Sensor Fusion and Tracking Toolbox™

User's Guide

MATLAB® & SIMULINK®

R2020a

MathWorks®

How to Contact MathWorks

Latest news:	www.mathworks.com
Sales and services:	www.mathworks.com/sales_and_services
User community:	www.mathworks.com/matlabcentral
Technical support:	www.mathworks.com/support/contact_us
Phone:	508-647-7000

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

Sensor Fusion and Tracking Toolbox™ User's Guide

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)

Contents

1 Tracking Scenarios

2 Radar Detections

3 Multi-Object Tracking

Tracking Scenarios

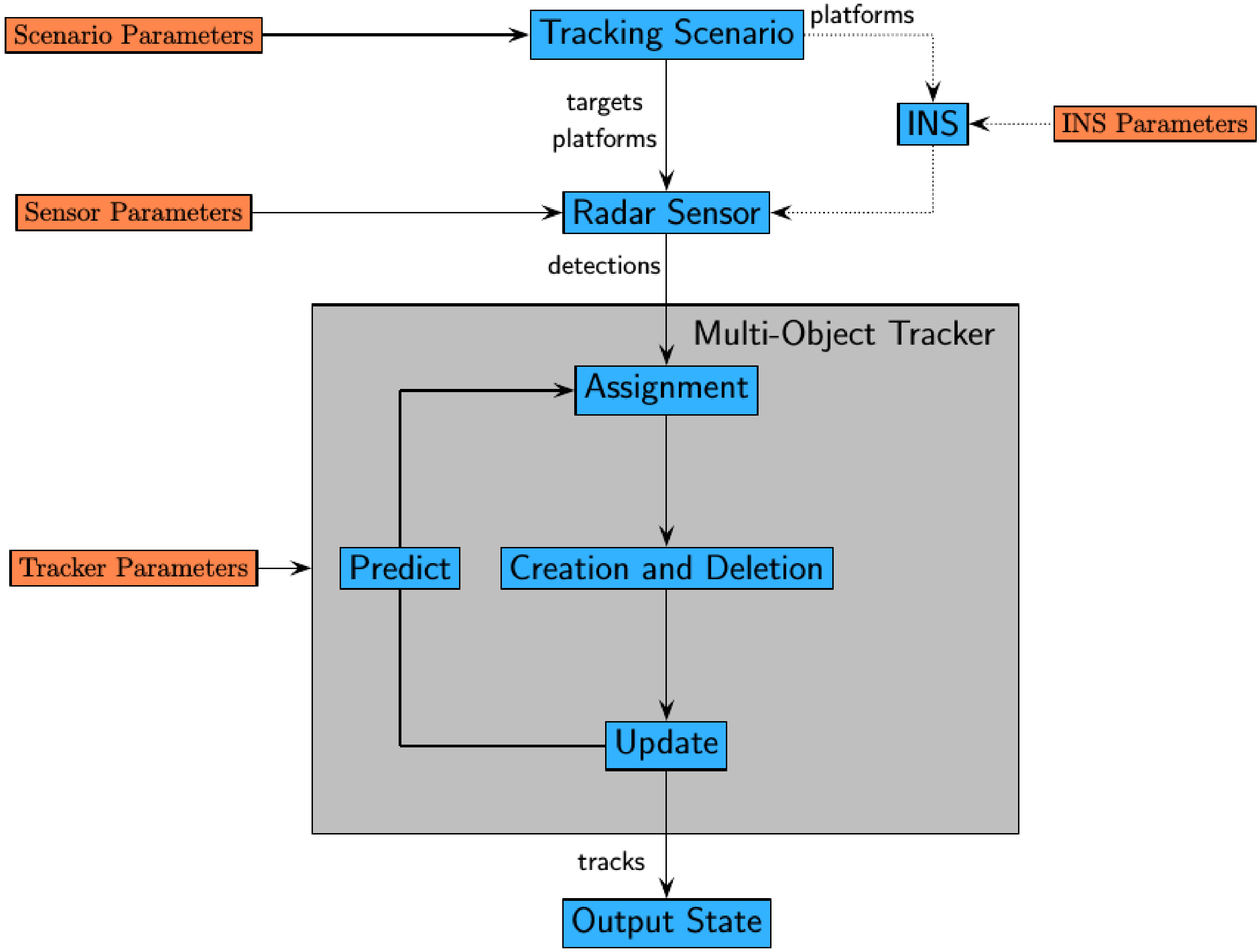
Tracking Simulation Overview

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

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

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

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


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

Creating a Tracking Scenario

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

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

Platform Properties

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

At any time during the simulation, you can retrieve the current values of platform properties using the `platformPoses` and `platformProfiles` methods of the `trackingScenario` object. Both the `platformPoses` and `platformProfiles` methods return properties of all platforms with respect to the scenario's NED frame. You can also use the `pose` method of the `Platform` to return the properties of one specific platform. In addition, the `Platform.targetPoses` method, while similar, returns properties of other platforms with respect to a specified platform.

Create Tracking Scenario with Two Platforms

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

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

Create two platforms.

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

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

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

Platform 2 follows a straight path for one second.

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

Verify the number of platforms in the scenario.

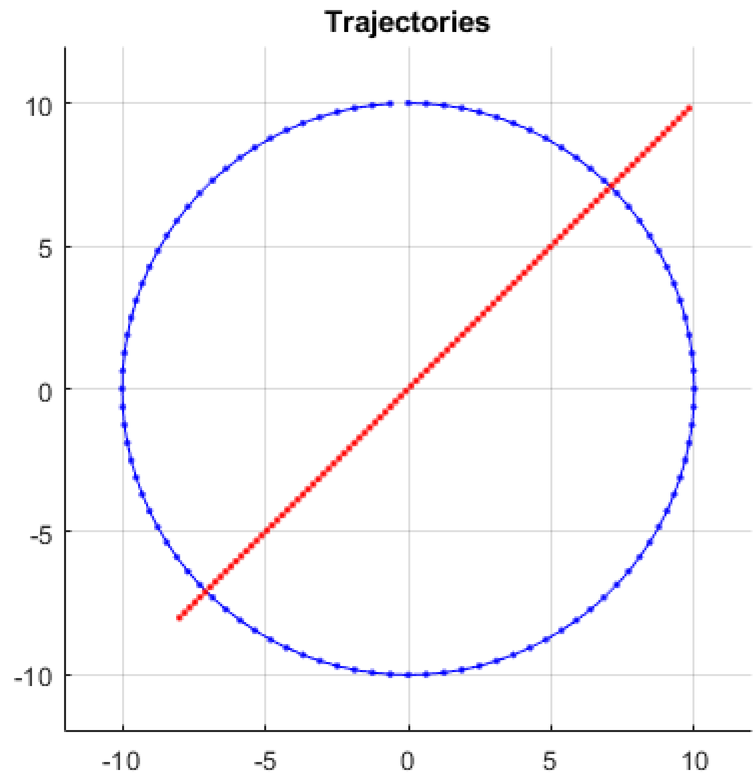
```
disp(sc.Platforms)
```

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

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

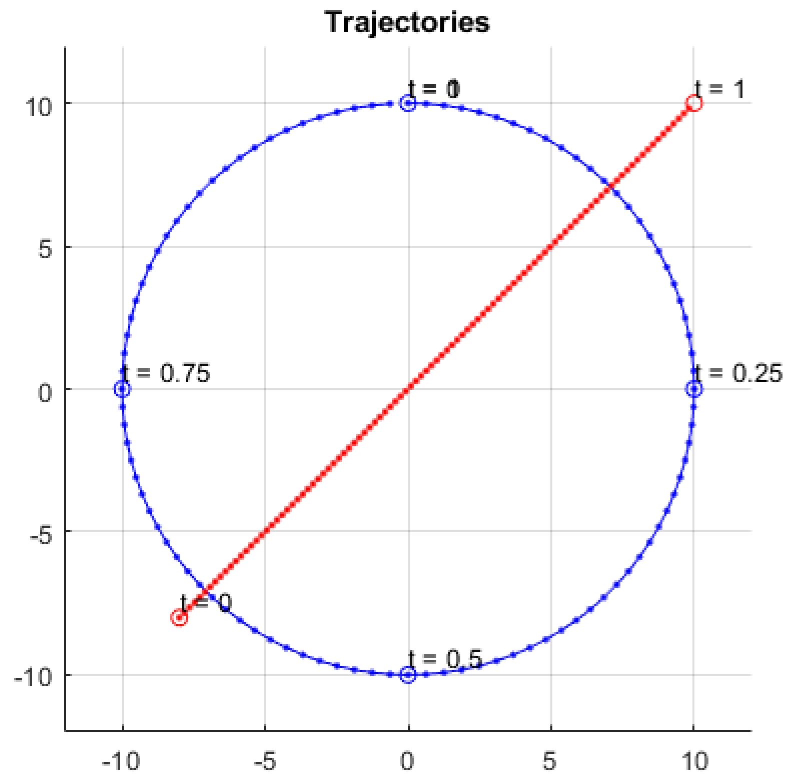
```
figure
grid
axis equal
axis([-12 12 -12 12])
line1 = animatedline('DisplayName','Trajectory 1','Color','b','Marker','.');
line2 = animatedline('DisplayName','Trajectory 2','Color','r','Marker','.');
title('Trajectories')
p1 = pose(platfm1);
p2 = pose(platfm2);
addpoints(line1,p1.Position(1),p1.Position(2));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

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



Plot the waypoints for both platforms.

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


Trajectories
t = 0
t = 1
t = 1
t = 0.75
t = 0.25
t = 0
t = 0.5
10
5
0
-5
-10
-10
-5
0
5
10

Radar Detections

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

Simulate Radar Detections

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

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

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

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

Create Radar Sensor

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

```
radar1 = monostaticRadarSensor( ...
    'UpdateRate',updaterate, ...          % Hz
    'ReferenceRange', 111.0e3, ...        % m
    'ReferenceRCS', 0.0, ...              % dBsm
    'HasMechanicalScan',true, ...
    'MaxMechanicalScanRate',scanrate, ... % deg/s
    'HasElectronicScan',false, ...
    'FieldOfView',fov, ...                % [az;el] deg
    'HasElevation',false, ...
    'HasRangeRate',false, ...
    'AzimuthResolution',1.4, ...          % deg
    'RangeResolution', 135.0)                % m
dets = radar1(targets,simtime);
```

Convenience Syntaxes

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

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

- `sensor = monostaticRadarSensor('Sector')` creates a `monostaticRadarSensor` object that mechanically scans a 90° azimuth sector. Setting `HasElevation` to `true`, points the radar antenna towards the center of the elevation field of view. You can change the `ScanMode` to `'Electronic'` to electronically scan the same azimuth sector. In this case, the antenna is not mechanically tilted in an electronic sector scan. Instead, beams are stacked electronically to process the entire elevation spanned by the scan limits in a single dwell.
- `sensor = monostaticRadarSensor('Raster')` returns a `monostaticRadarSensor` object that mechanically scans a raster pattern spanning 90° in azimuth and 10° in elevation upwards from the horizon. You can change the `ScanMode` property to `'Electronic'` to perform an electronic raster scan in the same volume.
- `sensor = monostaticRadarSensor('No scanning')` returns a `monostaticRadarSensor` object that stares along the radar antenna boresight direction. No mechanical or electronic scanning is performed.

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

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

Radar Sensor Parameters

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

```
help monostaticRadarSensor
```

at the command line.

Sensor location parameters.

Sensor Location

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

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

Sensitivity parameters.

Sensitivity Parameters

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

Sensor resolution and bias parameters.

Resolution Parameters

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

Enabling parameters.

Enabling Parameters

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

Scan parameters.

Scan Parameters

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

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

Range and range rate parameters.

Range and Range Rate Parameters

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

Detector Input

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

The structure contains these fields.

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

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

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

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

Target Poses

platformID
ClassID
Position
Velocity
Yaw
Pitch
Roll
AngularVelocity

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

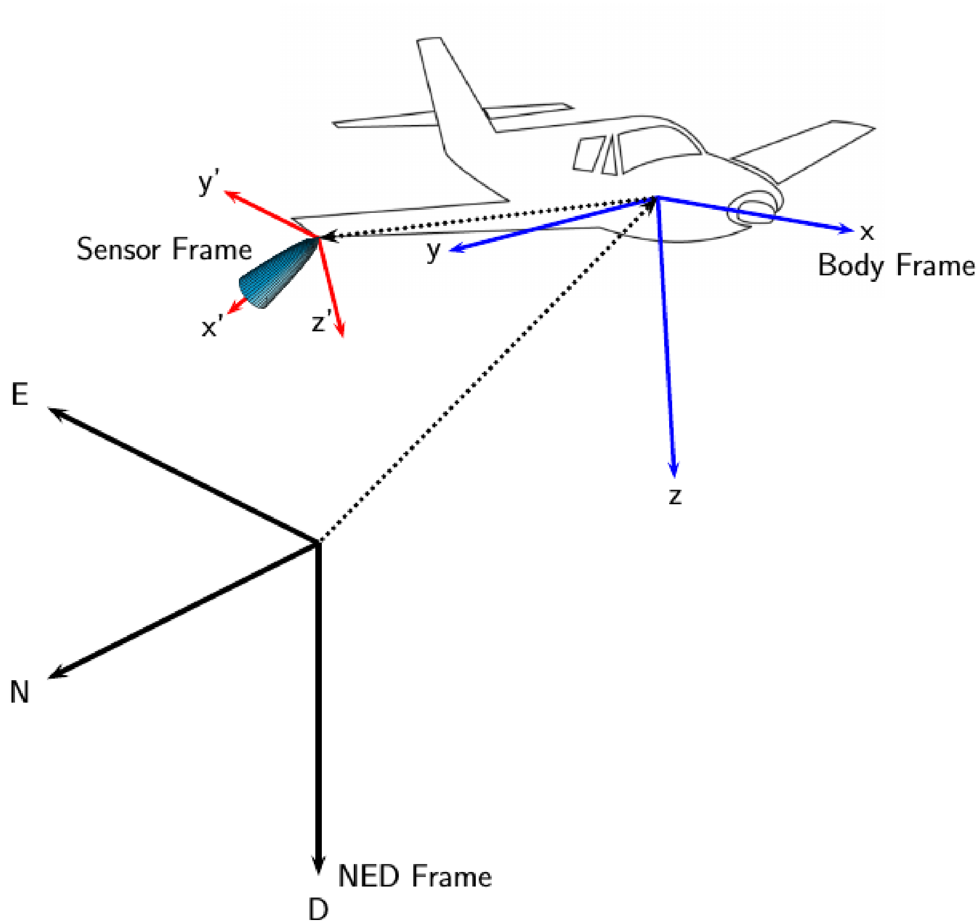
Platform Profiles

PlatformID
ClassID
RCSPattern
RCSAzimuthAngles
RCSElevationAngles

Radar Sensor Coordinate Systems

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

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



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

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

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

INS

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

Detections

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

objectDetection Structure

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

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

Measurement Coordinates

<table>
<tr><th>DetectionCoordinates</th><th>Measurement and Measurement Noise Coordinates</th></tr>
<tr><td>'Scenario'</td><td rowspan="3">Coordinate Dependence on HasRangeRate
<table>
<tr><th>HasRangeRate</th><th>Coordinates</th></tr>
<tr><td>true</td><td>[x;y;z;vx;vy;vz]</td></tr>
<tr><td>false</td><td>[x;y;z]</td></tr>
</table></td></tr>
<tr><td>'Body'</td></tr>
<tr><td>'Sensor rectangular'</td></tr>
<tr><td>'Sensor spherical'</td><td>Coordinate Dependence on HasRangeRate and HasElevation
<table>
<tr><th>HasRangeRate</th><th>HasElevation</th><th>Coordinates</th></tr>
<tr><td>true</td><td>true</td><td>[az;el;rng;rr]</td></tr>
<tr><td>true</td><td>false</td><td>[az;rng;rr]</td></tr>
<tr><td>false</td><td>true</td><td>[az;el;rng]</td></tr>
<tr><td>false</td><td>false</td><td>[az;rng]</td></tr>
</table></td></tr>
</table>

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

Each struct takes the form:

MeasurementParameters

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

ObjectAttributes

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

Multi-Object Tracking

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

Tracking and Tracking Filters

Multi-Object Tracking

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

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

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

- Decide which tracker to use.

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

  See the “Tracking Closely Spaced Targets Under Ambiguity” 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” and “Tracking Maneuvering Targets” 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” example.

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

Multi-Object Tracker Properties

trackerGNN Properties

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

trackerGNN Properties

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

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

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

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

trackerGNN Input

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

trackerGNN Input

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

trackerGNN Output

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

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

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

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

The fields contained in the `struct` are:

trackerGNN Output `struct`

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

Introduction to Estimation Filters

Background

Estimation Systems

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

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

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

$$y_k = h(x_k)$$

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

Noise Distribution

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

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

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

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

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

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

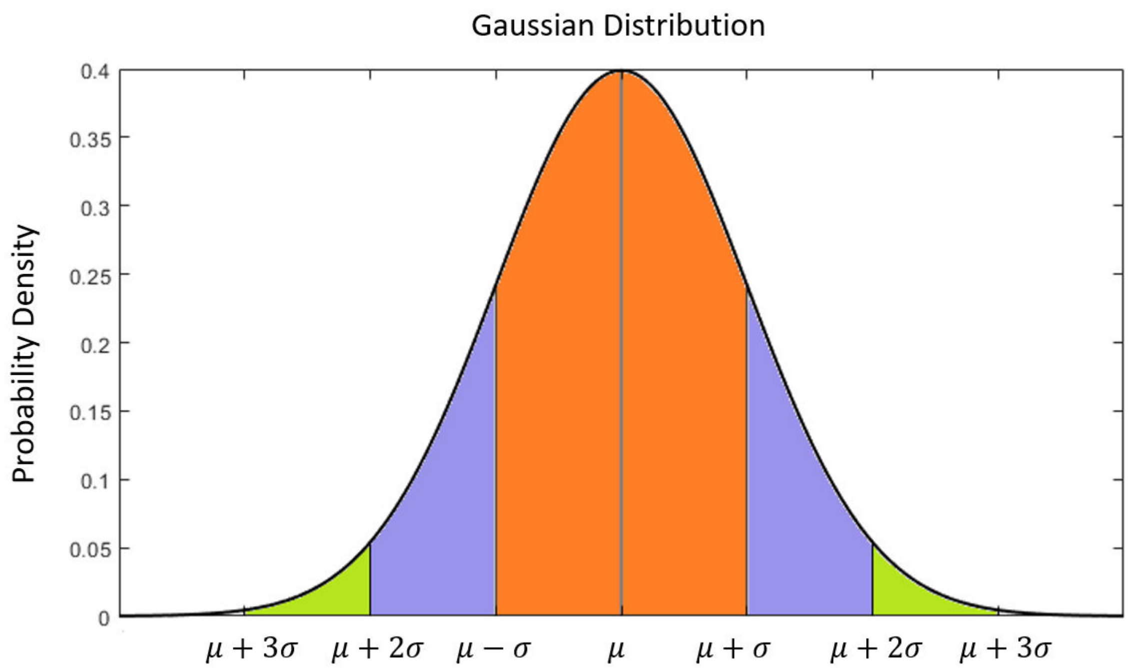
$$\mu = E[x]$$

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

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

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

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



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

Filter Design

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

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

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

Prediction

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

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

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

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

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

Correction

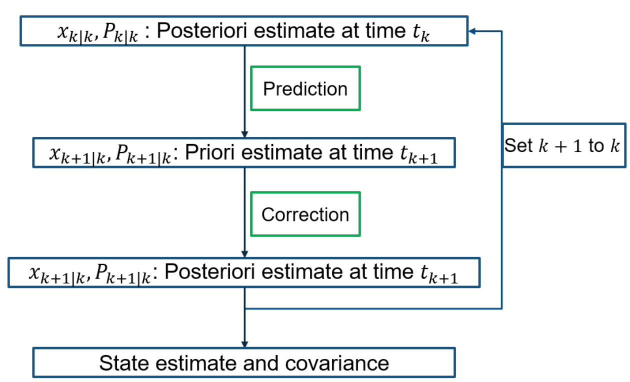
In the correction step, the filter uses measurements to correct the state estimate through measurement feedback. Basically, the difference between the true measurement and the predicted measurement is added to the state estimate after it is multiplied by a feedback gain matrix. For example, in an extended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

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

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

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

General Algorithms for Kalman Filters



Estimation Filters in Sensor Fusion and Tracking Toolbox

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

Kalman Filter

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

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

$$y_k = H_k x_k + v_k$$

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

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

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

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

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

The correction equations for the measurement update are:

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

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

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

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

Since the dimension of the inverted matrix is equal to that of the estimated state, this calculation requires some computation efforts for a high dimensional system. For more details, see “Linear Kalman Filters” on page 3-20.

Alpha-Beta Filter

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

Extended Kalman Filter

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

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

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

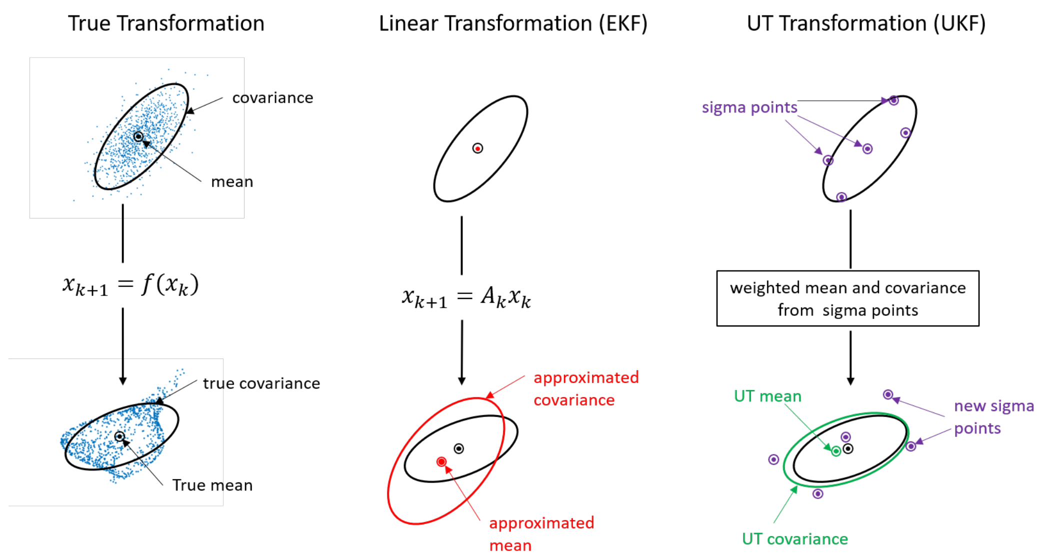
If the true dynamics of the estimation system are close to the linearized dynamics, then using this linear approximation does not yield significant errors for a short period of time. For this reason, an EKF can produce relatively accurate state estimates for a mildly nonlinear estimation system with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

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

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

Unscented Kalman Filter

The unscented Kalman filter (`trackingUKF`) uses an unscented transformation (UT) to approximately propagate the covariance distribution for a nonlinear model. The UT approach samples the covariance Gaussian distribution at the current time, propagates the sample points (called sigma points) using the nonlinear model, and approximates the resulting covariance distribution assumed to be Gaussian by evaluating these propagated sigma points. The figure illustrates the difference between the actual propagation, the linearized propagation, and the UT propagation of the uncertainty covariance.



Compared to the linearization approach taken by an EKF, the UT approach results in more accurate propagation of covariance and leads to more accurate state estimation, especially for highly nonlinear systems. UKF does not require the derivation and calculation of Jacobian matrices. However, UKF requires the propagation of $2n+1$ sigma points through the nonlinear model, where n is the dimension of the estimated state. This can be computationally expensive for high dimensional systems.

Cubature Kalman Filter

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

Gaussian-Sum Filter

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

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

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

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

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" for an example.

Interactive Multiple Model Filter

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

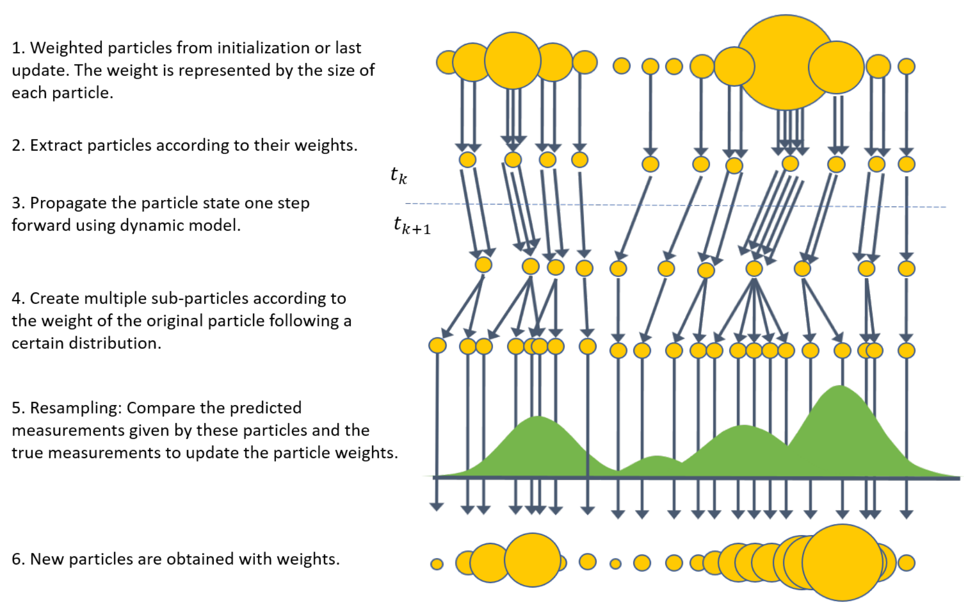
$$x_{k+1}^{i} = f_i(x_k^i, w_k^i)$$

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

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

Linear Kalman Filters

In this section...

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

State Equations

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

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

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

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

If you define the state as

$$x_1 = x$$

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

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

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

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

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

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

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

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

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

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

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

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

The state equation can be generalized to

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

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

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

Measurement Models

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

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

$$z_k = H_k x_k + w_k$$

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

Linear Kalman Filter Equations

Without noise, the dynamic equations are

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

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

$$z_k = H_k x_k$$

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

Filter Loop

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

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

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

   Propagate the covariance matrix as well.

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

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

   Then predict the measurement at the updated time.

   $$z_{k+1|k} = H_{k+1} x_{k+1|k}$$
2. Use the difference between the actual measurement and predicted measurement to correct the state at the updated time. The correction requires computing the Kalman gain. To do this, first compute the measurement prediction covariance (innovation)

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

   Then the Kalman gain is

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

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

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

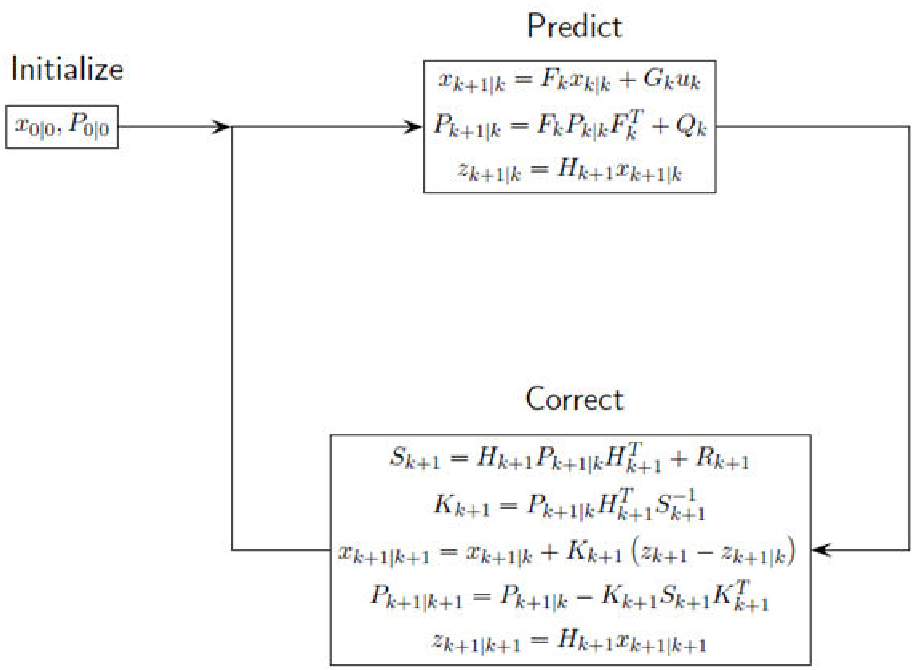
   where K_{k+1} is the Kalman gain. The corrected state is often called the *a posteriori* estimate of the state because it is derived after the measurement is included.

   Correct the state covariance matrix

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

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

This figure summarizes the Kalman loop operations.



Constant Velocity Model

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

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

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

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

Constant Acceleration Model

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

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

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

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

See Also

Objects
`trackingKF`

Extended Kalman Filters

In this section...

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

State Update Model

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

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

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

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

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

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

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

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

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

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

Measurement Model

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

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

The Jacobian of the measurement with respect to the state is

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

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

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

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

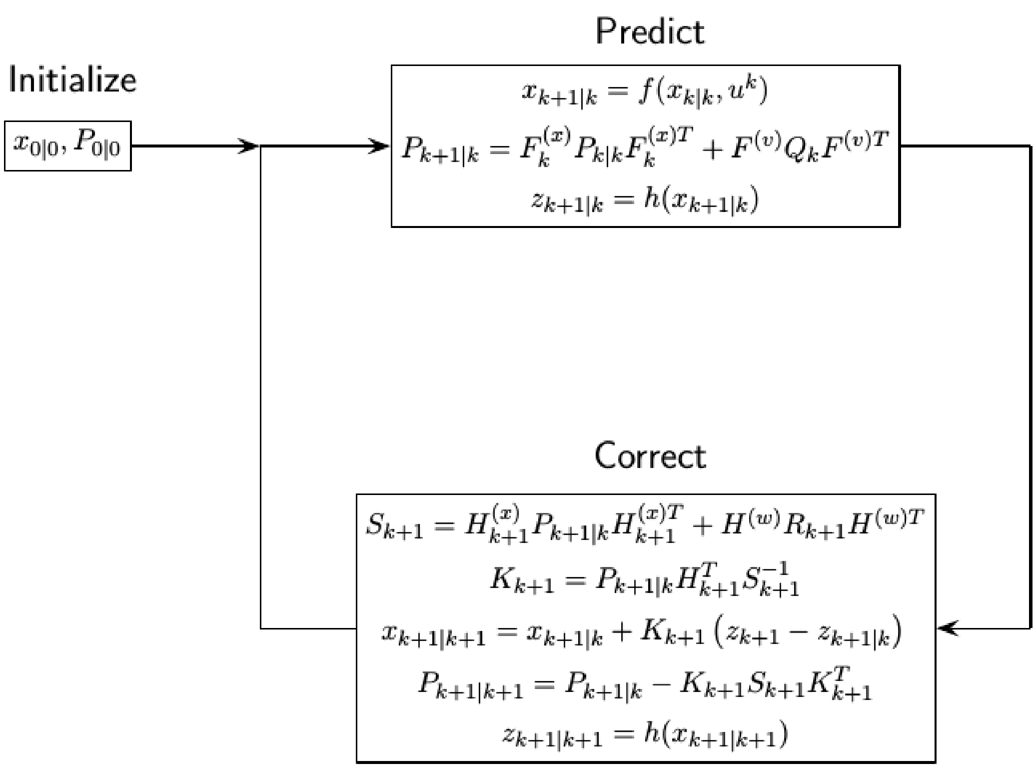
$$z_k = h(x_k, t) + v_k$$

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

Extended Kalman Filter Loop

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

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



Predefined Extended Kalman Filter Functions

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

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

See Also

Objects
`trackingEKF`

Introduction to Multiple Target Tracking

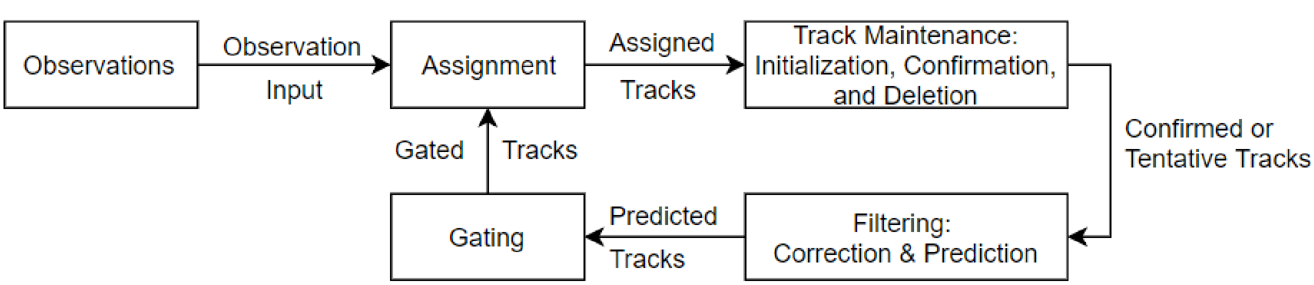
Background

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

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

- Target or detection distribution — If targets are sparsely distributed, then associating a target to its corresponding detection is relatively easy. However, if targets or detections are densely distributed, the assignment becomes ambiguous because assigning a target to a detection or a nearby detection rarely makes any differences on the cost.
- Probability of detection (P_d) of the sensor — P_d describes the probability that a target is detected by the sensor if the target is within the field of view of the sensor. If the P_d of a sensor is small, then the true target might not generate any detection during a sensor scan. As a result, the track represented by the true target may steal detections from other tracks.
- Sensor resolution — Sensor resolution determines the sensor's ability to distinguish between detections from two targets. If the sensor resolution is low, then two targets in proximity might only give rise to one detection. This violates the common assumption that each detection can only be assigned to one track and results in unresolvable assignment conflicts between tracks.
- Clutter or false alarm rate of the sensor — False alarms introduce additional possible assignments and therefore increase the complexity of data assignment.
- The number of targets and detections — The number of possible assignments increases exponentially as the number of targets and detections increases. Therefore, obtaining an optimal assignment requires more computations.

Elements of an MTT System



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

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

1. The internal filter (such as a Kalman filter) predicts the confirmed or tentative tracks from the previous step to the current step.
2. The tracker uses the predicted estimate and covariance to form a validation gate around the predicted track.
3. The detections falling within the gate of a track are considered as candidates for assignment to the track.
4. An assignment algorithm (based on the specific tracker, such as GNN or TOMHT) determines the track-to-detection association.
5. Based on the assignment, the tracker executes track maintenance, including initialization, confirmation, and deletion:
   - Unassigned observations can initiate new tentative tracks.
   - A tentative track becomes confirmed if the quality of the track satisfies the confirmation criteria.
   - Low-quality tracks are deleted based on the deletion criteria.
6. The new track set (tentative and confirmed) is predicted to the next scan step to form validation gates.

Detections

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

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

High-resolution sensors may generate multiple detections per target, which requires partitioning the detections into one representative detection before feeding to assignment-based trackers (such as `trackerGNN`, `trackerJPDA`, and `trackerTOMHT`). See “Extended Object Tracking and Performance Metrics Evaluation” for more details.

Gating and Assignment

For details about gating and assignment, see “Introduction to Assignment Methods in Tracking Systems” on page 3-33, which provides a comprehensive introduction of assignment methods. This section only covers the basics of gating and assignment used in the three assignment-based trackers, `trackerGNN`, `trackerJPDA`, and `trackerTOMHT`.

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

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

- `trackerGNN` — Global nearest data association. Based on likelihood theory, the goal of the GNN method is to minimize an overall distance function that considers all track-to-detection assignments.
- `trackerJPDA` — Joint probability data association. The JPDA method applies a soft assignment, such that detections within the validation gate of a track can all make weighted contributions to the track based on their probability of association.
- `trackerTOMHT` — Track-oriented multiple hypothesis tracking. Unlike GNN and JPDA, MHT is a deferred decision approach, which allows difficult data association situations to be postponed until more information is received.

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

- The GNN algorithm is the simplest to employ. It has low computational cost and can result in adequate performance for tracking sparsely distributed targets.
- The JPDA algorithm, which requires more computational cost, is also applicable for widely spaced targets. It usually performs better in a clutter environment than GNN.
- The TOMHT tracker, which requires heavily on computational resources, normally results in the best performance among all the three trackers, especially for densely distributed targets.

For more details, see the “Tracking Closely Spaced Targets Under Ambiguity” 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” 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 3-9.

Tracking Metrics

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

- You can use `trackAssignmentMetrics` to evaluate the performance of track assignment and maintenance. `trackAssignmentMetrics` provides indexes such as number of the track swaps, number of divergence steps, and number of redundant assignments.
- You can use `trackErrorMetrics` to evaluate the accuracy of tracking. `trackErrorMetrics` provides multiple root mean square (RMS) error values, which numerically illustrate the accuracy performance of the tracker.
- You can use `trackOSPAMetric` to compute the optimal subpattern assignment metric. `trackErrorMetrics` provides three scalar error components — localization error, labelling error, and cardinality error to evaluate tracking performance.

Non-Assignment-Based Trackers

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

See Also

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

References

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

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

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

Introduction to Assignment Methods in Tracking Systems

Background

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

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

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

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

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

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

2-D Assignment in Multiple Target Tracking

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

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

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

- Sensor resolution — Sensor resolution determines the sensor's ability to distinguish the detections from two targets. If the sensor resolution is low, then two targets in proximity may only give rise to one detection. This violates the common assumption that each detection can only be assigned to one track and results in unresolvable assignment conflicts between tracks.
- Clutter or false alarm rate of the sensor — False alarms introduce additional possible assignments and therefore increase the complexity of data assignment.

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

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

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

Gating

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

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

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

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

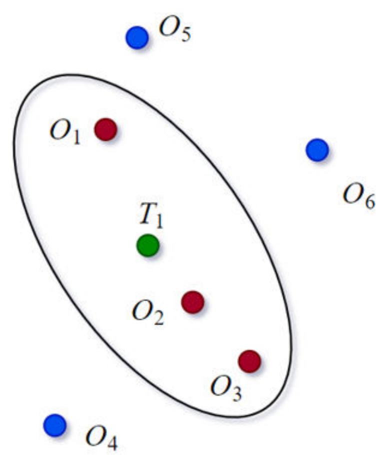


Figure 1. A gating example

Global Nearest Neighbor (GNN) Method

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

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

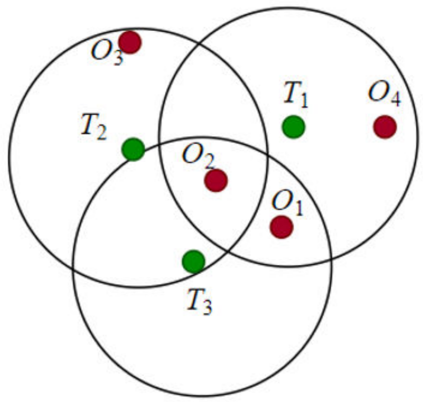


Figure 2. GNN with association conflicts

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

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

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

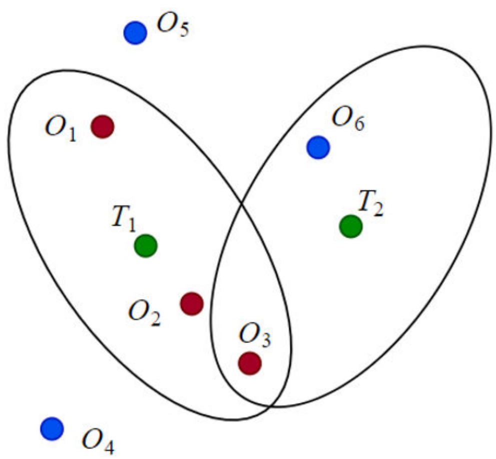
While the GNN method makes a rigid assignment of a detection to a track, the JPDA method applies a soft assignment so that detections within the validation gate of a track can all make weighted contributions to the track based on their probability of association.

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:

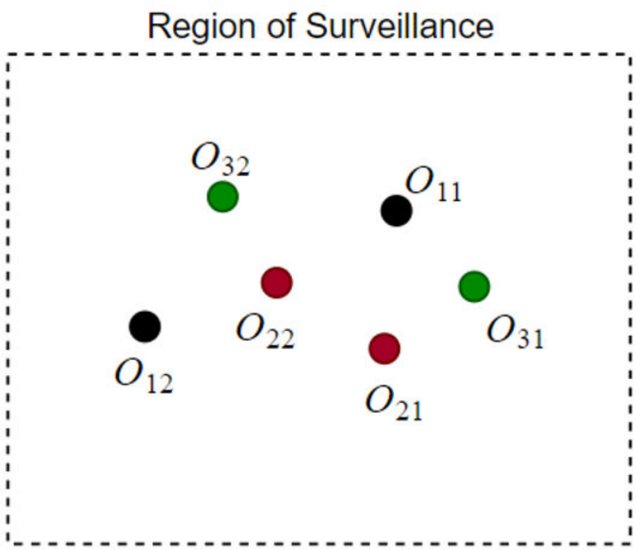
- Assign no detection to T_1 resulting in hypothesis T_{10}
- Assign O_1 to T_1 resulting in hypothesis T_{11}
- Assign O_2 to T_1 resulting in hypothesis T_{12}
- Assign O_3 to T_1 resulting in hypothesis T_{13}

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, $i = 1, 2, \ldots, M_1$. Use z_{ijk} to represent the track formation hypothesis of O_{1i}, O_{2j}, and O_{3k}. If the hypothesis is valid, then $z_{ijk} = 1$; otherwise, $z_{ijk} = 0$. As mentioned, c_{ijk} is used to represent the cost of z_{ijk} association. c_{ijk} is 0 for false alarms and negative for possible associations. The S-D optimization problem can be formulated as:

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

subject to three constraints:

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

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

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

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

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

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

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

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

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

See “Tracking Using Distributed Synchronous Passive Sensors” for the application of S-D assignment in static detection fusion.

See Also

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

References

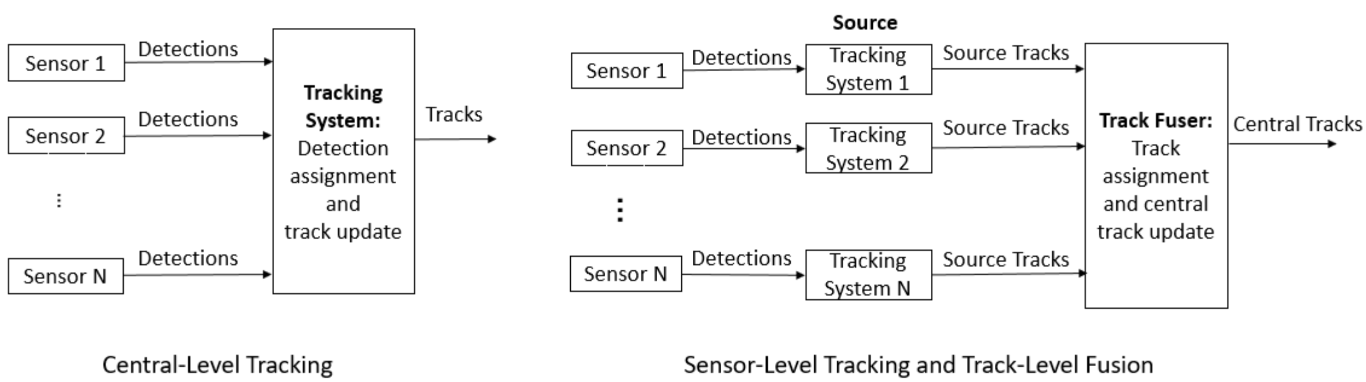
[1] Blackman, S., and R. Popoli. *Design and Analysis of Modern Tracking Systems.* Artech House Radar Library, Boston, 1999.

[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

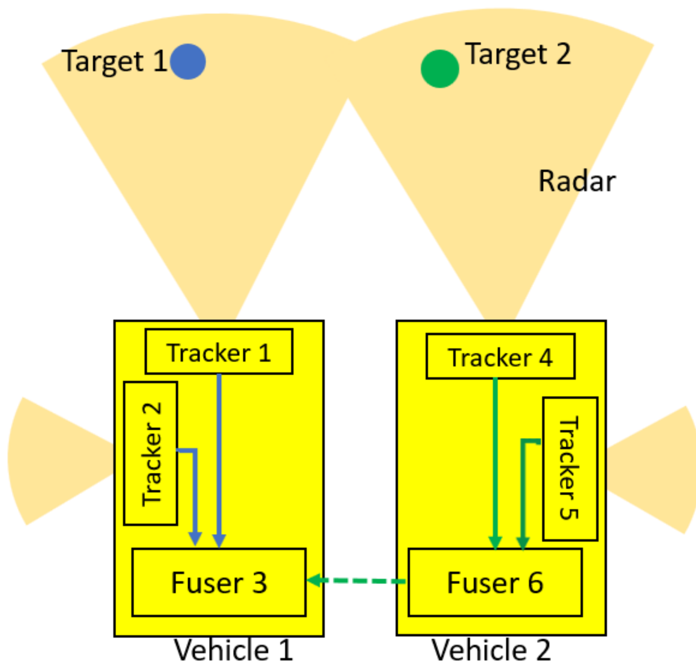
You can use the `trackFuser` in Sensor Fusion and Tracking Toolbox for the purpose of track-to-track fusion. The `trackFuser` System object™ provides two algorithms to combine source tracks considering the correction effects between different tracks. You can choose the algorithm by specifying the `StateFusion`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

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

References

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

Multiple Extended Object Tracking

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

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

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

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

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

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

- One detection per object.

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

  - A point detection per object.

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

    The Point Object Tracker approach shown in the first part of “Extended Object Tracking and Performance Metrics Evaluation” 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” 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 and Performance Metrics Evaluation” example, the GGIW-PHD Extended Object Tracker approach represents vehicle shapes as ellipses, and the Prototype Extended Object Tracker approach represents vehicle shapes as rectangles.

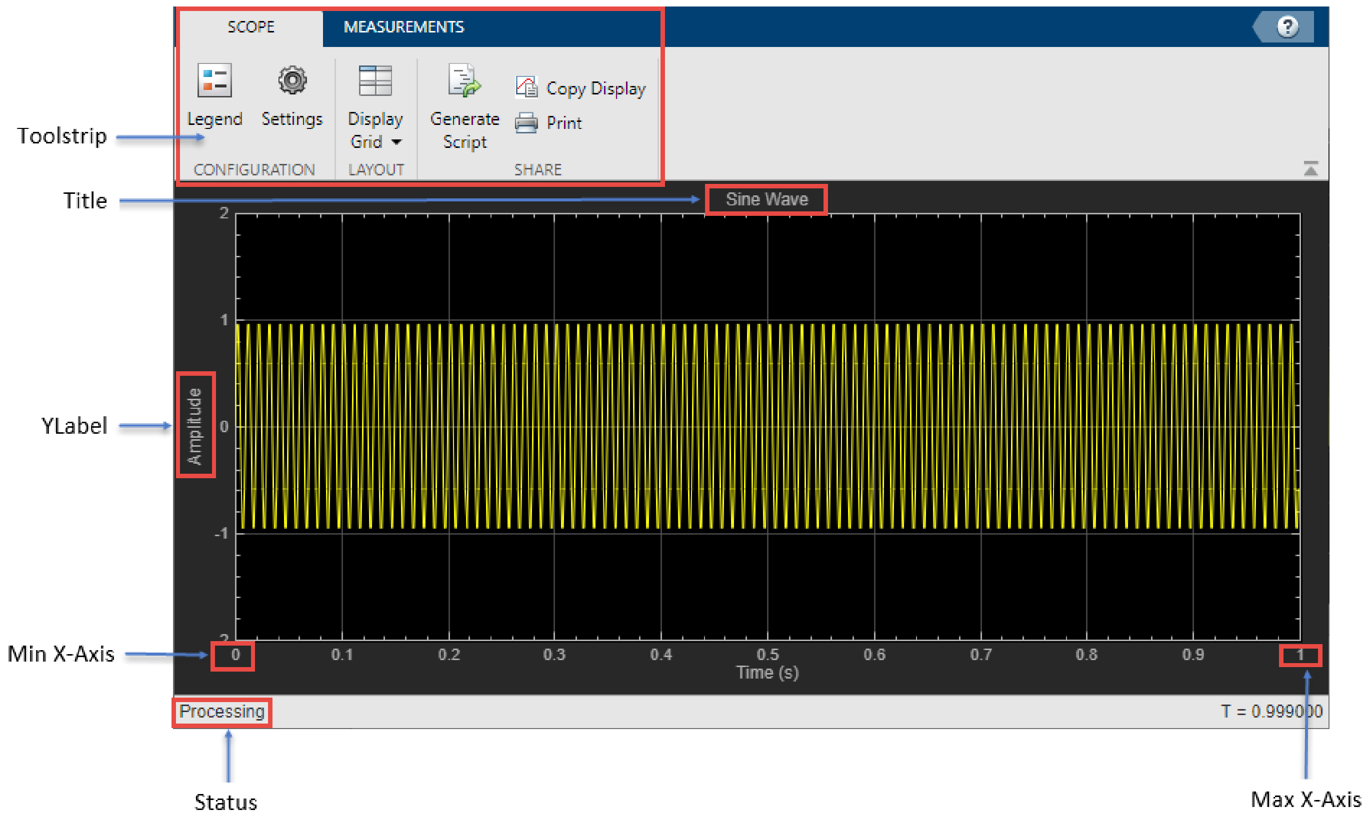
  In the “Marine Surveillance Using a PHD Tracker” example, the GGIW-PHD tracker models the ship shapes as ellipses.

Configure Time Scope MATLAB Object

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

Signal Display

This figure highlights the important aspects of the Time Scope window in MATLAB.



- **Min X-Axis** — Time scope sets the minimum *x*-axis limit using the value of the TimeDisplayOffset property. To change the **Time Offset** from the Time Scope window, click **Settings** () on the **Scope** tab. Under **Data and Axes**, set the **Time Offset**.
- **Max X-Axis** — Time scope sets the maximum *x*-axis limit by summing the value of the **Time Offset** property with the span of the *x*-axis values. If **Time Span** is set to `Auto`, the span of *x*-axis is `10`/SampleRate.

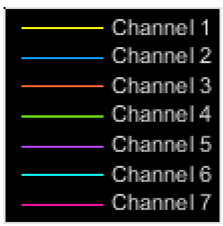
  The values on the *x*-axis of the scope display remain the same throughout the simulation.
- **Status** — Provides the current status of the plot. The status can be:
  - `Processing` — Occurs after you run the `step` function and before you run the `release` function.

   - `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** — The **Scope** tab contains buttons and settings to customize and share the time scope. The **Measurements** tab contains buttons and settings to turn on different measurement tools.

  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.

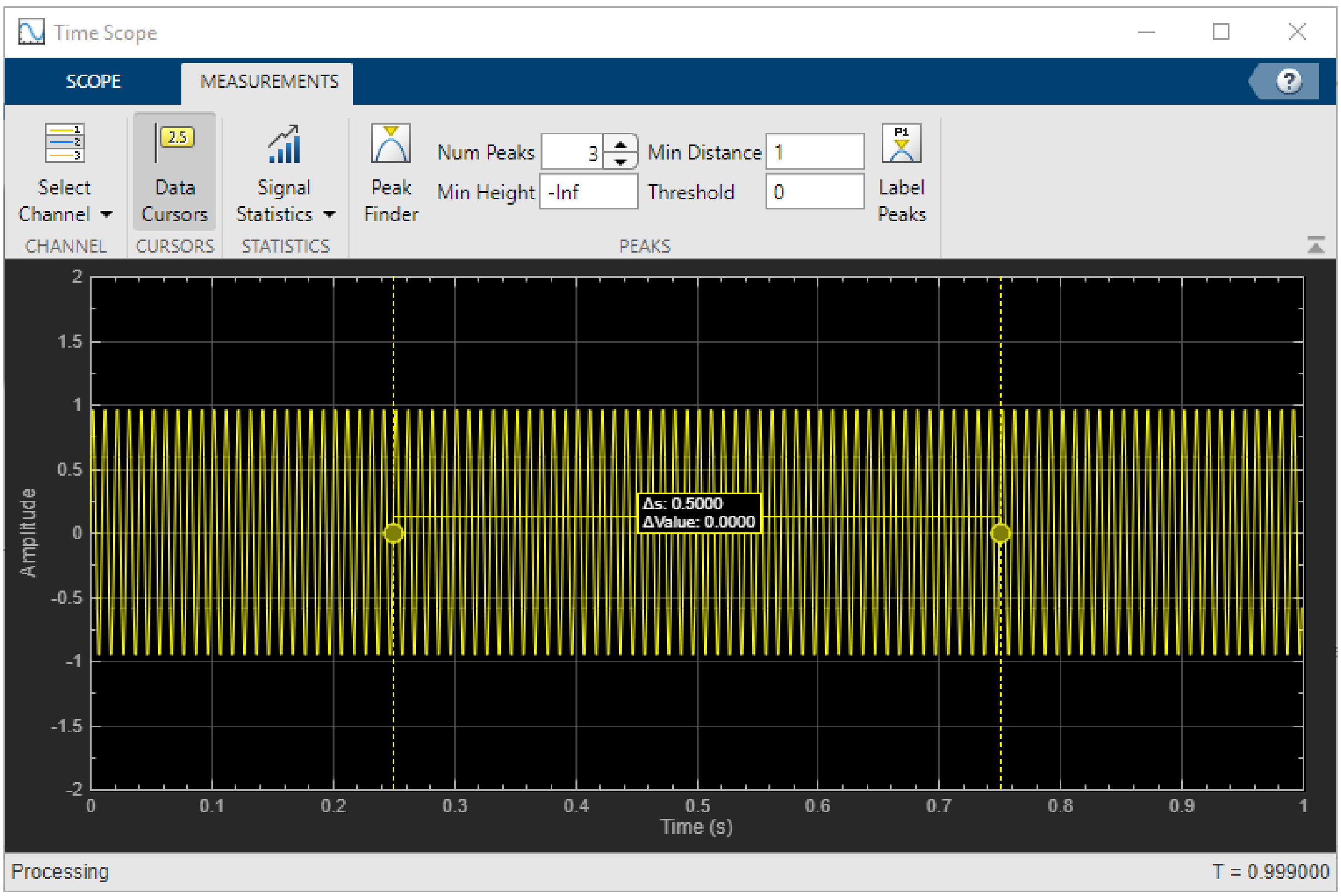
On the **Scope** tab, the **Layout** section allows you to modify the scope layout dimensions.

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

Use timescope Measurements

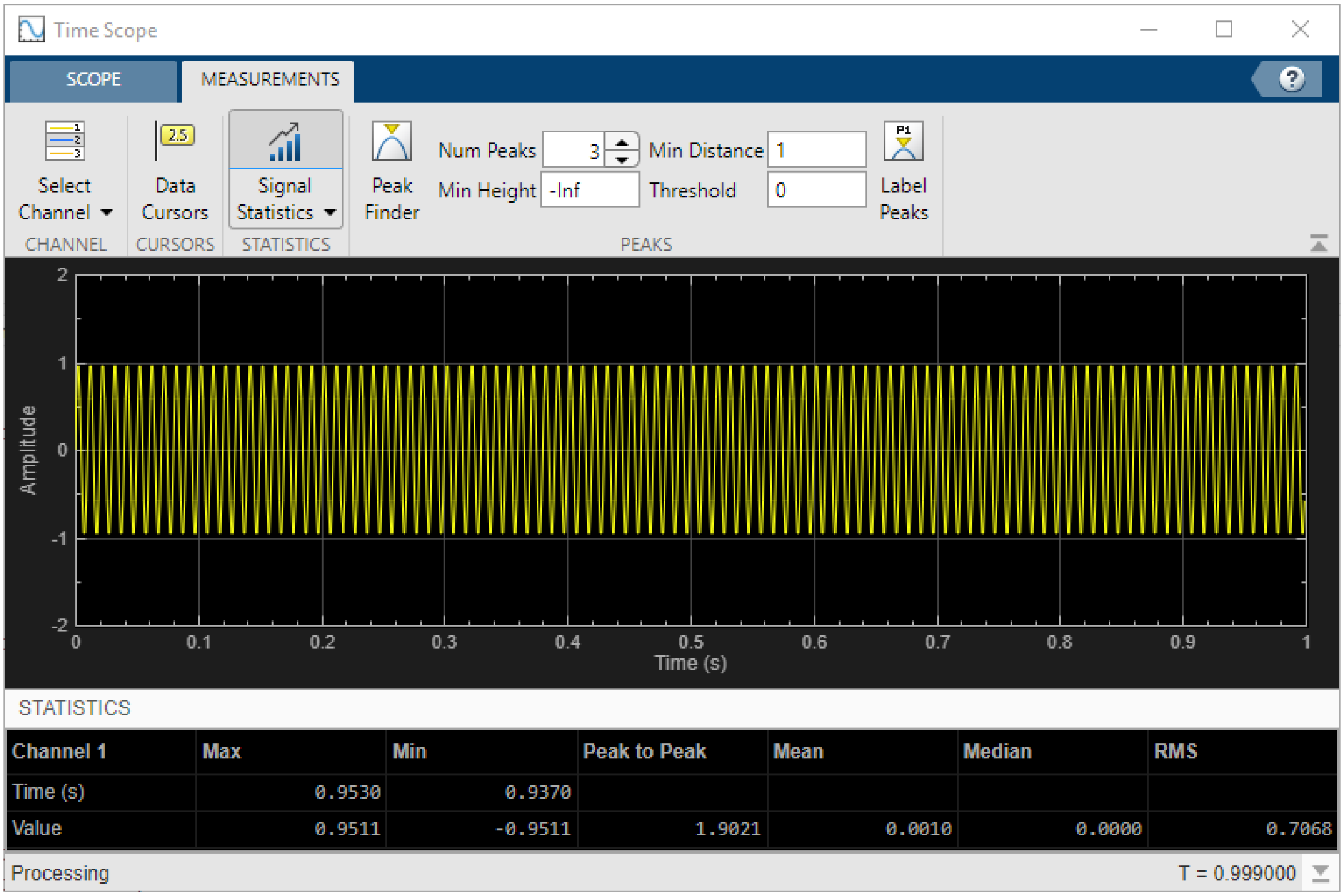
All measurements are made for a specified channel. By default, measurements are applied to the first channel. To change which channel is being measured, use the **Select Channel** drop-down on the **Measurements** tab.

Data Cursors



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

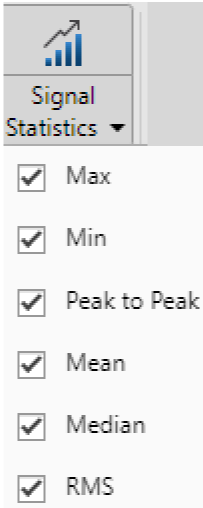
Signal Statistics



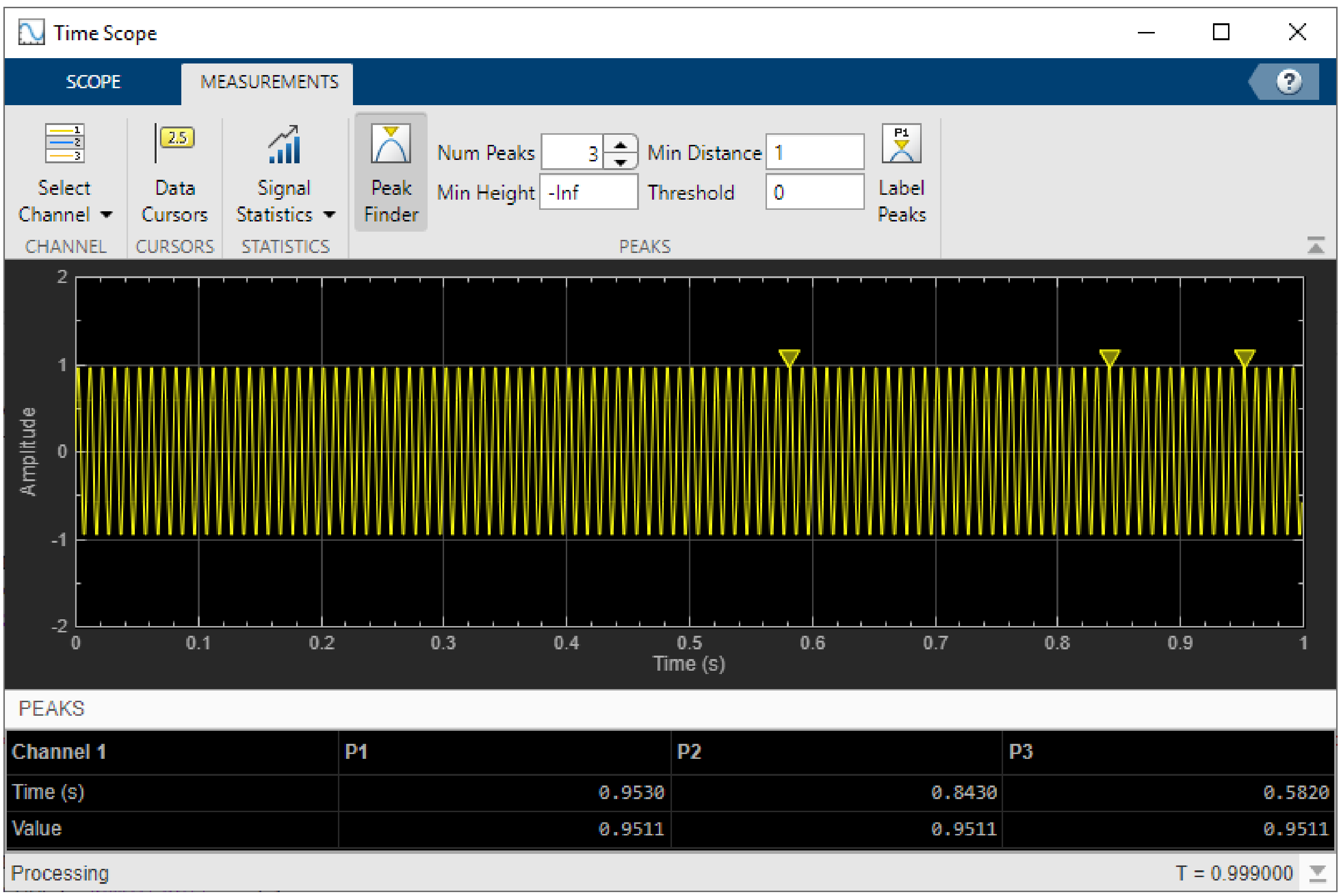
Use the **Signal Statistics** button to display various statistics about the selected signal at the bottom of the time scope window. You can hide or show the **Statistics** panel using the arrow button in the bottom right of the panel.

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

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



Peak Finder



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

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

You can customize several peak finder settings:

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

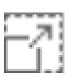
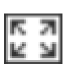
Share or Save the Time Scope

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

- **Generate Script** — Generate a script to re-create your time scope with the same settings. An editor window opens with the code required to re-create your `timescope` object.
- **Copy Display** — Copy the display to your clipboard. You can paste the image in another program to save or share it.
- **Print** — Opens a print dialog box from which you can print out the plot image.

Scale Axes

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

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

See Also

Objects
`timescope`