time of an object given its
% position and obtained TOA detections. It also computes the uncertainty in
% the estimate time.
globalParams = helperGetGlobalParameters;
emissionSpeed = globalParams.EmissionSpeed;
timeScale = globalParams.TimeScale;
n = numel(toaDetections);
emissionTime = zeros(n,1);
emissionTimeCov = zeros(n,1);
for i = 1:numel(toaDetections)
    % Calculate range from this receiver
    p0 = toaDetections{i}.MeasurementParameters.OriginPosition(:);
    r = norm(pos - p0);
    emissionTime(i) = toaDetections{i}.Measurement - r/emissionSpeed*timeScale;
    if nargout > 1
        rJac = (pos - p0)'/r;
        rCov = rJac*posCov*rJac';
        emissionTimeCov(i) = rCov./emissionSpeed^2*timeScale^2;
    end
end
% Gaussian merge each time and covariance estimate
time = mean(emissionTime);
if nargout > 1
    e = emissionTime - time;
    timeCov = mean(emissionTimeCov) + mean(e.^2);
end
end
end

```

helperMeasureTOA

```

function toa = helperMeasureTOA(posTime, params)
% This function calculates the expected TOA at a receiver given an object
% position and emission time and the receiver's measurement parameters
% (OriginPosition)
globalParams = helperGetGlobalParameters;
emissionSpeed = globalParams.EmissionSpeed;
timeScale = globalParams.TimeScale;
r = norm(posTime(1:3) - params.OriginPosition);
toa = posTime(4) + r/emissionSpeed*timeScale;
end

```

helperMeasureTDOA

```

function toa = helperMeasureTDOA(pos, params)
% This function calculates the expected TDOA measurement given object
% position and measurement parameters of a TDOA detection
globalParams = helperGetGlobalParameters;

```

```
emissionSpeed = globalParams.EmissionSpeed;  
timeScale = globalParams.TimeScale;  
r1 = norm(pos(1:3) - params(1).OriginPosition);  
r2 = norm(pos(1:3) - params(2).OriginPosition);  
toa = (r1 - r2)/emissionSpeed*timeScale;  
end
```

Copyright 2021 The MathWorks, Inc.

Design Fusion Filter for Custom Sensors

This example introduces how to customize sensor models used with an `insEKF` object.

Using the `insEKF` object, you can fuse measurement data from multiple types of sensors by using the built-in INS sensor models, including the `insAccelerometer`, `insGyroSpace`, `insMagnetometer`, and `insGPS` objects. Though these sensor objects cover a variety of INS sensor models, you may want to fuse measurement data from a different type of sensor. Using the `insEKF` object through an extended framework, you can define your own sensor models and motion models used by the filter. In this example, you learn how to customize three sensor models in a few steps. You can apply the similar steps for defining a motion model.

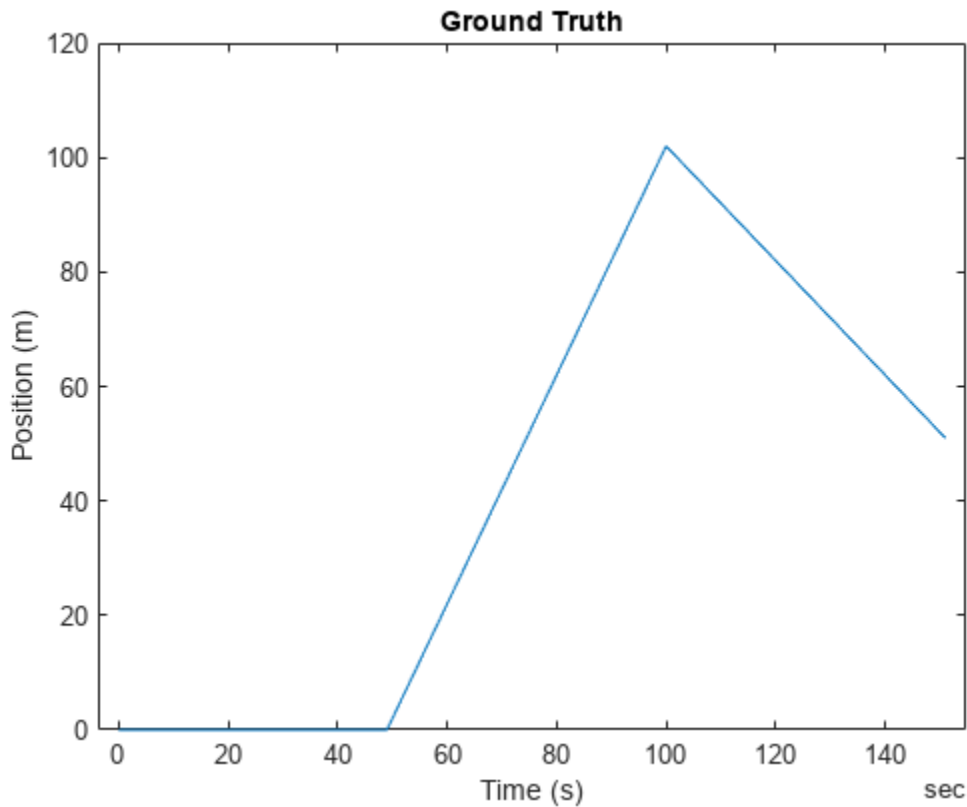
This example requires the Sensor Fusion and Tracking Toolbox or the Navigation Toolbox. This example also optionally uses MATLAB Coder to accelerate filter tuning.

Problem Description

Consider you are trying to estimate the position of an object that moves in one dimension. To estimate the position, you use a velocity sensor and fuse data from that sensor to determine the position. Typically, applying this approach can lead to a poor estimate of position because small errors in the velocity estimate can integrate to form larger errors in the position estimate. As shown later in this example, combining measurements from multiple sensors can improve the results.

To start, define a simple ground truth trajectory for the object and visualize how it moves.

```
groundTruth = exampleHelperMakeGroundTruth();  
plot(groundTruth.Time, groundTruth.Position);  
xlabel("Time (s)");  
ylabel("Position (m)");  
title("Ground Truth");
```



```
snapnow;
```

Simple Velocity Sensor

Consider the data from a simple velocity sensor that measures velocity but is corrupted by a small bias and additive white Gaussian noise.

```
velBias = exampleHelperVelocityWithBias(groundTruth);
```

For the simplest approach, you create a filter that treats the velocity measurements as just velocity plus white noise. This will likely not produce desirable results, but it is worth trying the simplest model first. To create a velocity sensor model for the `inSEKF` object, you need to define a class that inherits from the `positioning.INSSensorModel` interface class. At minimum you need to implement a `measurement` method which takes the sensor object and filter object as inputs. The `measurement` method returns a vector `z` as an estimate of measurement from the sensor, based on state variables.

```
classdef exampleHelperVelocitySensor < positioning.INSSensorModel
%EXAMPLEHELPERVELOCITYSENSOR A simple velocity sensor for inSEKF
% This class is for internal use only. It may be removed in the future.

% Copyright 2021 The MathWorks, Inc.

methods
function z = measurement(~, filt)
% The measurement is just the velocity estimate
z = stateparts(filt, 'Velocity');
```

```

        end
    end
end

```

This is the minimum required to define a sensor that works with the `insEKF` object. You could optionally define the measurement Jacobian in the `measurementJacobian` method, which can improve the calculation speed and possibly improve the estimation accuracy. Without implementing the `measurementJacobian` method, you are relying on the `insEKF` to numerically compute the Jacobian.

You build the filter with a simple 1-D constant velocity motion model defined using a `BasicConstantVelocityMotion` class. This class is an example of how to implement your own custom motion models by inheriting from the `positioning.INSMotionModel` interface class. Defining custom motion models requires implementing only a subset of the methods required to implement a custom sensor model.

You can now build the filter, tune its noise parameters, and see how it performs. You can name the Velocity sensor "VelocityWithBias" using the `insOptions` object.

```

basicOpts = insOptions(SensorNamesSource="property", SensorNames={'VelocityWithBias'});
basicfilt = insEKF(exampleHelperVelocitySensor, exampleHelperConstantVelocityMotion, ...
    basicOpts);

```

The object starts at rest with a position of zero. Set the state covariance for the position and velocity to a lower value before tuning.

```

statecovparts(basicfilt, "Position", 1e-2);
statecovparts(basicfilt, "Velocity", 1e-2);

```

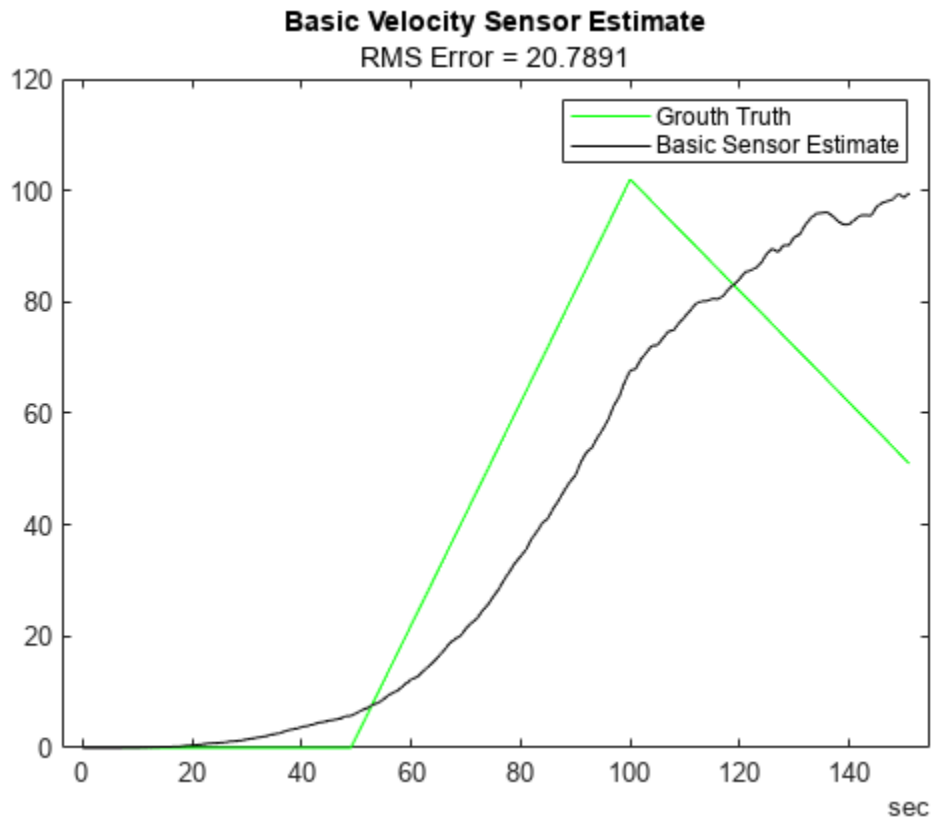
Tune the filter after specifying a noise structure and a `tunerconfig` object.

```

mn = tunernoise(basicfilt);
tunerIters = 30;
cfg = tunerconfig(basicfilt, MaxIterations=tunerIters, Display='none');
tunedmn = tune(basicfilt, mn, velBias, groundTruth, cfg);

estBasic = estimateStates(basicfilt, velBias, tunedmn);
figure;
plot(groundTruth.Time, groundTruth.Position, 'g', estBasic.Time, ...
    estBasic.Position, 'k')
title("Basic Velocity Sensor Estimate");
err = sqrt(mean((estBasic.Position - groundTruth.Position).^2));
subtitle("RMS Error = " + err);
legend("Ground Truth", "Basic Sensor Estimate");

```



```
snapnow;
```

The estimation results are poor as expected since the bias is not accounted for by the filter. When the unaccounted velocity bias is integrated into position, the estimate diverges quickly from ground truth.

Velocity Sensor With Bias

Now you can explore a more complex sensor model which accounts for the sensor bias. To do this, you define a `sensorstates` method to inform the `insEKF` to track the sensor bias in its state vector, saved in the `State` property of the filter. You can query the sensor bias with the `stateparts` object function of the filter and use the bias estimate in the `measurement` function.

```
classdef exampleHelperVelocitySensorWithBias < positioning.INSSensorModel
%EXAMPLEHELPERVELOCITYSENSORWITHBIAS Velocity sensor assuming constant bias
% This class is for internal use only. It may be removed in the future.
```

```
% Copyright 2021 The MathWorks, Inc.
```

```
methods
```

```
function s = sensorstates(~,~)
% Assume there is a constant bias corrupting the velocity
% measurement. Estimate that bias as part of the filter
% computation.
s = struct('Bias', 0);
```

```
end
```

```
function z = measurement(sensor, filt)
```

```

% Measurement is velocity plus bias. Obtain the velocity from
% the filter state.
velocity = stateparts(filt, 'Velocity');

% Obtain the sensor bias from the filter state by directly
% using the sensor object as an input to the stateparts object
% function. In this way, knowledge of the SensorName associated
% with this sensor is not required. See the reference page for
% stateparts for more details.
bias = stateparts(filt, sensor, 'Bias');

% Form the measurement
z = velocity + bias;
end
function dzdx = measurementJacobian(sensor, filt)
% Compute the Jacobian as the partial derivatives of the Bias
% state relative to all other states.
N = numel(filt.State); % Number of states

% Initialize a matrix of partial derivatives. This matrix has
% one row because the sensor state is a scalar.
dzdx = zeros(1,N);

% The partial derivative of z with respect to the Bias is 1. So
% put a 1 at the Bias index in the dzdx matrix:
bidx = stateinfo(filt, sensor, 'Bias');
dzdx(:,bidx) = 1;

% The partial derivative of z with respect to the Velocity is
% also 1. So put a 1 at the Bias index in the dzdx matrix:
vidx = stateinfo(filt, 'Velocity');
dzdx(:,vidx) = 1;
end
end
end

```

In the above you implemented the `measurementJacobian` method rather than relying on a numeric Jacobian. The Jacobian matrix contains the partial derivatives of the sensor measurement with respect to the filter state, which is an M -by- N matrix, where M is the number of elements in the vector returned by the measurement method and N is the number of states of the filter. Now you can build a filter with the custom sensor.

```

sensorVelWithBias = exampleHelperVelocitySensorWithBias;
velWithBiasFilt = insEKF(sensorVelWithBias, ...
    exampleHelperConstantVelocityMotion, basicOpts);

```

You can set the initial bias state of the filter with an estimate of the bias, based on data collected while the sensor is stationary, using the `stateparts` object function.

```

stationaryLen = 1e7;
stationary = timetable(zeros(stationaryLen,1), zeros(stationaryLen,1), ...
    TimeStep=groundTruth.Properties.TimeStep, ...
    VariableNames={'Position', 'Velocity'});
biasCalibrationData = exampleHelperVelocityWithBias(stationary);

stateparts(velWithBiasFilt, sensorVelWithBias, "Bias", ...
    mean(biasCalibrationData.VelocityWithBias));

```

Again, set the state covariance for these states to smaller values before tuning.

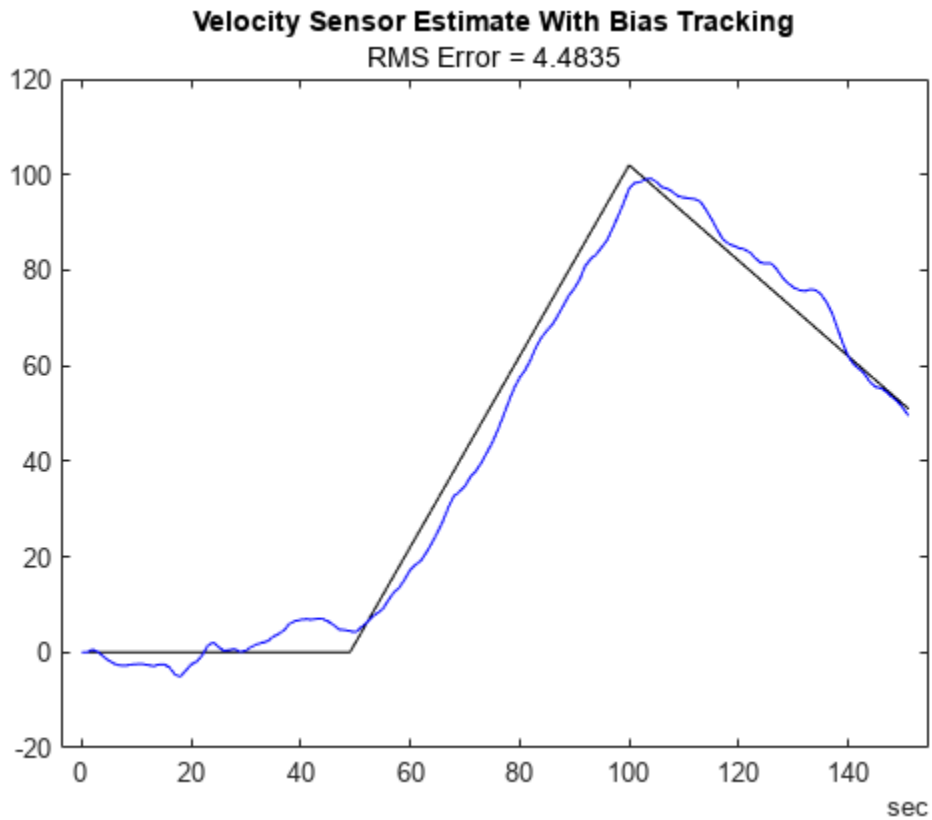
```
statecovparts(velWithBiasFilt, "Position", 1e-2);
statecovparts(velWithBiasFilt, "Velocity", 1e-2);
```

You can tune this filter as previously, but you can also tune it faster using MATLAB Coder. While the `tune` object function does not support code generation, you can use a MEX-accelerated cost function to greatly increase the tuning speed. You can specify the `tune` function to use a custom cost function through the `tunerconfig` input. The `inSEKF` object has object functions to help create a custom cost function. The `createTunerCostTemplate` object function creates a cost function, which tries to minimize the RMS error of state estimates, in a new document in the Editor. You can then generate code for that function with the help of the `tunerCostFcnParam` object function, which creates an example of the first argument to a cost function.

```
% Use MATLAB Coder to accelerate tuning by MEXing the cost function.
% To run the MATLAB Coder accelerated path, prior to running the example,
% type:
%   exampleHelperUseCodegenForCost(true);
% To avoid using MATLAB Coder, prior to the example, type:
%   exampleHelperUseCodegenForCost(false);

useCodegenForTuning = exampleHelperUseCodegenForCost();

if useCodegenForTuning
    createTunerCostTemplate(velWithBiasFilt); % A new cost function in the editor
    exampleHelperSaveCostFunction;
    p = tunerCostFcnParam(velWithBiasFilt);
    % Now generate a mex file
    codegen tunercost.m -args {p, velBias, groundTruth};
    % Use the Custom Cost Function
    cfg2 = tunerconfig(velWithBiasFilt, MaxIterations=tunerIters, ...
        Cost="custom", CustomCostFcn=@tunercost_mex, Display='none');
else
    % Use the default cost function
    cfg2 = tunerconfig(velWithBiasFilt, MaxIterations=tunerIters, ...
        Display='none'); %#ok<*UNRCH>
end
mn = tunernoise(velWithBiasFilt);
tunedmn = tune(velWithBiasFilt, mn, velBias, groundTruth, cfg2);
estWithBias = estimateStates(velWithBiasFilt, velBias, tunedmn);
figure;
plot(groundTruth.Time, groundTruth.Position, 'k', estWithBias.Time, ...
    estWithBias.Position, 'b')
title("Velocity Sensor Estimate With Bias Tracking");
err = sqrt(mean((estWithBias.Position - groundTruth.Position).^2));
subtitle("RMS Error = " + err);
```

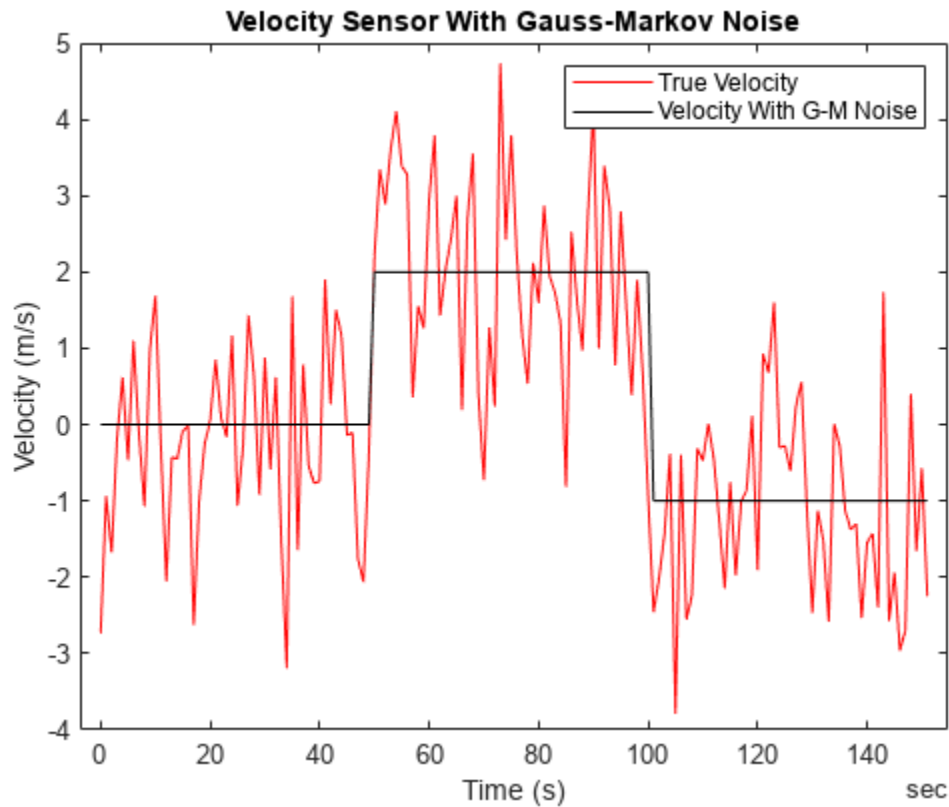
```
snapnow;
```

From the results, the position estimate has improved but is still not ideal. The filter estimates the bias, but any error in the bias estimate is treated as real velocity, which corrupts the position estimate. A more sophisticated sensor model can further improve the results.

Velocity Sensor with Gauss-Markov Noise

Consider a new velocity sensor model that is corrupted with first-order Gauss-Markov noise.

```
velGM = exampleHelperVelocityWithGaussMarkov(groundTruth);
figure
plot(velGM.Time, velGM.VelocityWithGM, 'r', groundTruth.Time, ...
     groundTruth.Velocity, 'k');
title("Velocity Sensor With Gauss-Markov Noise");
xlabel("Time (s)");
ylabel("Velocity (m/s)");
legend("True Velocity", "Velocity With G-M Noise");
```



snapshot;

First-order Gauss-Markov noise can be modeled in discrete time as:

$$x_k = (1 - \beta \cdot dt)x_{k-1} + w_k$$

where w_k is white noise. The equivalent continuous time formulation is:

$$\frac{d}{dt}x = -\beta x(t) + w(t)$$

where β is the time constant of the process, and $w(t)$ and $x(t)$ are the continuous time versions of the discrete state x_k and process noise w_k , respectively.

You can check that these are a valid discrete-continuous pair by applying Euler integration on the second equation between time k and $k-1$ to obtain the first equation. See Reference [1] for a further explanation. For simplicity, set β as 0.002 for this example. Since the Gauss-Markov process evolves over time you need to implement it in the `stateTransition` method of the `positioning.INSSensorModel` class. The `stateTransition` method describes how state elements described in `sensorstates` evolve over time. The `stateTransition` function outputs a structure with the same fields as the output structure of the `sensorstates` method, and each field contains the derivative of the state with respect to time. When the `stateTransition` method is not implemented explicitly, the `insEKF` object assumes the state is constant over time.

```
classdef exampleHelperVelocitySensorWithGM < positioning.INSSensorModel
%EXAMPLEHELPERVELOCITYSENSORWITHGM Velocity sensor with Gauss-Markov noise
```

```

% This class is for internal use only. It may be removed in the future.

% Copyright 2021 The MathWorks, Inc.

properties (Constant)
    Beta =0.002; % First-order Gauss-Markov time constant
end
methods
    function s = sensorstates(~,~)
        % Assume the velocity measurement is corrupted by a first-order
        % Gauss-Markov random process. Estimate the state of the
        % Gauss-Markov process as part of the filtering.
        s = struct('GMPProc', 0);
    end
    function z = measurement(sensor, filt)
        % Measurement of velocity plus Gauss-Markov process noise
        velocity = stateparts(filt, 'Velocity');
        gm = stateparts(filt, sensor, 'GMPProc');

        z = velocity + gm;
    end
    function dhdx = measurementJacobian(sensor, filt)
        % Compute the Jacobian of partial derivatives of the measurement
        % relative to all states.
        N = numel(filt.State); % Number of states

        % Initialize a matrix of partial derivatives. The matrix only
        % has one row because the sensor state is a scalar.
        dhdx = zeros(1,N);

        % Get the indices of the Velocity and GMPProc states in the
        % state vector.
        vidx = stateinfo(filt, 'Velocity');
        gmidx = stateinfo(filt, sensor, 'GMPProc');
        dhdx(:,gmidx) = 1;
        dhdx(:,vidx) = 1;
    end
    function sdot = stateTransition(sensor, filt, ~, varargin)
        % Define the state transition function for each sensorstates
        % defined in this class. Since the insEKF is a
        % continuous-discrete EKF, the output is the derivative of the
        % state in the form of a structure with field names matching
        % the field names of the output structure of sensorstates. The
        % first-order Gauss-Markov process has a state transition of
        %     sdot = -1*beta*s
        % where beta is the reciprocal of the time constant of the
        % process.
        sdot.GMPProc = -1*(sensor.Beta) * ...
            stateparts(filt, sensor, 'GMPProc');
    end
    function dfdx = stateTransitionJacobian(sensor, filt, ~, varargin)
        % Compute the Jacobian of the partial derivatives of the GMPProc
        % state transition relative to all states.

        % Find the number of states and initialize an array of the same
        % size with all elements equal to zero.
        N = numel(filt.State);
        dfdx.GMPProc = zeros(1,N);
    end
end

```

```

        % Find the index of the Gauss-Markov state, GMProc, in the
        % state vector.
        gmidx = stateinfo(filt, sensor, 'GMProc');

        % The derivative of the GMProc stateTransition function with
        % respect to GMProc is -1*Beta
        dfdx.GMProc(gmidx) = -1*(sensor.Beta);
    end
end
end

```

Note here a `stateTransitionJacobian` method is also implemented. The method outputs a structure with its fields containing the partial derivatives of `sensorstates` with respect to the filter state vector. If this function is not implemented, the `insEKF` computes these matrices numerically. You can try commenting out this function to see the effects of using the numeric Jacobian.

```

gmOpts = insOptions(SensorNamesSource="property", SensorNames={'VelocityWithGM'});
filtWithGM = insEKF(exampleHelperVelocitySensorWithGM, ...
    exampleHelperConstantVelocityMotion, gmOpts);

```

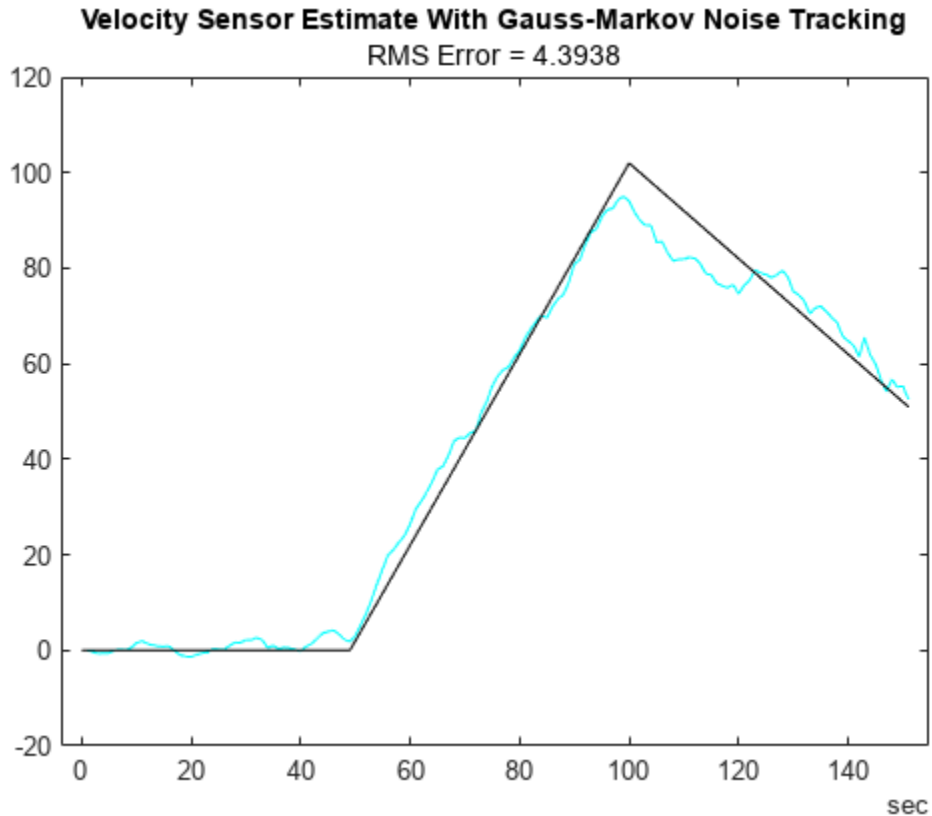
You can re-tune the filter following any of the processes above. Estimate the state by using the `estimateStates` object function.

```

statecovparts(filtWithGM, "Position", 1e-2);
statecovparts(filtWithGM, "Velocity", 1e-2);
gmMeasNoise = tunernoise(filtWithGM);
cfg3 = tunerconfig(filtWithGM, MaxIterations=tunerIters, Display='none');
gmTunedNoise = tune(filtWithGM, gmMeasNoise, velGM, groundTruth, cfg3);

estGM = estimateStates(filtWithGM, velGM, gmTunedNoise);
figure
plot(estGM.Time, estGM.Position, 'c', groundTruth.Time, ...
    groundTruth.Position, 'k');
title("Velocity Sensor Estimate With Gauss-Markov Noise Tracking");
err = sqrt(mean((estGM.Position - groundTruth.Position).^2));
subtitle("RMS Error = " + err);

```



```
snapnow;
```

The filter again cannot distinguish between the Gauss-Markov noise and the true velocity. The Gauss-Markov noise is not observable independently and thus corrupts the position estimate. However, you can reuse the sensors designed above with measurement data from multiple sensors to get a better position estimate.

Multi-sensor fusion

To get an accurate estimate of position, you use multiple sensors. You apply the sensor models you have already designed in a new `insEKF` filter object.

```
combinedOpts = insOptions(SensorNamesSource="property", SensorNames={'VelocityWithBias', 'VelocityWithGM'});
sensorWithBias = exampleHelperVelocitySensorWithBias;
sensorWithGM = exampleHelperVelocitySensorWithGM;
filtCombined = insEKF(sensorWithBias, sensorWithGM, ...
    exampleHelperConstantVelocityMotion, combinedOpts);

% Combine the two sets of sensor measurements.
sensorData = synchronize(velBias, velGM, "first");
% Initialize the bias state with an estimate.
stateparts(filtCombined, sensorWithBias, "Bias", ...
    mean(biasCalibrationData.VelocityWithBias));

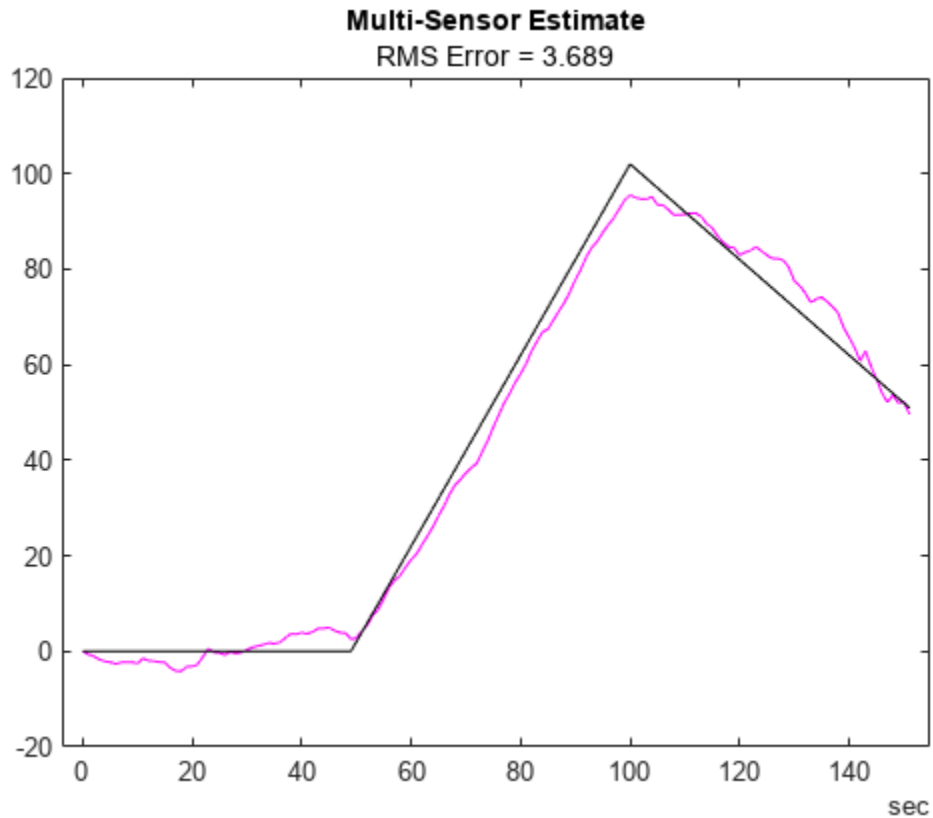
mnCombined = tunernoise(filtCombined);
cfg4 = tunerconfig(filtCombined, MaxIterations=tunerIters, Display='none');
combinedTunedNoise = tune(filtCombined, mnCombined, sensorData, ...
```

```

    groundTruth, cfg4);

estBoth = estimateStates(filtCombined, sensorData, combinedTunedNoise);
figure;
plot(estBoth.Time, estBoth.Position, 'm', groundTruth.Time, ...
     groundTruth.Position, 'k');
title("Multi-Sensor Estimate");
err = sqrt(mean((estBoth.Position - groundTruth.Position).^2));
subtitle("RMS Error = " + err);

```



```

snapnow;

```

Because the filter uses two sensors, both the bias and Gauss-Markov noise are observable. As a result, the filter obtains a more accurate velocity estimate and thus more accurate position estimate.

Conclusion

In this example, you learned the `inSEKF` framework and how to customize sensor models used with the `inSEKF` object. Implementing a custom motion model is almost the same process as implementing a new sensor, except that you inherit from the `positioning.INSMotionModel` interface class instead of the `positioning.INSSensorModel` interface class. Designing custom sensor fusion filters is straightforward with the `inSEKF` framework and it is simple to build reusable sensors to use across projects.

References

[1] A Comparison between Different Error Modeling of MEMS Applied to GPS/INS Integrated Systems. A. Quinchia, G. Falco, E. Falletti, F. DAVIS, C. Ferrer, Sensors 2013.

Estimate Orientation Using GPS-Derived Yaw Measurements

This example shows how to define and use a custom sensor model for the `inSEKF` object along with built-in sensor models. Using a custom yaw angle sensor, an accelerometer, and a gyroscope, this example uses the `inSEKF` object to determine the orientation of a vehicle. You use the velocity from a GPS receiver to compute the yaw of the vehicle. Following a similar approach as shown in this example, you can develop custom sensor models for your own sensor fusion applications.

This example requires either the Sensor Fusion and Tracking Toolbox or the Navigation Toolbox. This example also optionally uses MATLAB Coder to accelerate filter tuning.

Problem Description

You are trying to estimate the orientation of a vehicle while it is moving. The only sensors available are an accelerometer, a gyroscope, and a GPS receiver that outputs a velocity estimate. You cannot use a magnetometer because there is a large amount of magnetic interference on the vehicle.

Approach

There are several tools available in the toolbox to determine orientation including

- `ecompass`
- `imufilter`
- `ahrsfilter`
- `complementaryFilter`

However, these filters require some combination of an accelerometer, a gyroscope, and/or a magnetometer. If you need to determine the absolute orientation (relative to True North) and do not have a magnetometer available, then none of these filters is ideal. The `imufilter` and `complementaryFilter` objects fuse accelerometer and gyroscope data, but they only provide a relative orientation. Furthermore, while these filters can estimate and compensate for the gyroscope bias, there is no additional sensor to help track the gyroscope bias for the yaw angle, which can yield a suboptimal estimate. Typically, a magnetometer is used to accomplish this. However, as mentioned, magnetometers cannot be used in situations with large, highly varying magnetic disturbances. (Note that the `ahrsfilter` object can handle mild magnetic disturbances over a short period of time).

In this example, to get the absolute orientation you use the GPS velocity estimate to determine the yaw angle. This yaw angle estimate can serve the same purpose as a magnetometer without the magnetic disturbance issues. You will build a custom sensor in the `inSEKF` framework to model this GPS-based raw sensor.

Trajectory

First, you create a ground truth trajectory and simulate the sensor by using the `exampleHelperMakeTrajectory` and `exampleHelperMakeIMUGPSData` functions attached with this example

```
groundTruth = exampleHelperMakeTrajectory;  
originalSensorData = exampleHelperMakeIMUGPSData(groundTruth);
```

Retuned as a timetable, the original sensor data includes the accelerometer, gyroscope, and GPS velocity data. Transform the GPS velocity data into a yaw angle estimate.


```
velyaw = atan2(originalSensorData.GPSVelocity(:,2), originalSensorData.GPSVelocity(:,1));
```

The above results assume nonholonomic constraints. That is, they assume the vehicle is pointing in the direction of motion. This assumption is generally true some vehicles, such as a car, but not true for some other vehicles, such as a quadcopter.

Create Synthetic Yaw Angle Sensor Data

Yaw angle is difficult to work with because the yaw angle wraps at π and $-\pi$. The angle jumping at the wrapping bound could cause divergence of the extended Kalman filter. To avoid this problem, convert the yaw angle to a quaternion.

```
velyaw(:,2) = 0; % set pitch and roll to 0
velyaw(:,3) = 0;
qyaw = quaternion(velyaw, 'euler', 'ZYX', 'frame');
```

A common convention is to force the quaternion to have a positive real part

```
isNegQuat = parts(qyaw) < 0; % Find quaternions with a non-negative real part
qyaw(isNegQuat) = -qyaw(isNegQuat); % Invert the negative quaternions.
```

Note that when the `insEKF` object tracks a 4-element `Orientation` state as in this example, it assumes the `Orientation` state is a quaternion and enforces the quaternion to be normalized and have a positive real part. You can disable this rule by calling the state something else, like "Attitude".

Now you can build the timetable of the sensor data to be fused.

```
sensorData = removevars(originalSensorData, 'GPSVelocity');
sensorData = addvars(sensorData, compact(qyaw), 'NewVariableNames', 'YawSensor');
```

Create an `insEKF` Sensor for Fusing Yaw Angle

To fuse the yaw angle quaternion, you customize a sensor model and use it along with the `insAccelerometer` and `insGyroscope` objects in the `insEKF` object. To customize the sensor model, you create a class that inherits from the `positioning.INSSensorModel` interface class and implement the measurement method. Name the class `exampleHelperYawSensor`

```
classdef exampleHelperYawSensor < positioning.INSSensorModel
%EXAMPLEHELPERYAWSENSOR Yaw angle as quaternion sensor
% This class is for internal use only. It may be removed in the future.

% Copyright 2021 The MathWorks, Inc.

methods
function z = measurement(~, filt)
    % Return an estimate of the measurement based on the
    % state vector of the filter saved in filt.State.
    %
    % The measurement is just the quaternion converted from the yaw
    % of the orientation estimate, assuming roll=0 and pitch=0.

    q = quaternion(stateparts(filt, "Orientation"));
    eul = euler(q, 'ZYX', 'frame');
    yawquat = quaternion([eul(1) 0 0], 'euler', 'ZYX', 'frame');

    % Enforce a positive quaternion convention
    if parts(yawquat) < 0
```

```

        yawquat = -yawquat;
    end

    % Return a compacted quaternion
    z = compact(yawquat);
end
end
end
end

```

Now use a `exampleHelperYawSensor` object alongside an `insAccelerometer` object and an `insGyroscope` object to construct the `insEKF` object.

```

opts = insOptions(SensorNamesSource='property', SensorNames={'Accelerometer', 'Gyroscope', 'YawSensor'});
filt = insEKF(insAccelerometer, insGyroscope, exampleHelperYawSensor, insMotionOrientation, opts);

```

Initialize the filter using the `stateparts` and `statecovparts` object functions.

```

stateparts(filt, 'Orientation', sensorData.YawSensor(1,:));
statecovparts(filt, 'Accelerometer_Bias', 1e-3);
statecovparts(filt, 'Gyroscope_Bias', 1e-3);

```

Filter Tuning

You can use the `tune` object function to find the optimal noise parameters for the `insEKF` object. You can directly call the `tune` object function, or you can use a MEX-accelerated cost function with MATLAB Coder.

```

% Trim ground truth to just contain the Orientation for tuning.
trimmedGroundTruth = timetable(groundTruth.Orientation, ...
    SampleRate=groundTruth.Properties.SampleRate, ...
    VariableNames={'Orientation'});

% Use MATLAB Coder to accelerate tuning by MEXing the cost function.
% To run the MATLAB Coder accelerated path, prior to running the example,
% type:
%   exampleHelperUseAcceleratedTuning(true);
% To avoid using MATLAB Coder, prior to the example, type:
%   exampleHelperUseAcceleratedTuning(false);
% By default, the example will not tune the filter live and will not use
% MATLAB Coder.

% Select the accelerated tuning option.
acceleratedTuning = exampleHelperUseAcceleratedTuning();

if acceleratedTuning
    createTunerCostTemplate(filt); % A new cost function in the editor
    % Save and close the file
    doc = matlab.desktop.editor.getActive;
    doc.saveAs(fullfile(pwd, 'tunercost.m'));
    doc.close;
    % Find the first parameter for codegen
    p = tunerCostFcnParam(filt); %#ok<NASGU>
    % Now generate a mex file
    codegen tunercost.m -args {p, sensorData, trimmedGroundTruth};
    % Use the Custom Cost Function and run the tuner for 20 iterations
    tunerIters = 20;
    cfg = tunerconfig(filt, MaxIterations=tunerIters, ...
        Cost='custom', CustomCostFcn=@tunercost_mex, ...

```

```

    StepForward=1.5, ...
    ObjectiveLimit=0.0001, ...
    FunctionTolerance=1e-6, ...
    Display='none');

mnoise = tunernoise(filt);
tunedmn = tune(filt, mnoise, sensorData, trimmedGroundTruth, cfg);
else
% Use optimal parameters obtained separately.
tunedmn = struct(...
    'AccelerometerNoise', 0.7786515625000000, ...
    'GyroscopeNoise', 167.8674323338237, ...
    'YawSensorNoise', 1.003122320344434);

adp = diag([...
    1.2650000000000000;
    1.181989791398048;
    0.735171900658607;
    0.7650000000000000;
    0.026248409763699;
    0.154586330266264;
    31.823154516336434;
    0.000546245218270;
    5.517012554348883;
    0.8090859375000000;
    0.139035477206961;
    41.340145917279159;
    0.5928750000000000]);
filt.AdditiveProcessNoise = adp;
end

```

Fuse Data and Compare to Ground Truth

Batch fuse the data with the `estimateStates` object function.

```
est = estimateStates(filt, sensorData, tunedmn)
```

```
est=13500x5 timetable
```

Time	Orientation	AngularVelocity			Accelerom	
0 sec	1x1 quaternion	0.00034524	0.00030616	0.00029964	-1.2407e-08	-6.203
0.01 sec	1x1 quaternion	0.00037159	0.00035602	0.00071519	9.5542e-09	-0.00
0.02 sec	1x1 quaternion	0.00092989	0.00085552	0.0012341	-1.4108e-09	-1.33
0.03 sec	1x1 quaternion	0.0016774	0.001124	0.0017931	1.4709e-09	0.00
0.04 sec	1x1 quaternion	0.001733	0.0013256	0.0024162	7.0237e-09	9.29
0.05 sec	1x1 quaternion	0.0019492	0.0017904	0.0031902	-2.11e-09	5.13
0.06 sec	1x1 quaternion	0.0025321	0.0022415	0.0039831	-1.3683e-08	0.00
0.07 sec	1x1 quaternion	0.00252	0.0024167	0.0047767	-9.7718e-09	4.28
0.08 sec	1x1 quaternion	0.003163	0.0028615	0.0057521	-1.9926e-08	0.00
0.09 sec	1x1 quaternion	0.0032297	0.0026613	0.006758	5.1045e-09	0.00
0.1 sec	1x1 quaternion	0.0034874	0.0028447	0.0077077	6.8099e-09	0.00
0.11 sec	1x1 quaternion	0.0034133	0.0029448	0.00877	1.3428e-08	-0.00
0.12 sec	1x1 quaternion	0.0037193	0.0031047	0.0099134	1.7482e-08	-0.00
0.13 sec	1x1 quaternion	0.0041719	0.0036375	0.011024	-1.1761e-09	-1.11
0.14 sec	1x1 quaternion	0.0041973	0.0037822	0.012239	1.1026e-09	-0.00
0.15 sec	1x1 quaternion	0.0044736	0.0039896	0.013347	-4.9256e-10	-0.00

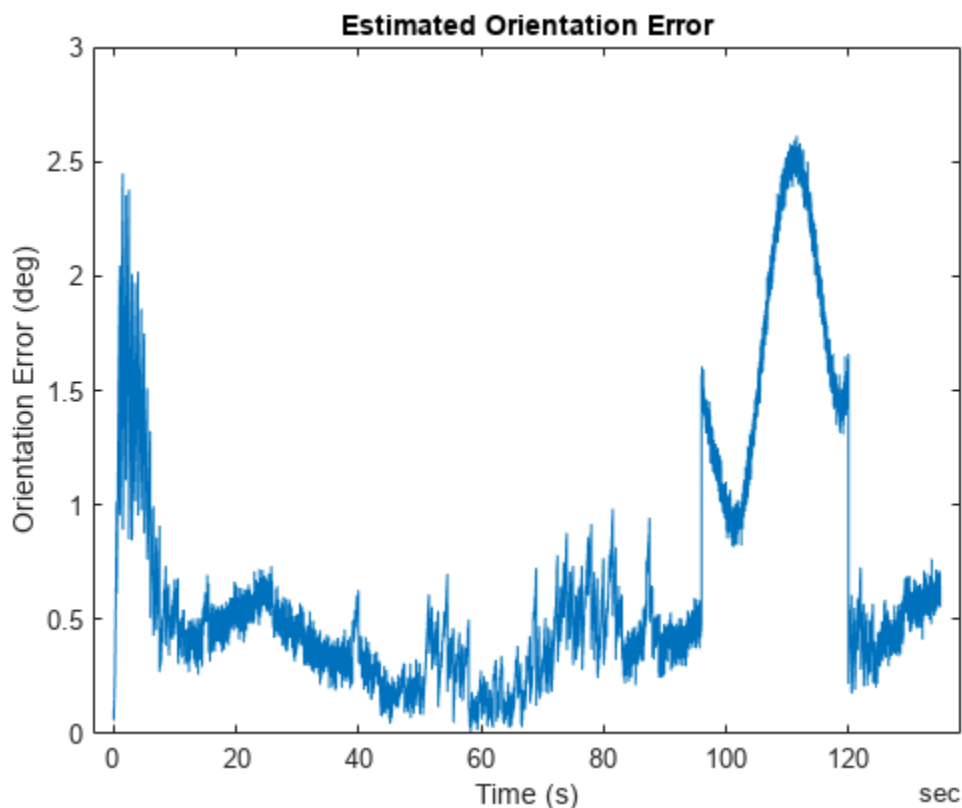
Compare the estimated orientation to ground truth orientation. Plot the quaternion distance between the estimated and true orientations.

```

% Convert compacted quaternions to regular quaternions
estOrient = quaternion(est.Orientation);

d = rad2deg(dist(estOrient, groundTruth.Orientation));
plot(groundTruth.Time, d);
xlabel('Time (s)')
ylabel('Orientation Error (deg)');
title('Estimated Orientation Error');

```



```

snapnow;

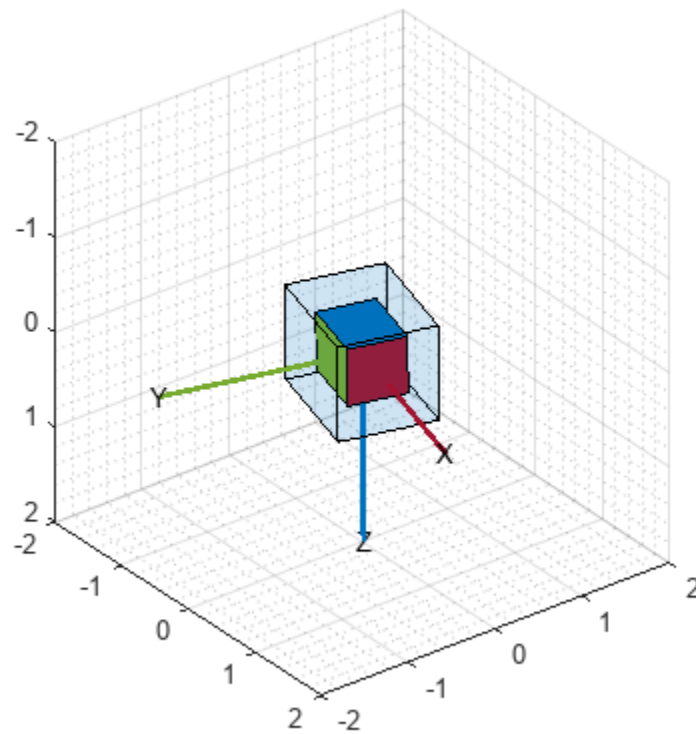
```

From the results, the estimated orientation error is low overall. There is a small bump in the error around 100 seconds, which is likely due to a slight inflection in the ground truth yaw angle at the time. But overall, this is a good orientation estimate result for many applications. You can visualize the estimated orientation using the `poseplot` function.

```

p = poseplot(estOrient(1));
for ii=2:numel(estOrient)
    p.Orientation = estOrient(ii);
    drawnow limitrate;
end

```



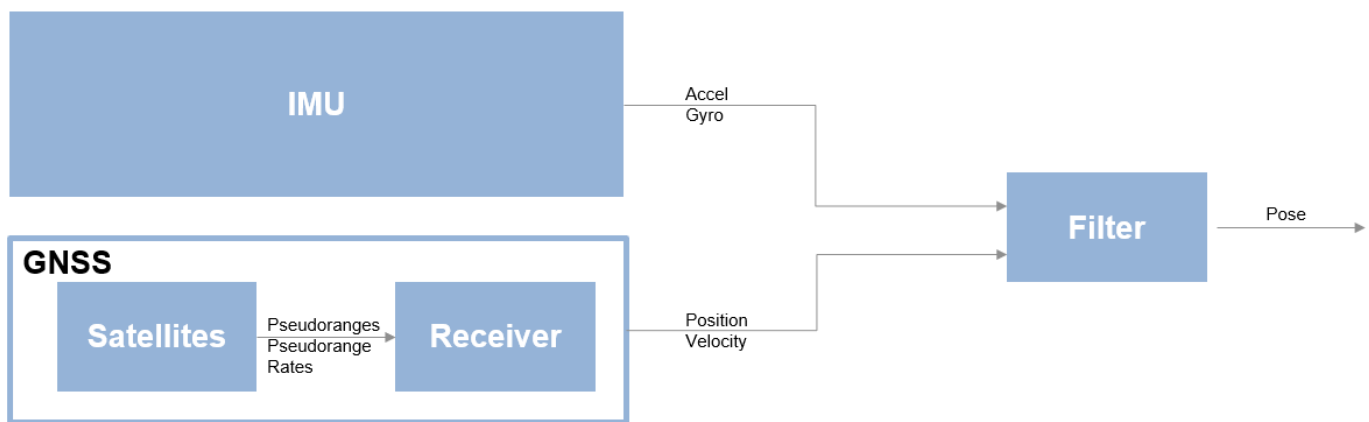
Conclusion

In this example, you learned how to customize a sensor and add it to the `insEKF` framework. Custom sensors can integrate with the built-in sensors like `insAccelerometer` and `insGyroscope`. You also learned how to use the `tune` object function to find optimal noise parameters and improve the filtering results.

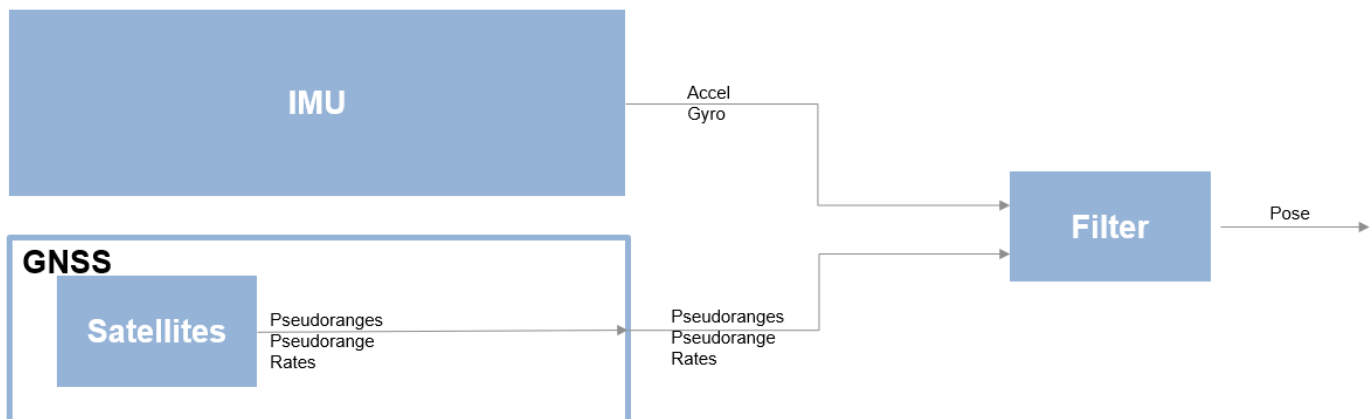
Ground Vehicle Pose Estimation for Tightly Coupled IMU and GNSS

This example shows how to estimate the position and orientation of a ground vehicle by building a tightly coupled extended Kalman filter and using it to fuse sensor measurements. A tightly coupled filter fuses inertial measurement unit (IMU) readings with raw global navigation satellite system (GNSS) readings. In contrast, a loosely coupled filter fuses IMU readings with filtered GNSS receiver readings.

Loosely Coupled Filter Diagram



Tightly Coupled Filter Diagram



Though a tightly coupled filter requires additional processing, you can use it when fewer than four GNSS satellite signals are available, or when some satellite signals are corrupted by interference such as multipath noise. This type of noise can occur when a ground vehicle is traveling through a portion of road with many obstructions, such as an urban canyon, where many surfaces reflect a combination of signals back into the receiver and interfere with the direct signal. Because the ground vehicle is in an urban environment with a lot of opportunity for multipath noise, this example uses a tightly coupled filter.

Define Filter Input

Specify these parameters for the sensor simulation:

- 1 Trajectory motion quantities
- 2 IMU sampling rate
- 3 GNSS receiver sampling rate
- 4 RINEX navigation message file

The RINEX navigation message file contains GNSS satellite orbital parameters used to compute satellite positions and velocities. The satellite positions and velocities are important for generating the GNSS pseudorange and pseudorange rates.

```
load("routeNatickMATtightlyCoupled","pos","orient","vel","acc","angvel","lla0","imuFs","gnssFs");
navfilename = "GODS00USA_R_20211750000_01D_GN.rnx";
```

Create the IMU sensor object. Set the axes misalignment and constant bias values to 0 to simulate calibration.

```
imu = imuSensor(SampleRate=imuFs);
loadparams(imu,"generic.json","GenericLowCost9Axis");
imu.Accelerometer.AxesMisalignment = 100*eye(3);
imu.Accelerometer.ConstantBias = [0 0 0];
imu.Gyroscope.AxesMisalignment = 100*eye(3);
imu.Gyroscope.ConstantBias = [0 0 0];
```

Read the RINEX file by using the `rinexread` (Navigation Toolbox) function. Use only the first set of GPS satellites from the file, and set the initial time for the simulation to the first time step in the file.

```
data = rinexread(navfilename);
[~,idx] = unique(data.GPS.SatelliteID);
navmsg = data.GPS(idx,:);
t0 = navmsg.Time(1);
```

Load the noise parameters, `paramsTuned`, for the IMU sensor and GNSS receiver from the `tunedNoiseParameters` MAT file.

```
load("tunedNoiseParameters.mat","paramsTuned");
accelNoise = paramsTuned.AccelerometerNoise;
gyroNoise = paramsTuned.GyroscopeNoise;
pseudorangeNoise = paramsTuned.exampleHelperINSGNSSNoise(1);
pseudorangeRateNoise = paramsTuned.exampleHelperINSGNSSNoise(2);
```

Set the RNG seed to produce repeatable results.

```
rng default
```

Create Filter and Filter Sensor Models

To create the tightly coupled filter by using the `insEKF` (Navigation Toolbox) object, you must define the conversion of filter states to raw GNSS measurements. Use the `exampleHelperINSGNSS` helper function to define this conversion.

```
gnss = exampleHelperINSGNSS;
gnss.ReferenceLocation = lla0;
```

Define an IMU model by creating accelerometer and gyroscope models using the `insAccelerometer` (Navigation Toolbox) and `insGyroscope` (Navigation Toolbox) objects, respectively.

```
accel = insAccelerometer;
gyro = insGyroscope;
```

Create the filter by using the IMU sensor model, the raw GNSS sensor model, and a 3-D pose motion model represented as an `insMotionPose` (Navigation Toolbox) object.

```
filt = insEKF(accel,gyro,gnss,insMotionPose);
```

Set the initial states of the filter using tuned parameters from `paramsTuned`.

```
filt.State = paramsTuned.InitialState;
filt.StateCovariance = paramsTuned.InitialStateCovariance;
filt.AdditiveProcessNoise = paramsTuned.AdditiveProcessNoise;
```

Estimate Vehicle Pose

Create a figure in which to view the position estimate for the ground vehicle during the filtering process.

```
figure
posLLA = ned2lla(pos,lla0,"ellipsoid");
geoLine = geoplot(posLLA(1,1),posLLA(1,2),".",posLLA(1,1),posLLA(1,2),".");
geolimits([42.2948 42.3182],[-71.3901 -71.3519])
geobasemap topographic
legend("Ground truth","Filter estimate")
```

Allocate a matrix in which to store the position estimate results.

```
numSamples = size(pos,1);
estPos = NaN(numSamples,3);
estOrient = ones(numSamples,1,"quaternion");
imuSamplesPerGNSS = imuFs/gnssFs;
numGNSSSamples = numSamples/imuSamplesPerGNSS;
```

Set the current simulation time to the specified initial time.

```
t = t0;
```

Fuse the IMU and raw GNSS measurements. In each iteration, fuse the accelerometer and gyroscope measurements to the GNSS measurements separately to update the filter states, with the covariance matrices defined by the previously loaded noise parameters. After updating the filter state, log the new position and orientation states. Finally, predict the filter states to the next time step.

```
for ii = 1:numGNSSSamples
    for jj = 1:imuSamplesPerGNSS
        imuIdx = (ii-1)*imuSamplesPerGNSS + jj;
        [accelMeas,gyroMeas] = imu(acc(imuIdx,:),angvel(imuIdx,:),orient(imuIdx,:));

        fuse(filt,accel,accelMeas,accelNoise);
        fuse(filt,gyro,gyroMeas,gyroNoise);

        estPos(imuIdx,:) = stateparts(filt,"Position");
        estOrient(imuIdx,:) = quaternion(stateparts(filt,"Orientation"));
```



```

    t = t + seconds(1/imuFs);
    predict(filt,1/imuFs);
end

```

Update the satellite positions and raw GNSS measurements.

```

gnssIdx = ii*imuSamplesPerGNSS;
recPos = posLLA(gnssIdx,:);
recVel = vel(gnssIdx,:);
[satPos,satVel,satIDs] = gnssconstellation(t,"RINEXData",navmsg);
[az,el,vis] = lookangles(recPos,satPos);
[p,pdot] = pseudoranges(recPos,satPos(vis,:),recVel,satVel(vis,:));
z = [p; pdot];

```

Update the satellite positions on the GNSS model.

```

gnss.SatellitePosition = satPos(vis,:);
gnss.SatelliteVelocity = satVel(vis,:);

```

Fuse the raw GNSS measurements.

```

fuse(filt,gnss,z, ...
    [pseudorangeNoise*ones(1,numel(p)),pseudorangeRateNoise*ones(1,numel(pdot))]);
estPos(gnssIdx,:) = stateparts(filt,"Position");
estOrient(gnssIdx,:) = quaternion(stateparts(filt,"Orientation"));

```

Update the position estimation plot.

```

estPosLLA = ned2lla(estPos,lla0,"ellipsoid");
set(geoLine(1),LatitudeData=posLLA(1:gnssIdx,1),LongitudeData=posLLA(1:gnssIdx,2));
set(geoLine(2),LatitudeData=estPosLLA(1:gnssIdx,1),LongitudeData=estPosLLA(1:gnssIdx,2));
drawnow limitrate
end

```



Validate Results

Calculate the RMS error to validate the results. The tightly coupled filter uses the provided sensor readings to estimate the ground vehicle pose with a relatively low RMS error.

```
posDiff = estPos - pos;
posRMS = sqrt(mean(posDiff.^2));
disp(['3-D Position RMS Error - X: ', num2str(posRMS(1)), ', Y:', ...
      num2str(posRMS(2)), ', Z: ', num2str(posRMS(3)), ' (m)'])
```

3-D Position RMS Error - X: 0.85213, Y:0.62864, Z: 0.9604 (m)

```
orientDiff = rad2deg(dist(estOrient, orient));
orientRMS = sqrt(mean(orientDiff.^2));
disp(['Orientation RMS Error - ', num2str(orientRMS), ' (degrees)'])
```

Orientation RMS Error - 4.077 (degrees)

Estimate Orientation Using AHRS Filter and IMU Data in Simulink

This example shows how to stream IMU data from sensors connected to Arduino® board and estimate orientation using AHRS filter and IMU sensor.

Required MathWorks Products

- MATLAB®
- Simulink®
- Simulink support Package for Arduino Hardware
- Either Navigation Toolbox™ or Sensor Fusion and Tracking Toolbox™

Hardware Required

1. Any of the Arduino board given below:

- Arduino Leonardo
- Arduino Mega 2560
- Arduino Mega ADK
- Arduino Micro
- Arduino Nano 3.0
- Arduino Uno
- Arduino Due
- Arduino MKR1000
- Arduino MKR WIFI 1010
- Arduino MKR ZERO
- Arduino Nano 33 IoT
- Arduino Nano 33 BLE Sense

2. IMU sensor with accelerometer, gyroscope, and magnetometer. In this example, X-NUCLEO-IKS01A2 sensor expansion board is used. The LSM6DSL sensor on the expansion board is used to get acceleration and angular rate values. The LSM303AGR sensor on the expansion board is used to get magnetic field value. The sensor data can be read using I2C protocol.

Hardware Connection

X-NUCLEO-IKS01A2 shield comes with Arduino UNO connectors, which makes it easy to interface with UNO board. For other boards, connect the SDA, SCL, 3.3V, and GND pin of the Arduino board to the respective pins on the sensor shield.

Hardware Configuration in the model

The example uses two models, `AnalyseIMUData.slx` and `EstimateOrientationUsingAHRSandIMU.slx`. Both the models are preconfigured to work with Arduino UNO. If you are using a different Arduino board, change the hardware board by doing the following steps:

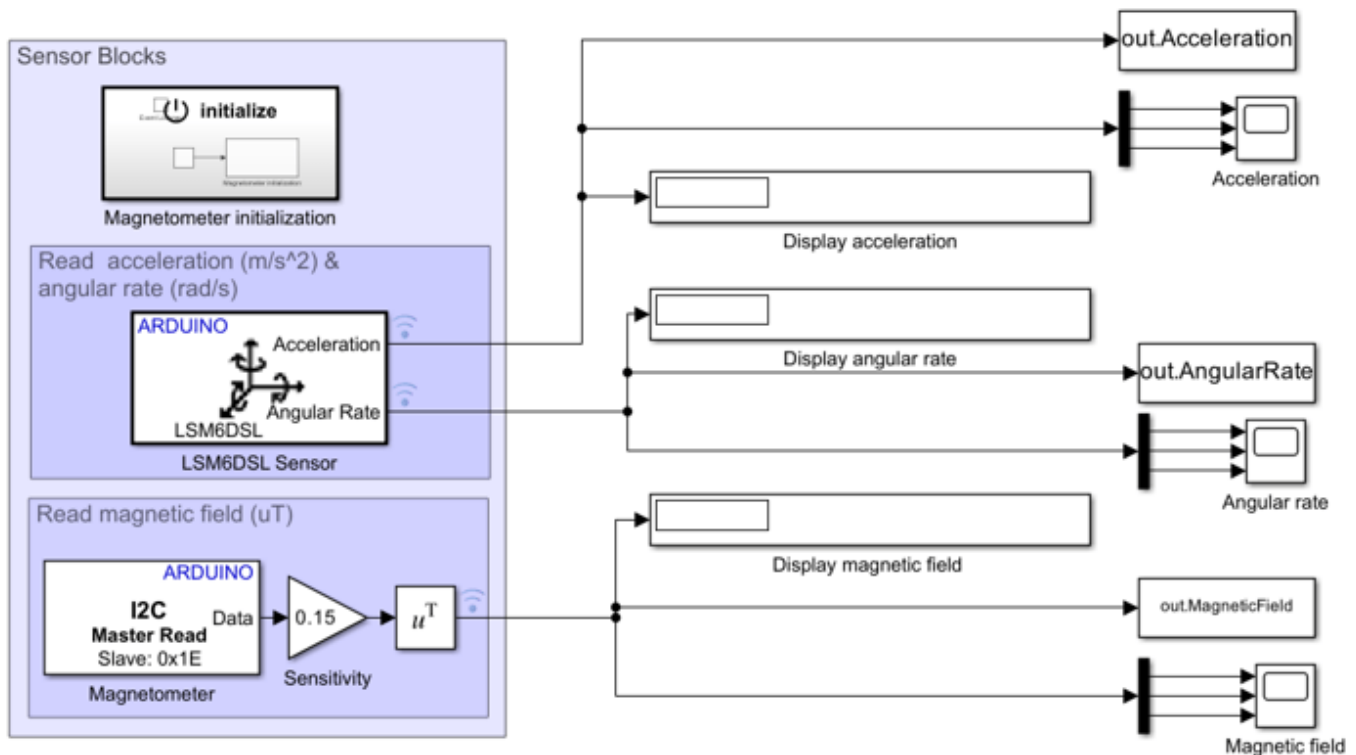
1. Click **Hardware Settings** in the **Hardware** tab of the Simulink toolbar.
2. In the Configurations Parameters dialog box, select **Hardware Implementation**.
3. From the **Hardware board** list, select the type of Arduino board that you are using.
4. Click **Apply**. Click **OK** to close the dialog box.

Task 1 - Read and Calibrate Sensor Values

This section describes how to read and calibrate the sensor values for the orientation estimation algorithm. To read and analyze values, use the model `AnalyseIMUData.slx`.

```
open_system('AnalyseIMUData.slx');
```

Analyse IMU Data



Copyright 2022 The MathWorks, Inc.

Reading acceleration and angular rate from LSM6DSL Sensor

Simulink Support Package for Arduino Hardware provides LSM6DSL IMU Sensor (Simulink Support Package for Arduino Hardware) block to read acceleration and angular rate along the X, Y and Z axis from LSM6DSL sensor connected to Arduino. The block outputs acceleration in m/s² and angular rate in rad/s. The sensor can be further configured by selecting the options given on the block mask.

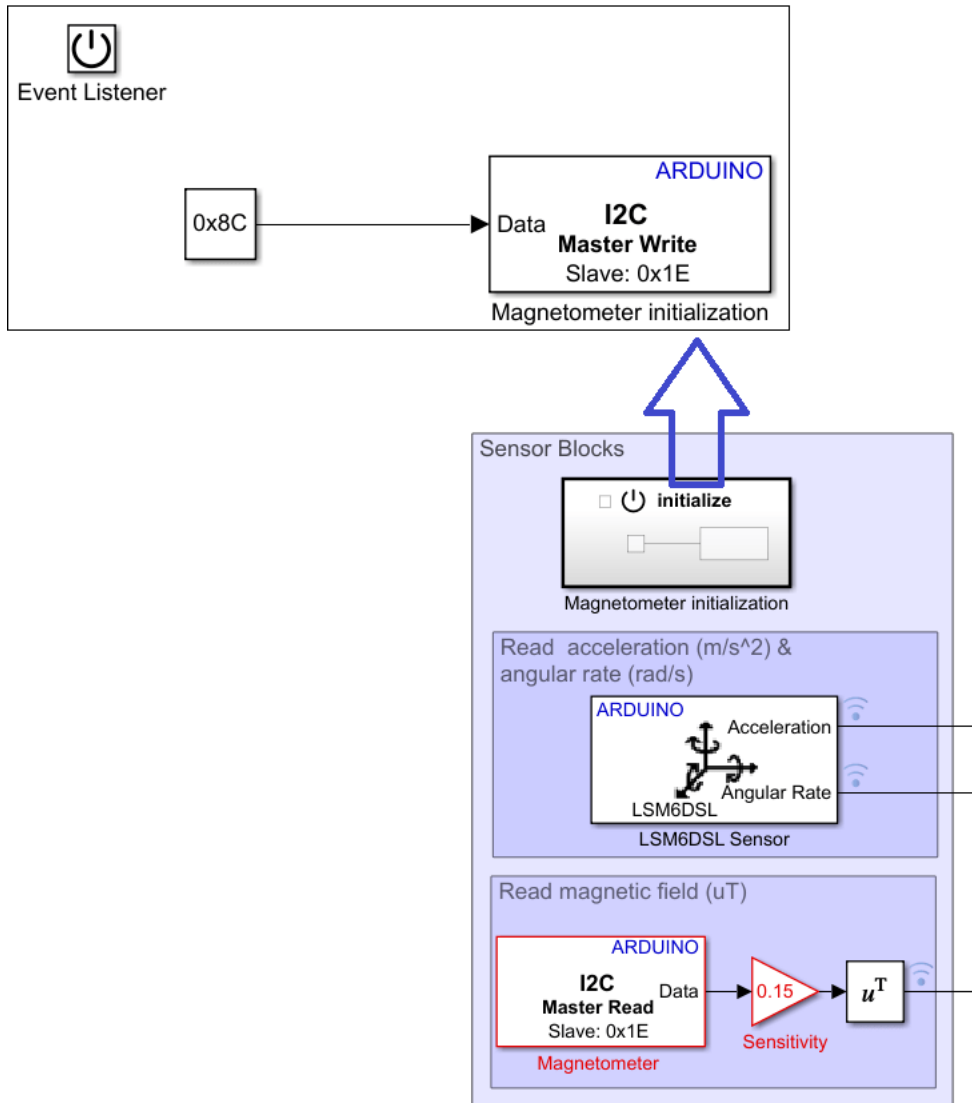
Reading sensor values form LSM303AGR

To read magnetic field values using LSM303AGR sensor, the example uses I2C Read (Simulink Support Package for Arduino Hardware) and I2C Write (Simulink Support Package for Arduino Hardware) blocks in the Support Package.

The I2C Address of the LSM303AGR sensor is 0x1E. This address is specified as *Slave address* in I2C Read/I2C Write block that is used to configure and read the magnetic field value from the sensor.

Depending on the Output Data Rate (ODR) required, a value needs to be written to CFG_REG_A_M register (0x60) of the sensor. To see the available ODRs, refer the LSM303AGR datasheet. This is a one-time operation required to initialize the sensor, and it is done using the Initialize Function (Simulink) block in the model.

The magnetic field is read from the output registers (0x68 - 0x6D) of the sensor using the I2C Read block and it is converted to microtesla as required by the example.



Perform Magnetometer calibration

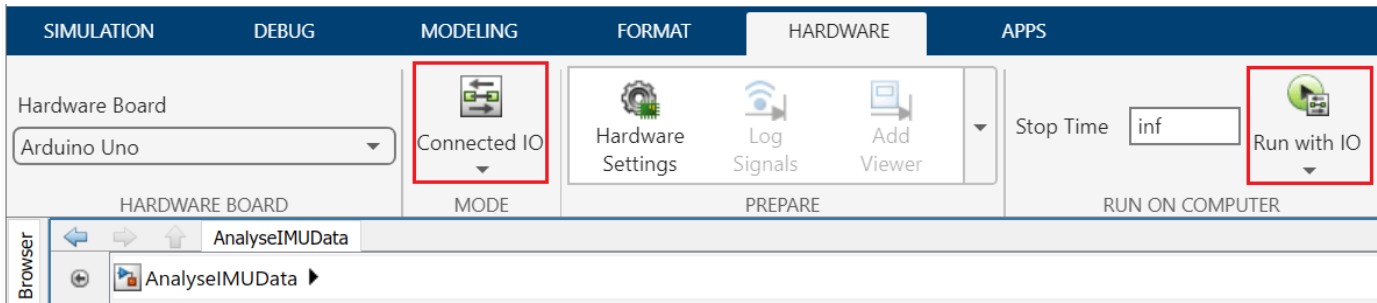
To get accurate measurements from the sensor, the sensor need to be calibrated. In this section, we consider magnetometer calibration for compensating hard iron distortions. Hard iron effects are stationary interfering magnetic noise sources. Often, these come from other metallic objects on the circuit board with the magnetometer. These distortions can be corrected by subtracting the correction value from the magnetometer readings for each axis.

To find the correction values, do the following:

1. Open the model *AnalyseIMUData*. The model uses the *To workspace* block (out.MagneticField in the model) to log magnetometer data.

The model is already configured to run in **Connected IO** mode. The simulation using Connected IO allows you to run your algorithm in Simulink with peripheral data from the hardware. For more details, refer to “Communicate with Hardware Using Connected IO” (Simulink Support Package for Arduino Hardware).

2. Run Connected IO by clicking the Run button corresponding to **Run with IO** under the **Hardware** tab.



3. While the model is running, rotate the sensor from 0 to 360 degree along each axis.

4. Click the Stop button to stop the Connected IO Simulation

6. The Magnetic field values are logged in the MATLAB base workspace as **out.MagneticField** variable. Use the `magcal` function on the logged values in MATLAB command window to obtain the correction coefficients.

```
[softIronFactor, hardIronOffset] = magcal(out.MagneticField);
```

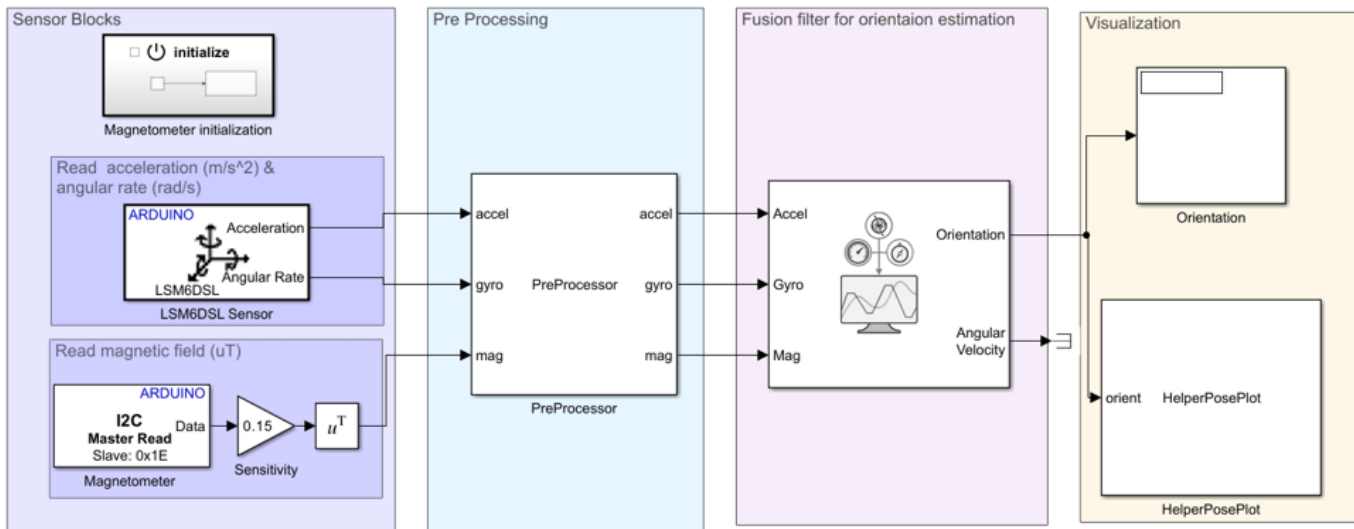
Note: The correction values change with the surroundings.

Task 2. Fuse Sensor Data with AHRS Filter

This section describes how to fuse the sensor data to estimate the orientation. Use the model *EstimateOrientationUsingAHRSAndIMU* for this section.

```
open_system("EstimateOrientationUsingAHRSandIMU.slx");
```

Estimate Orientation using AHRS filter and IMU sensor



Copyright 2022 The MathWorks, Inc.

Sensor Blocks

The first part of the model is for reading sensor values, which is described in the previous section. If you make changes to sensor blocks in the previous task, make the corresponding changes in the blocks in this model as well.

PreProcessor Block

The Preprocessor block in the model accepts acceleration, angular rate, and magnetic field from the sensor and magnetic field correction values. The block outputs the calibrated and axis-aligned sensor values.

Modify the values in the Constant block *Magnetometer correction values*, which is the input to the Preprocessor block, with the correction values (`hardIronOffset`) obtained from the step 6 in **Perform Magnetometer Calibration** section.

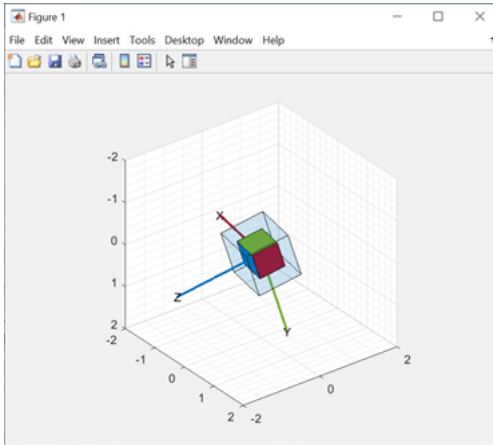
The axes of the accelerometer, gyroscope, and magnetometer in the sensor may not be aligned with each other. Specify the index and sign of x-, y-, and z-axis of each sensor on the PreProcessor block mask, so that the sensor is aligned with the North-East-Down (NED) coordinate system when it is at rest. In this example, the magnetometer Y-axes is changed while the accelerometer and gyroscope axes remain fixed.

Filter Block

To estimate orientation with IMU sensor data, an AHRS (Navigation Toolbox) block is used. The AHRS block fuses accelerometer, magnetometer, and gyroscope sensor data to estimate device orientation. The AHRS block has tunable parameters. Tuning the parameters based on the specified sensors being used can improve performance. For more details, refer to **Tuning Filter Parameters** section in "Estimate Orientation Through Inertial Sensor Fusion" (Navigation Toolbox).

Visualization Block

To visualize the orientation in Simulink, this example provides a helper block, *HelperPosePlot*. The block plots the pose specified by the quaternion or rotation matrix.

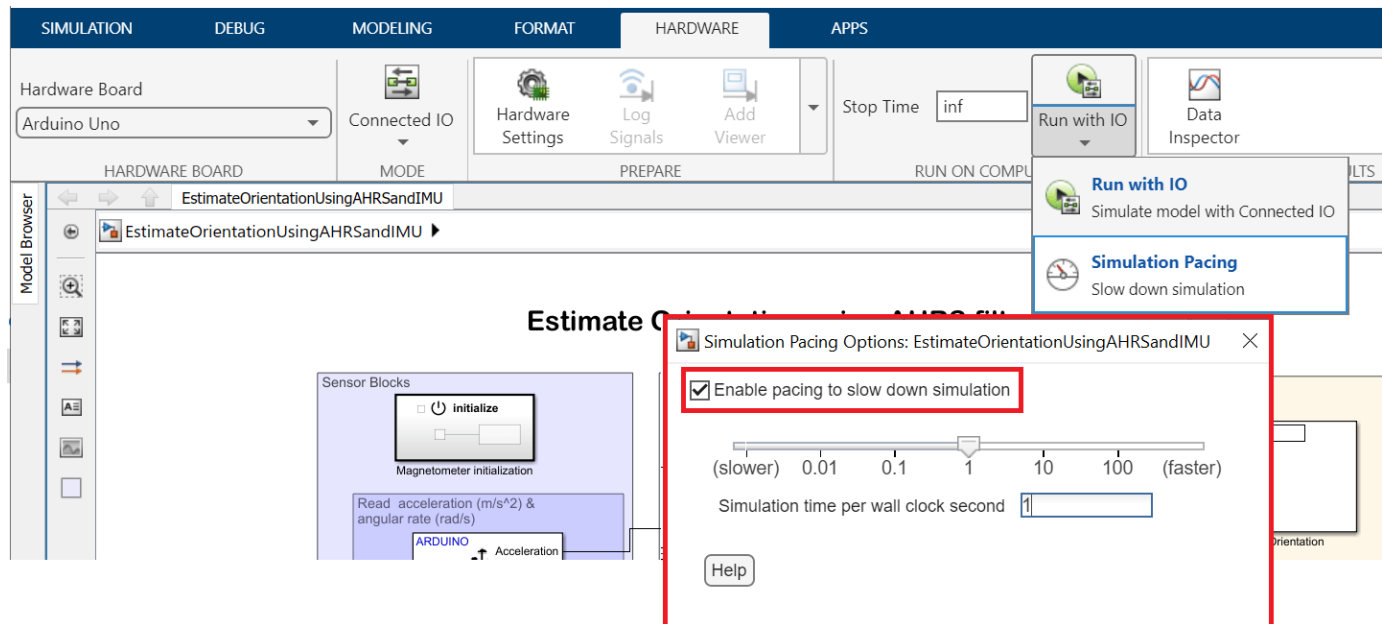


Validate the Model Design Using Connected IO

You can simulate the model in Connected IO to validate the model design before generating the code and deploying the model on Arduino board. This communication between the model and Arduino does not require any code generation or model deployment, thus accelerating the simulation process. For more information on Connected IO, see “Communicate with Hardware Using Connected IO” (Simulink Support Package for Arduino Hardware). The model is already configured to run in Connected IO mode.

This application requires the sensor data to be acquired in real time. To get real time data with Connected IO, you need to enable pacing. To acquire real time data from hardware, do the following:

1. On the Simulink toolbar, click the **Simulation** tab and set the Simulation mode to **Normal**.
2. To run this model in the Connected IO mode, click the **Hardware** tab, go to the **Mode** section, and select **Connected IO**.
3. On the **Hardware** tab, open the dropdown **Run with IO** in the **Run on Computer** section, and select **Simulation Pacing**.
4. Select **Enable pacing to slow down simulation**.



5. Click the Run icon corresponding to **Run with IO** to start the Connected IO Simulation.

Move the sensor and check if the motion in the figure is matching the motion of the sensor.

6. To stop running the model, click the Stop icon corresponding to **Run with IO**.

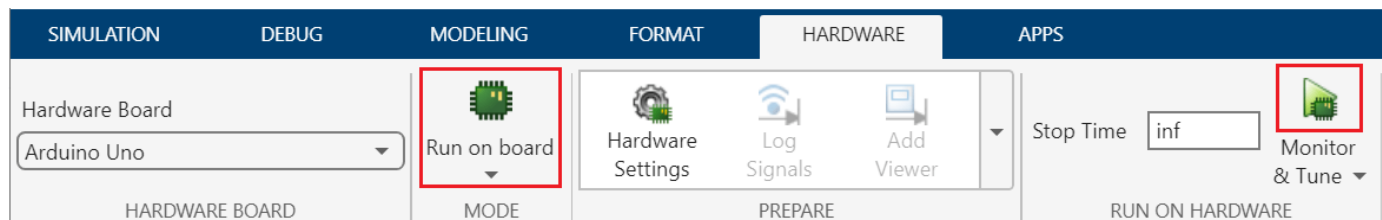
Run the Model in External Mode

After you successfully simulate the model in Connected IO, simulate the model in External mode. Unlike Connected IO, the model is deployed as a C code on the hardware. The code obtains real-time data from the hardware. In external mode, the data acquisition and parameter tuning are done while the application is running on the hardware.

Note: Ensure that the Arduino board you are using has sufficient memory to run the application on hardware. The boards like Arduino Uno, which have low memory, cannot support this application.

Note: The block *HelperPosePlot* is not supported for external mode workflow. To view the orientation in external mode, use other dashboards in Simulink like Scope, Display blocks, and so on.

1. To run External Mode, click the **Hardware** tab, go to the Mode section, select **Run on board (External mode)** and then click **Monitor & Tune**.



The lower left corner of the model window displays status while Simulink prepares, downloads, and runs the model on the hardware.

Move the sensor and verify the orientation values.

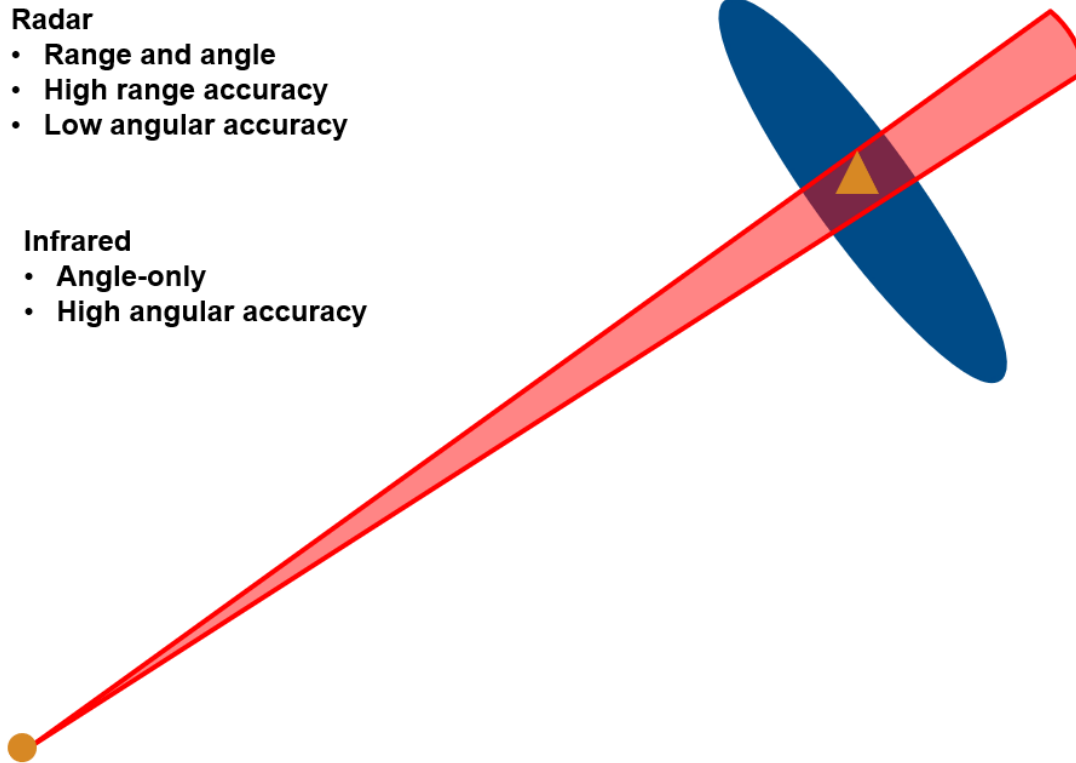
2. To stop running the model, click Stop corresponding to **Monitor and Tune**.

Angle and Position Measurement Fusion for Marine Surveillance

This example shows how to fuse position measurements from radar and angle-only measurements from infrared sensors for marine surveillance. The example illustrates the steps to generate a geo-referenced marine scenario, simulate radar and infrared measurements, and configure a multi-object probability hypothesis density (PHD) tracker to estimate the location and kinematics of objects in the close vicinity to a harbor.

Introduction

Radar and Infrared sensors provide complementary sensing capabilities to assist object tracking. Radar provides both range and angle measurements from an object and offers robust operation in all weather conditions. However, the angular accuracy of radar measurements is typically low. On the other hand, an infrared sensor provides only angular measurements from the object, but with a much higher angular accuracy as compared to a radar. Therefore, the fusion of radar and infrared measurements produces a more robust estimate about the location of an object. You can see visually, the fusion of angular covariance from infrared (red) and radar position covariance (blue) produces an intersection with a much smaller covariance as compared to the radar position covariance.



Radar and IR

Setup Scenario and Visualization

In this section, you setup a marine scenario using the `trackingScenario` object. The scenario consists of small civilian boats close to the base station as well as a large cargo vessel exiting the harbor. You simulated these objects as point targets to generate at most one detection per object at every time step. You set the `IsEarthCentered` property of `trackingScenario` to `true` to simulate geo-referenced scenarios. Further, you use the `geoTrajectory System` object™ to specify trajectories of each object. You mount a mechanically scanning radar and infrared sensor on a stationary base station near the harbor. These sensors provide 180-degree coverage facing the east direction. For more details on the setup, see the helper function, `helperCreateMarineGeoScenario`, included with this example.

```
% Set random seed for reproducible results
rng(2022);
```

```
% Create scenario
scenario = helperCreateMarineGeoScenario;
```

You use the `trackingGlobeViewer` object to visualize the ground truth, instantaneous sensor coverages, detections, and the tracks generated by the tracker. To generate a birds-eye view of the scene, you set the camera position 7 kilometers above the ground using the `campos` object function.

The image below shows the trajectory of all objects in the scene in blue and each blue diamond represents the starting position of the corresponding trajectory.

```
% Create a uifigure
sz = get(groot, 'ScreenSize');
fig = uifigure(Position = [0.1*sz(3) 0.1*sz(4) 0.8*sz(3) 0.8*sz(4)]);

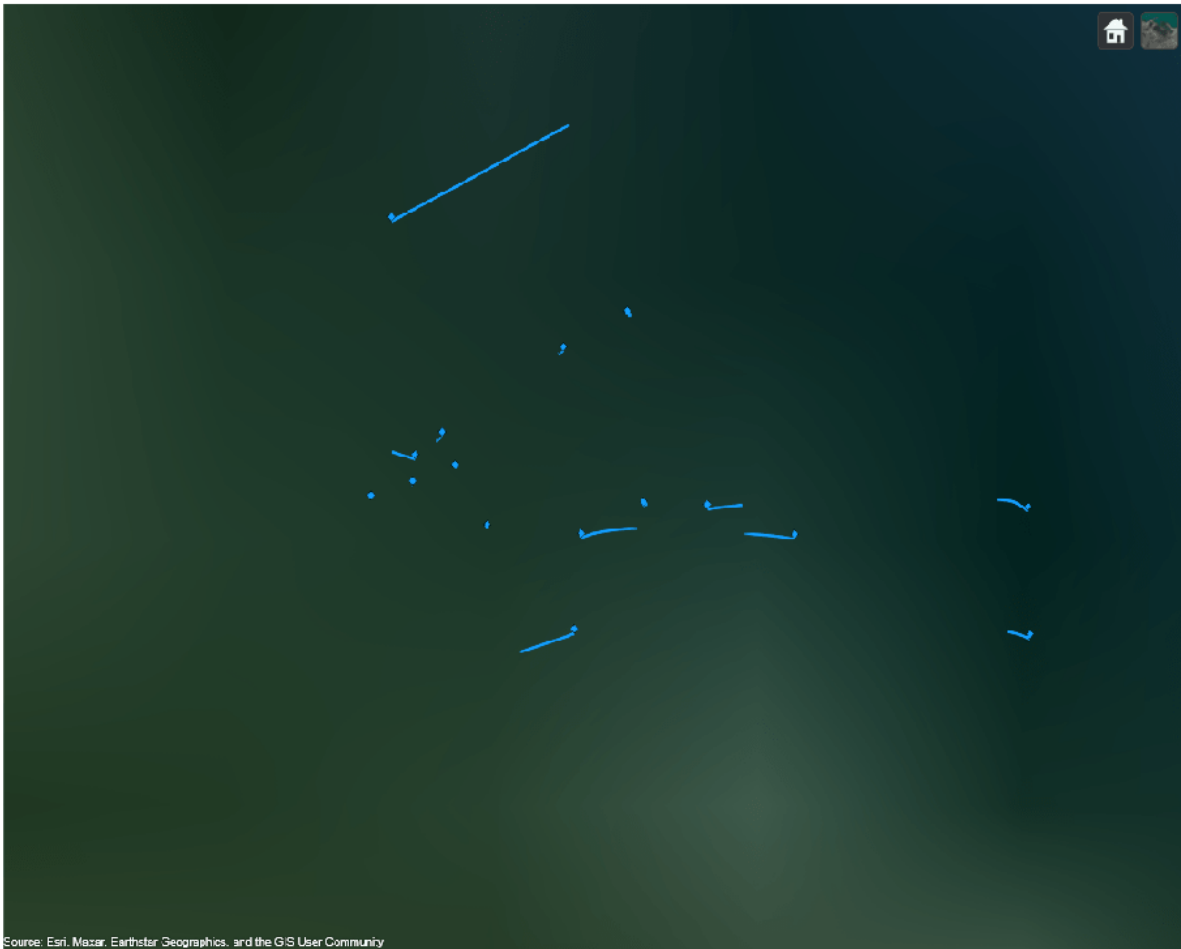
% Create tracking globe viewer
viewer = trackingGlobeViewer(fig, NumCovarianceSigma=1);

% Clear helper with persistence
clear helperPlotAngleOnlyDetection;

% Set position of camera
meanLat = mean(cellfun(@(x)x.Position(1), scenario.Platforms));
meanLong = mean(cellfun(@(x)x.Position(2), scenario.Platforms));
campos(viewer, [meanLat meanLong 7e3]);

% Plot all platforms
plotPlatform(viewer, [scenario.Platforms{:}], ...
    TrajectoryMode="Full", ...
    LineWidth=2, ...
    Marker="d", ...
    Color=[0.0745 0.6235 1.0000]);

% Take snapshot
drawnow;
snapshot(viewer);
```



Source: Esri, Maxar, Earthstar Geographics, and the GIS User Community

Setup Tracker and Performance Metric

In this section, you set up a Gaussian-mixture probability hypothesis density (GM-PHD) multi-object tracker to track the objects using radar and infrared measurements. You also set up the OSPA-on-OSPA (optimal subpattern assignment) or OSPA(2) metric to evaluate the results from the tracker based on the simulated ground truth.

GM-PHD Tracker

You configure the multi-object tracker using the `trackerPHD` System object™, which uses a probability hypothesis density (PHD) filter to estimate the object states. In this example, you represent the state of each object using a point target model and use the Gaussian mixture representation of PHD filter (GM-PHD). The `trackerPHD` object requires definition of sensor configurations using the `trackingSensorConfiguration` object. You can directly create `trackingSensorConfiguration` objects from all the sensors in the scenario using the following command in MATLAB.

```
sensorConfigs = trackingSensorConfiguration(scenario);
```

Additionally, you need to set the `SensorTransformFcn` and `FilterInitializationFcn` properties after constructing the configurations depending on the state-space of the tracks. In this example, you use a constant velocity model (`constvel`, `cvmeas`) to describe the object motion. Therefore, you set the `SensorTransformFcn` to `@cvmeas` to enable transformation from a constant-velocity state space ($[x \ v_x \ y \ v_y \ z \ v_z]$) to the sensor's detectability space.

```
sensorConfigs{1}.SensorTransformFcn = @cvmeas;
sensorConfigs{2}.SensorTransformFcn = @cvmeas;
```

You configure the radar to initialize birth components from weakly-associated detections. As the infrared sensor does not provide complete observations of the objects, you configure it to add no birth components in the environment. You define these behaviors for the tracker by using the `FilterInitializationFcn` property of the `trackingSensorConfiguration`. For more information about birth density and the GM-PHD tracker, refer to the Algorithms section of `trackerPHD`.

The two helper functions, `initRadarPHD` on page 6-1011 and `initIRPHD` on page 6-1012, use the `initcvgmphd` function to initialize the constant-velocity GM-PHD filter. As the radar and infrared sensors provide measurements of different sizes, you pad the infrared measurements with a dummy value to produce same size measurements as required by the tracker. The measurement model and its Jacobian defined using helper functions, `cvmeasPadded` on page 6-1012 and `cvmeasjacPadded` on page 6-1012, inform the tracker about these dummy measurements.

% Define a function to initialize the PHD from each sensor

```
sensorConfigs{1}.FilterInitializationFcn = @initRadarPHD;
sensorConfigs{2}.FilterInitializationFcn = @initIRPHD;
```

% Construct the PHD tracker

```
tracker = trackerPHD(SensorConfigurations=sensorConfigs,...% Sensor configurations
    HasSensorConfigurationsInput=true,...% Sensor configurations change with time
    MaxNumComponents=5000,...% Maximum number of PHD components
    MergingThreshold=200,...% Merging Threshold
    AssignmentThreshold=150 ...% Threshold for adding birth components
);
```

OSPA(2) Metric

In this example, you use the OSPA(2) metric to measure the performance of the tracker. The OSPA(2) metric allows you to evaluate the tracking performance by calculating a metric over history of tracks as opposed to instantaneous estimates in the traditional OSPA metric. As a result, the OSPA(2) metric penalizes phenomenon such as track switching and fragmentation more consistently. You configure the OSPA(2) metric by using the `trackOSPAMetric` object and setting the `Metric` property to "OSPA(2)". You choose absolute error in position as the base distance between a track and truth at a time instant by setting the `Distance` property to "posabserr".

% Define OSPA metric object

```
ospaMetric = trackOSPAMetric(Metric="OSPA(2)",...
    Distance="posabserr",...
    CutoffDistance=10);
```

Run Scenario and Track Objects

In this section, you run the scenario in a loop, generate detections and configurations from the sensor models and feed these data to the multi-object GM-PHD tracker. You pad infrared detections with a dummy measurement using the `padDetectionsWithDummy` on page 6-1012 helper function. To qualitatively assess the performance of the tracker, you visualize the results on the globe. To quantitatively assess the performance, you also evaluate the OSPA(2) metric.


```

% Initialize OSPA metric values
numSamples = scenario.StopTime*scenario.UpdateRate + 1;
ospa = nan(numSamples,1);
counter = 1;

while advance(scenario)
    % Current time
    time = scenario.SimulationTime;

    % Generate detections and updated sensor configurations
    [detections, configs] = detect(scenario);

    % Make infrared measurements of same size
    detections = padDetectionsWithDummy(detections);

    % Update tracker
    tracks = tracker(detections, configs, time);

    % Update globe viewer
    updateDisplay(viewer, scenario, detections, tracks);

    % Obtain ground truth. Last platform is the sensing platform
    poses = platformPoses(scenario);
    gTruth = poses(1:end-1);

    % Calculate OSPA metric
    ospa(counter) = ospaMetric(tracks, gTruth);
    counter = counter + 1;

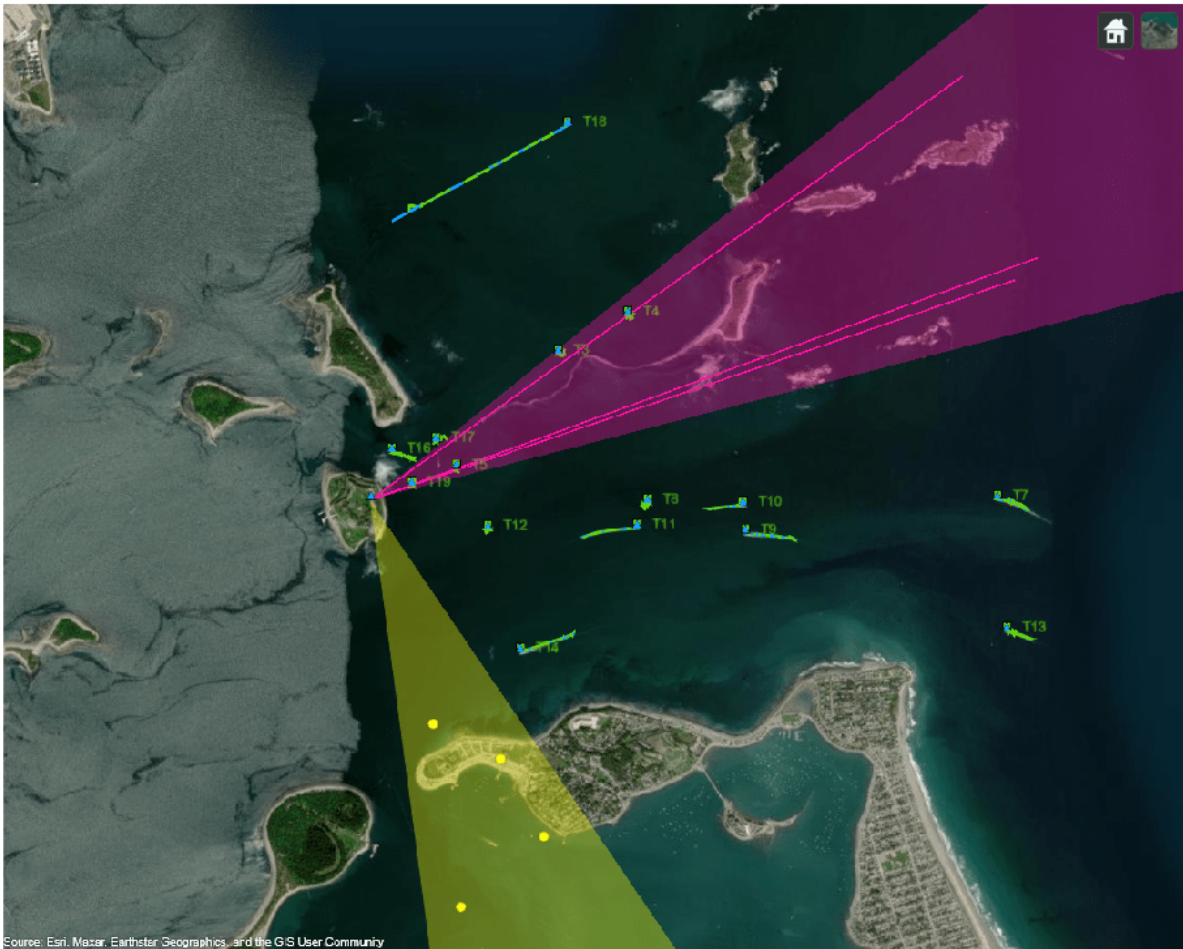
    % Take snapshot of Platform 10 at 5.9 seconds
    if abs((time - 5.9)) < 0.05
        snap = zoomOnPlatform(viewer, scenario.Platforms{10}, 25);
    end
end

```

Results

The image below shows the ground truth as well as the track history for all the platforms. The blue lines represent the ground truth trajectories and the green lines represent the tracks. The yellow and purple coverage areas represent the instantaneous coverage of the radar and infrared sensor, respectively. The purple lines represent the angle-only detections from the infrared sensor. Notice that the tracker is able to maintain tracks on all platforms.

```
snapshot(viewer);
```



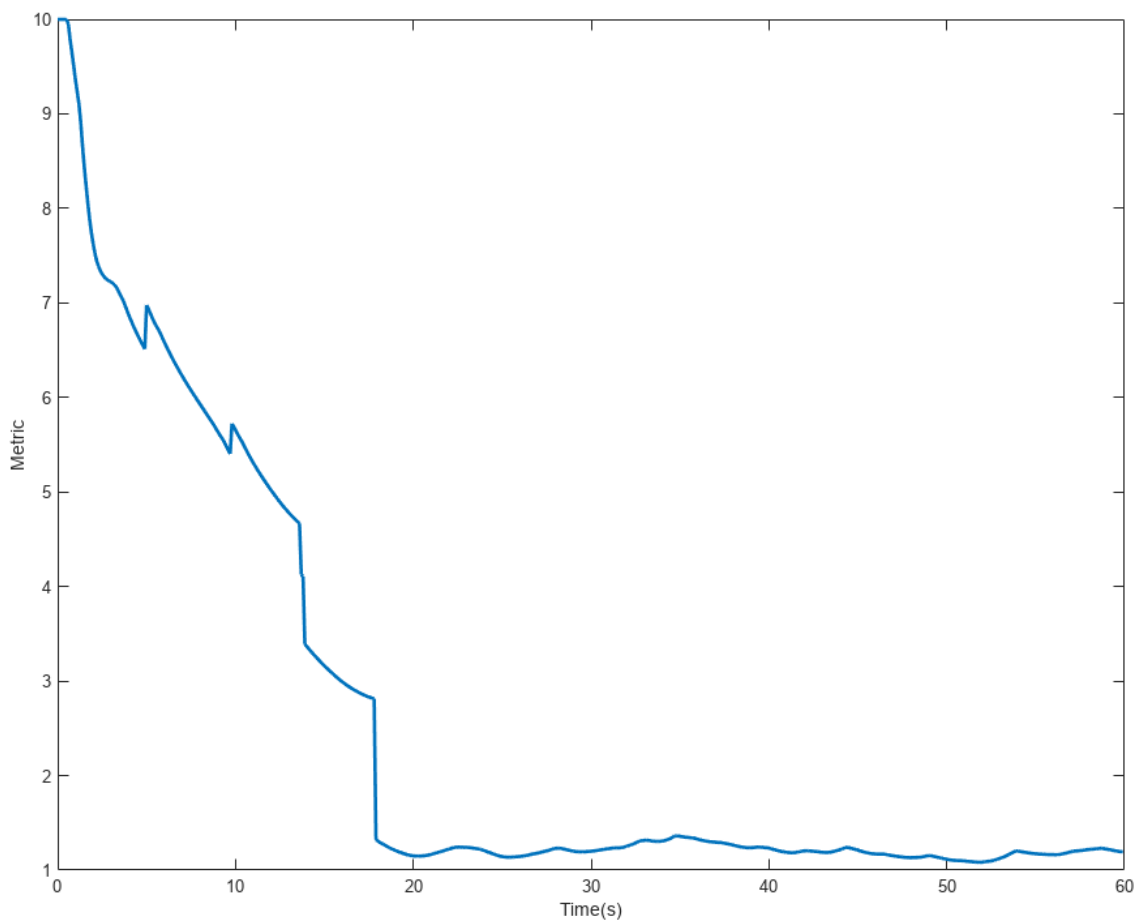
In the next image you see the estimated track on the 8th platform in the scenario. Notice that the tracker maintains a track close to the actual position of the object and closely follows the true trajectory of the platform.

```
zoomOnPlatform(viewer, scenario.Platforms{8}, 250);
```



You can also quantitatively measure the tracking performance by plotting the OSPA(2) metric as a function of time. Notice that the OSPA(2) metric is less than 2 after 15 seconds of simulation. This indicates that the tracker is able to accurately track the positions of the objects. It also indicates that events like missed targets, false tracks as well as track switching and fragmentation did not occur during the simulation.

```
time = linspace(0,scenario.StopTime,numSamples);  
plot(time, ospa,"LineWidth",2);  
xlabel("Time(s)");  
ylabel("Metric");
```



As mentioned, the accurate angle information from the infrared sensor allows the tracker to estimate the position of objects with much higher accuracy. The image below is the snapshot of a platform at about 5.9 seconds. From the image, you can observe the difference in the position uncertainty between the radar measurement and the track estimate. The track's uncertainty (denoted with a green ellipse) is much smaller as compared to the radar measurement uncertainty (denoted with a yellow ellipse). This demonstrates the effectiveness of fusing radar and infrared measurements.

```
imshow(snap);
```



Summary

In this example, you learned how to simulate a geo-referenced marine scenario and generate synthetic measurements using infrared and radar sensors. You also learned how to fuse position and angle-only measurements using a GM-PHD multi-object tracker. You assessed the tracking results by visualizing the tracks and ground truth on the globe and calculating the OSPA(2) metric.

Supporting Functions

initRadarPHD Initialize a constant velocity GM-PHD filter from radar detection.

```
function phd = initRadarPHD(varargin)
% initcvgmphd adds 1 component when detection is an input. It adds no
% components with no inputs.
phd = initcvgmphd(varargin{:});
phd.ProcessNoise = 5*eye(3);

% Update measurement model to use padded measurements
```

```
phd.MeasurementFcn = @cvmeasPadded;
phd.MeasurementJacobianFcn = @cvmeasjacPadded;
end
```

initIRPHD Initialize a constant velocity GM-PHD filter from an IR detection.

```
function phd = initIRPHD(varargin)
% This adds no components
phd = initcvgmphd;
phd.ProcessNoise = 5*eye(3);

% Update measurement model to use padded measurements
phd.MeasurementFcn = @cvmeasPadded;
phd.MeasurementJacobianFcn = @cvmeasjacPadded;
end
```

cvmeasPadded Measurement model with padded dummy variable to output measurements as three-element vectors.

```
% Padded measurement model
function z = cvmeasPadded(x,varargin)
z = cvmeas(x,varargin{:});
numPads = 3 - size(z,1);
numStates = size(z,2);
z = [z;zeros(numPads,numStates)];
end
```

cvmeasjacPadded Jacobian of padded measurement model

```
% Jacobian of padded measurement model
function H = cvmeasjacPadded(x,varargin)
H = cvmeasjac(x,varargin{:});
numPads = 3 - size(H,1);
numStateElements = size(H,2);
H = [H;zeros(numPads,numStateElements)];
end
```

padDetectionsWithDummy Pads infrared detections with a dummy value.

```
function detections = padDetectionsWithDummy(detections)
for i = 1:numel(detections)
% Infrared detections have SensorIndex = 2
if detections{i}.SensorIndex == 2
% Add dummy measurement
detections{i}.Measurement = [detections{i}.Measurement;0];

% A covariance value of 1/(2*pi) on the dummy measurement produces
% the same Gaussian likelihood as the original measurement without
% dummy
detections{i}.MeasurementNoise = blkdiag(detections{i}.MeasurementNoise,1/(2*pi));
end
```

```
end
end
```

updateDisplay Update the globe viewer

```
function updateDisplay(viewer, scenario, detections, tracks)
```

```
% Define position and velocity selectors
```

```
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];
```

```
% Color order
```

```
colorOrder = [1.0000    1.0000    0.0667
              0.0745    0.6235    1.0000
              1.0000    0.4118    0.1608
              0.3922    0.8314    0.0745
              0.7176    0.2745    1.0000
              0.0588    1.0000    1.0000
              1.0000    0.0745    0.6510];
```

```
% Plot Platforms
```

```
plotPlatform(viewer, [scenario.Platforms{:}], 'ECEF',Color=colorOrder(2,:),TrajectoryMode='Full')
```

```
% Plot Coverage
```

```
covConfig = coverageConfig(scenario);
covConfig(2).Range = 10e3;
plotCoverage(viewer,covConfig, 'ECEF',Color=colorOrder([1 7],:),Alpha=0.2);
```

```
% Plot Detections
```

```
sIdx = cellfun(@(x)x.SensorIndex,detections);
plotDetection(viewer, detections(sIdx == 1), 'ECEF',Color=colorOrder(1,:));
helperPlotAngleOnlyDetection(viewer, detections(sIdx == 2),Color=colorOrder(7,:));
```

```
% Plot Tracks
```

```
plotTrack(viewer,tracks, 'ECEF',PositionSelector=posSelector,...
          VelocitySelector=velSelector,LineWidth=2,Color=colorOrder(4,:));
drawnow;
```

```
end
```

zoomOnPlatform Take a snapshot of the platform from a certain height.

```
function varargout = zoomOnPlatform(viewer, platform, height)
```

```
currentPos = campos(viewer);
currentOrient = camorient(viewer);
pos = platform.Position;
campos(viewer,[pos(1) pos(2) height]);
drawnow;
[varargout{1:nargout}] = snapshot(viewer);
campos(viewer,currentPos);
camorient(viewer,currentOrient);
```

```
end
```

Introduction to Class Fusion and Classification-Aided Tracking

This example shows how to use Sensor Fusion and Tracking Toolbox to perform class (or classifier) fusion with a multi-object tracker. You also learn how to use classification to improve the data association of a tracker.

Introduction

Detections generated from regular sensors usually contain kinematic information of the targets, such as measurement and measurement noise. Some sensors also have the capability of identifying class of targets and outputting target classification labels. In the Sensor Fusion and Tracking Toolbox, you represent detections in the form of `objectDetection` objects. With the `objectDetection` object, you specify kinematic information by using the `Measurement`, `MeasurementNoise`, and `MeasurementParameters` properties. Meanwhile, you can specify classification information by using the `ObjectClassID` and `ObjectClassParameters` properties. See the “Convert Detections to objectDetection Format” on page 6-435 example for how to use the `objectDetection` object to create detections.

In most cases, a classifier outputs the classes of detections based on a fixed set of possible labels. For example, `{ClassA, ClassB, ..., ClassZ}`. A classifier usually outputs the classification results in one of the three formats [1]:

- 1 *Crisp* - The output contains a single label, for instance 'ClassB'.
- 2 *Ranked* - The output provides a ranked list of all labels, from the most likely one to the least likely one, for instance 'ClassB > ClassC > ClassA > ...'.
- 3 *Soft* - The output consists of scores or probabilities for all labels.

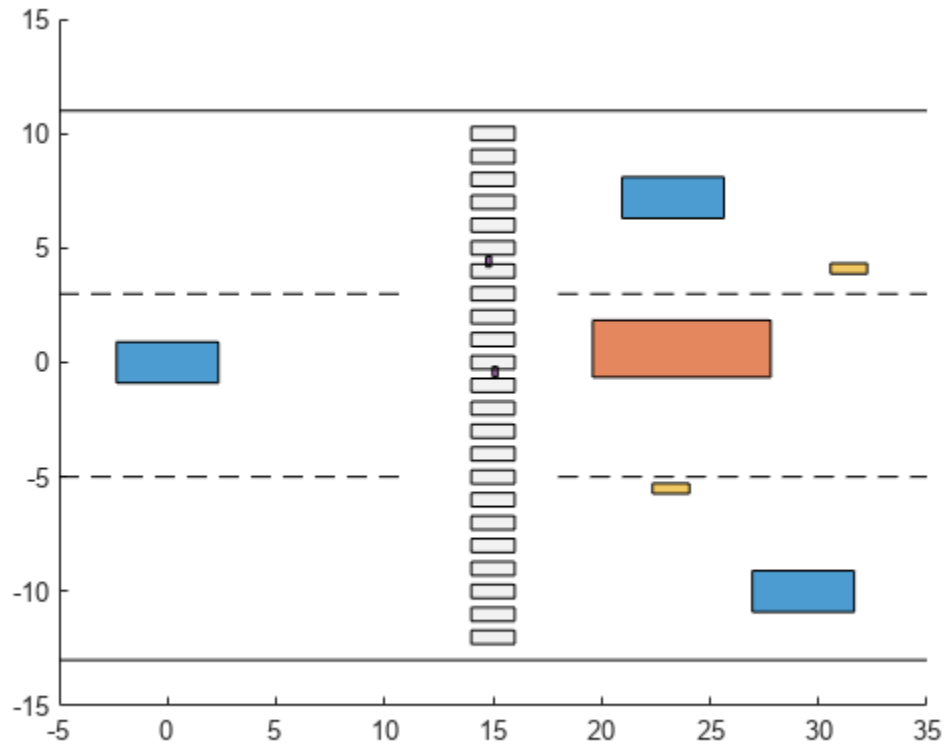
You can always convert a Soft or Ranked output to a Crisp output by selecting the most likely class. The Sensor Fusion and Tracking Toolbox uses the Crisp representation.

Class Fusion

In the context of object tracking, you can use class fusion for two main objectives. First, maintain and improve the class estimation for each tracked object over time. Second, fuse different and possibly conflicting classification results that are assigned to the same object.

In this example, consider the 4 following classes 'Car', 'Truck', 'Bicycle', and 'Pedestrian' as well as a simple traffic crosswalk scenario. In the scenario, two pedestrians are walking on the crosswalk, whereas a truck and two other cars stop on the right side of the crosswalk. Additionally, two bicycles are in between the truck and the cars. The ego vehicle is located at coordinates [0, 0]. Use the `helperClassFusionDisplay` class to visualize the scenario.

```
helperClassFusionDisplay.plotClassFusionScene();
```

The ego vehicle is equipped with two vision detectors (camera paired with a classifier). Each vision detector provides detections with a crisp classification. The two vision detectors have different classification accuracies, represented by their respective confusion matrices, normalized by row.

$$C_1 = \begin{bmatrix} 0.90 & 0.05 & 0.03 & 0.02 \\ 0.05 & 0.90 & 0.03 & 0.02 \\ 0.20 & 0.20 & 0.30 & 0.30 \\ 0.20 & 0.20 & 0.30 & 0.30 \end{bmatrix} \text{ and } C_2 = \begin{bmatrix} 0.30 & 0.20 & 0.02 & 0.02 \\ 0.05 & 0.30 & 0.03 & 0.03 \\ 0.02 & 0.03 & 0.90 & 0.05 \\ 0.02 & 0.03 & 0.05 & 0.90 \end{bmatrix}$$

The first confusion matrix denotes that if the true class of a target is 'Truck' the classifier classifies the detection 90% of the time as a 'Truck', 5% of the time as 'Car', 3% of the time as bicycle, and finally 2% of the time as a pedestrian. Therefore, the first detector performs well when classifying Trucks and Cars but performs poorly when classifying bicycles and pedestrians. The second detector, on the contrary, is more accurate on 'Bicycles' and 'Pedestrians' classification.

Load the synthetic vision detection data into the workspace. For each vision detector, inspect the confusion matrix saved in the `ObjectClassParameters` property of the first `objectDetection` object at the first time step.

```
load("visionDetectionLog.mat", "datalog");
disp(datalog(1).Detections{1}.ObjectClassParameters.ConfusionMatrix);

0.9000    0.0500    0.0300    0.0200
0.0500    0.9000    0.0300    0.0200
0.2000    0.2000    0.3000    0.2000
0.2000    0.2000    0.3000    0.3000
```

```
disp(datalog(1).Detections2{1}.ObjectClassParameters.ConfusionMatrix);  
  
    0.3000    0.2000    0.2000    0.2000  
    0.2000    0.3000    0.2000    0.2000  
    0.0300    0.0200    0.9000    0.0500  
    0.0300    0.0200    0.0500    0.9000
```

To fuse the detections across time, create a `trackerGNN` object and set the `ClassFusionMethod` property to "Bayes". Also set the `InitialClassProbabilities` property to model an environment with a uniform a priori distribution of classes.

```
globalTracker = trackerGNN(ClassFusionMethod="Bayes", ...  
    InitialClassProbabilities=[0.25 0.25 0.25 0.25]);
```

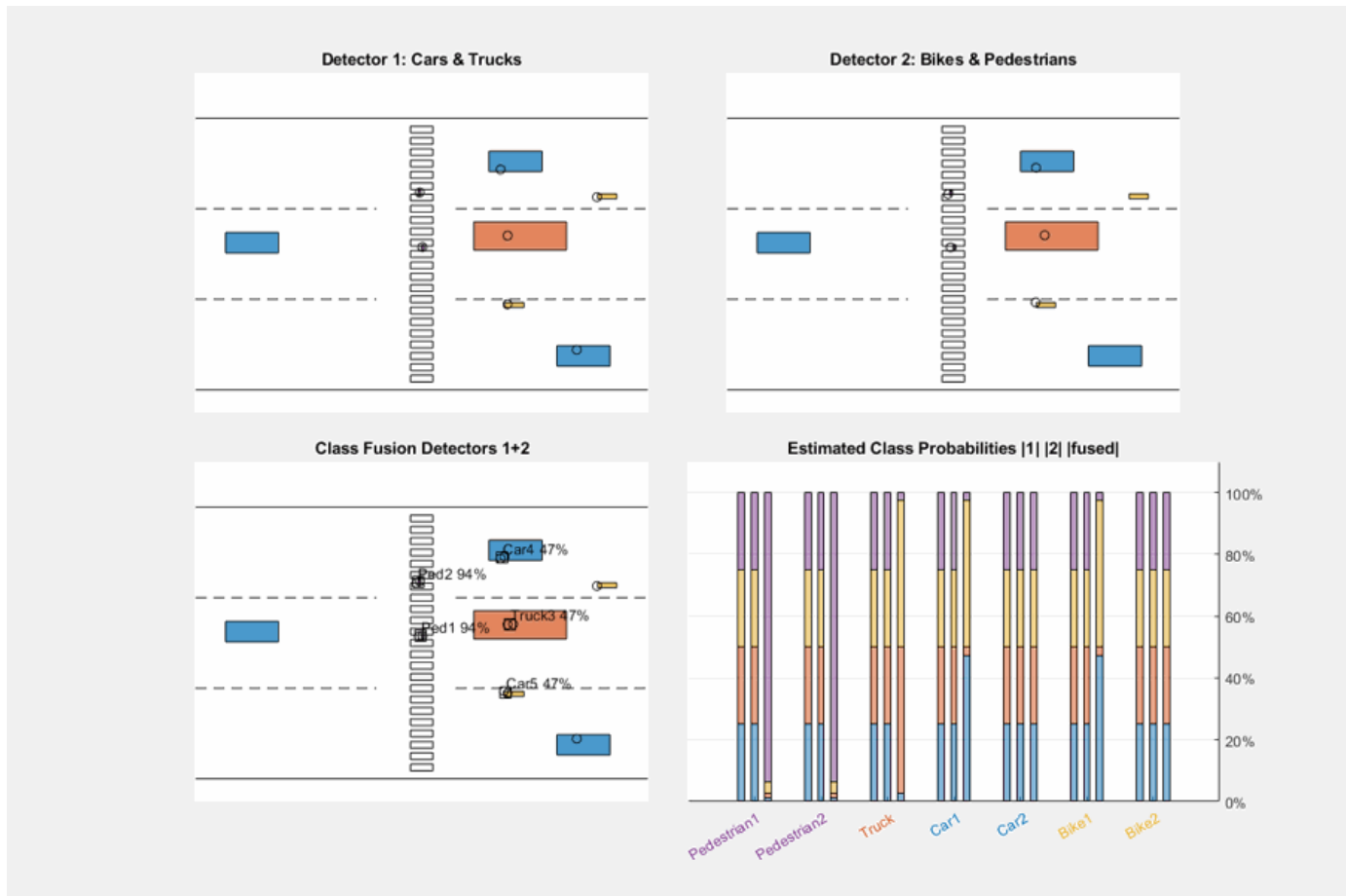
Clone the tracker twice for tracking with each individual detector.

```
localTracker1 = clone(globalTracker);  
localTracker2 = clone(globalTracker);
```

Simulate the tracking process and visualize the results using the `helperClassFusionDisplay` helper class

```
display = helperClassFusionDisplay();  
  
for i=1:numel(datalog)  
  
    time = datalog(i).Time;  
    dets1 = datalog(i).Detections1;  
    dets2 = datalog(i).Detections2;  
  
    tracks = globalTracker([dets1; dets2], time);  
    tracks1 = localTracker1(dets1, time);  
    tracks2 = localTracker2(dets2, time);  
  
    update(display, dets1, dets2, tracks, tracks1, tracks2);  
end
```

The GIF below shows the recorded animation of the scene.



The top left tile shows the tracking result based only on the detections from the first detector. From its confusion matrix, this detector provides accurate classification for cars and trucks, which allows the class estimation algorithms to accurately predict the classes of the two cars and the truck. However, this detector poorly classifies bicycles, which leads to wrong classifications for the two bicycles.

The top right tile shows the tracking result based only on detections from the second detector. In this case, all the pedestrians and bicycles are correctly classified.

The bottom left tile shows the results obtained when fusing detections from the two detectors. The high confidence of the first detector in classifying cars and trucks combined with the high confidence of the second detector in classifying pedestrians and bicycles provides optimal results.

The bar chart on the bottom right tile shows the class probability distributions for each target. The class probabilities for each target are stacked in a column. For each object, the left column corresponds to the first detector, the middle column corresponds to the second detector, and the right column corresponds to fused results from both detectors. At the end of the simulation, you see that each target is correctly estimated with 100% probability by the fusion of the two detectors.

Classification-Aided tracking

In this section, you explore a scenario in which measurement-to-track association benefits from classification data. This is often referred to as classification-aided tracking. Tracking closely spaced

targets that move in similar patterns is difficult because noisy kinematic measurements can have comparable likelihoods to be associated to the same target. This example uses the scenario proposed in [2], in which six targets fly in formation. Assume that there are four different target classes in total and that the six targets belong to class *Class1*, *Class2*, *Class3*, *Class4*, *Class3*, and *Class4*, respectively.

Load the classification-aided tracking scenario data from a MAT-file and inspect the confusion matrix.

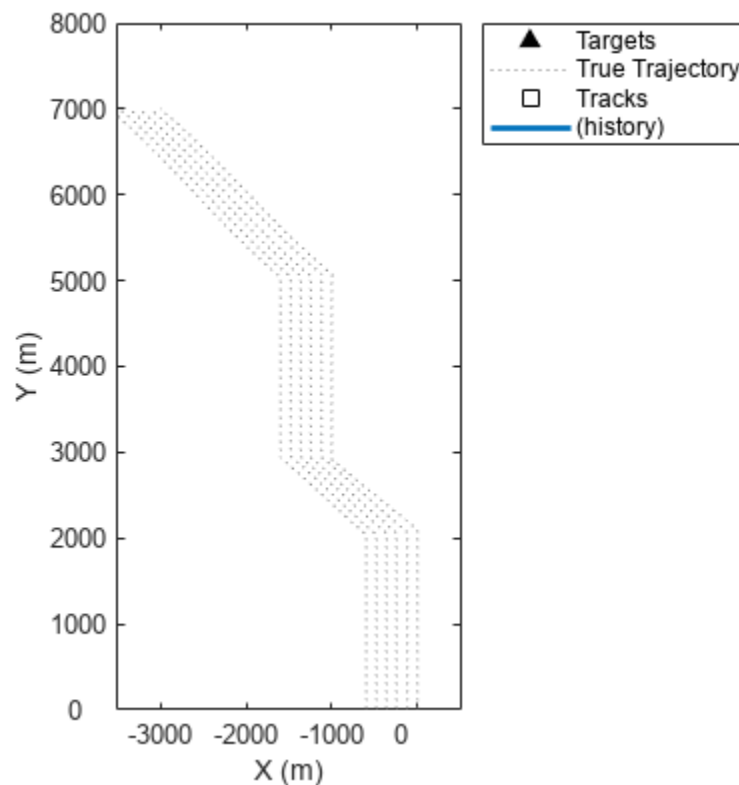
```
load('classAidedScenarioData.mat','scenario','allData');
disp(allData(1).Detections{1}.ObjectClassParameters.ConfusionMatrix);
```

```
0.8500    0.0500    0.0500    0.0500
0.0500    0.8500    0.0500    0.0500
0.0500    0.0500    0.8500    0.0500
0.0500    0.0500    0.0500    0.8500
```

From the confusion matrix, the classifier classifies correctly 85% of the time and misclassification are equiprobable.

Next, use a helper class to visualize the trajectories of the six targets. The targets are closely spaced and kinematic ambiguity of measurements is unavoidable.

```
helperClassAidedTrackingDisplay.plotClassAidedScene(scenario);
```



Configure a tracker with a default interacting multiple model (IMM filter) because the targets perform maneuvers in the scenario. Set the `ClassFusionMethod` property to "Bayes" and the `InitialClassProbabilities` property to the known distribution of target classes in the scenario.

```


gnn = trackerGNN(AssignmentThreshold = 100,...
    MaxNumTracks=10,...
    FilterInitializationFcn="initekfirm",...
    ClassFusionMethod="Bayes",...
    InitialClassProbabilities = [0.2 0.2 0.3 0.3]);

```

The `ClassFusionWeight` property defines how the tracker combines the kinematic and classification costs to obtain the overall data association cost.

$$\text{OverallCost} = (1 - \text{ClassFusionWeight}) * \text{KinematicCost} + \text{ClassFusionWeight} * \text{ClassCost}$$

The tracker then uses the assignment algorithm, specified by the `Assignment` property, to calculate the optimal assignment between detections and tracks based on the `OverallCost`. The `ClassFusionWeight` property ranges from 0 (kinematics only) to 1 (classification only). Move the slider control below to observe how the tracking performance varies with the weight change.

gnn.ClassFusionWeight = 0.7  ;

Simulate the tracking process, save the information analysis output from the tracker, and visualize the tracking results.

```

display2 = helperClassAidedTrackingDisplay(scenario);
infoLog = cell(1,numel(allData));
tracks = objectTrack.empty;

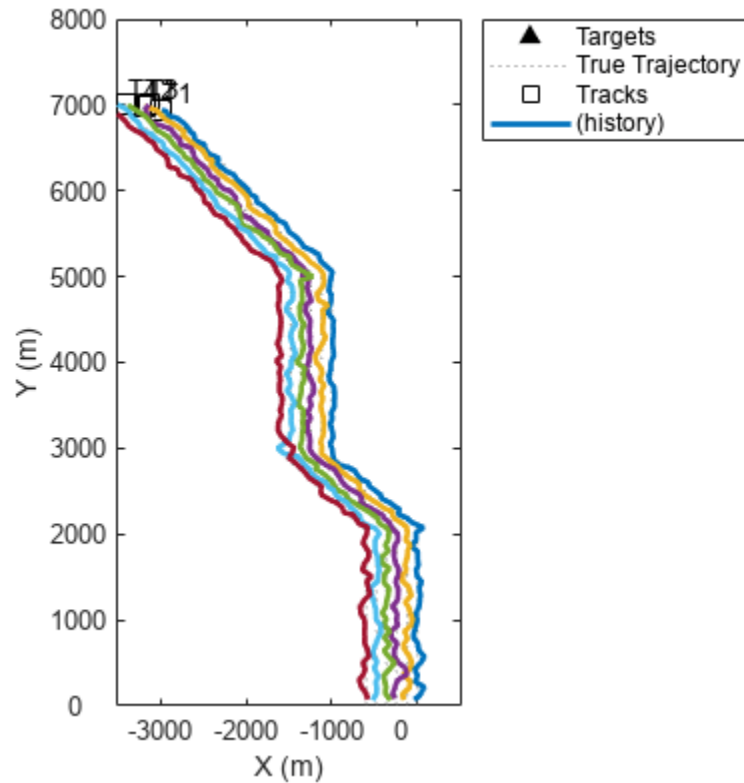
for i=1:numel(allData)

    time = allData(i).Time;
    dets = allData(i).Detections;

    % Update tracker
    [tracks, ~,~, info] = gnn(dets,time);

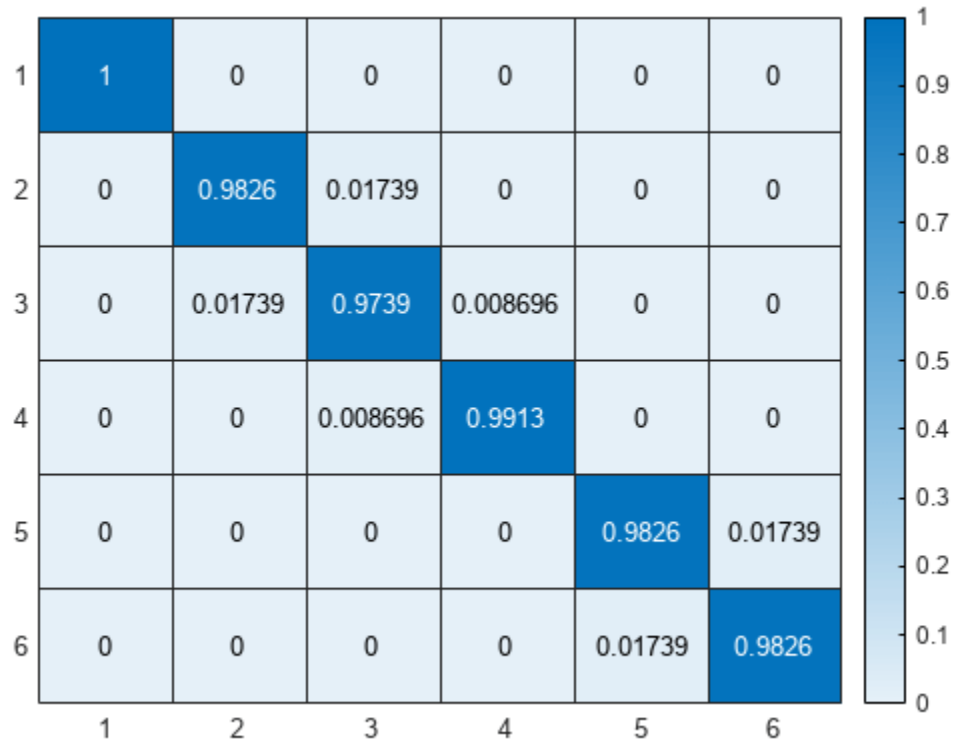
    update(display2, tracks);
    % Store info log
    infoLog{i} = info;
end

```



You can use a purity matrix plot to assess the data association performance in this scenario. In the purity matrix, the (i,j) entry represents the percentage of detections that originate from the true target j assigned to the estimated track i . A tracking algorithm with perfect association would lead to an identity purity matrix. The further away the main values are from the matrix diagonal, the worse the association is. Note that early track swap, can result in a purity matrix with main values on the secondary diagonal. This, however, still indicates a good data association. You can see such a case by setting `ClassFusionWeight` property to 0.6.

```
plotPurityMatrix(display2,infoLog)
```



Note that for this example, the best results are obtained for a value of 0.7. It is expected that using only kinematics will lead to poor tracking because of the nature of the scenario. However, relying only on classification is not a good option either, because any misclassification can result in false association and the kinematics must contribute in data association.

Conclusion

In this example you learned how to configure the `trackerGNN` object to fuse classified detections with a Bayesian Product class fusion algorithm. Class fusion that operates on crisp classifications with the knowledge of the confusion matrix allows you to estimate the probability of track classes. In the second part, you use classification information to improve data association in an ambiguous scenario. You learned balancing the kinematics and classification association costs can refine the overall tracking performance.

References

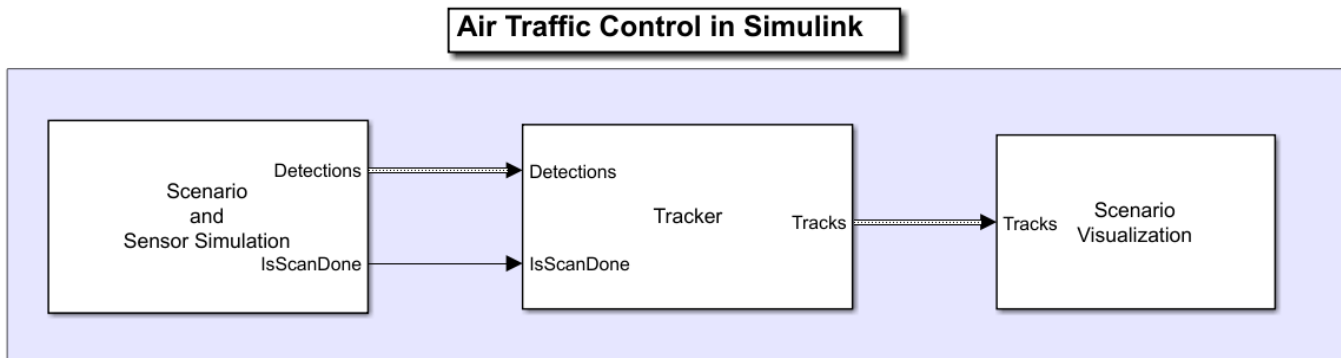
- [1] Kuncheva, Ludmila. *Fuzzy classifier design*. Vol. 49. Springer Science & Business Media, 2000.
- [2] Bar-Shalom, Yaakov, Thia Kirubarajan, and Cenk Gokberk. "Tracking with classification-aided multiframe data association." *IEEE Transactions on Aerospace and Electronic systems* 41, no. 3 (2005): 868-878.

Air Traffic Control in Simulink

This example shows how to create a tracking system for an air traffic control center using a global nearest neighbor (GNN) tracker in Simulink. This example closely follows the “Air Traffic Control” on page 6-2 MATLAB® example.

Overview of the Model

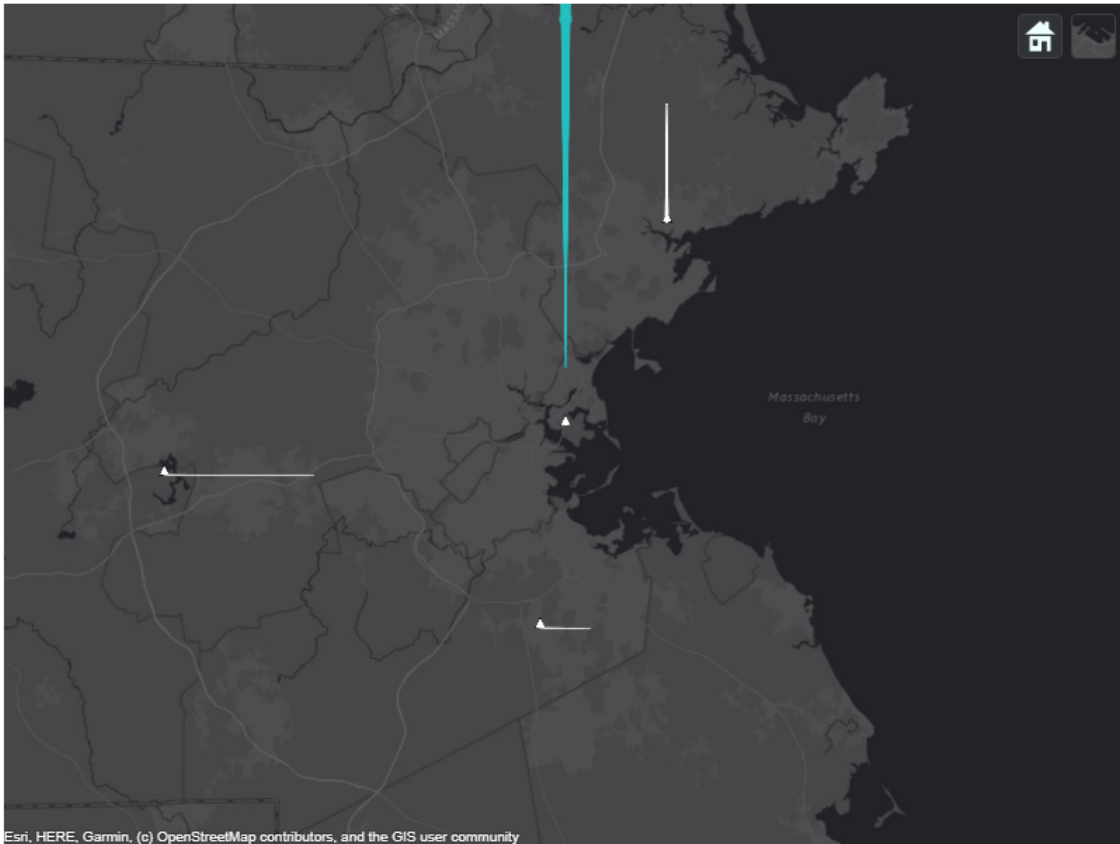
```
model = 'AirTrafficControlInSimulink';  
open_system(model);
```



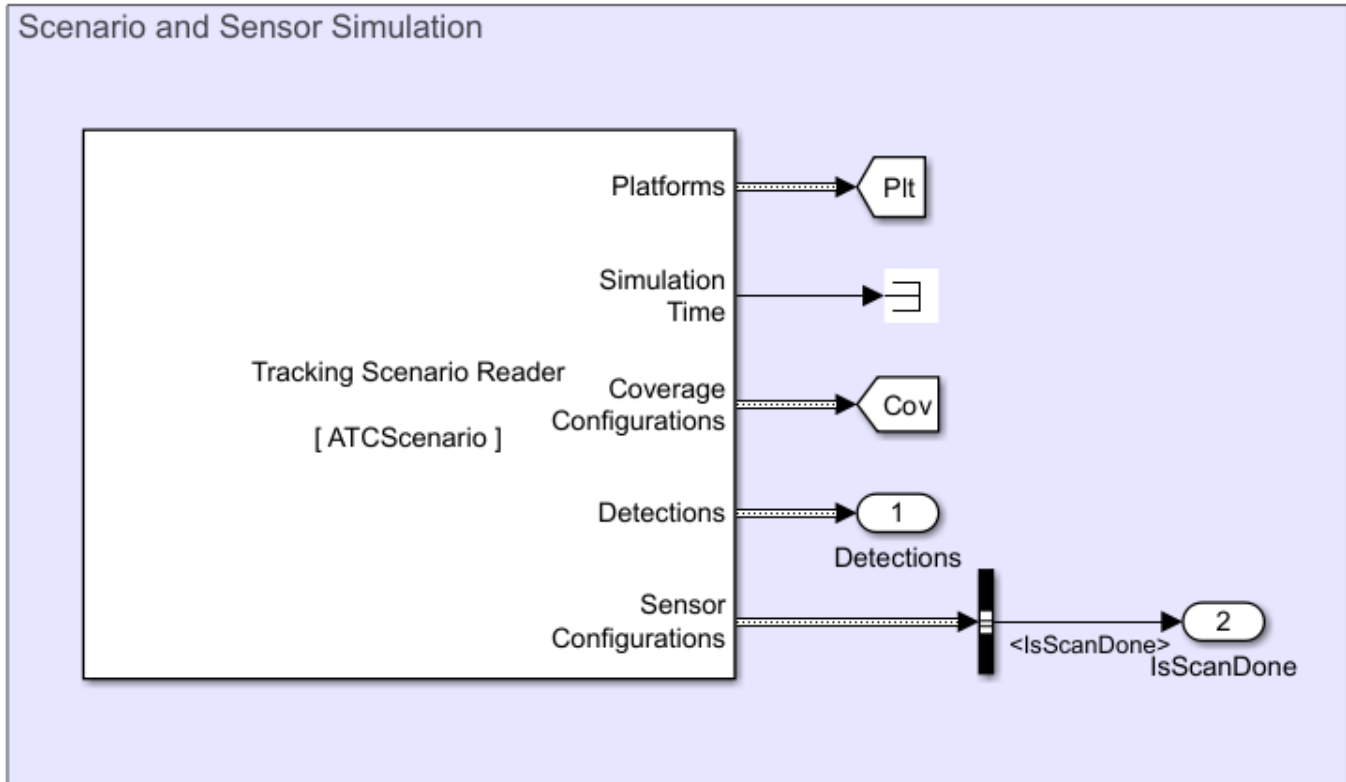
The model has three subsystems, each implementing a part of the workflow.

- Scenario and Sensor Simulation
- Tracker
- Scenario Visualization

Scenario and Sensor Simulation



You create the scenario using the `trackingScenario` object. You add a `platform` to represent the ATC tower and add three `platform`s to represent three airliners. One airliner approaches the ATC from a long range, another departs, and the third is flying tangential to the tower. You add an airport surveillance radar (ASR) to the ATC tower `platform`. 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. You use the `fusionRadarSensor` object to model the radar. See the `createATCScenario.m` file in the example folder for more details on the configuration of the scenario and radar.



The Tracking Scenario Reader block reads the tracking scenario object from the MATLAB workspace and generates simulation data. You set the sample time for the block as the reciprocal of the update rate of the radar sensor. You configure the block to additionally output coverage configurations, detections, and sensor configurations from the scenario. You also specify an initial seed to have reproducible results.

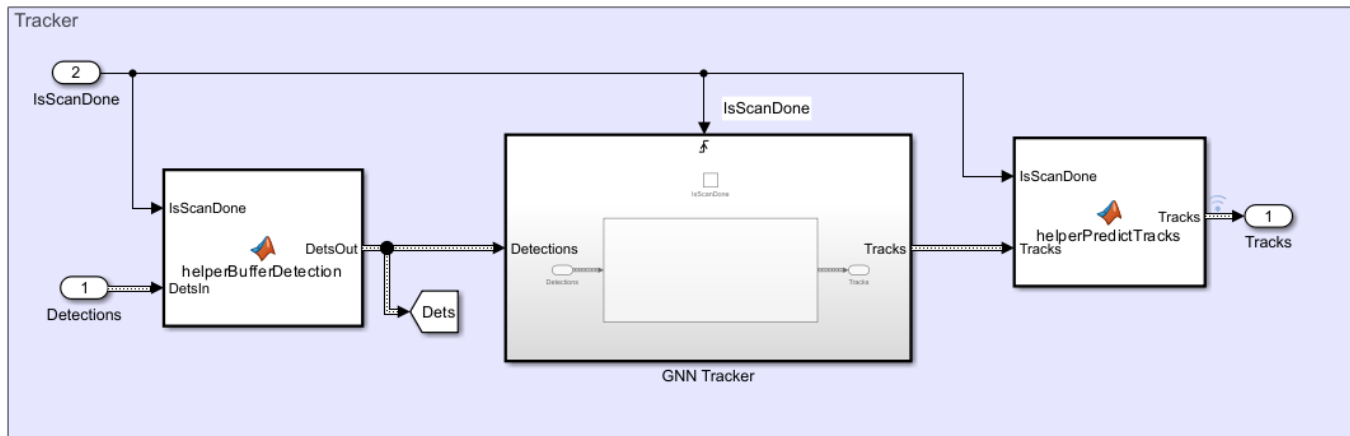
Scenario	Output Settings	Bus Settings
Source of scenario:	trackingScenario	
Workspace variable name:	ATCSenario	trackingScenario
		Browse
		Refresh Scenario Data
Source of platform pose:	Scenario	
Sample time (s):	1/updateRate	0.025
Coordinate system to report platform poses:	Cartesian	

Scenario	Output Settings	Bus Settings
Platforms		
<input type="checkbox"/> Include profiles information with platforms		
Sensors and emitters		
<input checked="" type="checkbox"/> Detections		
<input type="checkbox"/> Point clouds		
<input type="checkbox"/> Emissions		
Configurations		
<input checked="" type="checkbox"/> Coverage		
<input checked="" type="checkbox"/> Sensor		
<input type="checkbox"/> Emitter		
Random number generator settings		
Random number generation:	Specify seed	
Initial seed:	2020	

The radar sensor is modeled as a rotating radar which scans mechanically across the azimuth and elevation directions and completes a full scan within multiple updates. The `IsScanDone` bus element in the sensor configuration is returned as `true` when the sensor completes a full scan. You use the Bus Selector (Simulink) block to extract the `IsScanDone` element from the sensor configuration bus output.

Filter by name	Find	Selected elements	Up
<div style="border: 1px solid gray; padding: 5px;"> <p style="text-align: center;">Elements in the bus</p> <ul style="list-style-type: none"> ▼ Configurations <ul style="list-style-type: none"> SensorIndex IsValidTime IsScanDone FieldOfView RangeLimits RangeRateLimits > MeasurementParameters NumConfigurations </div>	<div style="border: 1px solid gray; padding: 5px;"> <p>Select>></p> <p>Refresh</p> </div>	<div style="border: 1px solid gray; padding: 5px;"> <p>Configurations.IsScanDone</p> </div>	<div style="border: 1px solid gray; padding: 5px;"> <p>Down</p> <p>Remove</p> </div>
		<input type="checkbox"/> Output as virtual bus	

Tracker



IsScanDone



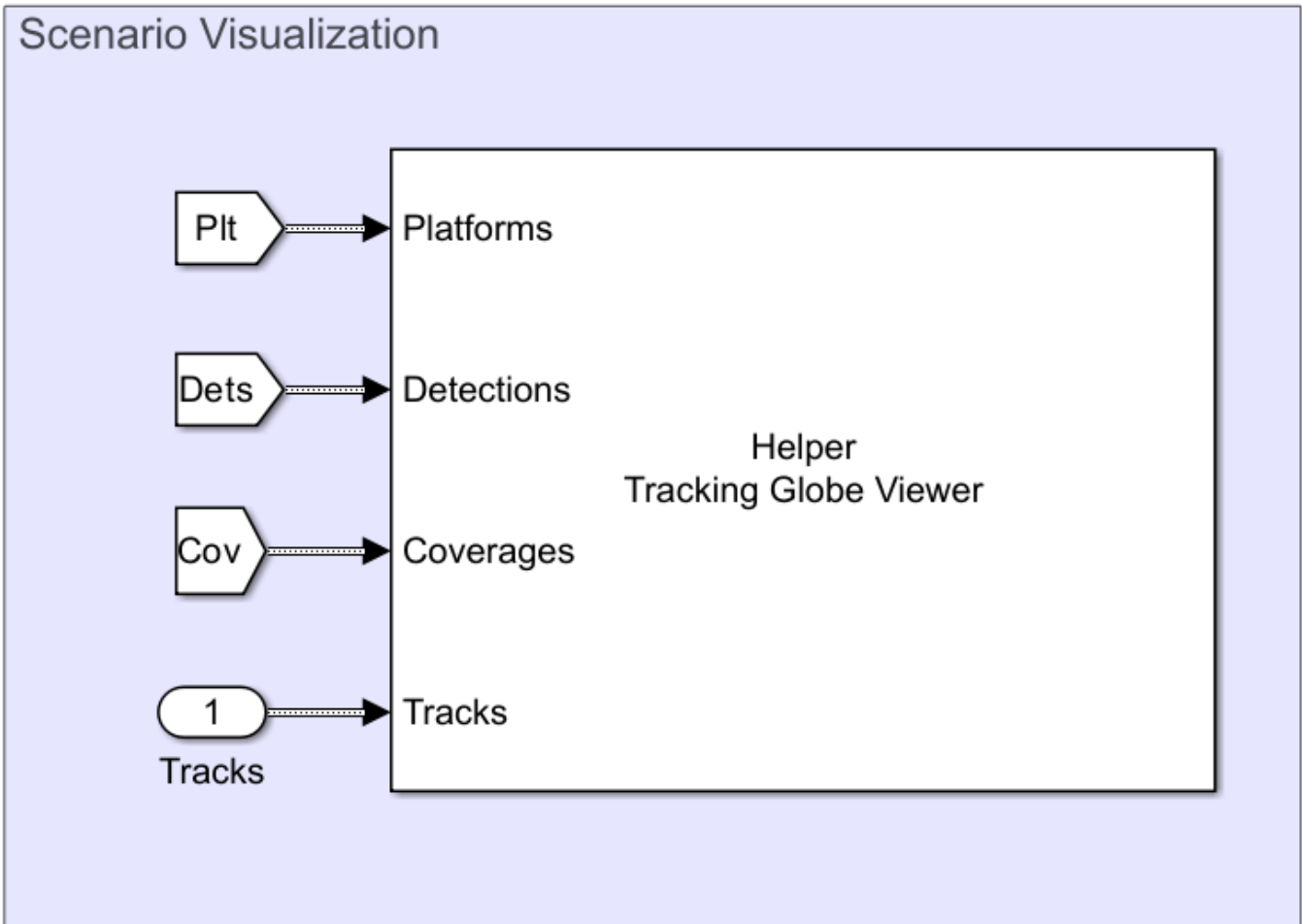
For each step in the scenario, the radar generates detections from targets in its field of view. After the radar completes a 360-degree scan in azimuth, the tracker updates with detections from the radar. Detections from the sensors are buffered until a full scan is complete. The buffer is implemented using a MATLAB Function (Simulink) block. When the `IsScanDone` flag is `true`, the buffer is released.

To ensure that the tracker only updates when a full sensor scan is complete, add a Global Nearest Neighbor Multi Object Tracker block in a Triggered Subsystem (Simulink) block. The block is triggered when `isScanDone` is true. When the full scan is not complete during the intermediate sensor updates, the tracks from previous time step are predicted to the current simulation time using a `constvel` motion model. The `helperPredictTracks` block is implemented using a MATLAB Function block.

Tracker Management	Track Logic	Port Setting
Tracker identifier:	1	
Filter initialization function:	initATCFilter	
Maximum number of tracks:	100	
Maximum number of sensors:	20	
Out-of-sequence measurements handling:	Terminate	
Track state parameters:	struct	
<input type="checkbox"/> Update track state parameters with time <input type="checkbox"/> Enable memory management		
Assignment		
Assignment algorithm name:	Auction	
Threshold for assigning detections to tracks:	[50.0, Inf] [50,Inf]	
Cluster tracks and detections for assignment:	off	
Tracker Management Track Logic Port Setting		
Type of track confirmation and deletion logic:	History	
Confirmation threshold [M N]:	[2, 3] [2,3]	
Deletion threshold [P Q]:	[5, 5] [5,5]	

You configure the tracker block by setting the assignment algorithm to `Auction`, and threshold for assigning detection to tracks to 50. You also set the filter initialization function to `initATCFilter`, which initializes a constant velocity extended Kalman filter for each new track. See `initATCFilter.m` in the example folder for more details. You use the default confirmation and deletion thresholds to confirm and delete tracks, respectively. A confirmation threshold [2, 3] means that a track is confirmed if a detection is assigned to it at least 2 times in the last 3 tracker updates and a deletion threshold [5, 5] means that a confirmed track is deleted if any detection is not assigned to the track in the last 5 tracker updates.

Visualization



Parameters	
Map reference location:	[42.366978, -71.022362, 50] <small>[42.367,-71.022,50]</small>
Display option for sensor coverages:	Beam
Covariance ellipse size in number of sigma:	2
Length of platform trajectory history line:	1000
Length of track history line:	1000
Track label scaling factor:	1.1
<input checked="" type="checkbox"/> Show dropped tracks on the globe	
Map to plot data:	streets-dark
Terrain of globe:	none
Tracking scenario object to plot static data:	ATCScenario <small>trackingScenario</small>
Tracking globe viewer uiFigure tag:	ATCInSimulinkVisualization
Scenario plotting function:	Custom
Specify custom plotting function:	customTGVPlotter

Implemented using the MATLAB System (Simulink) block, the Helper Tracking Globe Viewer block visualizes the scenario bases on the `trackingGlobeViewer` object. See the `helperScenarioVisualization.m` class in the example folder for the definition of the helper block.

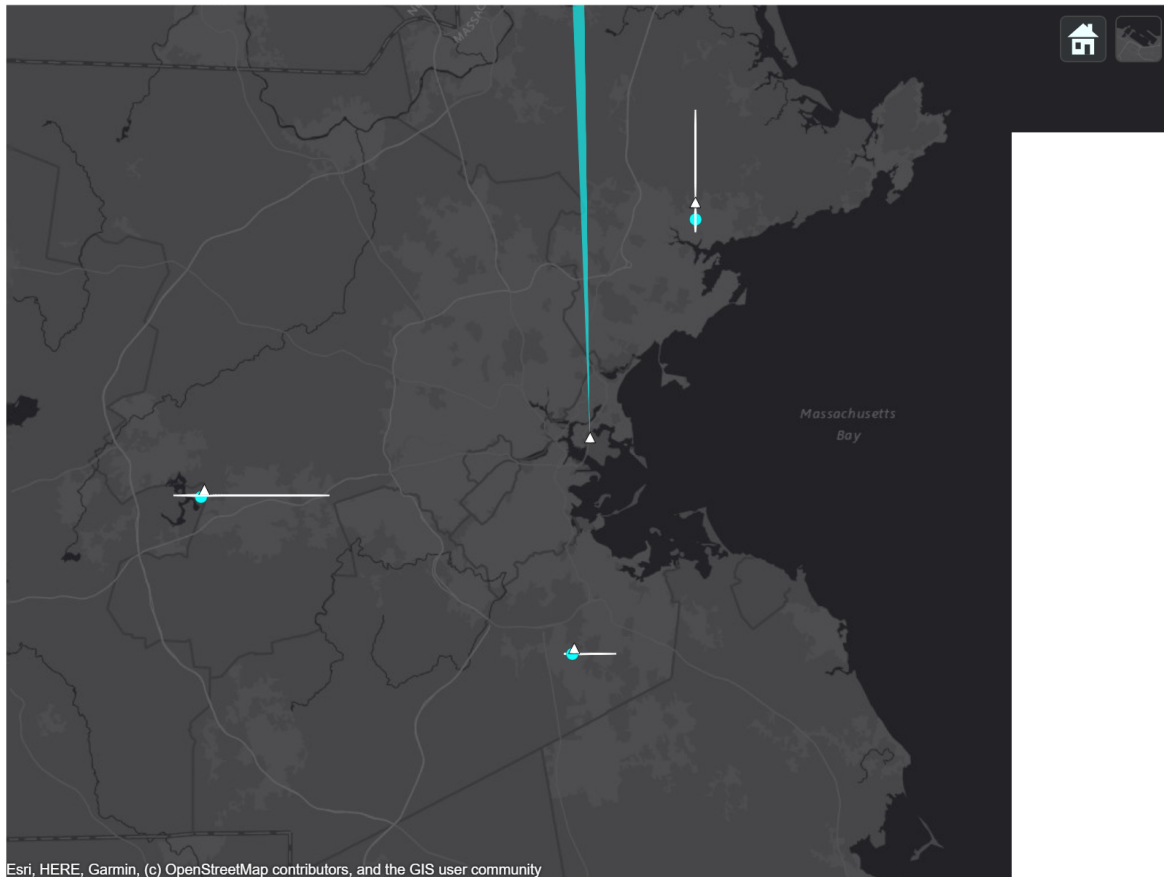
Simulate and Track Airliners

Simulate the model and obtain the output.

```
simOut = sim(model);
```

Show the first snapshot taken when the radar completes the second scan.

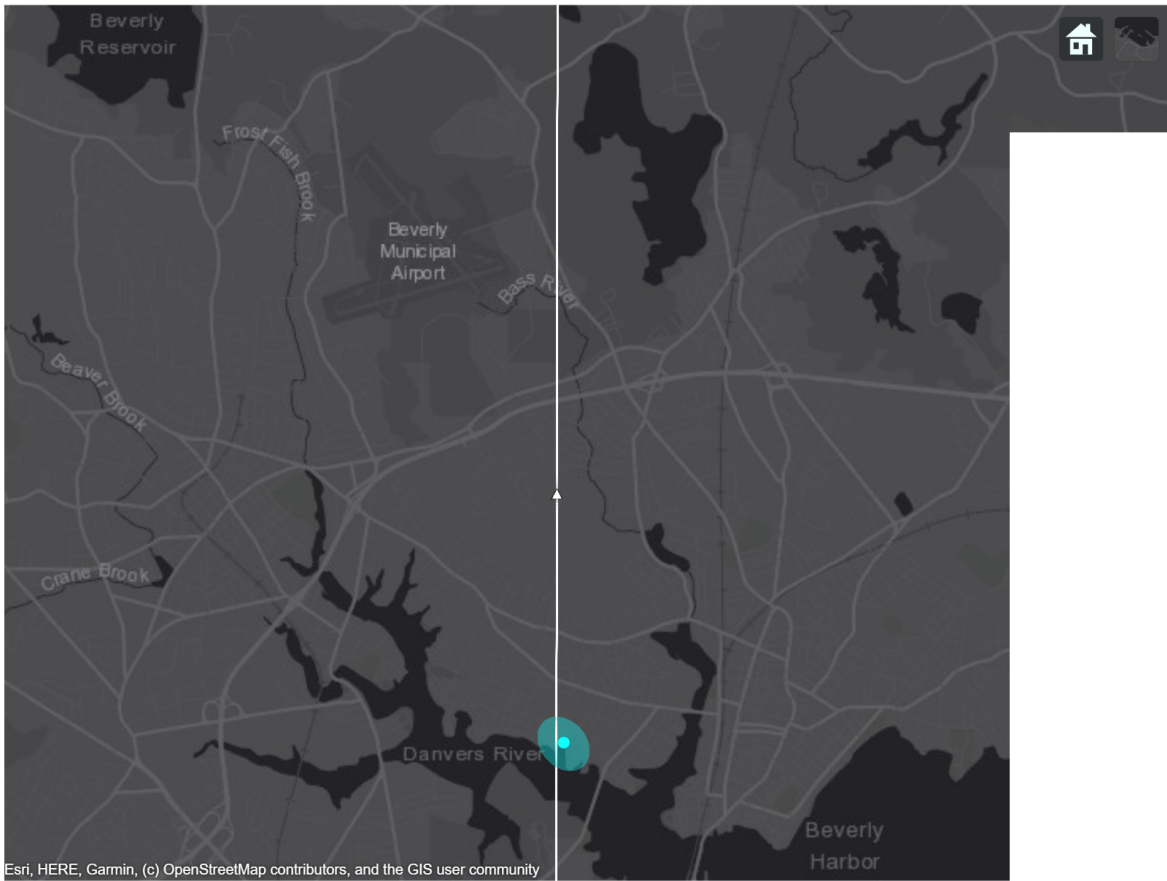
```
imshow(allsnaps{1})
```



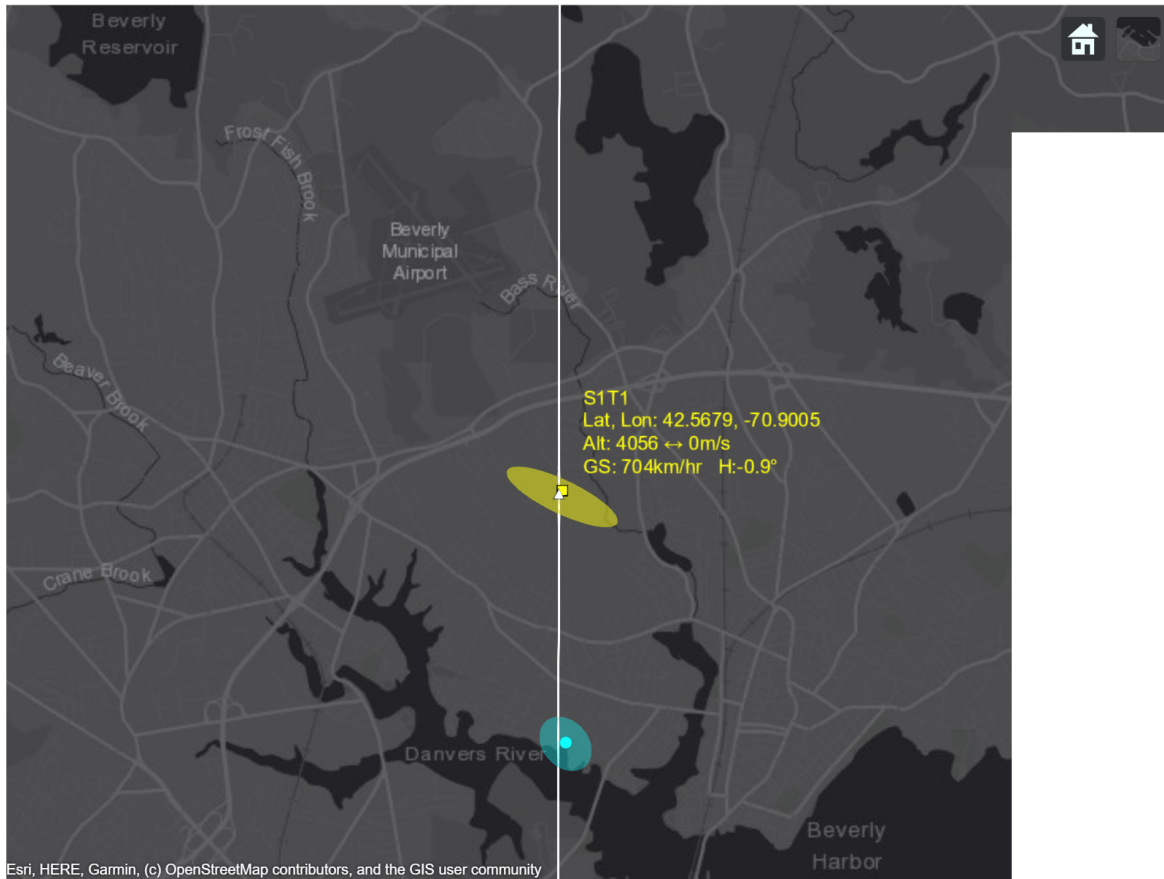
The figure above shows the scenario at the end of the second 360-degree scan of the radar. Radar detections, shown as light blue dots, are present for each of the simulated airliners. At this point, the tracker has already been updated by detections of one complete scan. Internally, the tracker has initialized tracks for each of the airliners. These tracks will be later confirmed and shown in the plot after more updates.

The next two snapshots show the tracking results for the outbound airliner.

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

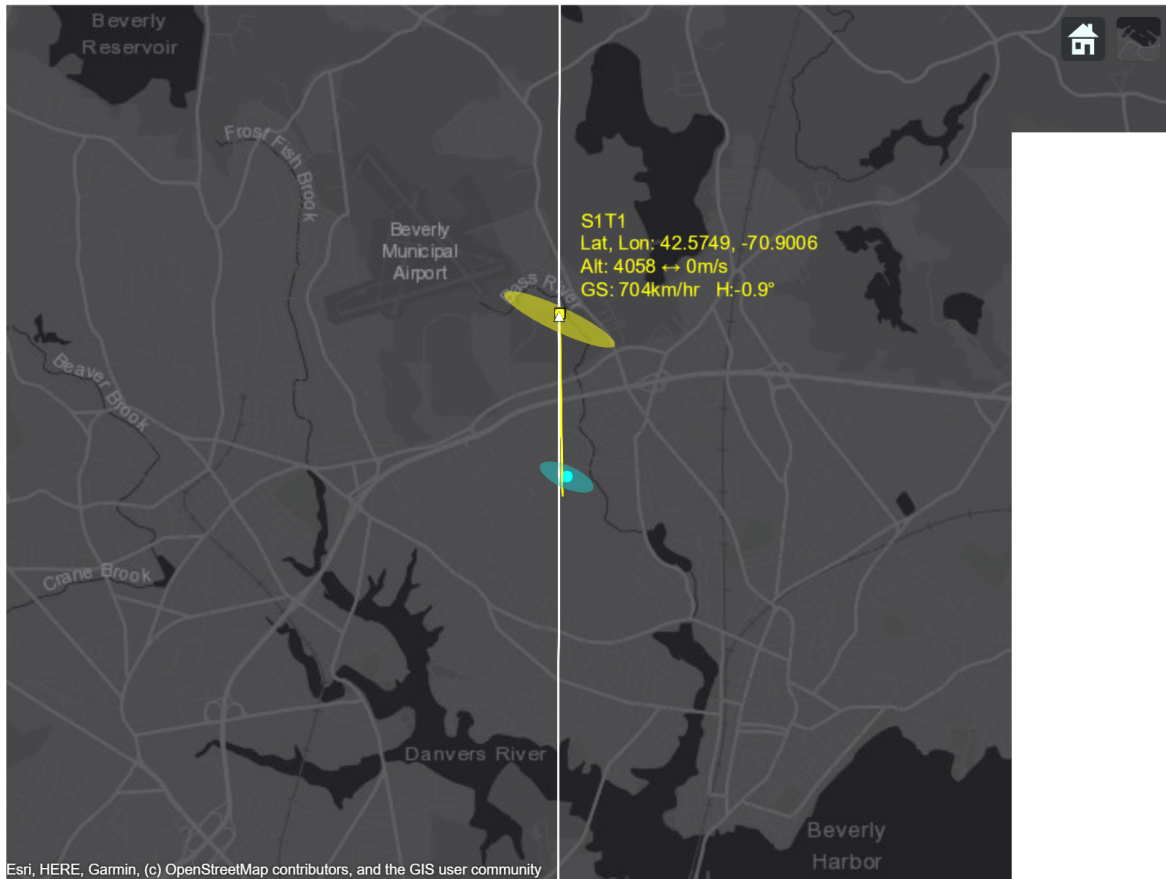



```
imshow(allsnaps{3})
```

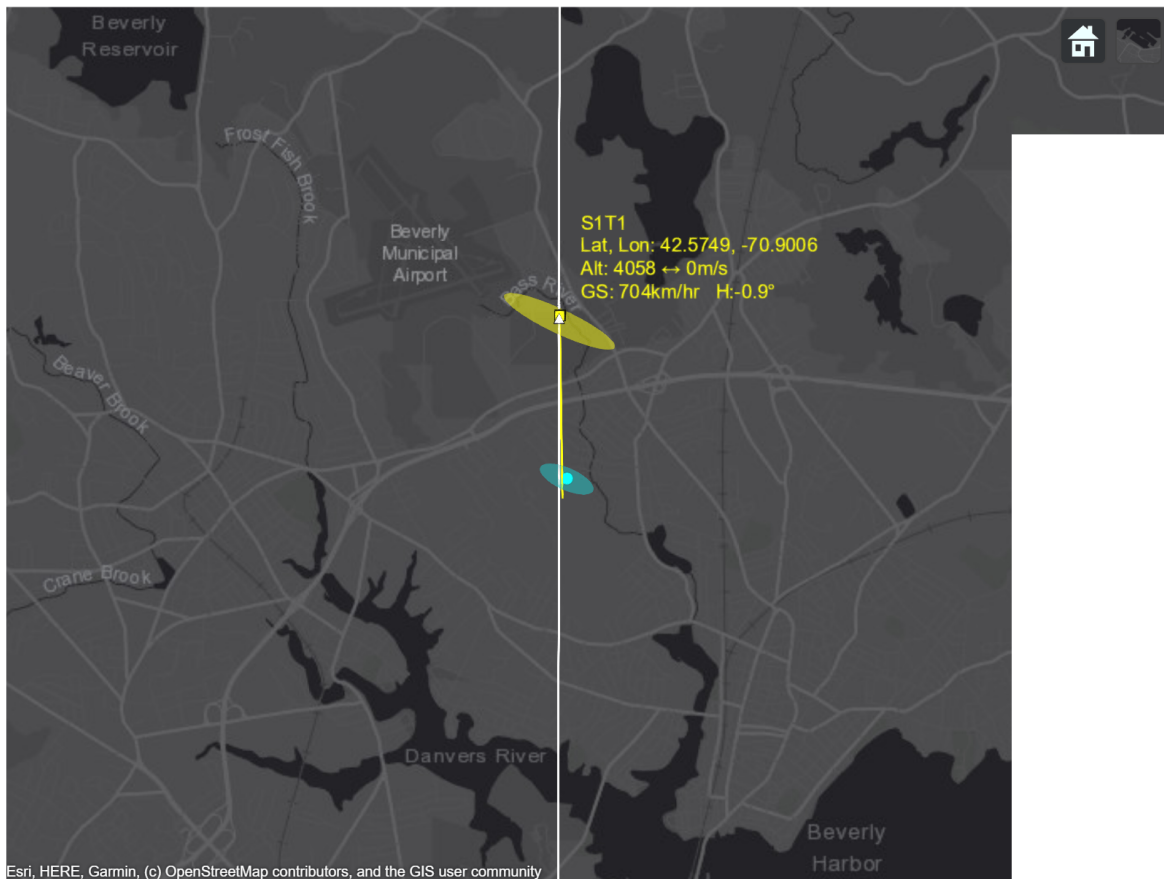


These two figures show the tracking results before and immediately after the tracker updates with the detections from second scan of the radar. The detection shown in the first figure is used to update and confirm the initialized track from the previous step. The next figure shows the confirmed track. The uncertainty of the track position estimate is shown as the yellow ellipse. After updating with only two detections, the tracker has established an accurate estimate of the outbound airliner. The true altitude of the airliner is 4 km traveling north at a speed of 700 km/hr.

```
imshow(allsnaps{4})
```

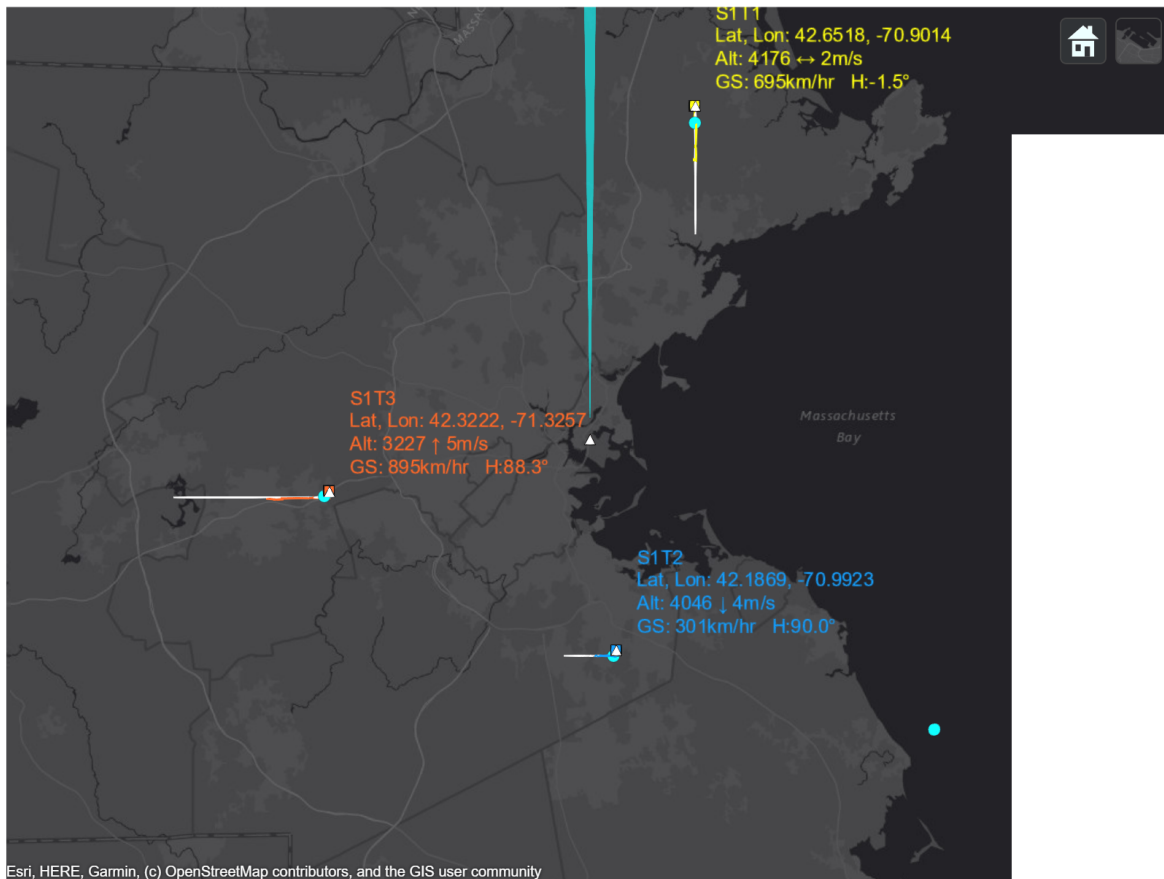


`imshow(allsnaps{5})`



The track for the outbound airliner 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 the last step. In the next figure, you can see the uncertainty of the track position is reduced after updating with the new detection. You also observe that after the third update, the track becomes closer to the airliner's true position.

```
imshow(allsnaps{6})
```



The final figure shows the state of all three tracks of the airliners at the end of the scenario. There is exactly one track for each airliner. The same track IDs are assigned to the airliners for the entire duration of the scenario, indicating that none of these tracks was dropped or switched during the scenario. The estimated tracks closely match the true position and velocity of the airliners.

```
disp(tabulateData(ATCScenario, simOut))
```

TrackID	Altitude		Heading		Speed	
	True	Estimated	True	Estimated	True	Estimated
"T1"	4000	4040	90	91	700	695
"T2"	4000	3964	0	0	300	301
"T3"	3000	3126	0	2	900	895

```
close_system(model,0)
```

Summary

This example shows how to design a tracking system for an air traffic control center in Simulink and configure a global nearest neighbor (GNN) tracker block to track the simulated targets using the radar detections. In this example, you learned how the 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.

Gesture Recognition Using Inertial Measurement Units

This example shows how to recognize gestures based on a handheld inertial measurement unit (IMU). Gesture recognition is a subfield of the general Human Activity Recognition (HAR) field. In this example, you use quaternion dynamic time warping and clustering to build a template matching algorithm to classify five gestures.

Dynamic time warping is an algorithm used to measure the similarity between two time series of data. Dynamic time warping compares two sequences, by aligning data points in the first sequence to data points in the second, neglecting time synchronization. Dynamic time warping also provides a distance metric between two unaligned sequences.

Similar to dynamic time warping, quaternion dynamic time warping compares two sequences in quaternion, or rotational, space [1] on page 6-1046.

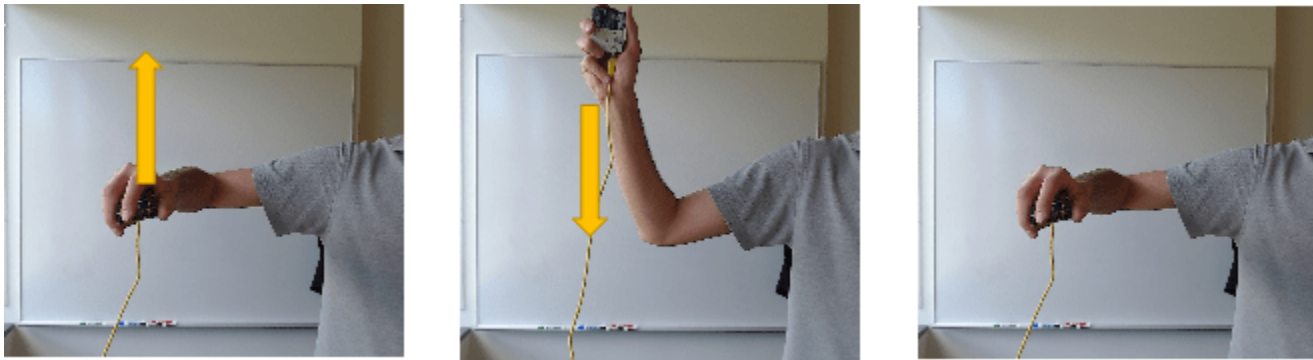
Quaternion dynamic time warping also returns a scalar distance between two orientation trajectories. This distance metric allows you to cluster data and find a template trajectory for each gesture. You can use a set of template trajectories to recognize and classify new trajectories.

This approach of using quaternion dynamic time warping and clustering to generate template trajectories is the second level of a two-level classification system described in [2] on page 6-1046.

Gestures and Data Collection

In this example you build an algorithm that recognizes and classifies the following five gestures. Each gesture starts at the same position with the right forearm level, parallel to the floor.

- up - Raise arm up, then return to level.



- down - Lower arm down, then return to level.



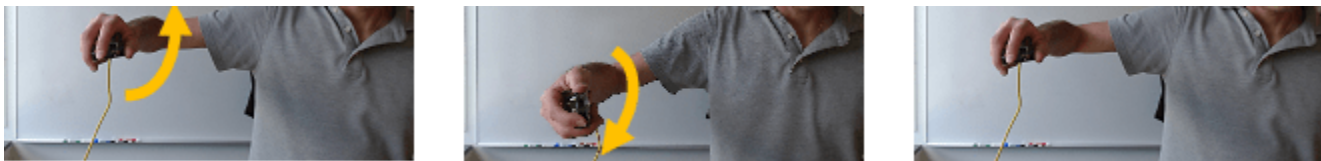
- left - Swipe arm left, then return to the center. Forearm is parallel to floor for the duration.



- right - Swipe arm right, then return to the center. Forearm is parallel to floor for the duration.



- twist - Rotate hand 90 degrees clockwise and return to the original orientation. Forearm is parallel to the floor for the duration.



Data for these five gestures are captured using the Arduino Support Package for MATLAB. Four different people performed the five gestures and repeated each gesture nine to ten times. The recorded data, saved as a table, contains accelerometer and gyroscope readings, sample rate, the gesture being performed, as well as the name of person performing the gesture.

Sensor Fusion Preprocessing

The first step is to fuse all the accelerometer and gyroscope readings to produce a set of orientation time series, or trajectories. You will use the `imufilter` System object to fuse the accelerometer and gyroscope readings. The `imufilter` System object is available in both the Navigation Toolbox and the Sensor Fusion and Tracking Toolbox. You can use a `parfor` loop to speed the computation. If you have the Parallel Computing Toolbox the `parfor` loop will run in parallel, otherwise a regular `for` loop will run sequentially.

```
ld = load("imuGestureDataset.mat");
dataset = ld.imuGestureDataset;
```



```

dataset = dataset(randperm(size(dataset,1)), :); % Shuffle the dataset

Ntrials = size(dataset,1);
Orientation = cell(Ntrials,1);
parfor ii=1:Ntrials
    h = imufilter("SampleRate", dataset(ii,:).SampleRate);
    Orientation{ii} = h(dataset.Accelerometer{ii}, dataset.Gyroscope{ii});
end

```

```

Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to the parallel pool (number of workers: 12).

```

```

dataset = addvars(dataset, Orientation, NewVariableNames="Orientation");
gestures = string(unique(dataset.Gesture)); % Array of all gesture types
people = string(unique(dataset.Who));
Ngestures = numel(gestures);
Npeople = numel(people);

```

Quaternion Dynamic Time Warping Background

Quaternion dynamic time warping [1] on page 6-1046 compares two orientation time series in quaternion-space and formulates a distance metric composed of three parts related to the quaternion distance, derivative, and curvature. In this example, you will only compare signals based on quaternion distance.

Quaternion dynamic time warping uses time warping to compute an optimal alignment between two sequences.

The following uses quaternion dynamic time warping to compare two sequences of orientations (quaternions).

Create two random orientations:

```

rng(20);
q = randrot(2,1);

```

Create two different trajectories connecting the two orientations:

```

h1 = 0:0.01:1;
h2 = [zeros(1,10) (h1).^2];
traj1 = slerp(q(1),q(2),h1).';
traj2 = slerp(q(1),q(2),h2).';

```

Note that though `traj1` and `traj2` start and end at the same orientations, they have different lengths, and they transition along those orientations at different rates.

Next compare and find the best alignment between these two trajectories using quaternion dynamic time warping.

```

[qdist, idx1, idx2] = helperQDTW(traj1, traj2);

```

The distance metric between them is a scalar

```

qdist
qdist = 1.1474

```

Plot the best alignment using the `idx1` and `idx2` variables

```

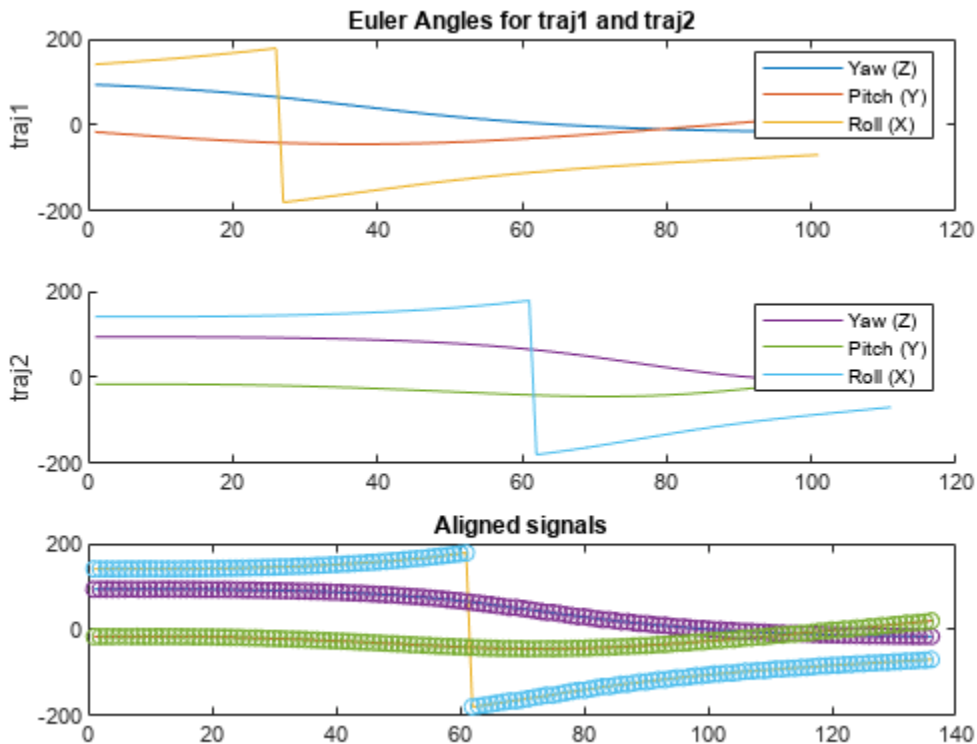
e1 = eulerd(traj1, "ZYX", "frame");
e2 = eulerd(traj2, "ZYX", "frame");

figure;
subplot(3,1,1);
plot(e1, "-");
legend("Yaw (Z)", "Pitch (Y)", "Roll (X)");
ylabel("traj1")

subplot(3,1,2);
set(gca, "ColorOrderIndex", 4);
hold on;
plot(e2, "-");
legend("Yaw (Z)", "Pitch (Y)", "Roll (X)");
ylabel("traj2")
hold off;

subplot(3,1,3);
plot(1:numel(idx1), e1(idx1,:), "-", ...
     1:numel(idx2), e2(idx2,:), "o" );
title("Aligned signals");
subplot(3,1,1);
title("Euler Angles for traj1 and traj2")

```



```

snapnow;

```

Training and Test Data Partitioning

The dataset contains trajectories from four test subjects. You will train your algorithm with data from three subjects and test the algorithm on the fourth subject. You repeat this process four times, each time alternating who is used for testing and who is used for training. You will produce a confusion matrix for each round of testing to see the classification accuracy.

```
accuracy = zeros(1,Npeople);
for pp=1:Npeople
    testPerson = people(pp);
    trainset = dataset(dataset.Who ~= testPerson,:);
    testset = dataset(dataset.Who == testPerson,:);
```

With the collected gesture data, you can use the quaternion dynamic time warping function to compute the mutual distances between all the recordings of a specific gesture.

```
% Preallocate a struct of distances and compute distances
for gg=1:Ngestures
    gest = gestures(gg);
    subdata = trainset(trainset.Gesture == gest,:);
    Nsubdata = size(subdata,1);
    D = zeros(Nsubdata);
    traj = subdata.Orientation;
    parfor ii=1:Nsubdata
        for jj=1:Nsubdata
            if jj > ii % Only calculate triangular matrix
                D(ii,jj) = helperQDTW(traj{ii}, traj{jj});
            end
        end
    end
    allgestures.(gest) = traj;
    dist.(gest) = D + D.'; % Render symmetric matrix to get all mutual distances
end
```

Clustering and Templating

The next step is to generate template trajectories for each gesture. The structure `dist` contains the mutual distances between all pairs of recordings for a given gesture. In this section, you cluster all trajectories of a given gesture based on mutual distance. The function `helperClusterWithSplitting` uses a cluster splitting approach described in [2] on page 6-1046. All trajectories are initially placed in a single cluster. The algorithm splits the cluster if the radius of the cluster (the largest distance between any two trajectories in the cluster) is greater than the `radiusLimit`.

The splitting process continues recursively for each cluster. Once a median is found for each cluster, the trajectory associated with that cluster is saved as a template for that particular gesture. If the ratio of the number of trajectories in the cluster to the total number of trajectories of a given gesture is less than `clusterMinPct`, the cluster is discarded. This prevents outlier trajectories from negatively affecting the classification process. Depending on choice of `radiusLimit` and `clusterMinPct` a gesture may have one or several clusters, and hence may have one or several templates.

```
radiusLimit = 60;
clusterMinPct = 0.2;
parfor gg=1:Ngestures
    gest = gestures(gg);
    Dg = dist.(gest); %#ok<*PFBNS>
```

```

[gclusters, gtemplates] = helperClusterWithSplitting(Dg, radiusLimit);
clusterSizes = cellfun(@numel, gclusters, "UniformOutput", true);
totalSize = sum(clusterSizes);
clusterPct = clusterSizes./totalSize;
validClusters = clusterPct > clusterMinPct;
tidx = gtemplates(validClusters);
tmpl = allgestures.(gest)(tidx);
templateTraj{gg} = tmpl;
labels{gg} = repmat(gest, numel(tmpl),1);
end
templates = table(vertcat(templateTraj{:}), vertcat(labels{:}));
templates.Properties.VariableNames = ["Orientation", "Gesture"];

```

The template variable is stored as a table. Each row contains a template trajectory stored in the Orientation variable and an associated gesture label stored in the Gesture variable. You will use this set of templates to recognize new gestures in the testset.

Classification System

Using quaternion dynamic time warping, compare the new gesture data from the test set can be compared to each of the gesture template. The system classifies the new unknown gesture as the class of the template that has the smallest quaternion dynamic time warping distance to the unknown gesture. If the distance to each template is beyond the radiusLimit, the test gesture is marked as unrecognized.

```

Ntest = size(testset,1);
Ntemplates = size(templates,1);
testTraj = testset.Orientation;
expected = testset.Gesture;
actual = strings(size(expected));

parfor ii=1:Ntest
    testdist = zeros(1,Ntemplates);
    for tt=1:Ntemplates
        testdist(tt) = helperQDTW(testTraj{ii}, templates.Orientation{tt});
    end
    [mind, mindidx] = min(testdist);
    if mind > radiusLimit
        actual(ii) = "unrecognized";
    else
        actual(ii) = templates.Gesture{mindidx};
    end
end
results.(testPerson).actual = actual;
results.(testPerson).expected = expected;
end

```

Compute the accuracy of the system in identifying gestures in the test set and generate a confusion matrix

```

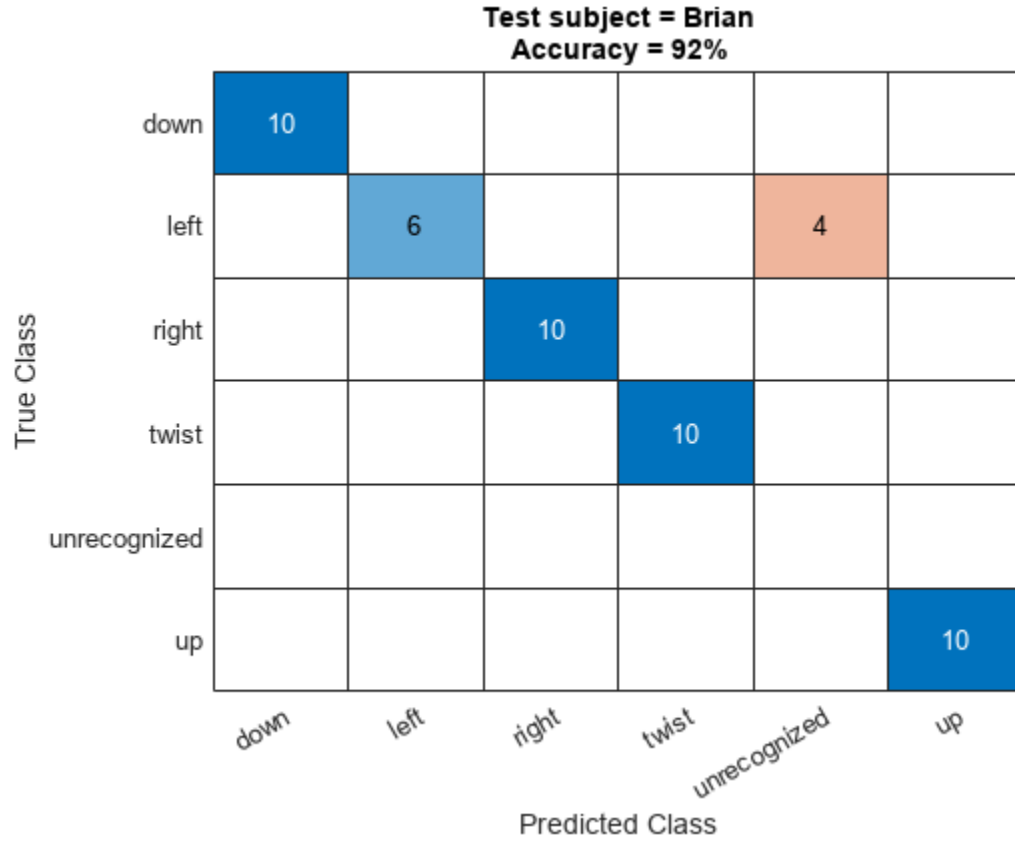
for pp=1:Npeople
    figure;
    act = results.(people(pp)).actual;
    exp = results.(people(pp)).expected;
    numCorrect = sum(act == exp);
    Ntest = numel(act);
    accuracy = 100 * numCorrect./Ntest;
end

```

```

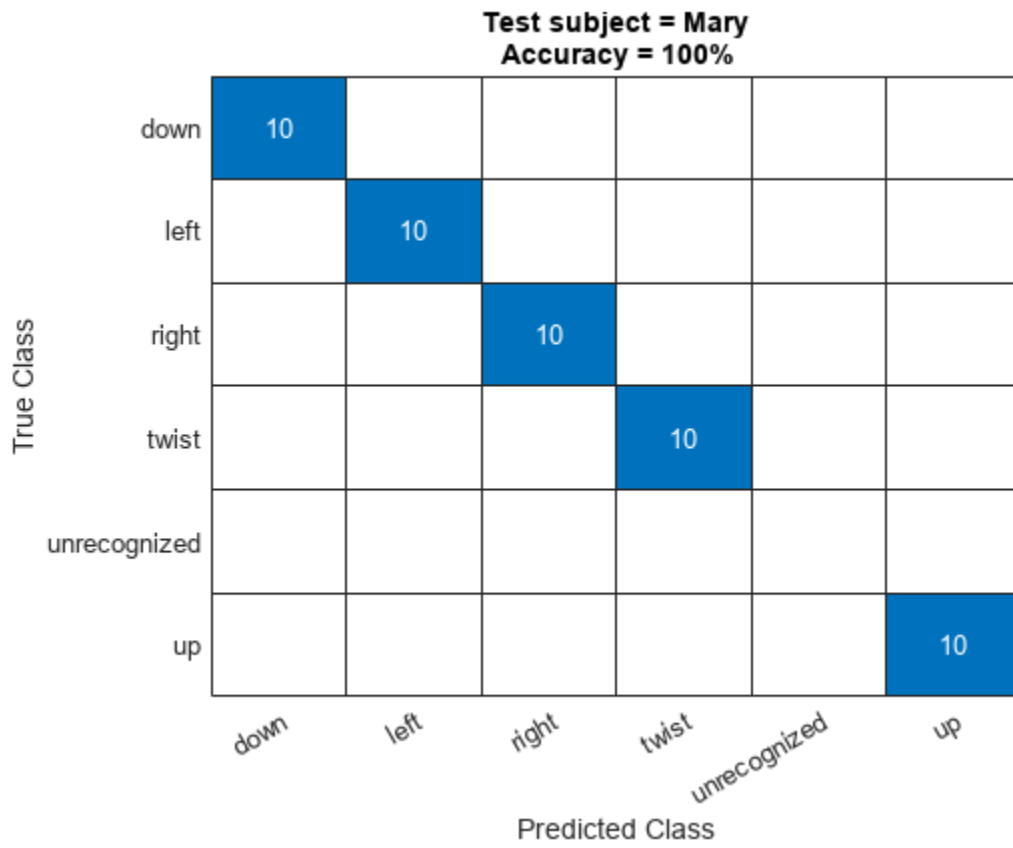
confusionchart([exp; "unrecognized"], [act; missing]);
title("Test subject = " + people(pp) + newline + "Accuracy = " + accuracy + "%" );
end
snapnow;

```



Test subject = Karsh
Accuracy = 88%

True Class	down	left	right	twist	unrecognized	up
down	10					
left		10				
right			10			
twist				4	6	
unrecognized						
up						10
	down	left	right	twist	unrecognized	up
	Predicted Class					



Test subject = Sandip
Accuracy = 94%

	down						
True Class	down	10					
	left		9			1	
	right			9		1	
	twist				10		
	unrecognized						
	up					1	9
		down	left	right	twist	unrecognized	up
		Predicted Class					

The average accuracy across all four test subject configurations is above 90%.

```
AverageAccuracy = mean(accuracy)
```

```
AverageAccuracy = 94
```

Conclusion

By fusing IMU data with the `imufilter` object and using quaternion dynamic time warping to compare a gesture trajectory to a set of template trajectories you recognize gestures with high accuracy. You can use sensor fusion along with quaternion dynamic time warping and clustering to construct an effective gesture recognition system.

References

- [1] B. Jablonski, "Quaternion Dynamic Time Warping," in *IEEE Transactions on Signal Processing*, vol. 60, no. 3, March 2012.
- [2] R. Srivastava and P. Sinha, "Hand Movements and Gestures Characterization Using Quaternion Dynamic Time Warping Technique," in *IEEE Sensors Journal*, vol. 16, no. 5, March 1, 2016.


Automatically Tune Tracking Filter for Multi-Object Tracker

This example shows how to automatically tune a tracking filter using the `trackingFilterTuner` object. After tuning, use the tuning results in a multi-target tracker to improve the tracking performance of the tracker.

Run a Tracker with an Untuned Filter

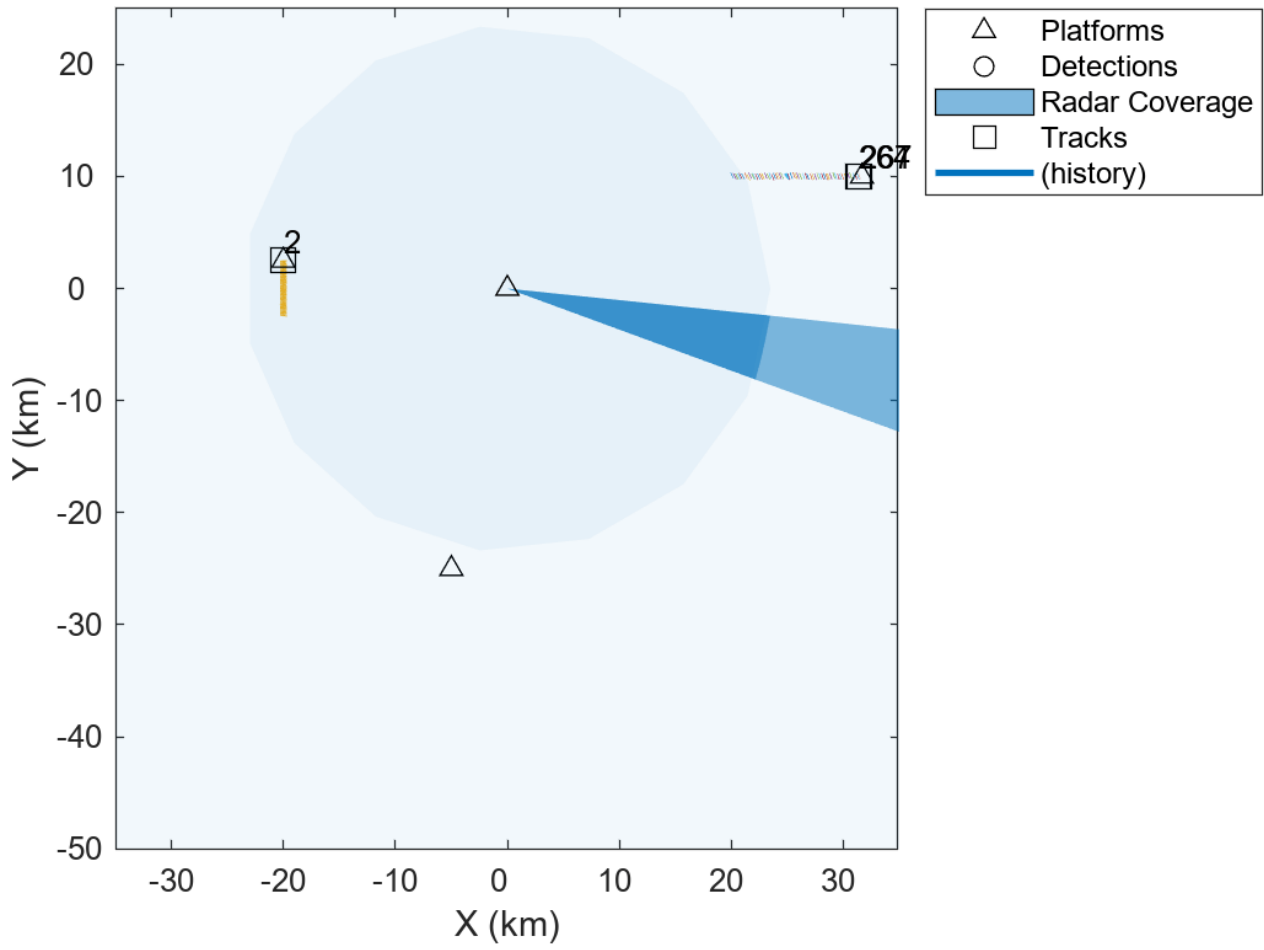
This example builds upon the scenario from the “Air Traffic Control” on page 6-2 example.

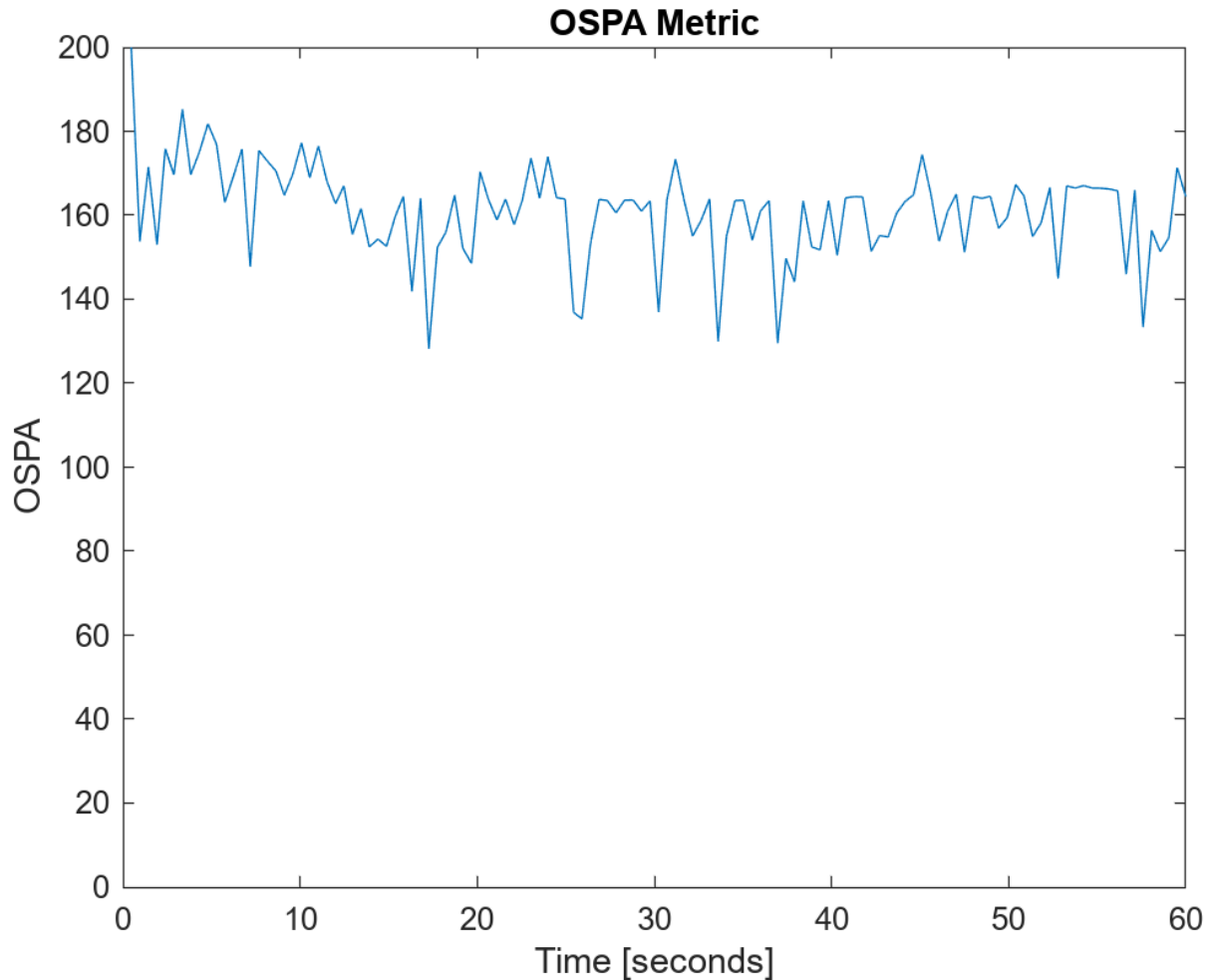
To study whether filter tuning is required, construct a `trackerGNN` System object with the default filter initialization function. Additionally, define an optimal sub-pattern association (OSPA) metric, `trackOSPAMetric`, to quantitatively analyze the tracking results. As a reminder, lower OSPA values mean better tracking quality. Use the slider to adjust the assignment threshold.

```
tracker = trackerGNN(AssignmentThreshold = 100  );  
metric = trackOSPAMetric(Distance = "posabserr", CutoffDistance = 200);
```

Compared with the original example, the radar is modified to have wider beam and a faster radar scanning speed, which allows it to generate more detections of each target in the scenario.

```
load("ATCScenario.mat", "scenario");  
helperRunScenario(scenario, tracker, metric);
```





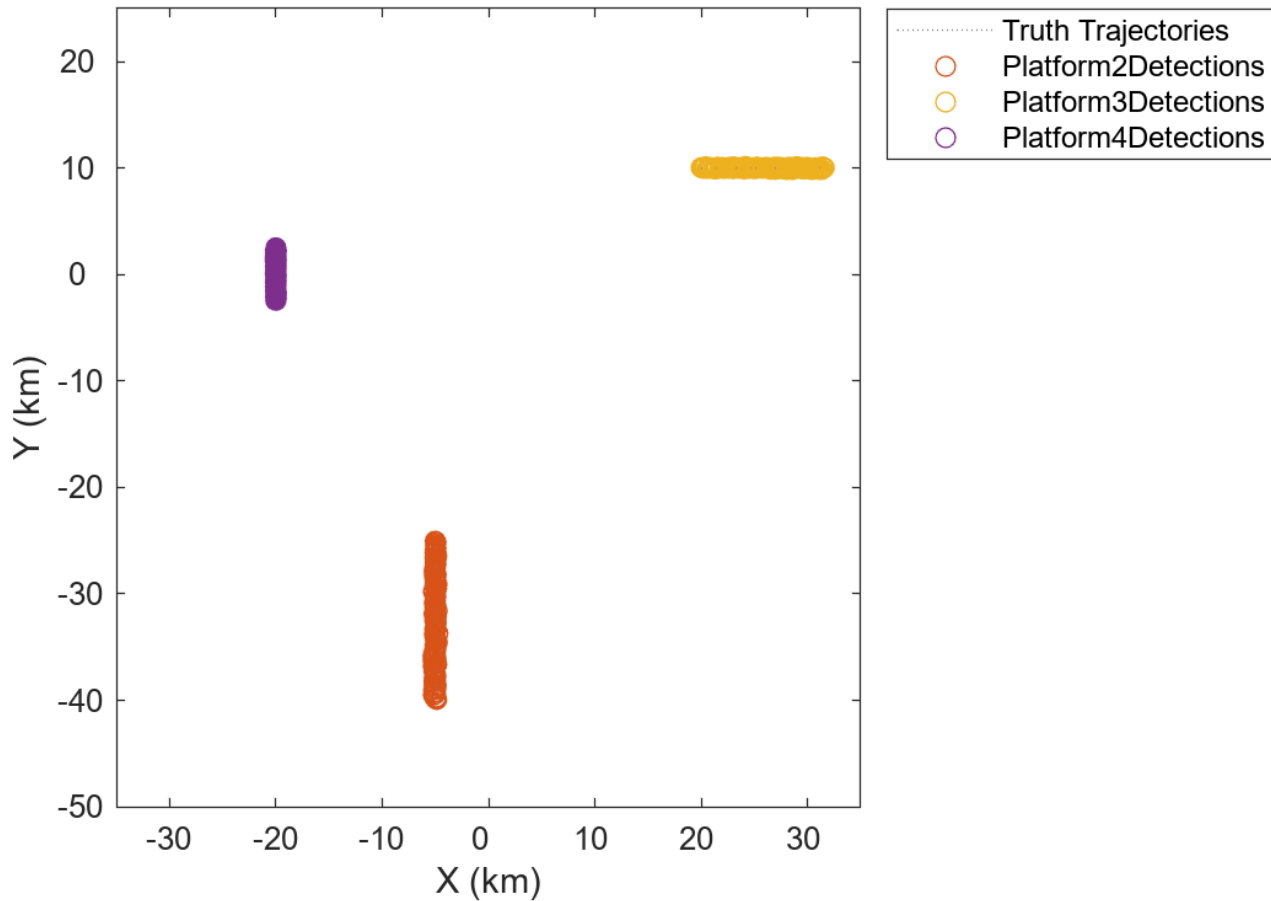
From the results, even when the assignment threshold is set at the highest, the tracker performs undesirably and is unable to track all the targets.

Create Tuning Data from the Scenario

To automatically tune a filter, use the `trackingFilterTuner` object with the history of truth objects and a log of detections per each truth object. The truth data is a cell array of `timetable` objects with each cell representing the history of one truth object. The detection log is a cell array. The i -th cell in the detection log is a cell array of `objectDetectionDelay` objects corresponding to the i -th truth object in the truth data.

To create the tuning data from the scenario, use the `monteCarloRun` function, which creates scenario recordings with different noise realizations. To convert the recordings to tuning data, use the `helperRecordingToTuningData` supporting function attached in the example. For brevity, this example directly loads a file with tuning data, created by two Monte Carlo runs of the scenario.

```
% recording = monteCarloRun(scenario,2);
% [detections, truthTable] = recordingToTuningData(recording);
load("ATCTuningData.mat","detections","truthTable");
helperVisualizeTuningData(detections,truthTable);
```



Tune a Filter with Default Tuning Parameters

To understand the difficulty of the tracker in tracking the fast-moving platforms, look at the filter that is initialized by the default `initcvekf` function used in the tracker.

```
sampleDetection = detections{1}{1};
filter = initcvekf(sampleDetection);
```

The filter uses the constant velocity motion model convention, in which the state vector defined as:

$\vec{x} = [x, v_x, y, v_y, z, v_z]^T$, where x , y , and z are position components in the rectangular frame and v_x , v_y , and v_z are the velocity components. The process noise of the filter represents the uncertainty about the velocity change, or acceleration, of the object.

```
disp(filter.ProcessNoise);
```

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

Similarly, the state covariance represents the uncertainty in the position and velocity components. The `initcvekf` function directly uses the detection measurement noise to initialize the position

components in the state uncertainty covariance. However, since the velocity components are not reported in the measurement, their respective uncertainty is set to 100 by default, representing a standard deviation of 10 m/s. Obviously, for airliners, the uncertainty should be higher.

```
disp(filter.StateCovariance);

1.0e+04 *

    0.9582         0    -0.1154         0    -0.0824         0
         0    0.0100         0         0         0         0
   -0.1154         0    0.0488         0    -0.3422         0
         0         0         0    0.0100         0         0
   -0.0824         0    -0.3422         0    4.1085         0
         0         0         0         0         0    0.0100
```

Create a `trackingFilterTuner` object.

```
tuner = trackingFilterTuner;
disp(tuner)

trackingFilterTuner with properties:

    FilterInitializationFcn: "initcvekf"
    TunablePropertiesSource: "Default"

                Cost: "RMSE"

                UseMex: 0
                UseParallel: 0
                Solver: "fmincon"
    SolverOptions: []
```

By default, when tuning a `trackingEKF` object that is initialized by the `initcvekf` function, the tuner tunes the `ProcessNoise` property of the filter and uses the root mean square error (RMSE) between the filter estimate and the truth as the cost. The process noise is usually the hardest to define, because often there is little information about how the object motion differs from the motion model used in the filter. The initial state covariance is the next hardest thing to tune, which you will tune in the next section.

To accelerate the tuning, set the `UseMex` property to `true` using the checkbox. This setting requires a MATLAB Coder license. Uncheck the box if that license is not available.

```
tuner.UseMex = ;
```

Similarly, set the `UseMex` property to `true` to use parallel processing, which requires a Parallel Computing Toolbox license. Uncheck the box if that license is unavailable.

```
tuner.UseParallel = .
```

You can use one of three solvers based on the `fmincon`, `particleswarm`, and `patternsearch` optimization algorithms. Choose your optimization algorithm based on the complexity of the optimization problem, the time allowed for it to run, and other criteria. You can refer to the Optimization Toolbox™ and Global Optimization Toolbox™ documentation to decide which solver to use.

The 'fmincon' solver requires the Optimization Toolbox license.

The 'particleswarm' or the 'patternsearch' solvers require the Global Optimization Toolbox license.

```
solver = fmincon;
```

Set the solver options to provide an iterative display of the solving progress over time. Use the slider to control the maximum number of iterations the solver can use. Setting the number of iterations higher requires more time for the tuner. To tune a filter to track all the targets, use the tuning data for all the targets.

```
tuner.Solver = solver;
tuner.SolverOptions = optimoptions(solver, Display = "iter", ...
    MaxIter = 15);
tic;
[tunedParams, bestCost] = tune(tuner,detections,truthTable);
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to the parallel pool (number of workers: 6).
Generating code.
Code generation successful.
```

```
Your initial point x0 is not between bounds lb and ub; FMINCON
shifted x0 to strictly satisfy the bounds.
```

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
0	7	7.848006e+01	0.000e+00	1.613e-01	
1	14	7.839574e+01	0.000e+00	1.852e-01	3.538e-01
2	21	7.810211e+01	0.000e+00	1.639e-01	1.463e+00
3	28	7.768567e+01	0.000e+00	1.813e-01	3.044e+00
4	35	7.720462e+01	0.000e+00	2.038e-01	4.543e+00
5	43	7.703294e+01	0.000e+00	2.229e-01	1.866e+00
6	50	7.686042e+01	0.000e+00	2.013e-01	1.277e+00
7	57	7.654589e+01	0.000e+00	1.457e-01	1.737e+00
8	64	7.603169e+01	0.000e+00	6.552e-02	4.124e+00
9	71	7.600035e+01	0.000e+00	6.463e-02	8.611e-01
10	78	7.589348e+01	0.000e+00	6.237e-02	1.662e+00
11	85	7.592252e+01	0.000e+00	4.813e-02	5.195e-01
12	92	7.591004e+01	0.000e+00	4.924e-02	1.346e-01
13	99	7.588873e+01	0.000e+00	3.311e-02	3.816e-01
14	106	7.586347e+01	0.000e+00	3.171e-02	1.262e+00
15	113	7.584819e+01	0.000e+00	2.707e-02	1.731e+00

```
Solver stopped prematurely.
```

```
fmincon stopped because it exceeded the iteration limit,
options.MaxIterations = 1.500000e+01.
```

```
disp("Time it took to tune the filter: " + toc);
```

```
Time it took to tune the filter: 156.134
```

```
disp("Optimized cost: " + bestCost);
```

```
Optimized cost: 75.8482
```

Note that the optimization process has not improved the cost much. That is expected, because in this case the main driver of the high cost is the poor initial velocity covariance, which was not part of the tuning.

The tuned properties for the `trackingEKF` object are returned as a structure with two fields: `ProcessNoise` and `StateCovariance`. Their values are displayed below. The state covariance is not tuned.

```
disp(tunedParams.ProcessNoise);
```

```
111.6332    28.3525    96.7912
 28.3525    48.7698    13.3867
 96.7912    13.3867    97.0984
```

```
disp(tunedParams.StateCovariance);
```

```
1.0e+04 *
 0.9582         0    -0.1154         0    -0.0824         0
         0    0.0100         0         0         0         0
 -0.1154         0    0.0488         0    -0.3422         0
         0         0         0    0.0100         0         0
 -0.0824         0    -0.3422         0    4.1085         0
         0         0         0         0         0    0.0100
```

To obtain the tuned filter, either set each property on the filter to its corresponding tuned value from the structure or use `setTunedProperties` object function of the filter.

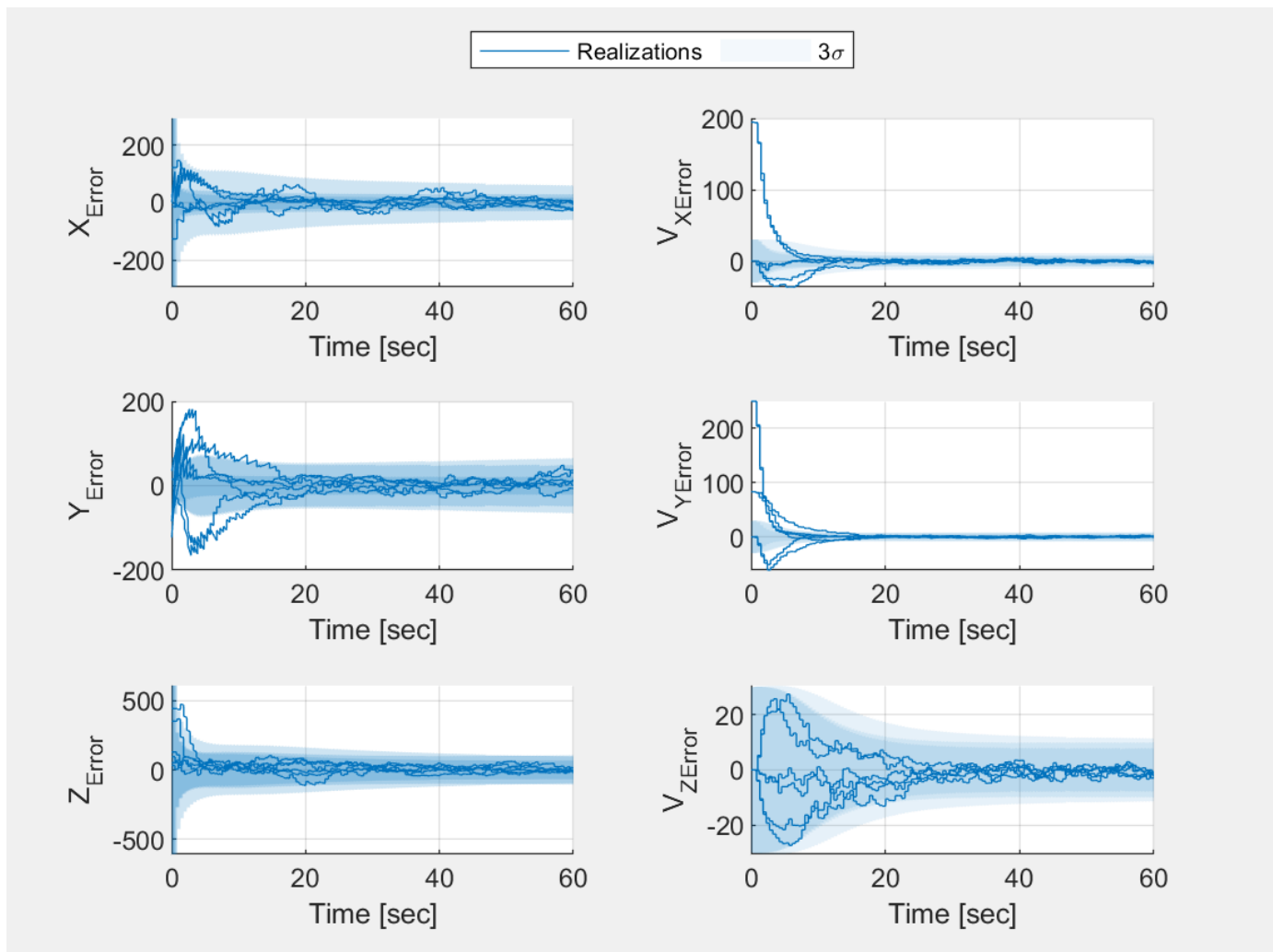
```
filter = feval(tuner.FilterInitializationFcn, sampleDetection);
setTunedProperties(filter,tunedParams);
disp(filter.ProcessNoise);
```

```
111.6332    28.3525    96.7912
 28.3525    48.7698    13.3867
 96.7912    13.3867    97.0984
```

Use the `plotFilterErrors` object function to show the estimate accuracy. The plots show the estimation errors of the position and the velocity in all motion dimensions. Each line represents the error between the estimate from one run of the filter and the associated truth. The bands represent the three standard deviations ("3-sigma") in that dimension obtained from the filter state covariance. For an unbiased filter, the filter estimate error should average around zero.

Furthermore, the estimate of a consistent Gaussian filter should fall within the 3-sigma bounds about 99% of the time. If the filter is overconfident, there will be more violations of the three-standard deviations bounds. If the filter is underconfident, the actual filter errors will be very small relative to the bounds. Having a consistent filter that provides the right measure of uncertainty is important in tracking.

```
plotFilterErrors(tuner);
```



Customize the Tuning

When tuning only the process noise, the filter estimate errors, especially for the velocity along the X and Y axes, far exceeds the three standard deviation bounds. The reason is that the initial state covariance value for velocity is too low. In other words, the filter is too confident about its velocity estimate. It also causes the process noise value to be too high after tuning, because the tuning algorithm tries to compensate for the large initial errors. As a result, the estimate follows the measurement more closely than it should and the filter error is large even after the filter converges.

To fix this issue, set the tuner to use custom properties. To create the filter tunable properties, create a same type of tracking filter object with the same motion model used in the tracking. Next, use the `tunableProperties` object function of the filter to construct a `tunableFilterProperties` object.

```
filter = feval(tuner.FilterInitializationFcn, sampleDetection);
tfp = tunableProperties(filter);
disp(tfp)
```

Tunable properties for object of type: trackingEKF


```

Property:      ProcessNoise
  PropertyValue: [1 0 0;0 1 0;0 0 1]
  TunedQuantity: Square root
  IsTuned:      true
    TunedQuantityValue: [1 0 0;0 1 0;0 0 1]
    TunableElements:   [1 4 5 7 8 9]
    LowerBound:        [0 0 0 0 0 0]
    UpperBound:        [10 10 10 10 10 10]
Property:      StateCovariance
  PropertyValue: [9582.02019480753 0 -1154.41787165046 0 -823.540813608376 0;0 100 0 0 0 0;-1154.41787165046 100 0 0 0 0;0 -823.540813608376 0 0 100 0;0 0 0 0 0 100]
  TunedQuantity: Square root of initial value
  IsTuned:      false

```

Using the filter tunable properties, define which properties should be tuned and how to tune each one. Use the `IsTuned` flag to set the tunability of the property. In this case, set the `StateCovariance` property to be tuned.

Similarly, define which elements to tune. To ensure the 6x6 state covariance matrix is positive semidefinite and symmetric as well as reduce the number of parameters to optimize from 36 down to 21, encode the covariance as a product of an upper triangular matrix and its transpose using the Cholesky decomposition. This is still a very large number of elements and would require a significant effort from the solver. However, the filter initialization function already uses the detection measurement noise to set the initial state covariance position uncertainty, so only the velocity elements need tuning. Moreover, the result above shows that the velocity error in the Z-component is already well contained.

Recall that the state vector used in the constant velocity model is defined as: $\vec{x} = [x, v_x, y, v_y, z, v_z]^T$, where x , y , and z are position components in the rectangular frame and v_x , v_y , and v_z are the velocity components. Therefore, to tune the uncertainty in the velocity components along the X and Y directions, only the (2,2), (2,4), and (4,4) elements of the Cholesky form should be tuned.

To further assist the tuner, define the lower and upper bound of each element. In this case, the graph above showed initial velocity errors that are less than 300 meters per second, so define the lower and the upper bound to 0 and 300, respectively. Note that the lower and upper bound vectors must have the same number of elements as the number of tunable elements, in this case, 3.

```

elements = sub2ind([6 6], [2 2 4], [2 4 4]); % Tune just the XY velocity covariance elements
lb = zeros(1,numel(elements));
ub = 300*ones(1,numel(elements));
setPropertyTunability(tfp, "StateCovariance", IsTuned = true, ...
    TunableElements = elements, LowerBound = lb, UpperBound = ub);

```

Similarly, there is no reason to tune all the elements of the process noise. The process noise is a 3x3 matrix, with the elements that control the X and Y process components being (1,1), (1,2), and (2,2). By reducing the number of elements to tune, the tuner can run faster.

Since airliners mostly fly straight level flight, use an upper bound of 10.

```

elements = sub2ind([3 3], [1 1 2], [1 2 2]); % Tune just the XY process elements
lb = zeros(1,numel(elements));
ub = 10*ones(1,numel(elements));
setPropertyTunability(tfp, "ProcessNoise", "IsTuned", true, ...
    "TunableElements", elements, "LowerBound", lb, "UpperBound", ub);

```

Finally, set the tuner custom tunable properties to the tunable filter properties.

```

tuner.TunablePropertiesSource = "Custom";
tuner.CustomTunableProperties = tfp;

solver = fmincon;
tuner.Solver = solver;
tuner.SolverOptions = optimoptions(solver, Display = "iter", ...
    MaxIter = 15 );
tic;
[tunedParams, bestCost] = tune(tuner,detections,truthTable);

```

Generating code.
Code generation successful.

Your initial point x_0 is not between bounds lb and ub ; FMINCON shifted x_0 to strictly satisfy the bounds.

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
0	7	8.116826e+01	0.000e+00	2.749e+00	
1	14	7.303610e+01	0.000e+00	1.375e+00	2.809e+00
2	21	6.933060e+01	0.000e+00	7.783e-01	3.039e+00
3	28	6.597938e+01	0.000e+00	3.954e-01	4.808e+00
4	35	6.375682e+01	0.000e+00	2.266e-01	5.858e+00
5	42	6.215164e+01	0.000e+00	1.328e-01	7.779e+00
6	49	6.112787e+01	0.000e+00	1.123e-01	9.377e+00
7	56	6.052563e+01	0.000e+00	1.073e-01	1.052e+01
8	63	6.021176e+01	0.000e+00	1.184e-01	9.795e+00
9	70	6.012935e+01	0.000e+00	1.139e-01	2.429e+00
10	77	5.990467e+01	0.000e+00	6.043e-02	7.234e+00
11	84	5.986403e+01	0.000e+00	1.322e-01	6.622e+00
12	91	5.984704e+01	0.000e+00	6.894e-02	4.802e-01
13	98	5.982958e+01	0.000e+00	6.043e-02	2.277e+00
14	105	5.980530e+01	0.000e+00	2.285e-02	5.648e+00
15	112	5.979078e+01	0.000e+00	2.116e-02	7.667e+00

Solver stopped prematurely.

fmincon stopped because it exceeded the iteration limit,
options.MaxIterations = 1.500000e+01.

```
disp("Tuning time: " + toc);
```

Tuning time: 71.7755

```
disp("Best cost: " + bestCost);
```

Best cost: 59.7908

```
setTunedProperties(filter,tunedParams);
disp("Tuned process noise is:");
```

Tuned process noise is:

```
disp(filter.ProcessNoise);
```

```

34.9996    50.4336         0
50.4336    83.5728         0
         0         0    1.0000

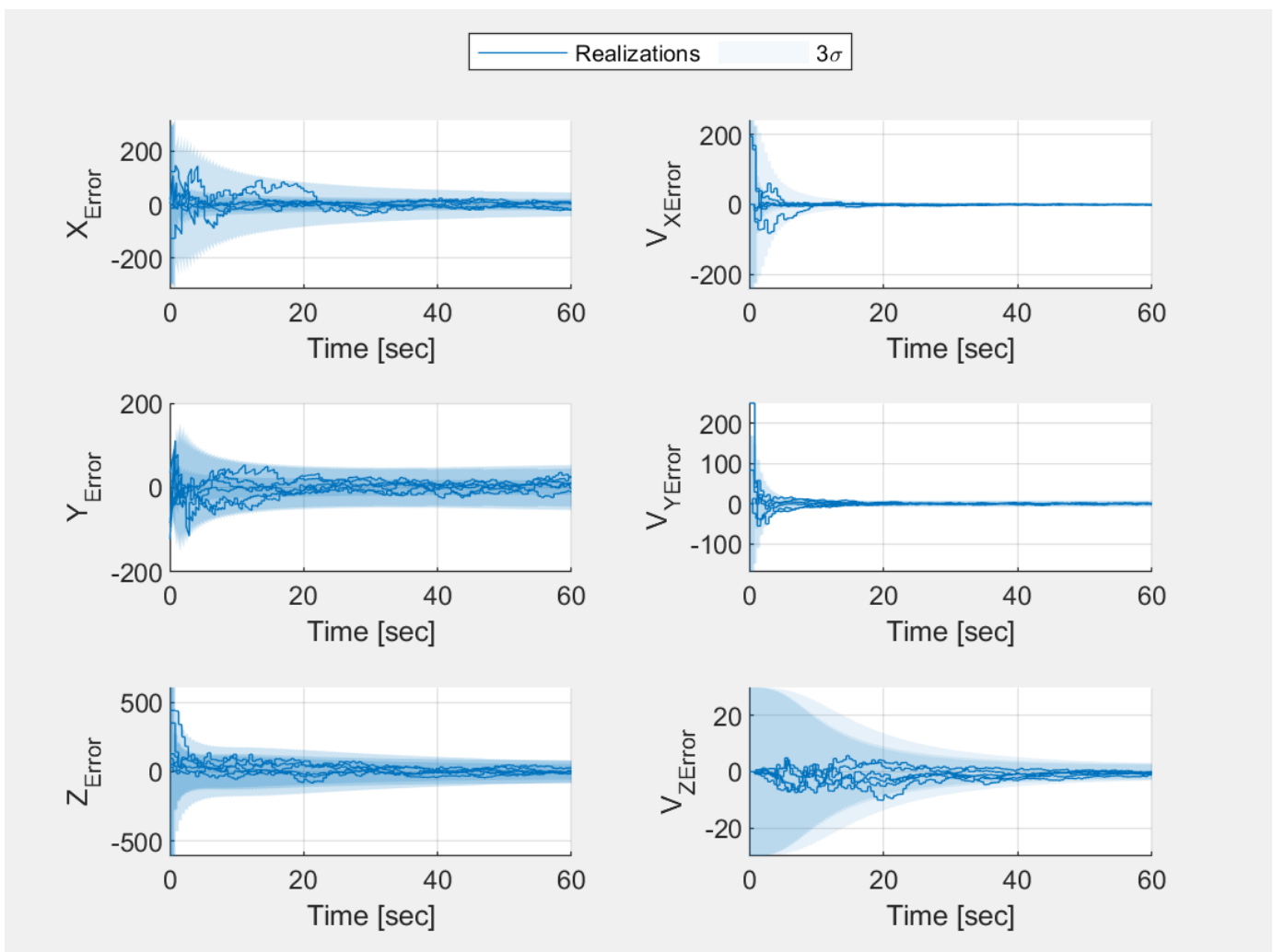
```

```

disp("Tuned initial state covariance is:");
Tuned initial state covariance is:
disp(filter.StateCovariance);
1.0e+04 *
    0.9582         0    -0.1154         0    -0.0824         0
         0    0.6434         0    0.0231         0         0
   -0.1154         0    0.0488         0    -0.3422         0
         0    0.0231         0    0.3186         0         0
   -0.0824         0    -0.3422         0         4.1085         0
         0         0         0         0         0    0.0100

plotFilterErrors(tuner);

```



Several changes indicate that the result of this tuning is much better than the result of tuning just the process noise. First, the value of the optimal RMSE cost is much lower than it was earlier. Second, the filter errors are all contained inside the 3-sigma bands throughout the scenario time. Finally, these errors and bands get narrower, indicating a good convergence for the filter estimate to the truth value.

Tune with a Custom Cost

The tuner has two built-in cost metrics: RMSE and normalized estimation error squared (NEES). The RMSE cost strives to minimize the average absolute error but may yield a filter that is either overconfident or underconfident about its estimate. The NEES cost strives to provide a consistent filter based on the cost proposed in [1]. Both RMSE and NEES account for errors in position and, if supplied in the truth table, in velocity.

There are cases where a custom cost allows better tuning. These cases are:

- 1 When using a truth table that contains a different set of data than position and velocity. Also, the tuner does not validate the truth table when using a custom cost.
- 2 When using a motion model that is not one of the built-in motion models: `constvelmscjac`, `constaccjac`, `constturnjac`, or `singer`.
- 3 When the cost must include additional elements or different metrics than the ones defined in the RMSE and NEES metrics.

The `helperCostNormalized` function attached with this example uses the augmented Mahalanobis distance proposed by Blackman and Popoli [2]. The cost is similar to NEES in the sense that it strives to yield a consistent filter by penalizing both underconfident filter estimates (large state covariance values) and overconfident filter estimates (small state covariance values).

Change the cost metric to the custom metric. You do not need to regenerate code.

```
tuner.Cost = "Custom";
tuner.CustomCostFcn = @(trkHistory, truth) helperCostNormalized(trkHistory, truth, "constvel");
tic;
[tunedParams, bestCost] = tune(tuner,detections,truthTable);
```

Your initial point `x0` is not between bounds `lb` and `ub`; `FMINCON` shifted `x0` to strictly satisfy the bounds.

Iter	F-count	f(x)	Feasibility	First-order optimality	Norm of step
0	7	5.972415e+01	0.000e+00	3.938e+00	
1	14	4.250375e+01	0.000e+00	1.074e+00	5.781e+00
2	21	3.997240e+01	0.000e+00	7.540e-01	2.021e+00
3	28	3.647123e+01	0.000e+00	3.655e-01	4.797e+00
4	35	3.470021e+01	0.000e+00	2.080e-01	4.774e+00
5	42	3.348664e+01	0.000e+00	1.137e-01	5.990e+00
6	49	3.273653e+01	0.000e+00	9.179e-02	6.968e+00
7	56	3.232928e+01	0.000e+00	1.024e-01	6.683e+00
8	63	3.218352e+01	0.000e+00	1.054e-01	3.375e+00
9	70	3.212378e+01	0.000e+00	1.031e-01	1.088e+00
10	77	3.186992e+01	0.000e+00	9.164e-02	6.811e+00
11	84	3.163440e+01	0.000e+00	2.814e-02	1.208e+01
12	91	3.164332e+01	0.000e+00	2.814e-02	7.510e-01
13	98	3.163372e+01	0.000e+00	2.814e-02	8.293e-01
14	105	3.157808e+01	0.000e+00	4.563e-02	4.516e+00
15	112	3.149118e+01	0.000e+00	1.438e-01	8.802e+00

Solver stopped prematurely.

```
fmincon stopped because it exceeded the iteration limit,
options.MaxIterations = 1.500000e+01.
```

```
disp("The time needed to tune the filter after code generation: " + toc);
```

The time needed to tune the filter after code generation: 57.4512

```
setTunedProperties(filter,tunedParams);  
disp("Tuned process noise is:");
```

Tuned process noise is:

```
disp(filter.ProcessNoise);
```

```
    69.5499    79.3440         0  
    79.3440    96.3635         0  
         0         0    1.0000
```

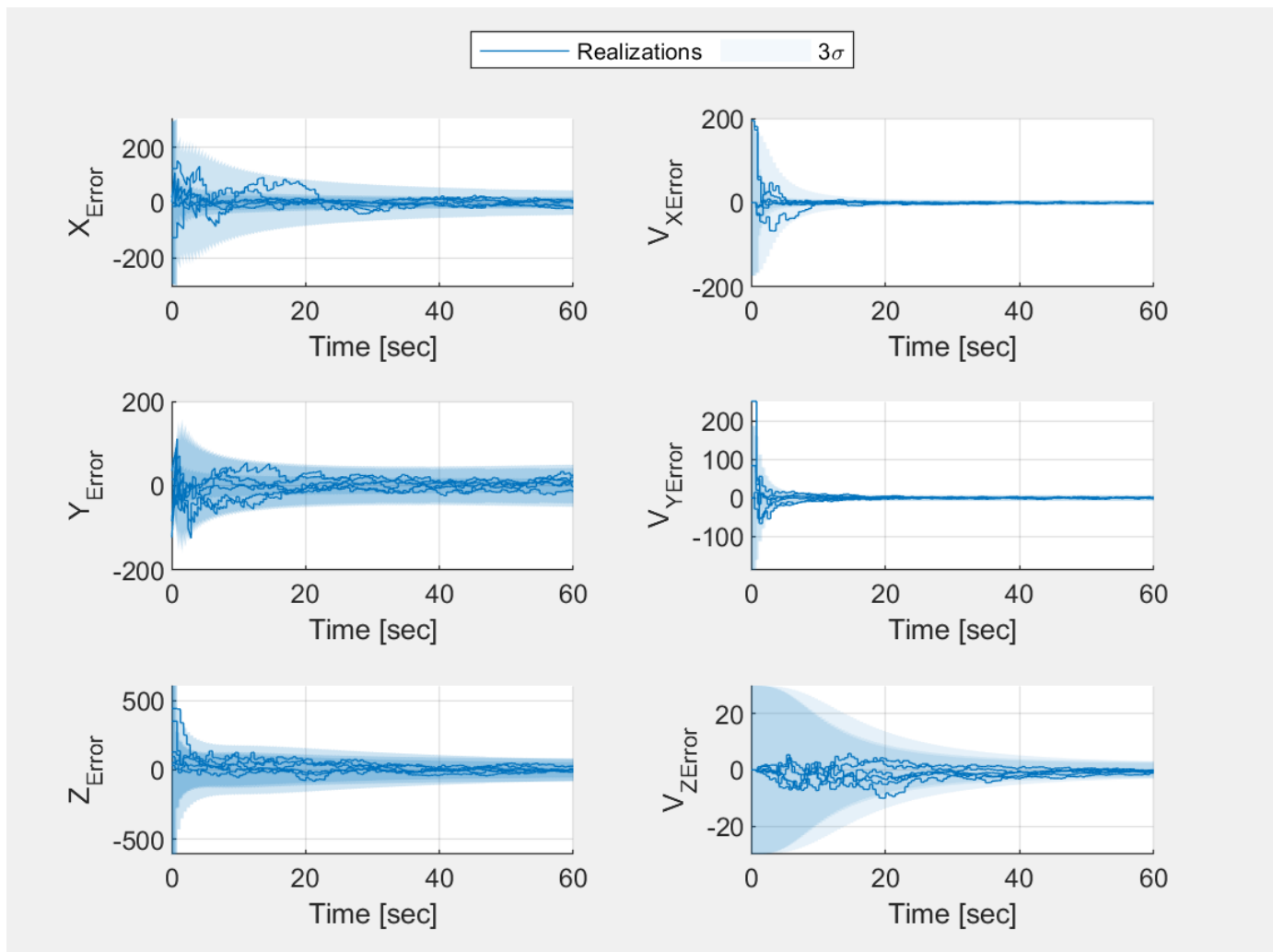
```
disp("Tuned initial state covariance is:");
```

Tuned initial state covariance is:

```
disp(filter.StateCovariance);
```

```
    1.0e+04 *  
    0.9582         0   -0.1154         0   -0.0824         0  
         0    0.3330         0    0.0280         0         0  
   -0.1154         0    0.0488         0   -0.3422         0  
         0    0.0280         0    0.3875         0         0  
   -0.0824         0   -0.3422         0    4.1085         0  
         0         0         0         0         0    0.0100
```

```
plotFilterErrors(tuner);
```



```
disp("Optimized cost: " + bestCost);
```

```
Optimized cost: 31.4912
```

Use the Tuned Filter in a Tracker

To observe the improvement in tracking quality that a tuned filter provides, use the tuning result in a multitarget tracker. First, use the `exportToFunction` object function of the tuner to export the filter initialization function.

```
exportToFunction(tuner, 'myTunedInitFcn');
```

Verify the tuned properties of the filter object that is generated by the tuned initialization function.

```
filter = myTunedInitFcn(sampleDetection);
disp(filter.ProcessNoise);
```

```
    69.5499    79.3440         0
    79.3440    96.3635         0
         0         0    1.0000
```

```
disp(filter.StateCovariance);
```

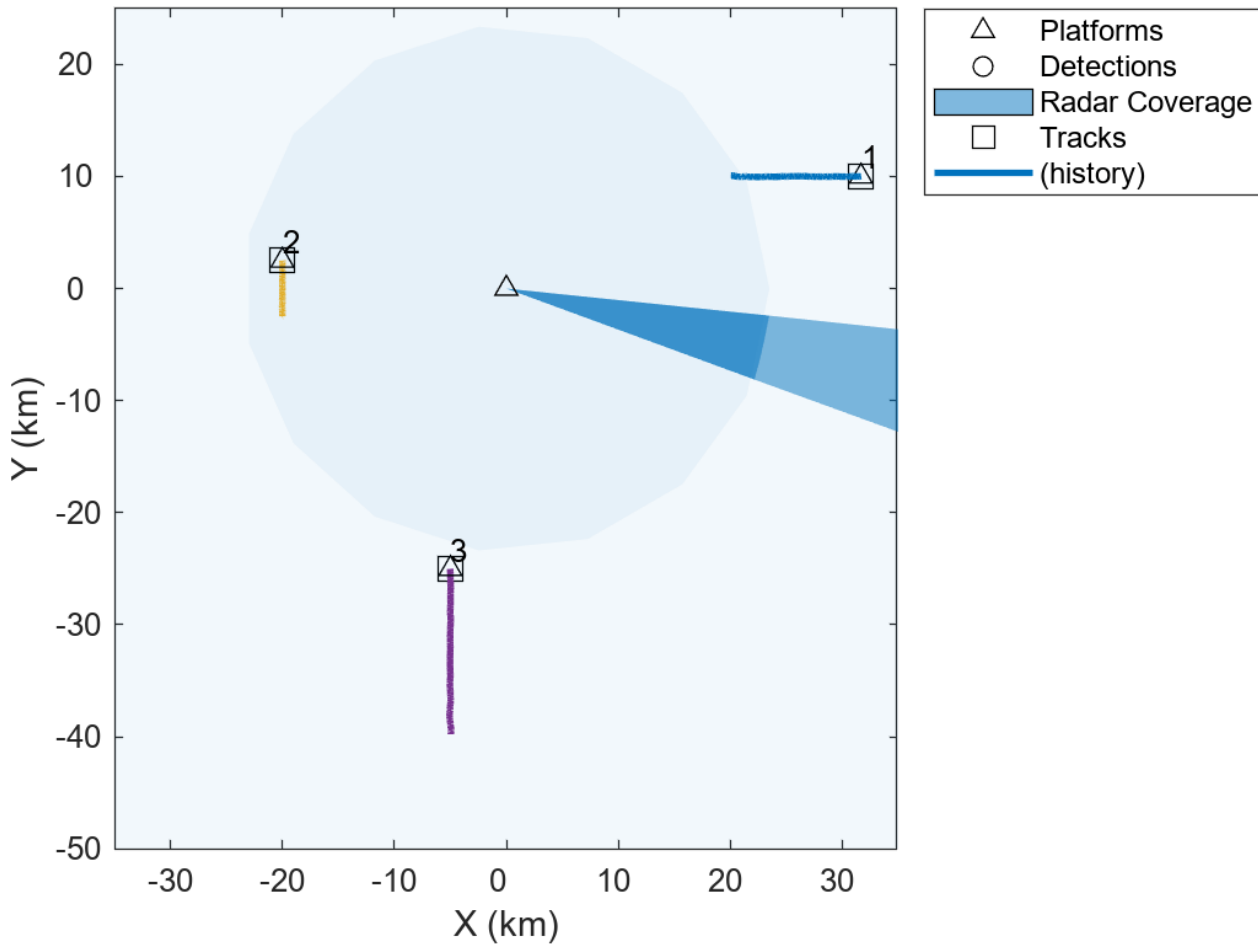
```

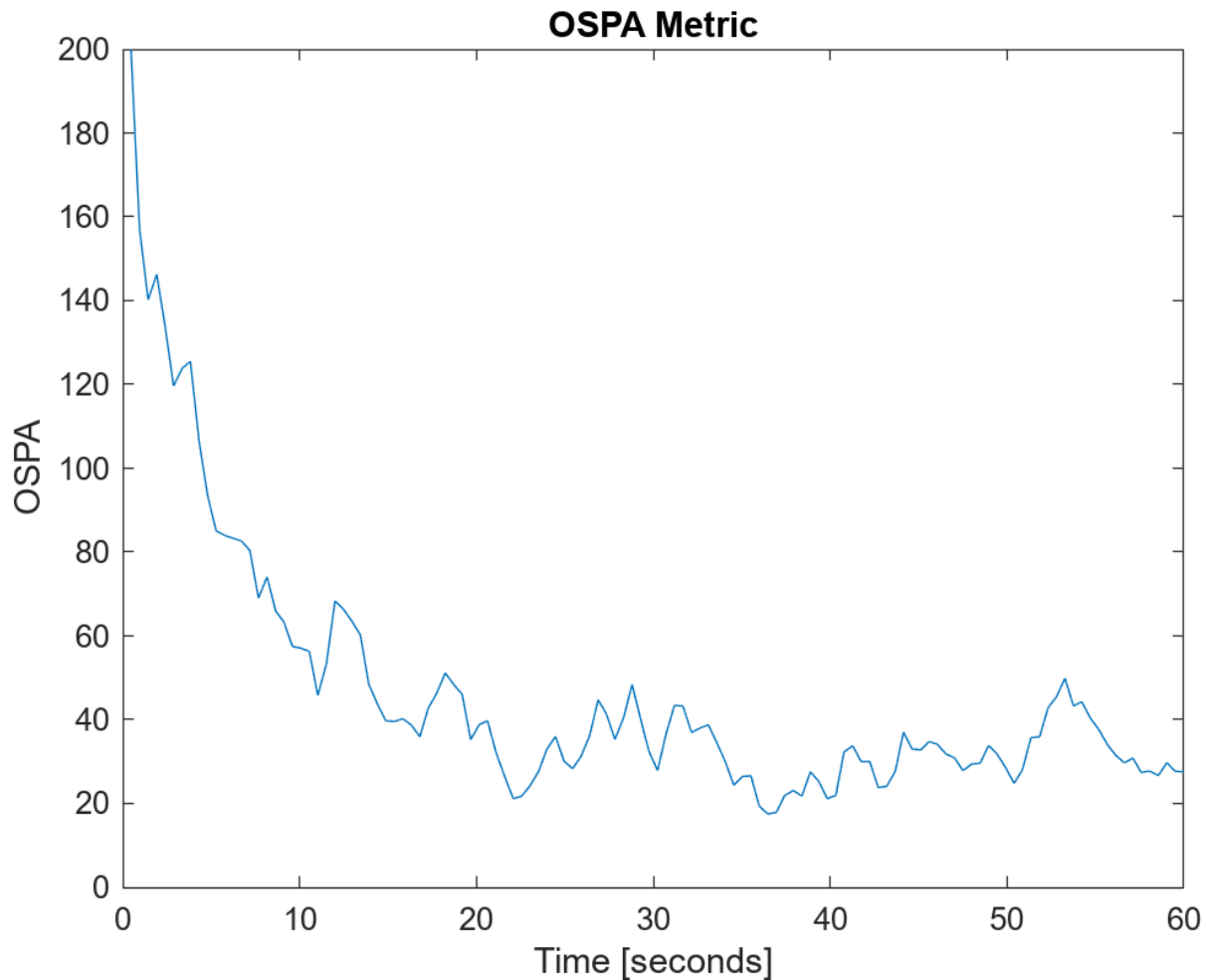
1.0e+04 *
  0.9582      0 -0.1154      0 -0.0824      0
    0      0.3330      0      0.0280      0      0
 -0.1154      0      0.0488      0 -0.3422      0
    0      0.0280      0      0.3875      0      0
 -0.0824      0 -0.3422      0      4.1085      0
    0      0      0      0      0      0.0100
    
```

Construct the tracker with this filter initialization function and observe the tracking results. Note that tuning the filter to better handle the initial state uncertainty allow you to reduce the `AssignmentThreshold` property. Having a smaller assignment threshold can help the tracker better assign detections to tracks.

```

tracker = trackerGNN(FilterInitializationFcn = "myTunedInitFcn", AssignmentThreshold = 50);
helperRunScenario(scenario, tracker, metric);
    
```





The OSPA metric plot above shows a much-improved tracking result. From the metric, once the tracks are established, around 5 seconds into the simulation, the average absolute position error is about 20-80 meters, which is a very good accuracy given the radar measurement accuracy. This result also agrees with the results of the tuning shown in previous sections.

Summary

In this example, you learned how to automatically tune a tracking filter using a tracking filter tuner. You learned how to control the tuned parameters, how to choose a cost for the tuning, and you used various optimization algorithms. You also learned how to export the tuned filter to a filter initialization function that returns a tuned filter, which can then be used with a multi-target tracker.

References

[1] Chen, Z., N. Ahmed, S. Julier, and C. Heckman, "Kalman Filter Tuning with Bayesian Optimization." *ArXiv:1912.08601 [Cs, Eess, Math]*, Dec. 2019. *arXiv.org*, <http://arxiv.org/abs/1912.08601>.

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

Supporting Functions

The functions included in this section are intended as helpers for this example and may be removed, modified, or renamed in a future release.

helperRecordingToTuningData

Create detection log and truth data from an array of trackingScenarioRecording objects.

```
function [detlog,truth] = helperRecordingToTuningData(recording)

detlog = extractDetections(recording);
truth = extractTruth(recording);

isemptyDetlog = cellfun(@(dl) isempty(dl), detlog);
detlog = detlog(~isemptyDetlog);
truth = truth(~isemptyDetlog);
end

function truth = extractTruth(recording)
data = recording.RecordedData;
numTruths = numel(data(1).Poses);
Time = seconds([data.SimulationTime]');
t = cell(1,numTruths);
poses = [data.Poses];
for j = 1:numTruths
    positions = reshape([poses(j,:).Position],3,[]);
    velocities = reshape([poses(j,:).Velocity],3,[]);
    t{j} = timetable(Time,positions,velocities,'VariableNames',{'Position','Velocity'});
end
truth = repmat(t,1,numel(recording));
end

function detlog = extractDetections(recording)
numRuns = numel(recording);
numTruths = numel(recording(1).RecordedData(1).Poses);
detlog = repmat({},1,numTruths * numRuns);
logIdx = 0;
for runIdx = 1:numRuns
    data = recording(runIdx).RecordedData;
    detections = vertcat(data.Detections);
    for truthIdx = 1:numTruths
        platID = data(1).Poses(truthIdx).PlatformID;
        fromThisPlat = cellfun(@(d) d.ObjectAttributes{1}.TargetIndex == platID, detections);
        logIdx = logIdx + 1;
        detlog{logIdx} = detections(fromThisPlat);
    end
end
end
```

helperVisualizeTuningData

This function visualizes the tuning data and returns the theaterPlot visualization object.

```
function helperVisualizeTuningData(detections, truthTable)
thp = theaterPlot(AxesUnits=["km","km","km"],XLimits=[-35000 35000],YLimits=[-50000 25000]);
trp = trajectoryPlotter(thp,DisplayName="Truth Trajectories");
```

```

numTruths = numel(truthTable);
pos = cell(1,numTruths);
for i = 1:numTruths
    pos{i} = truthTable{i}.Position;
end
plotTrajectory(trp,pos);

knownPlatforms = [];
existingDetPlotters = {};
colors = colororder;
for j = 1:numTruths
    d = vertcat(detections{j}{:});
    if ~isempty(d)
        platIndex = d(1).ObjectAttributes{1}.TargetIndex;
        if ~any(platIndex == knownPlatforms)
            dtp = detectionPlotter(thp,DisplayName=("Platform"+platIndex+"Detections"),MarkerEdge
                knownPlatforms(end+1) = platIndex; %#ok<*AGROW>
                existingDetPlotters{end+1} = dtp;
        else
            dtp = existingDetPlotters{platIndex == knownPlatforms};
        end
        detpos = arrayfun(@(d) d.Measurement, d, UniformOutput=false);
        plotDetection(dtp,[detpos{:}]);
    end
end
end

```

helperRunScenario

A function to run a recording, track the objects, visualize the results, and analyze tracking quality.

```

function helperRunScenario(scenario, tracker, metric)
% Initialize plotters
persistent thp plp dep cvp tap
if isempty(thp)
    thp = theaterPlot(AxesUnits=["km", "km", "km"],XLimits=[-35000 35000],YLimits=[-50000 25000]);
end
if isempty(plp)
    plp = platformPlotter(thp,DisplayName="Platforms");
end
if isempty(dep)
    dep = detectionPlotter(thp,DisplayName="Detections");
end
if isempty(cvp)
    cvp = coveragePlotter(thp,DisplayName="Radar Coverage");
end
if isempty(tap)
    tap = trackPlotter(thp, DisplayName="Tracks", ConnectHistory="on", ColorizeHistory="on");
end

% Reset all objects
clearPlotterData(thp);
restart(scenario);
reset(tracker);
reset(metric);

% Modify the radar to rotate faster and provide more detections
radar = scenario.Platforms{1}.Sensors{1};

```

```

release(radar);
radar.FieldOfView(1) = 14;
radar.MaxMechanicalScanRate(1) = 750;

% Initialize variables. There are 3215 timestamps in this scenario
ospa = zeros(1,3215);
trackerTimes = zeros(1,3215);
detBuffer = {};
tracks = objectTrack.empty;
index = 0;

% For repeatable results, use a constant RNG seed
r = rng(0, 'twister');
h = onCleanup(@() rng(r));

while advance(scenario)
    time = scenario.SimulationTime;
    poses = platformPoses(scenario);
    covcon = coverageConfig(scenario);
    [detections, sencon] = detect(scenario);

    if ~sencon.IsScanDone
        detBuffer = vertcat(detBuffer,detections);
    else
        tracks = tracker(detBuffer, time);
        index = index + 1;
        ospa(index) = metric(tracks,poses(2:4));
        trackerTimes(index) = time;
        detBuffer = {};
    end

    if isempty(tracks)
        trkpos = zeros(0,3);
        trkIDs = string.empty;
    else
        trkpos = getTrackPositions(tracks,"constvel");
        trkIDs = string([tracks.TrackID]);
    end

    if isempty(detBuffer)
        detpos = zeros(0,3);
    else
        detpos = cellfun(@(db) db.Measurement, detBuffer, 'UniformOutput', false);
        detpos = [detpos{:}];
    end
    plotPlatform(plp, reshape([poses.Position],3,[]));
    plotCoverage(cvp, covcon);
    plotDetection(dep, detpos);
    plotTrack(tap, trkpos, trkIDs);
end

figure;
plot(trackerTimes(1:index),ospa(1:index));
ylim([0 200]);
title('OSPA Metric');
xlabel('Time [seconds]');
ylabel('OSPA');
end

```

helperCostNormalized

Calculate the Blackman and Popoli augmented Mahalanobis distance.

Given the error between the state at time step k and Monte Carlo run i and the state estimate at that time, $x_{\text{err}}(k, i) = x(k, i) - x_{\text{est}}(k, i)$, with a state covariance $P(k, i)$, the Blackman and Popoli cost is:

$$d(k, i) = x_{\text{err}}(k, i)' P(k, i)^{-1} x_{\text{err}}(k, i) + \log(\det(P(k, i))) \quad \forall k \in 1, \dots, T, \forall i \in 1, \dots, N$$

T is the number of timesteps. N is the number of Monte Carlo runs.

Note that the $\log(\det(P(k, i)))$ term is added to penalize large values of state covariance whereas the Mahalanobis cost term serves to penalize overconfident filter estimates.

The overall cost is the average of $d(k, i)$ over all timesteps and runs.

```
function cost = helperCostNormalized(trkHistory, truthTable, model)

% Process truthTable to position and velocity truths
posTruth = [truthTable.Position];
hasVelocity = ismember('Velocity', truthTable.Properties.VariableNames);
if hasVelocity
    velTruth = [truthTable.Velocity];
end

[numSteps, numRuns] = size(trkHistory);
c = zeros(numSteps, numRuns, 'like', posTruth);

for run = 1:numRuns
    [posEst, posCovs] = getTrackPositions(trkHistory(:, run), model);
    [velEst, velCovs] = getTrackVelocities(trkHistory(:, run), model);

    if hasVelocity
        poserror = posTruth - posEst;
        velerror = velTruth - velEst;
        stateerror = [poserror, velerror];
    else
        stateerror = posTruth - posEst;
    end

    for step = 1:numSteps
        if hasVelocity
            P = blkdiag(posCovs(:, :, step), velCovs(:, :, step));
        else
            P = posCovs(:, :, step);
        end
        c(step, run) = stateerror(step, :) / P * stateerror(step, :)' + log(det(P));
    end
end

cost = mean(c, "all");
end
```

Asynchronous Angle-only Tracking with GM-PHD Tracker

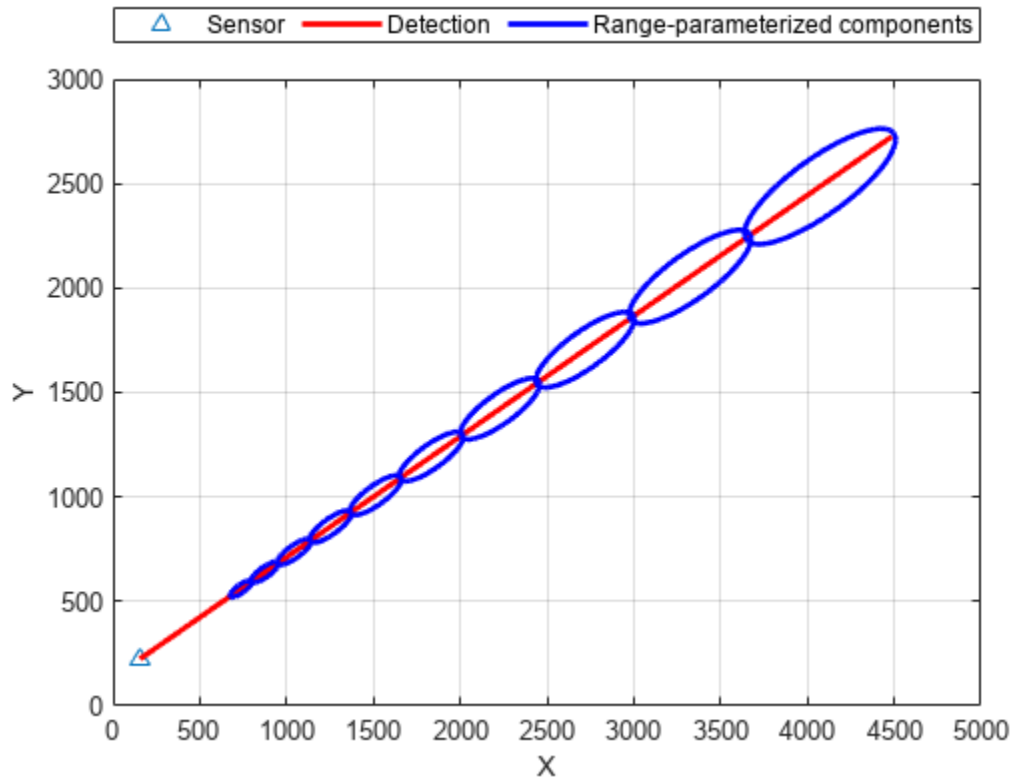
This example shows you how to track objects using angle-only measurements from spatially distributed and asynchronous passive sensors. Angle-only measurements from a single sensor provide incomplete observability for estimating the state of an object. When working with angle-only measurements, there are two main methods to completely observe the object state. In the case of a single sensor, the sensing platform needs to out-maneuver the object to obtain range information. Alternatively, you can use multiple spatially distributed passive sensors to gain observability of the object state

You can find the single sensor workflow in “Passive Ranging Using a Single Maneuvering Sensor” on page 6-241 example. For the case that angle-only measurements from multiple sensors are received at the same time (synchronous sensors), you can use a *static fusion before tracking* [1] architecture illustrated in the “Tracking Using Distributed Synchronous Passive Sensors” on page 6-227 example. In this example, you track objects using asynchronous passive sensors, which do not provide information at the same timestamp.

Introduction

In the asynchronous multi-sensor angle-only tracking problem, since detections from all sensors are not available at the same time step, the prior distribution of object states must be modeled by using detections from a single sensor. A popular technique to initialize prior distribution using angle-only detections is by using range-parameterization. The range-parameterization technique defines the prior distribution as a Gaussian mixture and initializes many Gaussian components at different range hypothesis for the object. The range-parameterization process from a single angle-only detection is shown in the image below.

```
helperAsyncAngleOnlyTrackingDisplay.plotRangeParameterization();
```



The formulation of Gaussian mixture probability hypothesis density (GM-PHD) filter offers a natural approach to integrate the range-parameterization. By using the range-parameterization technique to describe the adaptive birth intensity of the GM-PHD filter, you can model the prior distribution using angle-only measurements from a single sensor [2]. When the filter is updated with detections from other sensors, this prior distribution has higher likelihood of being closer to the intersections of angle-only measurements. As multiple angle-only measurements can intersect at both true and ghost object locations, the GM-PHD filter tries to disambiguate between them using two main sources of information. First, components at true target locations have higher likelihood to associate with measurements from multiple sensors than those at ghost locations. Second, components at ghost object locations may deviate from the expected prior motion model defined for the objects.

Setup Scenario

In this example, you use a similar scenario as shown in the “Tracking Using Distributed Synchronous Passive Sensors” on page 6-227 example. The relative placement of sensors and objects in this scenario is same as that of an example in [1]. The scenario consists of five equally-spaced objects observed by three passive sensors. You use the `radarEmitter` object and the `fusionRadarSensor` to simulate radar emission and detection respectively. Set the `DetectionMode` property of `fusionRadarSensor` to `ESM` to model a passive radar sensor. Each sensor has a field of view of 180 degrees in azimuth and produces about 2 to 3 false alarms per step. The sensor update rate is 1 Hz and the sensors have a clock offset of 1/3 seconds between each other. The `HasNoise` property of the sensors is set to `false` to first generate noise-free detections. After that, you add white noise with a variance of 2 square degrees to the measurements. Note that the estimation accuracy and ghost disambiguation highly depends on the measurement accuracy. For more details on scenario creation, refer to the helper function, `helperCreatePassiveAsyncScenario`, attached with this example.

```

% For reproducible results
rng(2022)

% Create scenario
scenario = helperCreatePassiveAsyncScenario;

% Create display
display = helperAsyncAngleOnlyTrackingDisplay;

```

Setup Tracker and Metric

In this section, you set up a Gaussian-mixture probability hypothesis density (GM-PHD) multi-object tracker to track the objects using angle-only measurements from asynchronous sensors. You also set up the OSPA-on-OSPA (Optimal Subpattern Assignment) or OSPA(2) metric [3] to evaluate results from the tracker based on the simulated ground truth.

GM-PHD Tracker

You configure the tracker using the `trackerPHD` System Object™. The `trackerPHD` object requires the definition of sensor configurations as `trackingSensorConfiguration` objects. You can obtain this information from the simulated sensor models in the scenario. In addition to that, you also define the range-parameterized birth intensity by setting the `FilterInitializationFcn` to the helper function, `initcvAngleOnlyGMPHD`, included in this example script. This function creates range-parameterized Gaussian mixture with 10 components ranging from 600 and 5000 meters from the sensor. You use these sensor configurations to construct the PHD tracker.

```

% Construct sensor configurations
sensorConfigs = trackingSensorConfiguration(scenario,...
    FilterInitializationFcn=@initcvAngleOnlyGMPHD);

% Construct tracker
tracker = trackerPHD(SensorConfigurations=sensorConfigs,...
    HasSensorConfigurationsInput=true,...
    BirthRate=0.1,...
    MaxNumComponents=5000,...
    ExtractionThreshold=0.85,...
    ConfirmationThreshold=0.95);

```

OSPA(2) metric

You setup the OSPA(2) metric to measure the performance of the tracker. The OSPA(2) metric allows you to evaluate the tracking performance over a history of tracks as opposed to instantaneous results in the traditional OSPA metric. As a result, the OSPA(2) metric penalizes phenomenon such as track switching and fragmentation more consistently. You configure the OSPA(2) metric by using the `trackOSPAMetric` object and setting the `Metric` property to "OSPA(2)". You choose absolute error in position as the base distance between a track and truth at a time instant by setting the `Distance` property to "posabserr".

```

% Define OSPA metric calculator
ospaMetric = trackOSPAMetric(Metric="OSPA(2)",...
    Distance="posabserr",...
    WindowLength=25,...
    CutoffDistance=150);

```

Run Scenario and Track Objects

In this section, you run the scenario in a loop, generate data from asynchronous sensors and feed the data to the multi-object tracker. For qualitative assessment of tracking performance, you visualize the

ground truth and the estimated tracks. Further, to understand how the PHD tracker can discriminate between ghosts and true objects, you plot the PHD estimate in the 2-D position space. For quantitative assessment of the tracking performance, you evaluate the OSPA(2) metric.

```

% Initialize variables
ospa2 = zeros(0,1);
cardOspa2 = zeros(0,1);
locOspa2 = zeros(0,1);
tracks = struct.empty(0,1);
phd = initcvgmphd;

% Measurement accuracy (deg^2)
measNoise = 2;

% Run the scenario
while advance(scenario)
    % Current time
    time = scenario.SimulationTime;

    % Generate emissions from object
    [emTx, emConfigs] = emit(scenario);

    % Generate detections from emissions
    [detections, configs] = detect(scenario, emTx, emConfigs);

    % Add noise to detections
    detections = addNoise(detections, measNoise);

    if isLocked(tracker) || ~isempty(detections)
        % Update tracker
        tracks = tracker(detections, configs, time);

        % Get PHD filter from the tracker
        phd = helperAccessPHDFilterInformation.getFilter(tracker);
    end

    % Obtain ground truth
    platPoses = platformPoses(scenario);
    gTruth = platPoses(1:5);

    % Calculate OSPA(2) metric
    [ospa2(end+1,1), cardOspa2(end+1,1), locOspa2(end+1,1)] = ospaMetric(tracks, gTruth); %#ok<SAGI

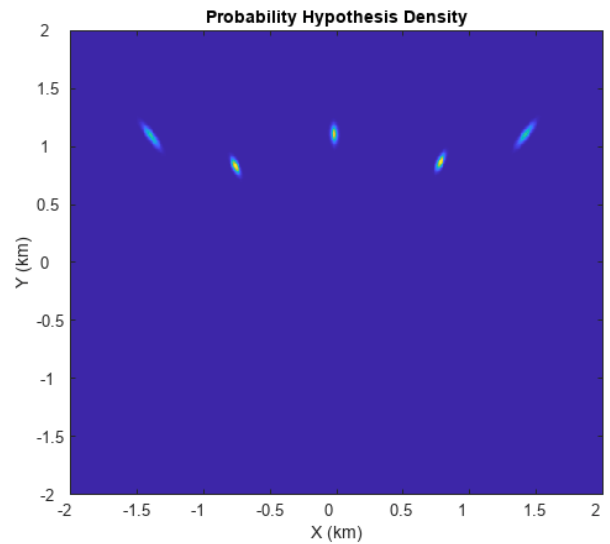
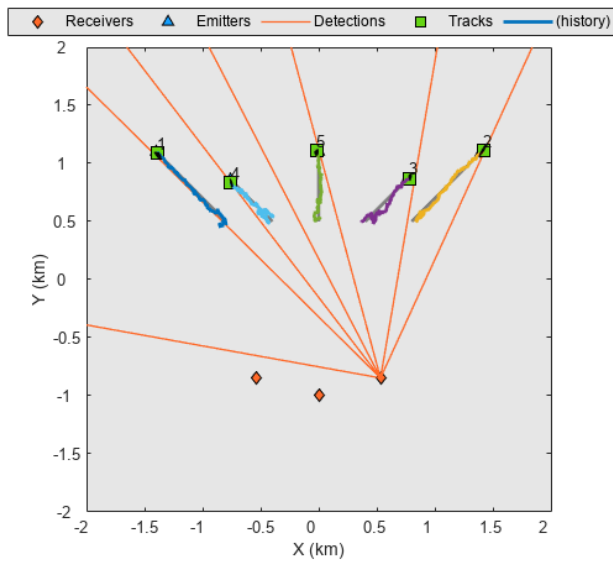
    % Update display
    display(scenario, tracks, detections, phd);
end

```

Results

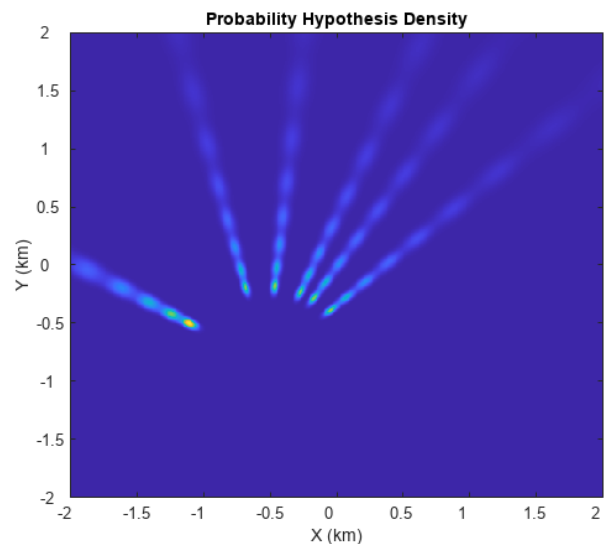
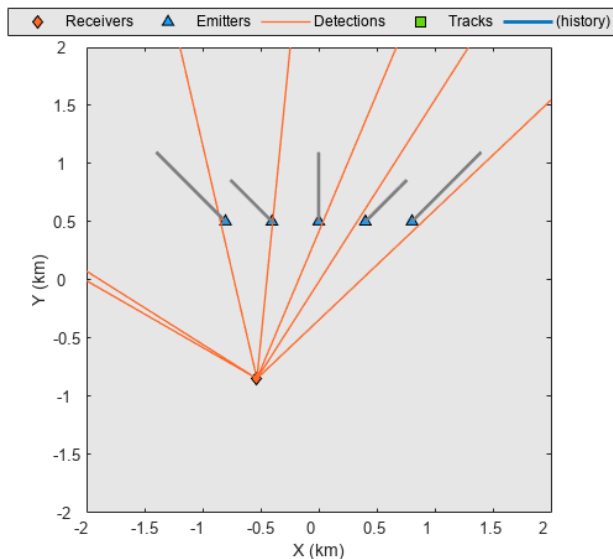
The images below show the ground truth, the true trajectories of the object, estimated tracks, and the PHD estimate in the 2-D position space. The first snapshot shows the result at the last time step of the scenario. You can see that tracker was able to maintain a track on every object. Also, the PHD estimate has strong peaks at true target locations, which implies that the tracker is confident about its estimate and does not have ambiguity between potential ghost targets and true targets.

```
showSnaps(display, time);
```

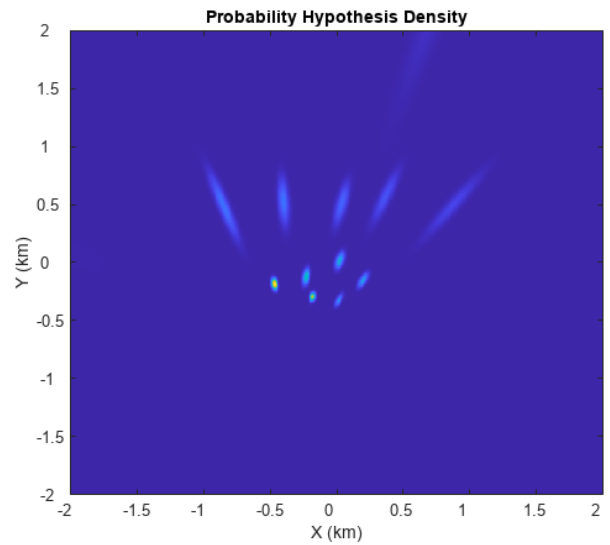
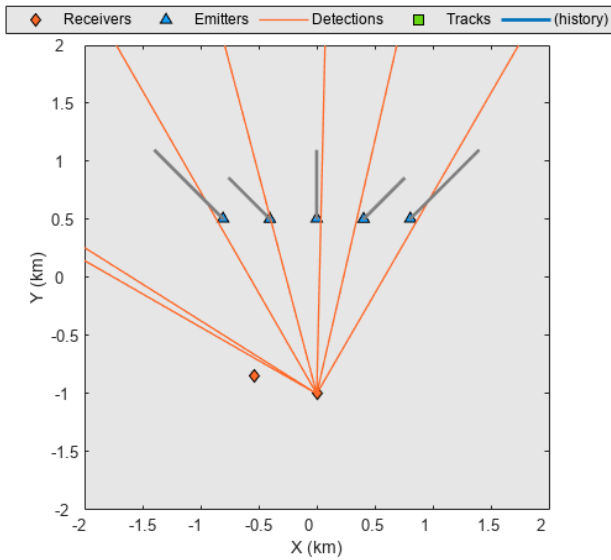
In the next few snapshots, you observe how the PHD is initialized at the first step from a single sensor and updated with new information from other sensors. In the first snapshot at time of 1/3 seconds, the peaks of the PHD follow the range-parameterized components based on the detections received from the first sensor.

```
showSnaps(display,1/3);
```

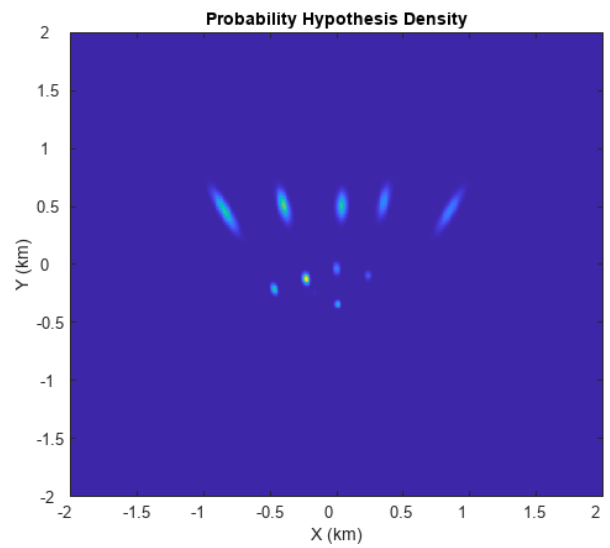
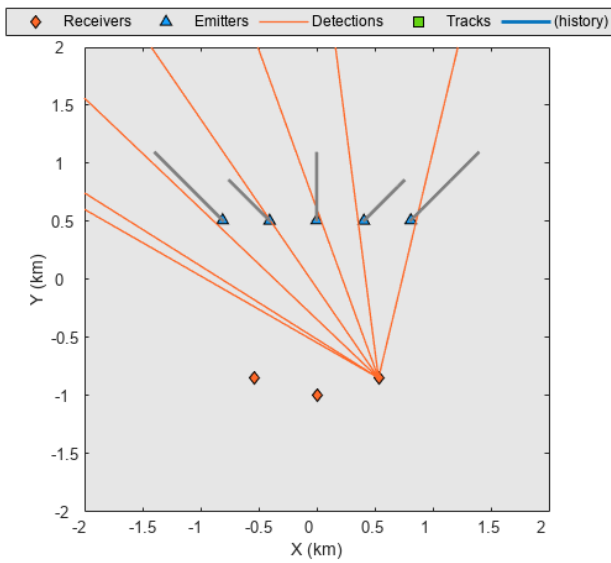


In the next two snapshots at time of 2/3 seconds and time of 1 second, notice how the PHD is updated with detections from the second and third sensor, respectively. After updating with detections from the second sensor, the PHD estimate is ambiguous because peaks from ghost objects are also present on the PHD. These peaks lie close to intersections between detections from the first sensor and the second sensor. Similarly, after the updating with detections from the third sensor, stronger peaks appear at positions close to the intersections of all three sensor measurements. After 1 second, the tracker estimate is still ambiguous as peaks are still present at some ghost object locations.

```
showSnaps(display,2/3);
```

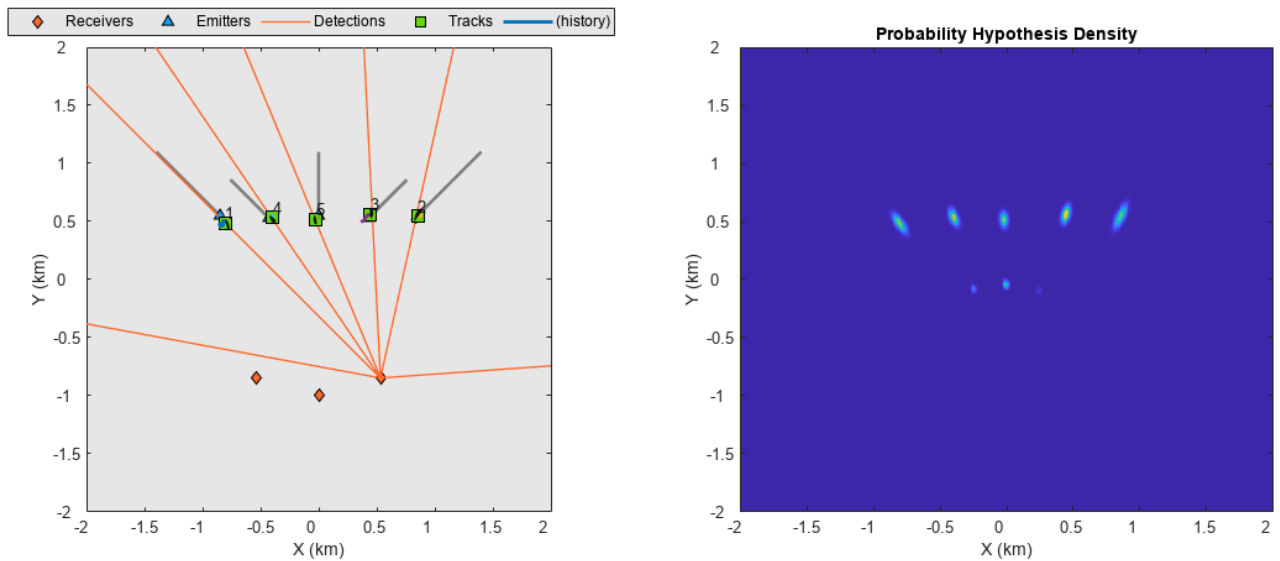


```
showSnaps(display,1);
```



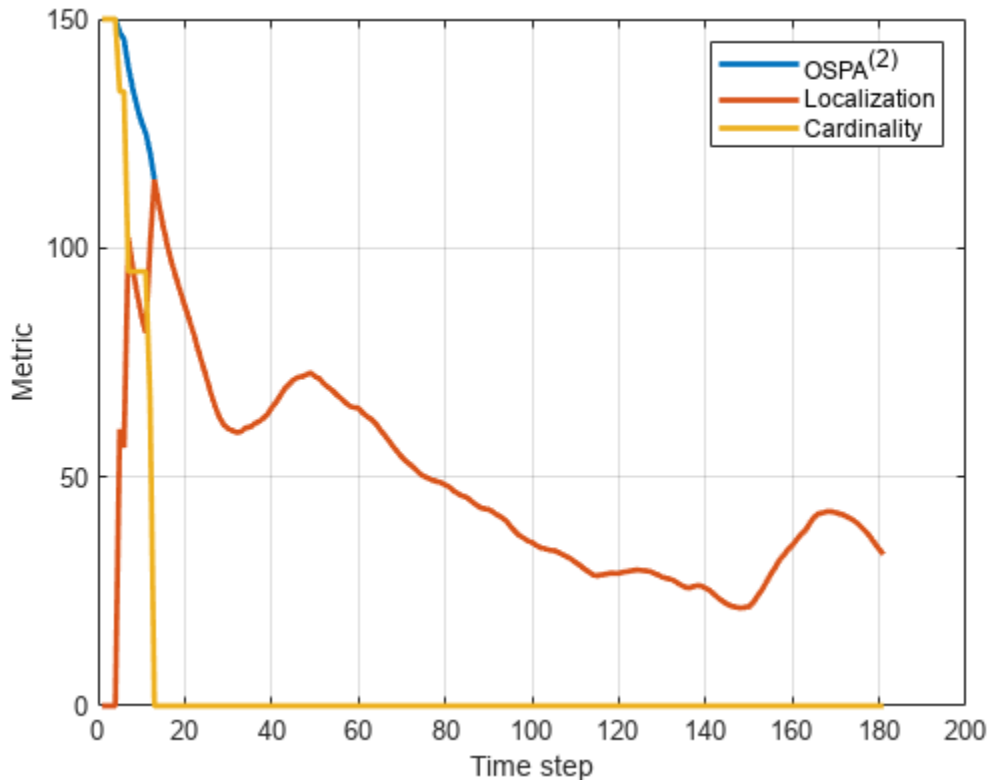
The tracker converges on the true peaks around 5 seconds as shown in the snapshot below. At this time, the tracker extracts tracks from these peaks, which are display in the plot on the left.

```
showSnaps(display,5);
```



You also plot the OSPA(2) metric as a function of time to quantitatively assess the performance of the tracker. Notice that the OSPA(2) metric converges to a value less than 50, illustrating the average estimation accuracy per ground truth. The OSPA(2) metric is composed of two components, a localization component which measures the estimation accuracy of the trajectories and a cardinality component which measures the mismatch between numbers of true and estimated targets. The cardinality component of the metric shows that no false tracks or missed targets were present during this scenario simulation.

```
figure();
plot([ospa2 cardospa2 locospa2], 'LineWidth', 2);
grid on;
xlabel('Time step');
ylabel('Metric');
legend({'OSPA^{(2)}', 'Localization', 'Cardinality'});
```



Summary

In this example, you learned how to simulate a scenario with asynchronous passive sensors and generate angle-only measurements. You simulated angle-only measurements from the sensors and fused them using a GM-PHD tracker. You qualitatively assessed the performance of the algorithms by visualizing the estimated tracks as well as the PHD. You also quantitatively assessed the performance by calculating the OSPA(2) metric.

References

- [1] Bar-Shalom, Yaakov, Peter K. Willett, and Xin Tian. "Tracking and Data Fusion: A Handbook of Algorithms." (2011).
- [2] Hamidi, Dimitri, et al. "Angle-Only, Range-Only and Multistatic Tracking Based on GM-PHD Filter." 2021 IEEE 24th International Conference on Information Fusion (FUSION). IEEE, 2021.
- [3] Beard, Michael, Ba Tuong Vo, and Ba-Ngu Vo. "OSPA (2): Using the OSPA metric to evaluate multi-target tracking performance." 2017 International Conference on Control, Automation and Information Sciences (ICCAIS). IEEE, 2017.

Supporting Functions

addNoise

This function adds noise to azimuth measurement of 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

```

initcvAngleOnlyGMPHD

This function initializes a range-parameterized Gaussian mixture PHD filter from an angle-only detection.

```

function filter = initcvAngleOnlyGMPHD(detection)
% Define range-parameterization parameters
numComponents = 10;
Rmin = 600;
Rmax = 5000;

% No components added to predictive birth intensity
filter = initcvgmphd;
if nargin == 0
    return;
end

% Use initrpekf to easily get states and covariances of the filter bank
rpekf = initrpekf(detection{1}, numComponents, [Rmin Rmax]);

% Add components to the PHD filter
for i = 1:numComponents
    % Gaussian component for one range
    thisFilter = gmphd(rpekf.TrackingFilters{i}.State,...
        rpekf.TrackingFilters{i}.StateCovariance);

    % Append the filter to total PHD filter resulting in a Gaussian mixture.
    append(filter, thisFilter);
end
end

```

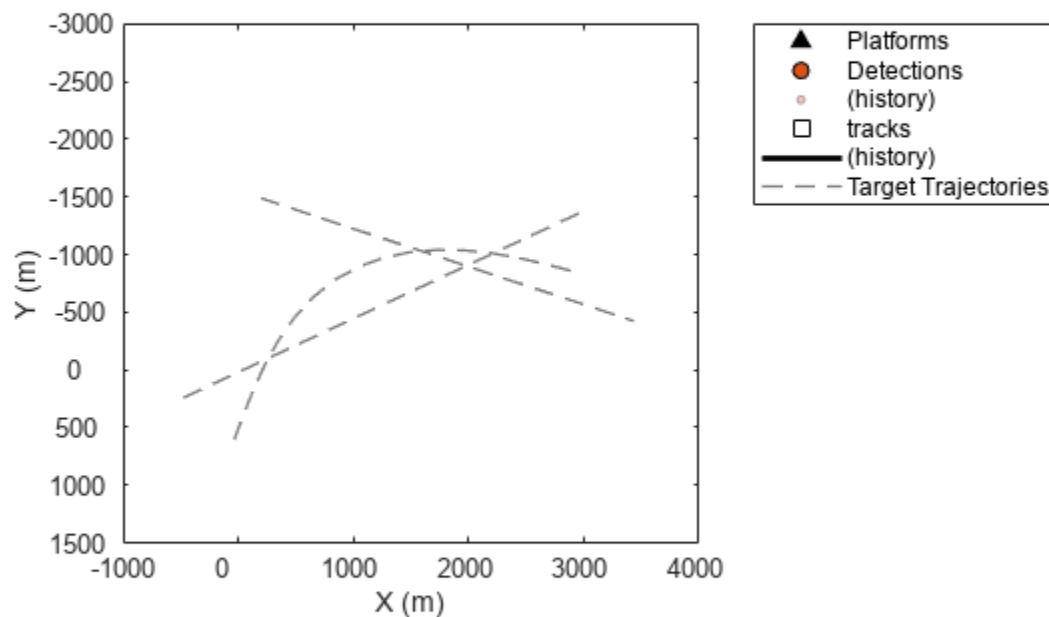
Analyze Track and Detection Association Using Analysis Info

This example shows how to use the analysis info output of the `trackerGNN` and `trackerJPDA` System objects to derive useful quantities about the assignments between tracks and detections.

Scenario and simulation

Create and simulate a simple tracking scenario and save the histories of detections, tracks, and analysis info. The scenario contains three targets with crossing trajectories. Targets are moving at a constant velocity. Use a monostatic radar sensor to generate positional detections of the targets.

```
scenario = createScenario();  
[platp, detp, trackp] = createPlotters(scenario);
```



Create a trackerGNN object with the default configuration. Increase the association gate by setting the AssignmentThreshold property to 100 to preserve all possible associations.

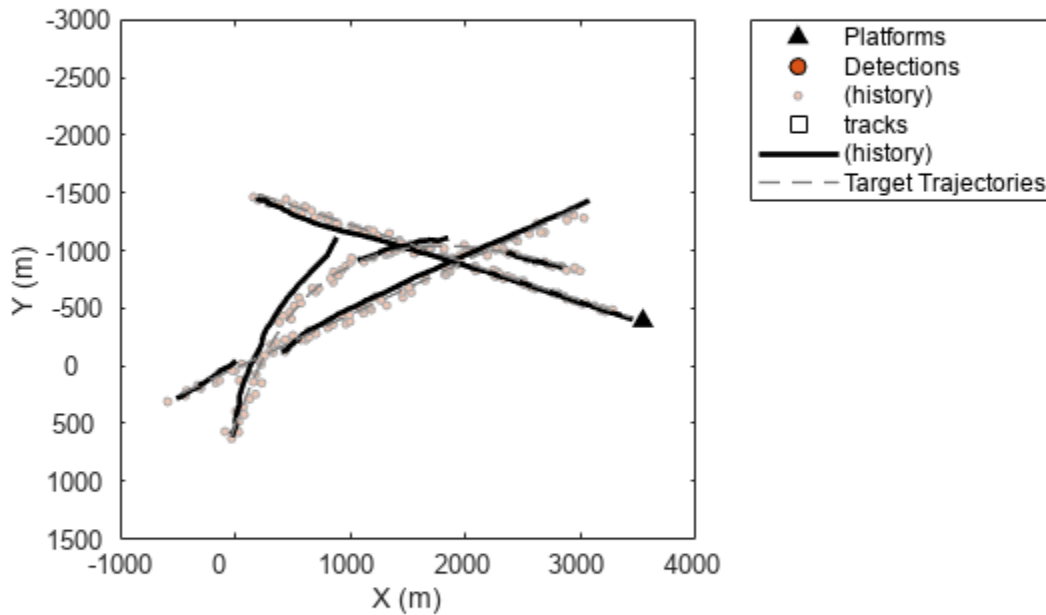
```
tracker = trackerGNN(AssignmentThreshold=100);
detlog = {};
tracklog = {};
infolog = {};

rng(2022); %For repeatable results
while advance(scenario)
    % Generate sensor data
    [dets, configs, sensorConfigPIDs] = detect(scenario);

    % Update tracker
    [tracks, ~, ~, info] = tracker(dets, scenario.SimulationTime);

    % Update plots
    [truePosition, meas, meascov, trackpos, trackcov, trackids] = readData(scenario, dets, tracks);
    plotPlatform(platp,truePosition);
    plotDetection(detp,meas,meascov);
    plotTrack(trackp,trackpos,trackcov,trackids);
    drawnow

    % Log data
    detlog{end+1} = dets;
    tracklog{end+1} = tracks';
    infolog{end+1} = info; %#ok<*SAGROW>
end
```



The figure above shows the detection and track history for the entire simulation. From the results, the tracker created multiple tracks but some targets were only partially tracked. To further understand the results, you can use the collected information to analyze the association between the detections and the tracks.

History of Assigned Detections for trackerGNN

You parse the info output to retrieve the list of detections associated to each track. First, you obtain the IDs of all the tracks.

```
alltrackids = unique(arrayfun(@(x) x.TrackID, [tracklog{:}])))
```

```
alltrackids = 1x6 uint32 row vector
```


1 2 3 4 6 7

The tracker created six tracks during the simulation. Retrieve their assigned detections by querying the `Assignment` field of the info structure. Refer to the `trackerGNN` documentation for the definition of the `Assignment` matrix.

The function `getDetectionHistoryGNN` parses the info structure to find all the detections assigned to a given track. The function also returns the history of the tracks.

```
function [detHistory, trackHistory] = getDetectionHistoryGNN(infoLog, detLog, trackLog, tID)

detHistory = objectDetection.empty;
for i=1:numel(infoLog)

    curInfo = infoLog{i};
    existTrack = any(curInfo.TrackIDsAtStepBeginning == tID);
    if ~existTrack
        % Check if track was created
        if any(curInfo.InitiatedTrackIDs == tID)
            % Add initial detection
            detHistory(end+1) = detLog{i}{find(curInfo.InitiatedTrackIDs == tID)};
        end
        continue
    end

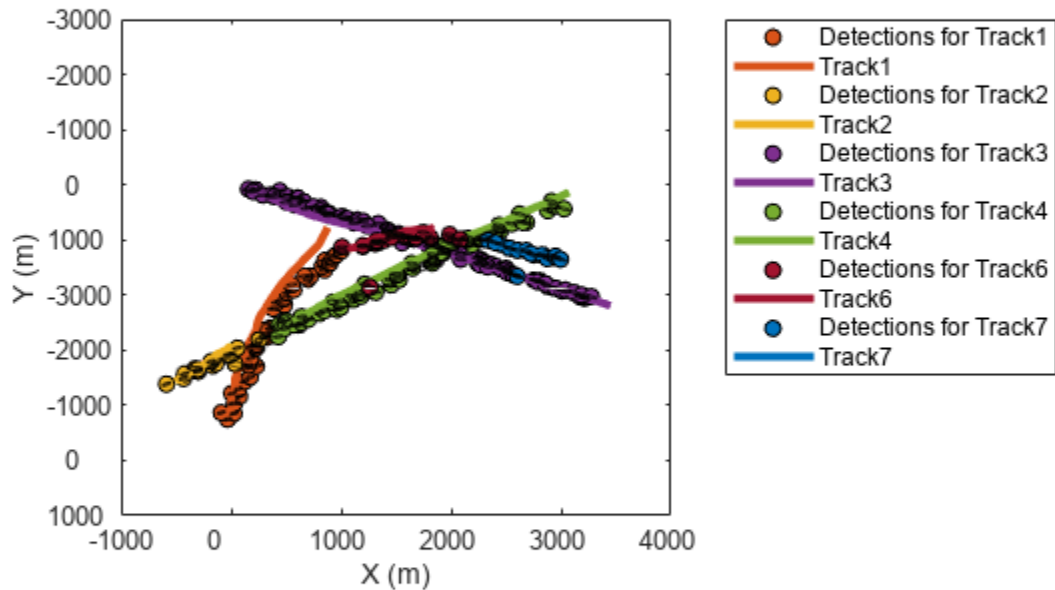
    if any(curInfo.DeletedTrackIDs == tID)
        %Track was deleted
        break
    end

    trackMatches = find(curInfo.Assignments(:,1) == tID );
    assignedDetectionIndices = curInfo.Assignments(trackMatches, 2);
    assignedDets = [detLog{i}{assignedDetectionIndices}];
    for j=1:numel(assignedDets)
        detHistory(end+1) = assignedDets(j);
    end
end

trackarray = [trackLog{:}];
alltrackids = [trackarray.TrackID];
trackHistory = trackarray(alltrackids == tID);
```

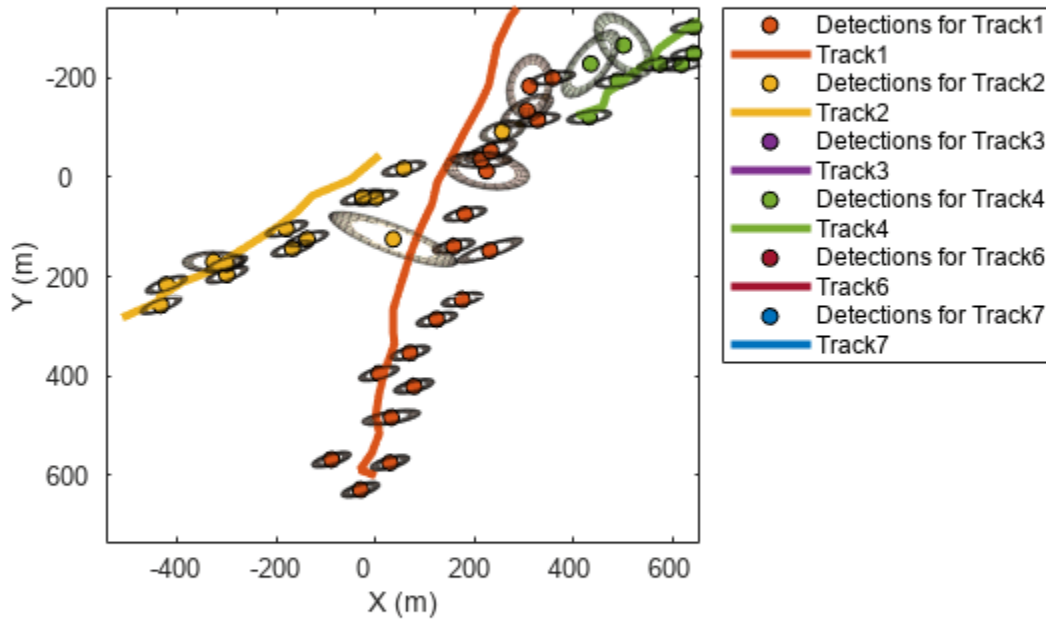
Use this function and the `plotTrackAndDets` function, attached in the example folder, to find and display each track and its assigned detections in the same color.

```
f=figure(Units="normalized",OuterPosition=[0.2 0.2 0.45 0.6]); axes(f);
for tid = alltrackids
    [detectionHistory, trackHistory] = getDetectionHistoryGNN(infoLog,detLog,trackLog,tid);
    plotTrackAndDets(f, detectionHistory, trackHistory);
end
```



Zoom in on the region where the first track breaks. Observe that the yellow track (Track2) breaks because the detections along the true target trajectory are assigned to the orange track (Track1). Additionally, the yellow track fell behind the last two assigned detections, which can be attributed to poor velocity estimates. You can observe similar track breaks in the figure. Note that tuning the tracking filter can potentially improve the capability of the tracker on maintaining tracks through the trajectory crossing.

```
xlim([-543 654]);  
ylim([-341 736]);
```



History of Assigned Detections for trackerJPDA

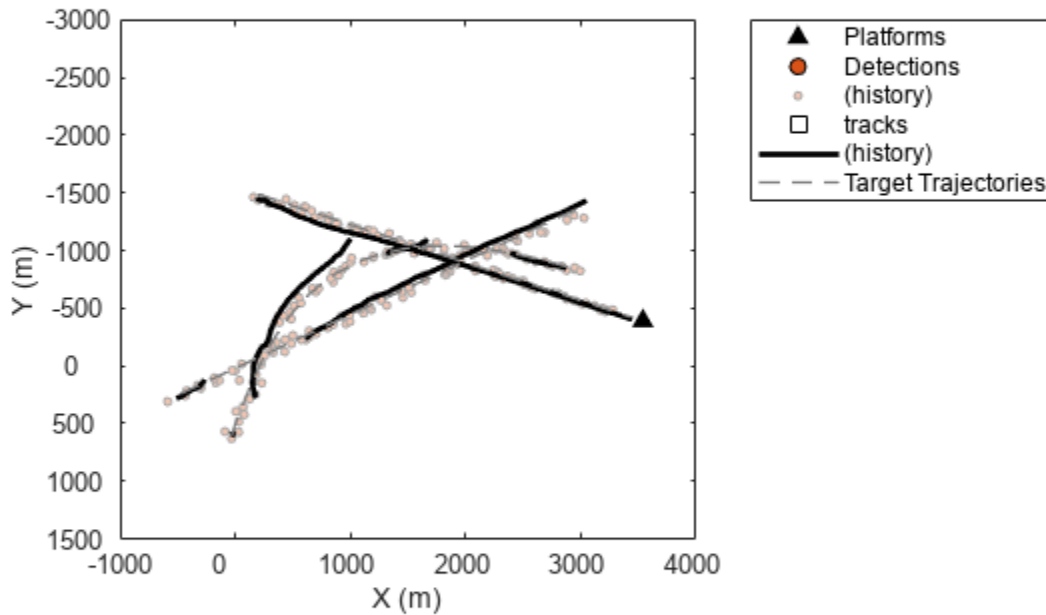
Simulate the same scenario but track targets using the `trackerJPDA` System object this time. Create the `trackerJPDA` object with the default tracking filter and the same assignment threshold as in the `trackerGNN` case.

```
tracker = trackerJPDA(AssignmentThreshold=100, ClutterDensity=5e-13, HitMissThreshold=0.1);

% Create a new plot
[platp, detp, trackp] = createPlotters(scenario);
hax = gca;

% Re-initialize logs
detlog = {};
tracklog ={};
```

```
infolog = {};  
  
restart(scenario);  
while advance(scenario)  
    % Generate sensor data  
    [dets, configs, sensorConfigPIDs] = detect(scenario);  
  
    % Update tracker  
    [tracks, ~, ~, info] = tracker(dets, scenario.SimulationTime);  
  
    % Update plots  
    [truePosition, meas, meascov, trackpos, trackcov, trackids] = readData(scenario, dets, tracks);  
    plotPlatform(platp,truePosition);  
    plotDetection(detp,meas,meascov);  
    plotTrack(trackp,trackpos,trackcov,trackids);  
    drawnow  
  
    % Log data  
    detlog{end+1} = dets;  
    tracklog{end+1} = tracks';  
    infolog{end+1} = info; %#ok<*SAGROW>  
end
```



The figure above shows the tracking results. Similar to the results obtained with `trackerGNN`, the tracker creates multiple tracks and has a few track breaks. Next, you use the analysis info of `trackerJPDA` to analyze the association history.

The analysis info of `trackerJPDA` contains the clustering results. The helper function `getDetectionHistoryJPDA` shows one approach to parse the info and retrieve the list of detections used to correct each track. Unlike `trackerGNN`, `trackerJPDA` can assign multiple detections to multiple tracks, with different probabilistic weights. In this section, use the `HitMissThreshold` property of the tracker as the probability threshold to declare a detection assigned to a track.

```
function [detHistory, trackHistory] = getDetectionHistoryJPDA(infoLog, detLog, trackLog, tID, pro
```

```
detHistory = objectDetection.empty;
for i=1:numel(infoLog)
```

```

curInfo = infoLog{i};
existTrack = any(curInfo.TrackIDsAtStepBeginning == tID);
isUnassigned = any(curInfo.UnassignedTracks == tID);
if ~existTrack || isUnassigned
    % Check if track was created
    if any(curInfo.InitializedTrackIDs == tID)
        % Add initial detection
        detHistory(end+1) = detLog{i}{find(curInfo.InitializedTrackIDs == tID)};
    end
    continue
end

if any(curInfo.DeletedTrackIDs == tID)
    % Track was deleted
    break
end

% Find the cluster with tID
hasTID = cellfun(@(x) any(x.TrackIDs == tID), curInfo.Clusters);
cluster = curInfo.Clusters{hasTID};
trackIndexInCluster = find(cluster.TrackIDs == tID );
detIndexInCluster = find(cluster.MarginalProbabilities(1:end-1,trackIndexInCluster) > probTh);
assignedDetectionIndices = cluster.DetectionIndices(detIndexInCluster);
assignedDets = [detLog{i}{assignedDetectionIndices}];
for j=1:numel(assignedDets)
    detHistory(end+1) = assignedDets(j);
end
end
trackLog = [trackLog{:}];
alltrackids = [trackLog.TrackID];
trackHistory = trackLog(alltrackids == tID);

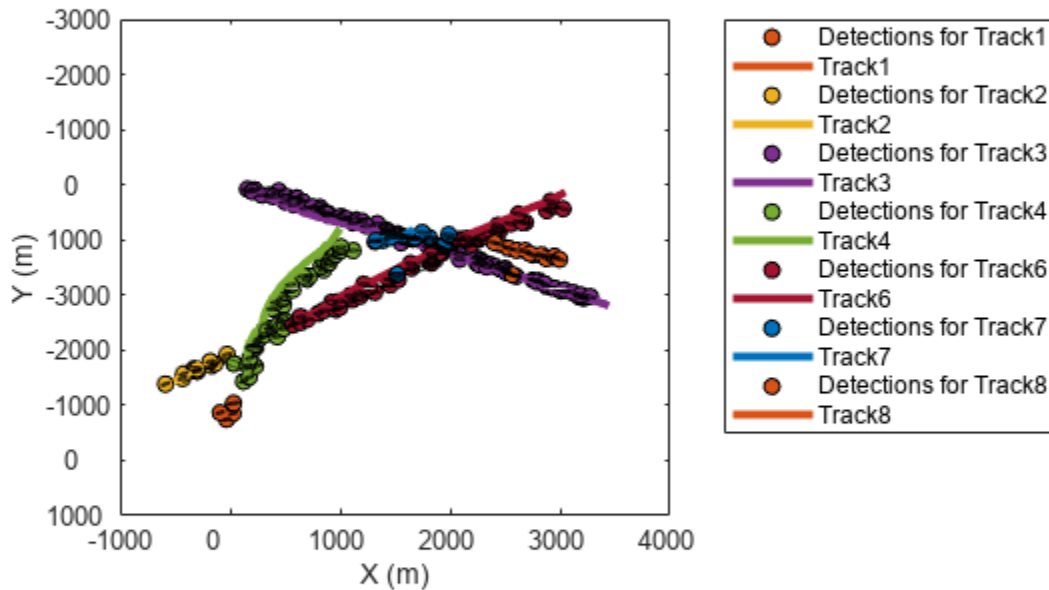
```

In a new figure, visualize each track and its history of detections.

```

f = figure(Units="normalized",OuterPosition=[0.2 0.2 0.45 0.6]); axes(f);
alltrackids = unique(arrayfun(@(x) x.TrackID, [tracklog{:}]));
for tid = alltrackids
    [detectionHistory, trackHistory] = getDetectionHistoryJPDA(infoLog,detlog,tracklog,tid, tracklog);
    plotTrackAndDets(f, detectionHistory, trackHistory);
end

```



Assignment Cost and Assignment Probabilities

The cost matrix is another useful information in the info output of the tracker. In `trackerJPDA`, each cluster report contains the matrix of assignment probabilities. Use this information to visualize each cluster and quantify the contribution of each detection to each track update. The `getClusterData` function, attached in the example folder, shows how to parse the info output to obtain the number of clusters, a list of track reports for each cluster, a list of detections for each cluster, and their respective association probabilities.

```
function [numClusters, clusterTracks, clusterDetections, clusterProbabilities] = getClusterData(
info = infolog{step};
detections = detlog{step};
numClusters = numel(info.Clusters);
```

```
% Retrieve tracks in cluster
initialTracks = tracklog{step -1};
clusterTracks = cell(1,numClusters);
clusterDetections = cell(1,numClusters);
clusterProbabilities = cell(1,numClusters);

for c=1:numClusters
    clusterTrackIDs = info.Clusters{c}.TrackIDs;
    clusterTracks{c} = initialTracks(ismember([initialTracks.TrackID],clusterTrackIDs));
    clusterDetections{c} = detections(info.Clusters{c}.DetectionIndices);
    clusterProbabilities{c} = info.Clusters{c}.MarginalProbabilities;
end
end
```

Next, use the `getClusterData` function to study the last track crossing which happens around step 59 of the simulation.

```
step = 59;
[numClusters, clusterTracks, clusterDetections, clusterProbabilities] = getClusterData(infolog, c

numClusters = 2

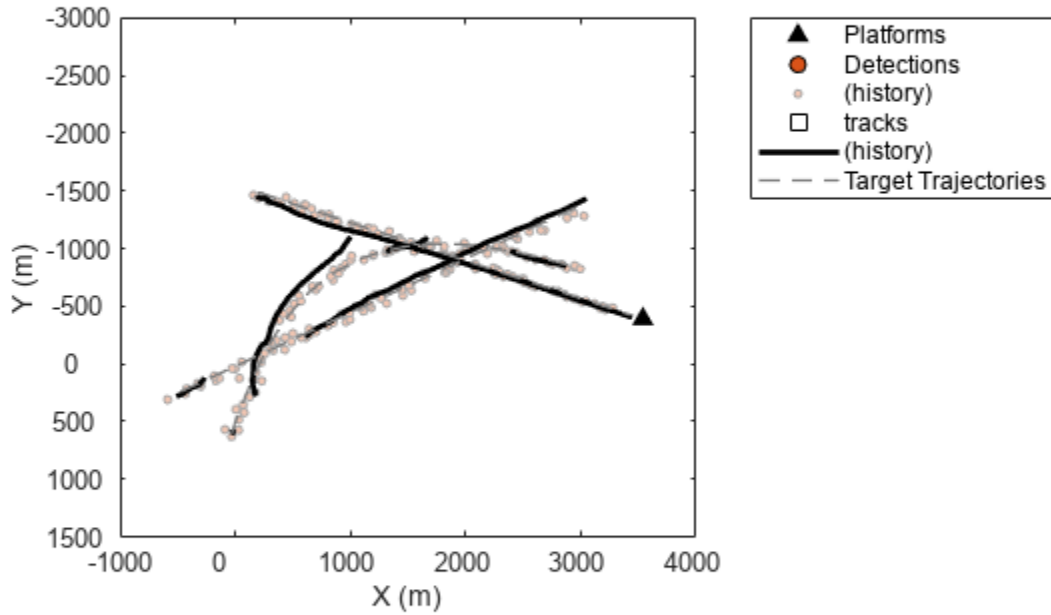
clusterTracks=1x2 cell array
    {1x1 objectTrack}    {1x2 objectTrack}

clusterDetections=1x2 cell array
    {1x1 objectDetection}    {1x2 objectDetection}

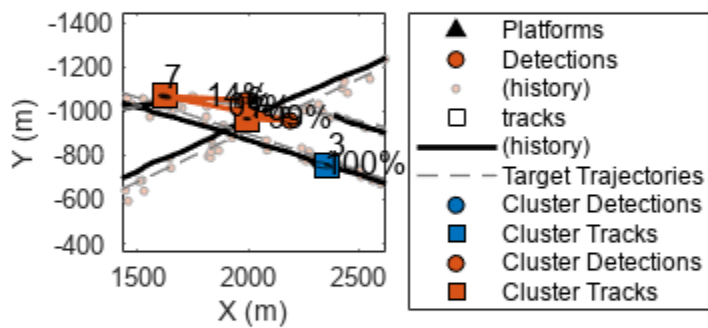
clusterProbabilities=1x2 cell array
    {2x1 double}    {3x2 double}
```

At this step, there were 2 clusters. The first cluster has 1 track and 1 detection. The second cluster has two tracks and two detections. Show the two clusters on a new figure.

```
f=figure(Units="normalized",OuterPosition=[0.2 0.2 0.45 0.6]);
copyobj(hax,f);
```

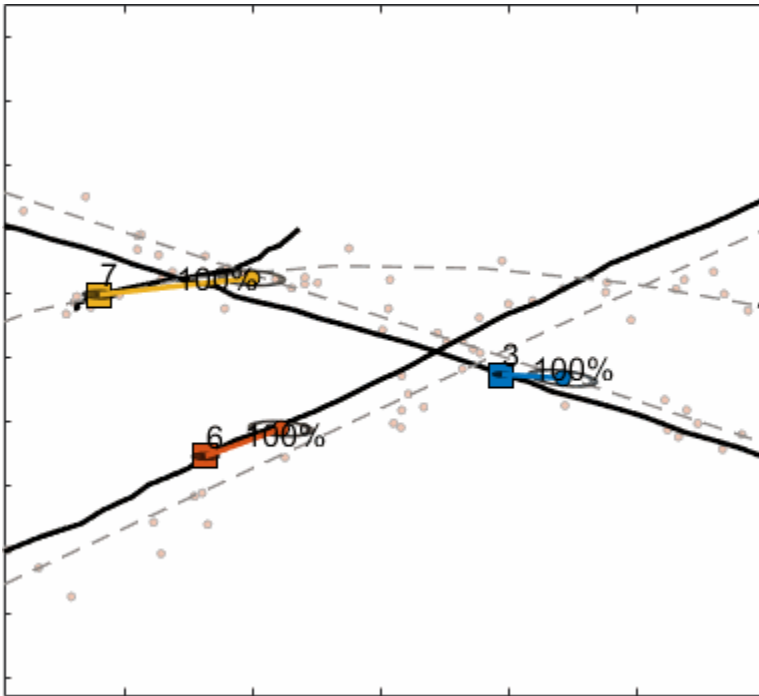



```
xlim([1428 2625]);  
ylim([-1442 -365]);  
colors = lines(numClusters);  
for c=1:numClusters  
    plotCluster(f, clusterTracks{c}, clusterDetections{c}, clusterProbabilities{c}, colors(c,:));  
end
```



Notice that Track 7 is only associated by 14% to one detection and 0 % to the second detection in its cluster. There is an 86% probability that the track was not associated to any detections.

The animation below shows the evolution of the two clusters from step 50 to 60.



Conclusion

In this example you learned different ways to utilize the analysis info output of `trackerGNN` and `trackerJPDA` System objects to analyze the data association results. In particular, you learned how to retrieve the history of detections that constitutes a track and how to inspect joint track association clusters in the `trackerJPDA` System object.

