Tracking radar targets with multiple reflection points

Rev 1.8

![Texas Instruments logo]

Texas Instruments, Incorporated

12500 TI Blvd

Dallas, TX 75074 USA

Version History

Date	Comment
April 13, 2017	**Initial version**
July 20, 2017	**Version 1.0**
Nov 17, 2017	**Version 1.1** - **Updated group and centroid covariance matrices calculation** **Updated configuration and added advanced configuration parameters**
Feb 6, 2018	**Version 1.2** **Updated configuration parameters description**
March 9, 2018	**Version 1.3** **Added derivations for tracking in 3D space**
September 6, 2018	**Version 1.4** **Added description of 3D or 2D library options** **Changes in configuration parameters to support 3D and 2D options**
October 18, 2018	**Version 1.5** **Changed Gating function from constant volume to constant gain with limiters.**
September 1, 2020	**Version 1.6** **Updates based on tracker version 0.110**
January 15, 2021	**Version 1.7** **Added C code Implementation Details** **Added more explanation for the configuration params** **Reorganized the document and moved some sections to the Appendix**
February 26, 2021	**Version 1.8** **Made changes based on feedback from Apps, systems and the Field team**

Table of Contents

1. Introduction and Scope

The purpose of this document is to provide a detailed description of the Group Tracker algorithm and its implementation. After a brief introduction in Section 1 of tracker preliminaries, such as the coordinate systems and tracker motion model, we describe in detail the main building blocks and underlying theory of the Group Tracker in Section 2. Section 3 explains the implementation details with reference to the provided GTRACK source code in the people counting demo application [1]. In Section 4, we briefly describe the tracker configuration parameters. For a detailed description on the user-configurable group tracker parameters and tuning guidelines please refer to the Group Tracker Tuning guide [2].

1.1. Tracking Module

In the Radar Processing stack, the tracking algorithms are the implementations of the localization processing Layer. Tracker is expected to work on the Detection layer outputs (i.e. point clouds) and provide localization information to the higher layers (e.g. classification layer).
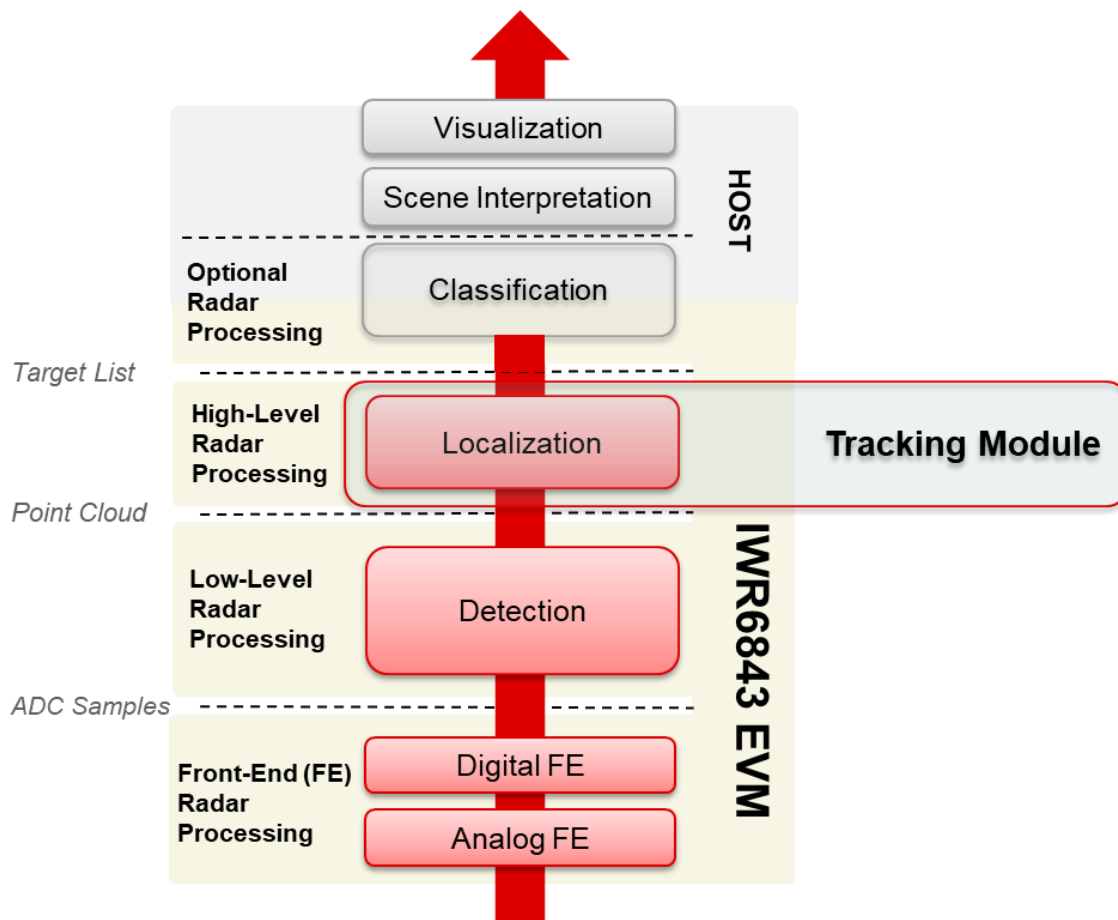

Figure 1. Radar Processing Layers

With high resolution radar sensors, the detection layer is capable of sensing multiple reflections from real life targets, delivering a rich set of measurement vectors (in some cases, thousands of vectors per frame), known as a *Point Cloud*. Each measurement vector represents a reflection point, with Range, Angle, and Radial velocity. Each measurement vector may also contain reliability information in the form of a SNR measure.

Tracking Layer takes the point cloud data as an input, performs target localization, and reports the results (a *Target List*) for further high-level radar processing (such as classification). Therefore, the output of the tracker is a set of trackable objects with certain properties (like position, velocity, physical dimensions, point density, and other features) that can be used by a classifier to make an identification decision.

1.2. Tracker Motion Model in 3D Space

We are interested in modeling the position, velocity, and acceleration of a target (referred to as the state vector), and the Kalman filter is used to "refine" the state vector estimates based on the motion model and measured values from the radar sensor.

For the convenience of target motion extrapolation, we chose to implement tracking in Cartesian coordinates. This allows for simple Newtonian linear prediction models. Tracking can operate in either 2D or 3D Cartesian spaces. For each space, we can use either a constant velocity model or a constant acceleration model. The constant velocity model assumes that the target velocity remains costant during a measurement interval. In constant acceleration model, the assumption is that for each discrete time step *T,* the position and velocity of the object can potentially change, and the acceleration would remain constant. In this document, we describe the constant acceleration model as we think it's reasonable to assume that the target position and velocity would change.

The state Transition Matrix F is used to specify the dynamics of the motion model. State transition matrix F_{3DA} for the constant acceleration model is given by

$$F_{3DA} = \begin{bmatrix} 1 & 0 & 0 & T & 0 & 0 & 0.5T^2 & 0 & 0 \\ 0 & 1 & 0 & 0 & T & 0 & 0 & 0.5T^2 & 0 \\ 0 & 0 & 1 & 0 & 0 & T & 0 & 0 & 0.5T^2 \\ 0 & 0 & 0 & 1 & 0 & 0 & T & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & T & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & T \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad (1\text{-}1)$$

The state of the Kalman filter at time instant **n** is given as

$$s(n) = Fs(n-1) + w(n) \qquad (1\text{-}2)$$

where the state vector $s(n)$ defined in Cartesian coordinates for constant acceleration model is :

$$s_{3DA}(n) \triangleq [x(n) \quad y(n) \quad z(n) \quad \dot{x}(n) \quad \dot{y}(n) \quad \dot{z}(n) \quad \ddot{x}(n) \quad \ddot{y}(n) \quad \ddot{z}(n)]^T \quad \text{(1-3)}$$

$w(n)$ is the vector of process noise with covariance matrix $Q(n)$ of size 9x9 for the constant acceleration model. In Kalman filtering terminology, the process noise matrix Q represents the deviation between the actual state of the system and the motion model.
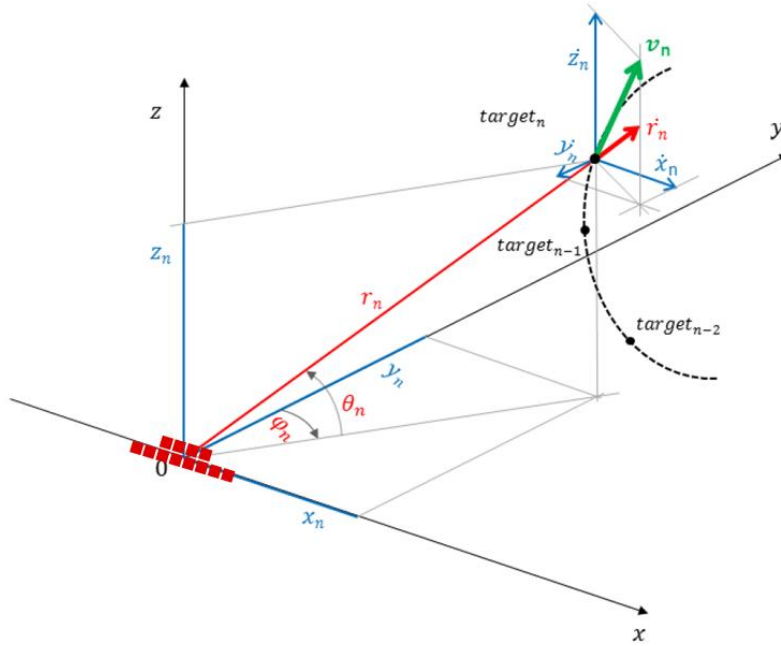
Figure 2. Tracking in 3D.

Figure 2 shows a typical tracking scenario where the center of the antenna virtual array of the radar sensor is considered to be the origin. Target trajectory is shown at times $n-2$, $n-1$, and n. Target is moving with velocity vector v. The input measurement vector $u(n)$ from the radar sensor is in speherical co-ordinates and includes range (r), azimuth (φ), elevation (θ), and radial velocity ($\dot{r}$)

$$u(n) = [r(n) \quad \varphi(n) \quad \theta(n) \quad \dot{r}(n)]^T \qquad \text{(1-4)}$$

The relationship between the state $s(n)$ of the Kalman filter and measurement vector $u(n)$ is expressed as:

$$u(n) = H(s(n)) + v(n), \qquad \text{(1-5)}$$

where H is a measurement matrix given by,

$$H\big(s(n)\big) = \begin{bmatrix} \sqrt{x^2 + y^2 + z^2} \\ \tan^{-1}(x, y) \\ \tan^{-1}\left(z, \sqrt{x^2 + y^2}\right) \\ \dfrac{x\dot{x}+y\dot{y}+z\dot{z}}{\sqrt{x^2+y^2+z^2}} \end{bmatrix}, \tag{1-6}$$

the function $\tan^{-1}(a, b)$ is defined as

$$\tan^{-1}(a, b) \triangleq \begin{cases} \tan^{-1}\left(\dfrac{a}{b}\right), & b > 0, \\ \dfrac{\pi}{2}, & b = 0, \\ \tan^{-1}\left(\dfrac{a}{b}\right) + \pi, & b < 0. \end{cases} \tag{1-7}$$

and $v(n)$ is vector of measurement noise with covariance matrix $R(n)$ of size 4×4.

In the above formulation, (in Eq 1.6), the measurement vector $u(n)$ is related to the state vector $s(n)$ via a non-linear relation. Because of this non-linearity, we use Extended Kalman filter (EKF), which linearizes the relation between $u(n)$ and $s(n)$ by retaining only the first term in the Taylor series expansion of $H(\cdot)$ [3]. The first order Taylor series expansion of $H(s(n))$ about $s_{apr}(n)$ is given as

$$H(s(n)) \approx H\big(s_{apr}(n)\big) + J_H\big(s_{apr}(n)\big)\big[s(n) - s_{apr}(n)\big] \tag{1-8}$$

where $s_{apr}(n)$ is a-priori estimation of state vector at time *n* based on measurements at time *n-1* and J_H is the Jacobian matrix given by

$$J_H(s) = \begin{bmatrix} \dfrac{\partial r}{\partial x} & \dfrac{\partial r}{\partial y} & \dfrac{\partial r}{\partial z} & \dfrac{\partial r}{\partial \dot{x}} & \dfrac{\partial r}{\partial \dot{y}} & \dfrac{\partial r}{\partial \dot{z}} \\ \dfrac{\partial \varphi}{\partial x} & \dfrac{\partial \varphi}{\partial y} & \dfrac{\partial \varphi}{\partial z} & \dfrac{\partial \varphi}{\partial \dot{x}} & \dfrac{\partial \varphi}{\partial \dot{y}} & \dfrac{\partial \varphi}{\partial \dot{z}} \\ \dfrac{\partial \theta}{\partial x} & \dfrac{\partial \theta}{\partial y} & \dfrac{\partial \theta}{\partial z} & \dfrac{\partial \theta}{\partial \dot{x}} & \dfrac{\partial \theta}{\partial \dot{y}} & \dfrac{\partial \theta}{\partial \dot{z}} \\ \dfrac{\partial \dot{r}}{\partial x} & \dfrac{\partial \dot{r}}{\partial y} & \dfrac{\partial \dot{r}}{\partial z} & \dfrac{\partial \dot{r}}{\partial \dot{x}} & \dfrac{\partial \dot{r}}{\partial \dot{y}} & \dfrac{\partial \dot{r}}{\partial \dot{z}} \end{bmatrix}. \tag{1-9}$$

For the specific case of 3D constant acceleration motion mdoel the Jacobian Matrix $J_H(s_{3DA})$ is given as (the calculation of partial derivaties is shown in the Appendix):

$$J_H(s_{3DA}) = \begin{bmatrix} \dfrac{x}{r} & \dfrac{y}{r} & \dfrac{z}{r} & 0 & 0 & 0 & 0 & 0 & 0 \\ \dfrac{y}{x^2+y^2} & -\dfrac{x}{x^2+y^2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\dfrac{x}{r^2}\dfrac{z}{\sqrt{x^2+y^2}} & -\dfrac{y}{r^2}\dfrac{z}{\sqrt{x^2+y^2}} & \dfrac{\sqrt{x^2+y^2}}{r^2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \dfrac{y(\dot{x}y-\dot{y}x)+z(\dot{x}z-\dot{z}x)}{r^3} & \dfrac{x(\dot{y}x-\dot{x}y)+z(\dot{y}z-\dot{z}y)}{r^3} & \dfrac{x(\dot{z}x-\dot{x}z)+y(\dot{z}y-\dot{y}z)}{r^3} & \dfrac{x}{r} & \dfrac{y}{r} & \dfrac{z}{r} & 0 & 0 & 0 \end{bmatrix} \tag{1-10}$$

After linearizing, the relationship between the state $s(n)$ and measurement vector $u(n)$ is :

$$u(n) = H\big(s_{apr}(n)\big) + J_H\big(s_{apr}(n)\big)\big[s(n) - s_{apr}(n)\big] + v(n), \tag{1-11}$$

2. Group Tracking

In this section, we describe the main building blocks and underlying theory of the Group Tracker. From a signal processing perspective, the group tracker can be considered as a module that spatially and temporally filters the detected points. Traditionally, clustering (spatial filtering) and tracking (temporal filtering) are done as separate modules. In the group tracker, both these functionalities are unified and hence the name "Group tracker" as it tracks a cluster of points (a.k.a. group) over time.

The following is some of the notation used in this section:

- $s_i(n)$ – State vector of tracking object i at time n. Each tracking object has its own state vector, which is predicted and updated independently. For simplicity reasons, we omit the tracking index.
- $s_{apr}(n)$ – A-priori (predicted) estimates of tracking state at time n.
- $P(n)$ – State vector estimation error covariance matrix at time n, defined as $P(n) =$ Cov $[s(n) - s_{apr}(n)]$.
- $P_{apr}(n)$ – A-priory (predicted) estimates of state vector covariance matrix at time n.
- F – State Transition Matrix.
- Q – Process Noise Covariance Matrix.
- C_G – Group residual Covariance Matrix
- R_m – Measurement Covariance Matrix (This is a diagonal Matrix)
- D – Group Dispersion Matrix
- R_c – Measurement error covariance matrix for the centroid (track)

2.1. High Level Algorithm Description

Real world radar targets (cars, pedestrians, walls, landing ground, etc.) are presented to a tracking processing layer as a set of multiple reflection points. Those detection points form a group of correlated measurements with range, angle, and radial velocity. Of course, at any time, there could be multiple real world targets. Therefore, a tracker is needed that is capable of working with multiple target groups. The group tracking approach is illustrated in Figure 3 below.
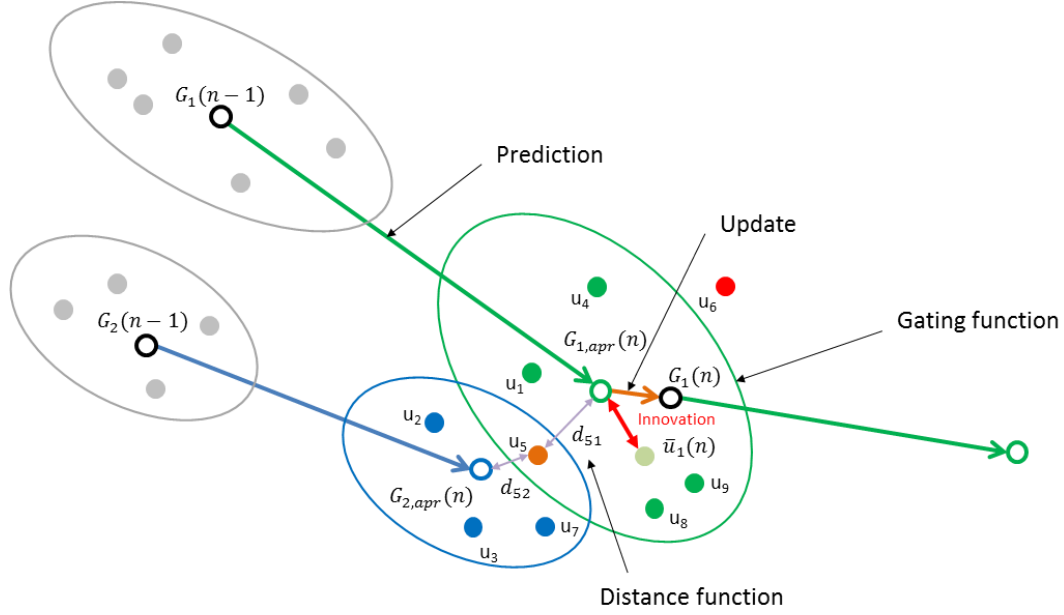
Figure 3. Group Tracking

An example of tracking two targets is shown in Figure 3 where at time instance $n-1$ the centroids of the two tracks are labelled as $G_1(n-1)$ and $G_2(n-1)$. A high level description of the tracking process is illustrated below

- In the prediction step of the tracker (described in section 2.2), the track centroids are predicted for time n based on the tracker motion model. The predicted track centroids are labelled as $G_{1,apr}(n)$ and $G_{2,apr}(n)$.

- A gate (measurement boundary) is constructed around each of the predicted centroids i.e. $G_{1,apr}(n)$ and $G_{2,apr}(n)$. The shape of this gate is determined by the gating function described in section 2.3.1

- A set of measurements $[u_1, u_2, \ldots \ldots u_9]$ obtained from the detection layer at time n need to be associated with a unique track. A distance metric is computed between the measurements and the predicted track centroids. A measurement is assigned to the track with the closest distance. For instance, in Figure 3, the measurement u_5 is within the gate of both track 1 and track 2. d_{51} is the distance metric between measurement u_5 and track 1 while d_{52} is between u_5 and track 2. The measurement u_5 has been assigned to track 1. Section 2.3 describes these steps in more detail.

- For each track, the mean $\bar{u}(n)$ of all associated measurements is computed. For track 1, the set of associated measurements is $[u_1, u_4, u_5, u_8, u_9]$ and the mean is donated by $\bar{u}_1(n)$

- The difference between the predicted centroid $G_{1,apr}(n)$ and mean of the measurments $\bar{u}_1(n)$ is called the innovation. The innovation is a measure of how much our predictions are different from our observed measurements.

- Using the innovation measure, the track centroid is is updated to $G_1(n)$ using the update steps of the group tracker explained in Section 2.5

11

- The above sequence of operation is repeated for the next time stamp.

The main functional blocks of the group tracker algorithm are shown in a Figure 4. The blocks shown in blue are classical extended Kalman filter (EKF) operations. The blocks shown in orange are additions to support multipoint grouping. A description of each of these blocks is given in subsequent sections.
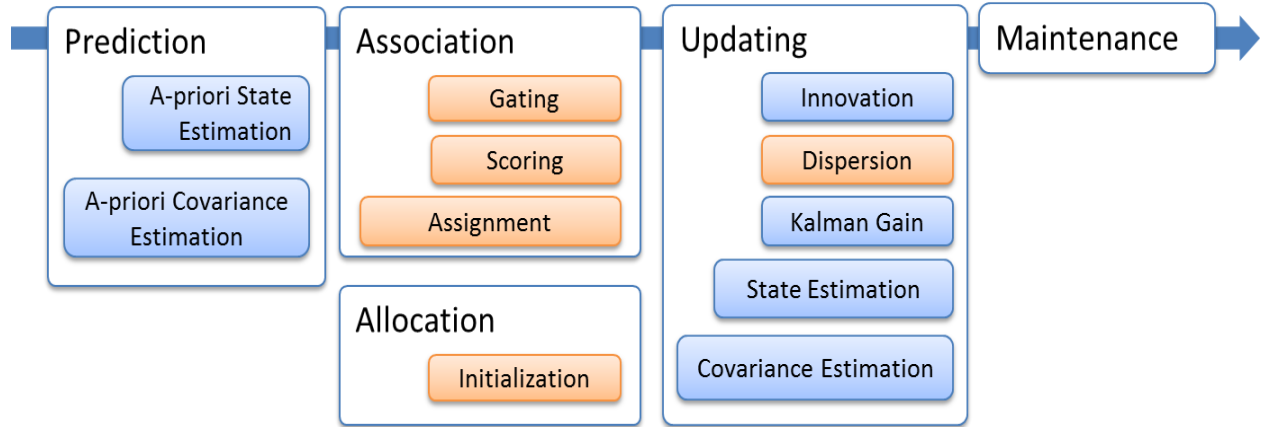
Figure 4. Tracking Block Diagram.

2.2. Prediction Step

As shown in Figure 4, the Kalman filter prediction step constitutes of computing the *a-priori* state $s_{apr}(n)$ and *a-priori* covariance $P_{apr}(n)$ estimations for each trackable object based on state and process covariance matrices estimated at time instance *n-1*. The a-*priori* state and *a-priori* covariance estimates are obtained using:

$$s_{apr}(n) = Fs(n-1) \tag{2-1}$$
$$P_{apr}(n) = FP(n-1)F^T + Q(n-1). \tag{2-2}$$

The above equations constitute the *prediction* step of the Kalman filter. $Q(n)$ is the process noise covariance matrix described in detail in Section Design of Process Noise Matrix6.4.3 in the appendix. In addition, we also compute $H\left(s_{apr}(n)\right)$ which converts the predicted *a-priori* states $s_{apr}(n)$ from cartesian to spherical coordinates. $H\left(s_{apr}(n)\right)$ is subsequently used in the upate step of the group tracker.

2.3. Association Step

The association step consists of associating radar measurements to a unique existing track. This is accomplished by 3 main steps namely Gating, Scoring and Assignment as shown in Figure 4.

Assume existence of one or more tracks and the associated predicted state vector. For each given track, we form a gate about the predicted centroid. The gate should account for (a) target maneuver, (b) dispersion of the group, and (c) measurement noise. We use the group residual covariance matrix C_G to

build an ellipsoid in 3D measurement space about the tracking group centroid. The ellipsoid will represent **a** *Gating function* (shown in 2D in Figure 3) to qualify individual measurements we observe at time *n*. The gating function design is explained in section 2.3.1.

In the scoring step, we assign a bidding score to each measurement that falls within the gate of a track. Subsequently, the assignment process assigns one measurement at a time to the closest track. This creates a set of measurements associated with each track. The scoring and assignment procedures are explained in section 2.3.2.

2.3.1. Gating Function

The gating function represents the amount of innovations we are willing to accept, given the current state of uncertainty that exists in a current track. As defined previously, the innovation is a measure of how much our predictions are different from our observed measurements. For a track i, and measurement vector u_j, the amount of innovation is given by

$$y_{ij} = u_j(n) - H_i\big(\hat{s}(n-1)\big) \tag{2-3}$$

To measure the amount of uncertainty, we define the group residual covariance matrix as

$$\boldsymbol{C_G} = \boldsymbol{J_H}\big(\boldsymbol{s}_{apr}(n)\big)\boldsymbol{P}_{apr}(n)\boldsymbol{J_H^T}\big(\boldsymbol{s}_{apr}(n)\big) + \boldsymbol{R_m} + \widehat{\boldsymbol{D}} \tag{2-4}$$

Note that this group covariance matrix $\boldsymbol{C_G}$ is between a member of the measurement group and the group centroid. The term $\boldsymbol{J_H}\big(\boldsymbol{s}_{apr}(n)\big)\boldsymbol{P}_{apr}(n)\boldsymbol{J_H^T}\big(\boldsymbol{s}_{apr}(n)\big)$ represents the uncertainty in the centroid due to target maneuvering, and is similar to the term used for individual target tracking. The term $\boldsymbol{R_m}$ is the diagonal matrix of the measurement variances and the term $\widehat{\boldsymbol{D}}$ is the estimation of group track dispersion matrix. (Please refer to section 3.8.3 for more details)

For each existing track i, and for all measurement vectors j we obtained at time n, we define a distance function d_{ij}^2, which represents the amount of innovation the new measurement adds to an existing track.

$$d_{ij}^2 \triangleq y_{ij}{}^T \, inv(C_{Gi}) \, y_{ij} \tag{2-5}$$

$$d_{ij}^2 = \big[u_j(n) - H_i\big(\hat{s}(n-1)\big)\big]^T inv\big(C_{Gi}(n)\big)\big[u_j(n) - H_i\big(\hat{s}(n-1)\big)\big] \tag{2-6}$$

We define the chi-squared test (because the sum of squares of M Gaussian random variables with zero mean is chi-square distribution with degree of freedom M) which limits the amount of innovation we are willing to accept as

$$d_{ij}^2 < G \tag{2-7}$$

The boundary condition represents an arbitrarily oriented ellipsoid centered at *a-priory* expectation of the measurement vector, while $\boldsymbol{G}$ represents the largest normalized distance from the measurement

that we are going to accept into the group. To avoid stability issues with this approach of keeping the gain **G** constant (the bubble captures new points, grows, which increases the chances to capture more points), we add a limiting factor. Under no circumstances can the targets capture points further than pre-configured limits.

2.3.2. Scoring Function and Assignment

Figure 5 illustrates the post gating situation, where measurement vectors $\{u_1, u_2, u_3, u_7\}$ pass the gating test for the green track, while vectors $\{u_3, u_4, u_5, u_8, u_9\}$ pass the gating test for the blue track.
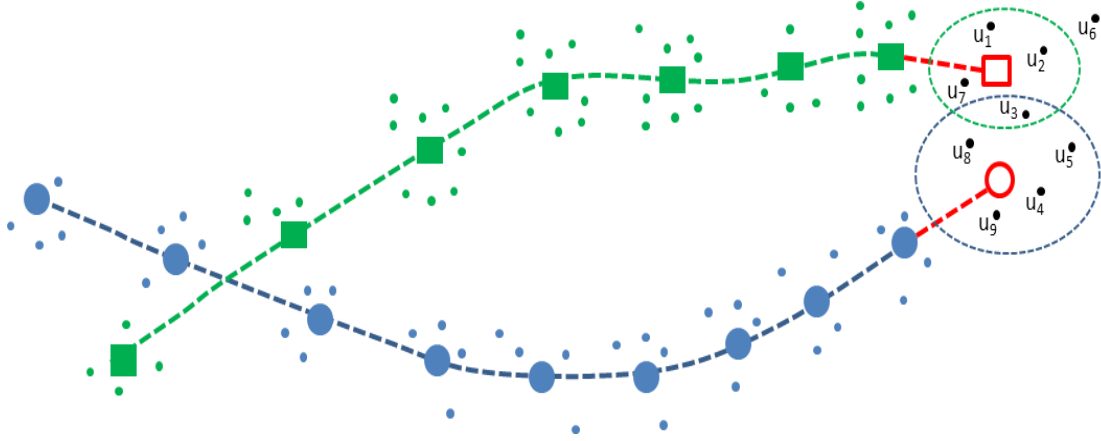
Figure 5. Scoring function Illustration

As shown in [4], the likelihood function (assuming Gaussian distribution for the residual) associated with assignment of observation j to track i is

$$g_{ij} = \frac{e^{-\frac{d_{ij}^2}{2}}}{(2\pi)^{M/2}\sqrt{|C_{Gi}|}} \tag{2-8}$$

Where, $|\boldsymbol{C}_{Gi}|$ is determinant of the residual covariance matrix for track i, $d_{ij}^2 = y_{ij}^T \, inv(\boldsymbol{C_{Gi}}) \, y_{ij}$, and y_{ij} is residual vector from observation j to track i.

To maximize g_{ij}, by taking the logarithm, we derive the scoring criteria we want to minimize:

$$B_{ij}^2 = \ln|\boldsymbol{C}_{Gi}| + d_{ij}^2 \tag{2-9}$$

Finally, the measurement j is assigned to the track i with which it has the best bidding score.

2.4. Allocation Step

For measurements not associated with any track (that are outside of any existing gate), a new group tracker is allocated and initialized. This is an iterative process, similar to the DBSCAN clustering algorithm [5]. It is significantly simpler since it is only done for the leftover measurements.

We first select an arbitraty point as a leading measurement and set a centroid (range/angle) equal to it. One candidate at a time, we first check whether the point is within measurement bounds (velocity check, followed by a distance check). If passed, the centroid is recalculated, and point is added to the cluster. Once finished, we perform a few qualifying tests for the cluster. These tests include whether the cluster contains a minimal number of measurement points, strong enough combined SNR, and/or minimal amount of dynamicity of the centroid. If passed, we create (allocate) a new tracking object and use the associated points to initialize dispersion matrices. Clusters that do not pass these tests are ignored.

2.5. Updating Step

The tracker update steps are outlined below. These steps are only done when we have enough dynamic points associated with a track. (Cases in which we have no measurement points associated with a track is described in the Implementation Section 3.8.2)

1. For each track, the mean $\bar{u}(n)$ of all associated measurements is first computed. If desired, this mean can also be a SNR weighted mean.

2. Compute the innovation (or measurement residual) vector using the mean of all measurements associated with it

$$y(n) = \bar{u}(n) - \boldsymbol{H}\left(s_{apr}(n)\right) \tag{2-10}$$

3. Compute the innovation (or measurement residual) covariance matrix

$$C(n) = \boldsymbol{J}_H\left(s_{apr}(n)\right)\boldsymbol{P}_{apr}(n)\boldsymbol{J}_H^T\left(s_{apr}(n)\right) + R_c(n) \tag{2-11}$$

Where $\boldsymbol{R}_c$ is the measurement error covariance matrix for the centroid used for Kalman Update and is given by

$$\boldsymbol{R}_C = \frac{\boldsymbol{R}_m}{N_A} + f\left(N_A, \widehat{N}\right)\widehat{\boldsymbol{D}} \tag{2-12}$$

$\widehat{\boldsymbol{D}}$ is the Group dispersion Matrix
$\boldsymbol{R}_m$ is the measurement covariance Matrix
N_A is the number or measurements associated with a given track
$\widehat{N}$ is the estimated number of elements in target tracked by the given track
$f\left(N_A, \widehat{N}\right)$ is the weighting factor given by $\frac{\widehat{N}-N_A}{(\widehat{N}-1)N_A}$ for the case with no false detections i.e. $N_A \leq \widehat{N}$.

Note that $f(N_A, \widehat{N}) = 0$ when all the elements are detected and $f(N_A, \widehat{N}) = 1$ when there is a single associated measurement.

From Eq 2.12, R_c is the sum of two factors. The first term $\frac{R_m}{N_A}$ represents the error in measuring the centroid due to radar measurement error. The more the number of measurements associated with a given track, the smaller is this error term. The second term represents the uncertainty due to the fact that not all the elements may have been observed. The estimated number of elements $\widehat{N}_n$ in a tracked target is estimated recursively as

$$\widehat{N}_n = (1 - \alpha_N)\widehat{N}_{n-1} + \alpha_N \widehat{N}_n \tag{2-13}$$

Group dispersion Matrix $\widehat{D}_n$ is estimated at time instance **n** as described below. For the measurement vector, $u(n) = [r(n) \ \varphi(n) \ \theta(n) \ \dot{r}(n) \ ]^T$, the dispersion matrix is given by

$$D = \begin{bmatrix} d_{rr}^2 & d_{r\varphi}^2 & d_{r\theta}^2 & d_{r\dot{r}}^2 \\ d_{\varphi r}^2 & d_{\varphi\varphi}^2 & d_{\varphi\theta}^2 & d_{\varphi\dot{r}}^2 \\ d_{\theta r}^2 & d_{\theta\varphi}^2 & d_{\theta\theta}^2 & d_{\theta\dot{r}}^2 \\ d_{\dot{r}r}^2 & d_{\dot{r}\varphi}^2 & d_{\dot{r}\theta}^2 & d_{\dot{r}\dot{r}}^2 \end{bmatrix} \tag{2-14}$$

Where $d_{r\theta}^2 = \frac{1}{N}\sum_{i=1}^{N}(a_i - \bar{a})(b_i - \bar{b})$ and $a, b \in \{r, \ \theta, \ \varphi, \ \dot{r} \}$ and $\bar{a}, \bar{b}$ are the means computed for a given group of measurements. The Group dispersion matrix is then updated as

$$\widehat{D}_n = (1 - \alpha_D)\widehat{D}_{n-1} + \alpha_D D \tag{2-15}$$

4. Compute Kalman Gain $K(n)$

$$K(n) = P_{apr}(n)J_H^T\left(s_{apr}(n)\right)inv[C(n)] \tag{2-16}$$

5. Compute a-posteriori State Vector $s(n)$

$$s(n) = s_{apr}(n) + K(n)y(n) \tag{2-17}$$

6. Compute a-posteriori Error Covariance Matrix $P(n)$

$$P(n) = P_{apr}(n) - K(n)J_H\left(s_{apr}(n)\right)P_{apr}(n)) \tag{2-18}$$

2.6. Maintenance Step

Each track goes through a life cycle of events. At the maintenance step we may decide to change the state or to delete a track that is not active any more. More details are in section 3.9 .

3. Group Tracker Implementation Details

This section describes the TI implementation details of the Group tracker described in section 2. We explain the use of the group tracker in People Counting Demo application where the group tracker is used to track and count the number of people.

3.1. Sensor Mount Configuration Geometry

The People Counting Demo, supports two sensor mount configurations (a) Wall-mount (b) Ceiling-mount. It's convenient to define mathematical spaces in which the measured data is obtained, tracker operates and the output is visualized. We define the following spaces which are applicable to both wall and ceiling mount configurations

- World Space W : {X_w, Y_w, Z_w} is Cartesian with origin **O** at floor level
- Tracker Space T : {X_t, Y_t, Z_t} is Cartesian with origin at sensor
- Point Cloud Space P: {r, φ, θ, $\dot{r}$} is Spherical with origin at the sensor where r is the radial distance, φ is the azimuth angle, θ the elevation angle and $\dot{r}$ is the radial velocity of a measured point with respect to the sensor axis.

The sensor mounting geometry and its relation to the different measurements spaces is shown in Figure 6. Please note that in the demo configurations

- Sensor location S_w = {0, 0, H} is specified in the World Cartesian Space where H is the height of the sensor above the ground

- It's also useful to define a rotation matrix. If the sensor is tilted clockwise by an angle Θ about the axis X_w, then the rotation Matrix $R_x(\Theta)$ is given as

$$R_x(\Theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\Theta & sin\Theta \\ 0 & -\sin\Theta & cos\Theta \end{bmatrix}$$

- All configuration boxes defined by the user are also in the World Cartesian Space. In Figure 6, the boundary box is specified by the co-ordinate values {x_1, x_2, y_1, y_2, z_1, z_2,}
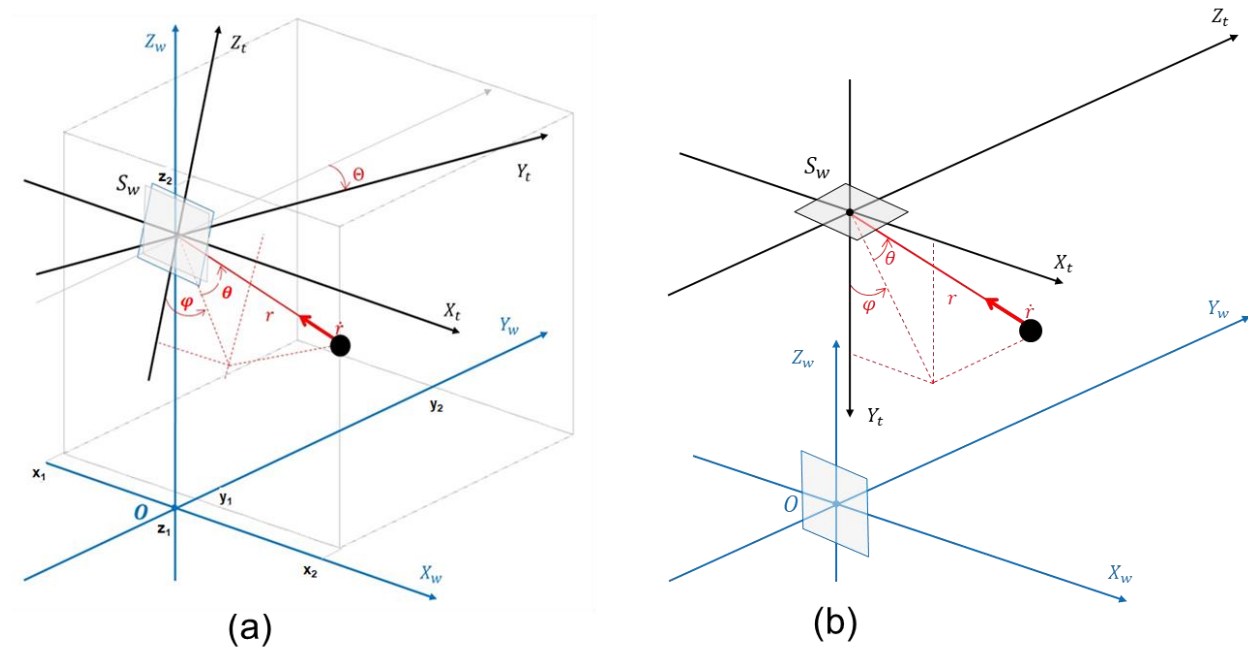
Figure 6. Sensor mounting geometry (a) Wall Mount Sensor (b) Ceiling Mount Sensor

As mentioned previously, the radar reports the point cloud data in spherical coordinates while the tracker operates in Cartesian in tracker Space. Moreover, the scene configuration and tracker output data is best visualized in the world space. Hence there would be several occasions when a transformation from one space to another is needed which are summarized in the table below

Transformation	Wall Mount	Ceiling Mount	Description
$P: \{r, \varphi, \theta\} \rightarrow$ $T: \{x_t, y_t, z_t\}$	$x_t = rCos(\theta)Sin(\varphi)$ $y_t = rCos(\theta)Cos(\varphi)$ $z_t = rSin(\theta)$	$x_t = rCos(\theta)Sin(\varphi)$ $y_t = rCos(\theta)Cos(\varphi)$ $z_t = rSin(\theta)$	Spherical to Cartesian conversion
$T: \{x_t, y_t, z_t\} \rightarrow$ $W: \{x_w, y_w, z_w\}$	$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = R_x(\Theta) \begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ H \end{bmatrix}$	$x_w = x_t$ $y_w = z_t$ $z_w = -y_t + H$	Tracker to World Space Useful for tracker output visualization
$W: \{x_w, y_w, z_w\} \rightarrow$ $T: \{x_t, y_t, z_t\}$	$\begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} = R_x(-\Theta) \left(\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ H \end{bmatrix} \right)$	$x_t = x_w$ $y_t = H - z_w$ $z_t = y_w$	World to Tracker Space Used for boundary box conversions

Table 1 : Co-ordinate Transformations

3.2.   Group tracker (GTRACK) Library

The tracking algorithm is delivered as a source code and pre-compiled target libraries with a ready-to-build makefile infrastructure. The application task creates an algorithm instance with configuration parameters that describe sensor, scenery, and behavior of radar targets. The configuration is passed to the algorithm instance at module create time. Algorithm is called once per frame from the application Task context. It takes as an input, the point cloud data from the detection layer, performs target localization, and outputs a set of trackable objects with certain properties (like position, velocity, physical dimensions).

The figure below shows the steps that the group tracker algorithm goes through during each frame call. Please refer to Section 2 for more details on the group tracker algorithm.
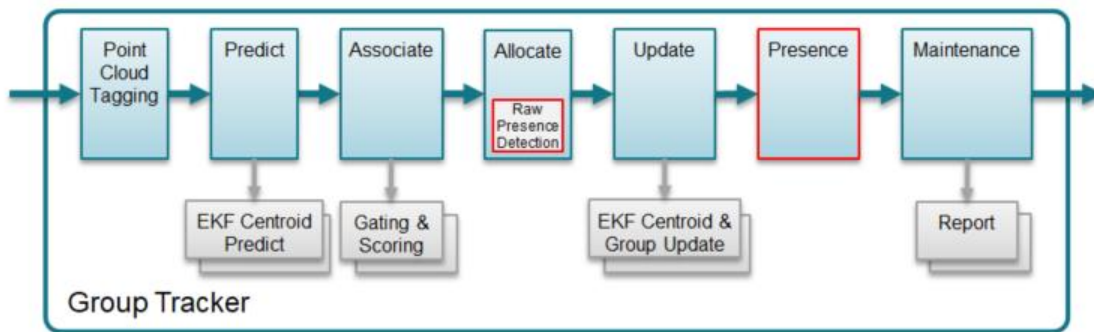
Figure 7. Group Tracking Algorithm

3.2.1. Module and Unit sub-layers

The group tracker algorithm is implemented with two internal software sublayers: MODULE and UNIT(s). An application can create multiple MODULE instances with different configuration parameters (for example, to track different classes of targets). Each MODULE instance creates and manages multiple UNIT(s) where each tracker UNIT represents a single tracking object. UNIT(s) inherits configuration parameters from the parent MODULE. In the people counting demo only one tracker MODULE is instantiated as we are only interested in one class of objects (i.e. people). The number of tracker UNIT(s) managed by the tracker MODULE corresponds to the number of people being actively tracked.

Corresponding to each functional block there is a MODULE-level function (with prefix gtrack_module) which implements the corresponding functionality (e.g. Prediction performed by gtrack_modulePredict). The MODULE-level functions in turn call the UNIT-level functions (with prefix gtrack_unit) which perform the corresponding tasks for all the active UNIT(s) maintained by the parent MODULE.

3.2.2. Gtrack library API usage

The pseudo-code below illustrates the implementation of the Group Tracker from an application perspective.

```
h = gtrack_create(params);                    // Creates an instance of the algorithm with a desired configuration
while(running) {
    gtrack_step(h, pointCloud, &targetList);  // Runs a single step of the given alrorithm instance with input point cloud
        data
}
gtrack_delete(h);                             // Delete the algorithm instance
```

- *gtrack_create()* creates a MODULE instance and pre-allocates resources for a maximum number of UNIT(s).
- *gtrack_step()* calls one single round of MODULE functions as illustrated in the pseudo-code below
- *gtrack_delete()* deletes the tracker instance and returns all the resources back to the system.

```
gtrack_step(h, pointCloud, &targetList) {
    gtrack_modulePredict(h,...);
    gtrack_moduleAssociate(h,...);
    gtrack_moduleAllocate(h,...);
    gtrack_moduleUpdate(h,...);
    gtrack_moduleReport(h,...);
}
```

Tracker UNIT(s) are created during *gtrack_moduleAllocate()* calls. The resources pre-allocated at MODULE create time are assigned to a new UNIT. All active UNIT(s) are called during each parent module function step calls. As an example, the pseudo-code below illustrates the implementation of the predict function:

```
gtrack_modulePredict(h,...) {
    for(each active unit) {
        gtrack_unit_predict(unit, ...);
    }
}
```

UNIT(s) can be deleted if not active during *gtrack_moduleUpdate()* calls and the resources returned back to the parent MODULE.

Table 2 lists the relevant C functions and configuration parameters for the different functional blocks of the Group tracker

Functional Blocks	Relevant Configuration Parameters	Relevant C functions
Co-ordinate Transformations		gtrack_sph2cart, gtrack_censor2world gtrack_cart2sph
Point Cloud Tagging	Scenery Parameters	gtrack_step, gtrack_sph2cart, gtrack_censor2world gtrack_isPointInsideBox
Track initialization	Gating Parameters, Max Acceleration Parameters	gtrack_create, gtrack_unitCreate, gtrack_unitStart gtrack_calcMeasurementLimits

Track Prediction	Gating Parameters,	gtrack_modulePredict, gtrack_unitPredict, gtrack_cartesian2spherical
Association	Gating Parameters,	gtrack_moduleAssociate, gtrack_unitScore, gtrack_computeMahalanobisPartial, gtrack_computeMahalanobis
Track Allocation	Allocation Parameters	gtrack_moduleAllocate, gtrack_unitStart gtrack_calcDistance
Track Update	Allocation Parameters	gtrack_moduleUpdate, gtrack_unitUpdate gtrack_unitStop gtrack_calcDim
Track Maintenance	State Transition Parameters	gtrack_unitEvent gtrack_unitUpdate gtrack_isPointInsideBox
Presence	Presence Detection Parameters	gtrack_modulePresence gtrack_moduleAllocate
Track report		gtrack_moduleReport gtrack_unitReport

Table 2 : Functional blocks of the Group tracker and the relevant C functions and configuration parameters

3.3. Point Cloud Tagging

Point cloud input is first tagged based on scene boundaries. Some points may get tagged as "outside the boundaries", and will be ignored in association and allocation processes. Boundary boxes in the configuration parameters are defined in the World-coordinates in Cartesian space while the radar measurements are in Sensor-coordinates (in Spherical space). Hence a transformation is required to determine if the measurement points fall within the boundary boxes specified by the user.

Point cloud tagging is implemented in *gtrack_step()*. The function *gtrack_sph2cart()* converts the measurement point in Spherical-to-Cartesian coordinates and the function *gtrack_censor2world()* converts from the Sensor- to-World coordinates.

```
if(inst->params.transormParams.transformationRequired) {
        gtrack_sph2cart(&point[n].vector, &pos);
        gtrack_censor2world(&pos, &inst->params.transormParams, &posW);
}
```

The position values in World-coordinates are returned in the variable **posW** and are subsequently checked if they are within the user-specified boundary boxes by the function *gtrack_isPointInsideBox().* All those points that are outside the boundary boxes are discarded by the tracker.

3.4. Tracker Initialization

Tracker configuration parameters are initialized at several different levels as summarized in Table 3. Tracker MODULE level parameters that would be common for all the track UNIT(s) within the parent module are set in the *gtrack_create()* function. Some group tracker parameters such as the motion model (e.g. 3D constant acceleration model explained in Section1.2) are initialized when the track unit is created through the function *gtrack_unitCreate()*. Finally, once a decision has been made to allocate a

track, parameters such as the initial estimate of the state vector, error covariance matrix, measurement error limits and measurement spreads are initialized in *gtrack_unitStart()*

Function name	Description	Initialized Parameters
gtrack_create()	Module level parameter values that would be constant for each of the tracking Units	Gating, Tracker state, velocity unrolling, allocation, scenery parameters, Process Noise Matric, State Transition Matrix Wall-mount or ceiling mount configuration
gtrack_unitCreate()	Configuration parameters that depend on specific use-cases i.e. ceiling mount vs wall mount	Motion model Parameters, Internal tracker configurations
gtrack_unitStart()	Internal tracker configurations that are initialized based on the current target position	State transition counters State vector Estimate and Covariance Matrix Measurement Error limits, Measurement Spread Estimation

Table 3 : Tracker Initialization

3.4.1. Initialization at tracker MODULE creation

gtrack_create() function creates a tracker module instance and pre-allocates resources for a maximum number of units. This function also sets parameter values that would be common for all the track units within the module. These parameters are

(a) Default parameters for the gating, state, velocity unrolling, allocation, scenery and presence are set. Some of these default parameters will be overwritten if the user has specified those through the CLI interface

(b) Determines if the tracker should operate in the Wall-mount or Ceiling-mount configuration based on the elevation Tilt of the sensor. If the elevation tilt is within the interval *69.5 ≤ elevation tilt ≤ 110.4*, then the group tracker is configured as Ceiling-mount. This elevation tilt is specified by the user through the scenery parameters (described in section 4.2.1).

```
thetaRot = inst->params.sceneryParams.sensorOrientation.elevTilt;
    if(fabs(thetaRot - 90.0f) < 20.5f)
        inst->isCeilingMounted = true;
    else
        inst->isCeilingMounted = false;
```

(c) Initialize the Process Noise Matrix **Q** and the State Transition Matrix **F**

3.4.2. Initialization at tracker UNIT creation

(a) For each track unit, the function *gtrack_create()* calls *gtrack_unitCreate()* to set the unit-level tracker parameters. For the 3D constant acceleration motion model the following group tracker internal parameters are configured

```
case GTRACK_STATE_VECTORS_3DA:
        inst->stateVectorType = GTRACK_STATE_VECTORS_3DA;
        inst->stateVectorDimNum = 3;
        inst->stateVectorDimLength = 3;
```

```
inst->stateVectorLength = 9;
inst->measurementVectorLength = 4;
inst->isAssociationGhostMarking = false;
inst->estSpreadAlpha = 0.01f;
```

(b) Wall-mount and ceiling-mount might need different configuration settings. These are set as below

```
if(fabs(90.0f - inst->sceneryParams->sensorOrientation.elevTilt) < 20.5f) {
            /* Ceiling mount specific initialization */
            inst->isAssociationHeightIgnore = true;
            inst->isEnablePointNumberEstimation = false;
            inst->isSnrWeighting = false;
            inst->confidenceLevelThreshold = 0.3f;
            inst->minVelocityStopNoPoints = 1.0f;
            inst->minVelocityStopNoDyn = 1.0f;
            inst->minStaticVelocitySlow = 2.0f;
        }
        else {
            /* Wall mount specific initialization */
            inst->isAssociationHeightIgnore = false;
            inst->isEnablePointNumberEstimation = true;
            inst->isSnrWeighting = true;
            inst->confidenceLevelThreshold = 0.5f;
            inst->minVelocityStopNoPoints = 0.5f;
            inst->minVelocityStopNoDyn = 0.5f;
            inst->minStaticVelocitySlow = 1.0f;
    }
```

3.4.3. Initialization at track allocation

Once a new set of measurement points passes allocation thresholds in the allocation module *(gtrack_moduleAllocate())* a new track will be allocated. *gtrack_unitStart()* is called once a decision has been made to allocate the track which initializes several internal tracker configurations for such as

(a) The state transition counters (i.e. *active2freeCount, sleep2freeCount, detect2activeCount, detect2freeCount*) are initialized to zero.
(b) The initial STATE of the track is set to DETECT (Note that the track is not ACTIVE as yet)
(c) The *confidenceLevel* of the track is set to 1 (0.5 if the track is behind another existing target. The function *gtrack_isPointBehindTarget()* is called to determine if the target is behind and the flag *isBehind* is appropriately set).
(d) Radial velocity handling is set to start with range-rate filtering and the initial *rangeRate* value for the track is set to the unrolled radial velocity.
(e) Centroid of the track is set to the mean value of the range, Doppler, Angle, and SNR of the members of the allocation set. Note that these mean values have already been calculated in the *gtrack_moduleAllocate()* for the allocation set.
(f) The State vector is initialized by converting the centroid values (range, azimuth angle, elevation angle, Doppler) in spherical co-ordinates to Cartesian. The initial value of acceleration parameters in the State vector are set to 0.

(g) State Vector Estimate Covariance Matrix is initialized

(h) Measurement Error Limits *(H_limits)* are initialized based on the *gatingParams* and the current target position (i.e. centroid range) using the function *gtrack_calcMeasurementLimits().*

(i) Measurement Spread Estimation *(estSpread)* is initialized based on the *gatingParams* and the current target position (i.e. centroid range) using the function *gtrack_calcMeasurementSpread()*

3.5.   Track Prediction

The MODULE level predict function *gtrack_modulePredict()* is called by *gtrack_step()* to perform Prediction. For each active track, the unit-level function *gtrack_unitPredict()* is called to implement the track prediction based on the pseudo-code below

```
gtrack_modulePredict(h,...) {
    for(each active unit) {
        gtrack_unit_predict(unit, ...);
    }
}
```

The flag *isTargetStatic* is checked to determine if the target is STATIC or DYNAMIC. This flag is updated in the Update step (explained in section 3.8). Within *gtrack_unitPredict()* , the prediction steps are only performed if the target is DYNAMIC. If the target is STATIC then the prediction step is skipped and current State Vector and Covariance Matrix are maintained as shown in the flow diagram in Figure 8
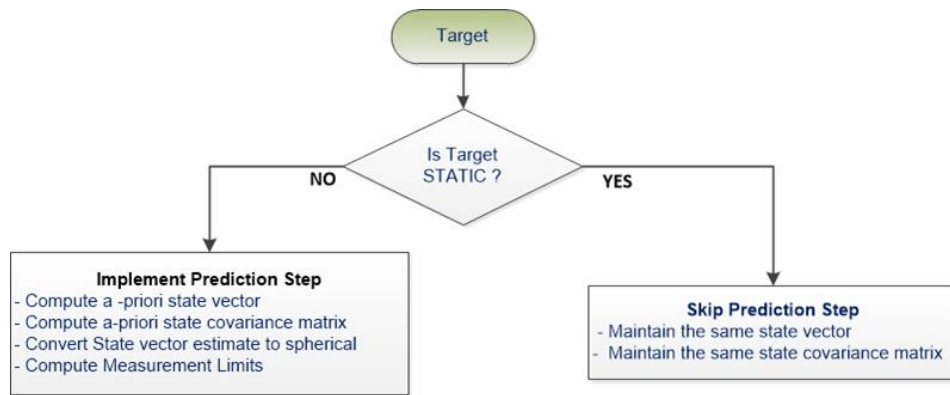
Figure 8 : Flow Diagram: Tracker Prediction

If the target is DYNAMIC then the following set of operations are done

(a) Compute the *a priori* state estimate $s_{apr}(n) = Fs(n-1)$

```
gtrack_matrixMultiply(SSIZE, SSIZE, 1, inst->F, inst->S_hat, inst->S_apriori_hat);
```

(b) Compute the *a priori* state vector covariance Matrix $P_{apr}(n) = FP(n-1)F^T + Q(n-1)$

```
gtrack_matrixMultiply(SSIZE, SSIZE, SSIZE, inst->F, inst->P_hat, temp1);
gtrack_matrixTransposeMultiply(SSIZE, SSIZE, SSIZE, temp1, inst->F, temp2);
gtrack_matrixAdd(SSIZE, SSIZE, temp2, inst->Q, temp3);
```

24

```
gtrack_matrixMakeSymmetrical(SSIZE, temp3, inst->P_apriori_hat);
```

(c) Transform the *a priori* state vector estimate from Cartesian-to-Spherical space i.e. $\boldsymbol{H}\left(s_{apr}(n)\right)$

$$\boldsymbol{H}\left(s_{apr}(n)\right) = \begin{bmatrix} \sqrt{x^2 + y^2 + z^2} \\ tan^{-1}(x,y) \\ tan^{-1}(z,\sqrt{x^2 + y^2}) \\ \dfrac{x\dot{x} + y\dot{y} + z\dot{z}}{\sqrt{x^2 + y^2 + z^2}} \end{bmatrix}$$

```
/* Convert from cartesian to spherical */
gtrack_cartesian2spherical(inst->currentStateVectorType, inst->S_apriori_hat, inst->H_s.array);
```

(d) Limits in the measurement space *H_limit* are computed using function *gtrack_calcMeasurementLimits()* for each track based on the user-configurable Gating parameters in Cartesian space (i.e. *<gatingParams.depth>, <gatingParams.width>, <gatingPrams.height>, <gatingParams.vel>)* and the estimated range from $H\left(s_{apr}(n)\right)$.

```
/* Compute measurement limits */
gtrack_calcMeasurementLimits(inst->H_s.vector.range,&inst->gatingParams->limits,&inst->H_limits.vector);
```

3.6. Association

The association function allows each tracking unit to indicate whether each measurement point is "close enough" (gating), and if it is, to provide the bidding value (scoring). Point is assigned to the highest bidder. Association function includes optional ghost tagging feature (only supported in the 2D version of the tracker). In some configurations (ex. 2D Wall mounted), we observed significant amount of multipath reflection points behind the target. The locations of the ghost targets are scene dependent and in general difficult to predict. Once enabled, the hypothetical ghost also bids for the association. If won, the point is tagged as the "ghost" point and will not be assigned to any target. The resultant ghost points will be ignored.

The function *gtrack_step()* calls the MODULE-level association function *gtrack_moduleAssociate()* to associate measurement points with existing targets. Within *gtrack_moduleAssociate(),* the function *gtrack_unitScore()* is called for each active track.

```
gtrack_moduleAssociate(h,….) {
    for (each active unit) {
    gtrack_unitScore(unit, ….)
    }
}
```

Association can be divided into three main functional blocks i.e. Gating, Scoring, and Assignment. All these are implemented in the *gtrack_unitScore()* function. *gtrack_unitScore()* will go through all the measurement points and will compute a *normalized distance* between the measurement point and the

centroid of the track for all the points that are within the tracks gating function. The assignment process assigns one measurement at a time to the closest track.

A number of FLAGs are set and utilized in the association process. A brief description of these is in the Table 4 below.

FLAGS	Applicable to	Description
isWithinLimits	point	Is the measurement point within Measurement Limits
isInsideGate	point	Is the measurement point within the Gating Limits
isGhostPoint	point	Is the measurement point coming from a potential Ghost Target
isDynamicPoint	point	Is the measurement point Dynamic
isUnitStatic	Target/Unit	Is the target static

Table 4 : Flags used in Tracker Association

Flowchart for the Association process is shown in Figure 9. Note that for simplicity the flowchart is only showing the main steps. Some additional details are provided in the description and code snippets below.
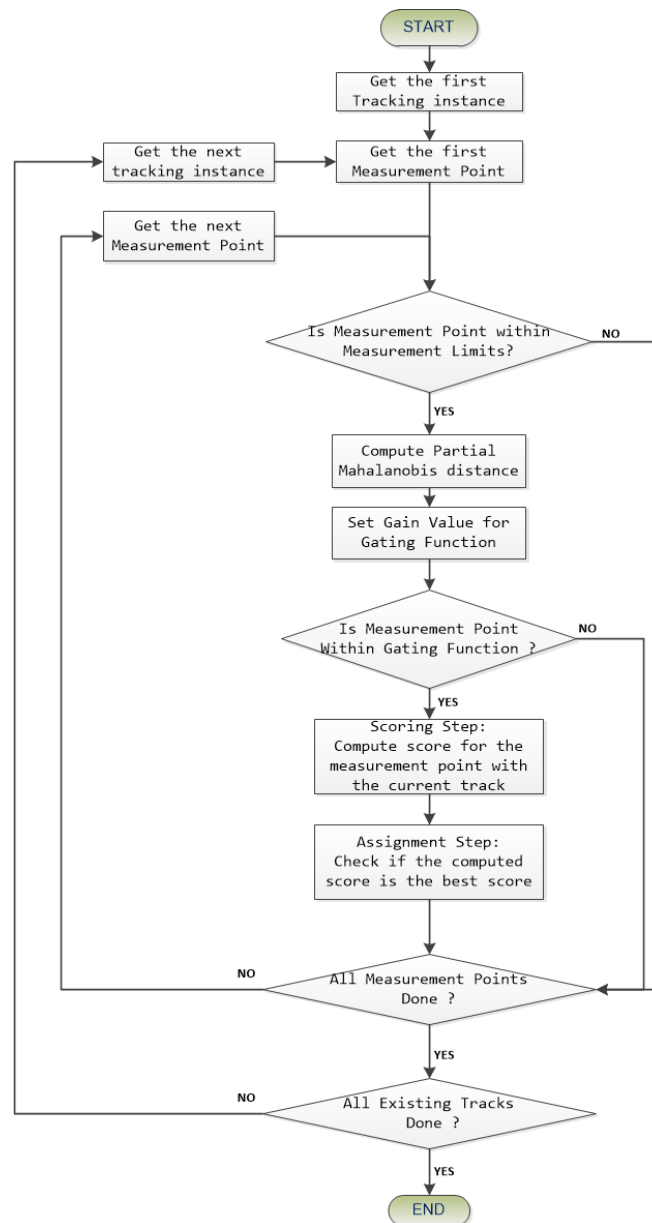
Figure 9. Flow Diagram: Association Steps

3.6.1. Gating steps

The Group Covariance Matrix (gC) is used to build an ellipsoid in 3D measurement space about the track centroid. This ellipsoid represents a gating function. The group Covariance Matrix (gC) is computed in the Tracker Update steps. The following steps are performed to determine if a measurement point is within the Gating Function of a given track

1) **Determine if the measurement point is within the Measurement limits (H_limits):** Note that the measurement limits (H_limits) were computed in the Prediction step described previously. The

27

measurement limit in the Doppler dimension is however modified based on the minimum value of measurement limit or measurement spread in Doppler dimension.

```
/* Add doppler agility */
limits.vector.doppler = GTRACK_MIN(2*inst->H_limits.vector.doppler, 2*inst->estSpread.vector.doppler);
```

Each measurement point is checked if it is within these measurement limits and the corresponding flag *isWithinLimits* is set accordingly

```
/* Gating */
 /* Limit check first */
 for(m=0; m<GTRACK_MEASUREMENT_VECTOR_SIZE; m++)
 {
    if(fabs(u_tilda.array[m]) > limits.array[m])
      {
       isWithinLimits = false;
       break;
       }
  }
```

2) **Compute Partial Mahalanobis distance:** If the measurement point *k* is within the measurement limits then Mahalanobis distance is computed between the track *j* and measurement point *k*. This Mahalanobis distance is calculated by ignoring the Doppler dimension hence we call it "partial Mahalanobis distance (*mdp*)". Note that computing the Partial Mahalanobis distance requires the inverse of the group covariance matrix **(*gC_inv*)** which is calculated in the Tracker Update step

```
/* Check whether point is within limits */
        if(isWithinLimits == true)
        {/* For the gating purposes we compute partial Mahalanobis distance, ignoring doppler */
            gtrack_computeMahalanobisPartial(u_tilda.array, inst->gC_inv, &mdp);
            ………………………………………………………………………………….
    }
```

3) **Set the threshold for the Gating:** For dynamic points, the gate value is set to the Gain value *<gatingParams.gain>* from the Gating parameters. However, for static point a gate value of 1 is used.

```
/* Gating Step */
 if(isDynamicPoint)
    gate = inst->G;
 else
    gate = 1.f;
```

Gain is defined as the amount the track dimensions can grow to create the gating function. Each of the dimensions X (width), Y (depth), Z (height) is multiplied by the gain to generate a gating function for the track. This value is set based on expected tracking errors, expected target maneuver and uncertainties in the detection layer. However, for static points we do not expect target maneuvering and hence we set the gate value to 1.

4) **Determine if the measurement point is within the Gating Function:** Partial Mahalanobis distance (*mdp*) is compared with the gate value and the flag *isInsideGate* is set accordingly

```
if(mdp < gate) {
/* Within the Gate, compute scoring function using all dimensions*/
  isInsideGate = true;
  …………………………………………………………………………………………….
  }
```

3.6.2. Scoring step

If the "Partial Mahalanobis Distance (*mdp*)" is less than the gate value then a bidding score is computed between measurement *j* and track *i* in the **Scoring** step. For the bidding score, the complete Mahalanobis distance (this time including the Doppler dimension) is computed. However, note that the Doppler difference is magnified x3.

Reason Doppler difference is magnified x3 when computing Mahalanobis Distance: Ideally, Doppler should be treated the same as the other geometric dimensions (range, azimuth, elevation). However, if there are two targets close together then the physical resolution of the system might not be enough to separate them. In these scenarios, we want to exploit any subtle differences in the Doppler dimensions between the targets. Hence these Doppler differences are x3 magnified.

```
if(mdp < gate) {
            /* Within the Gate, compute scoring function using all dimensions*/
            isInsideGate = true;
            /* Magnify doppler factor */
            u_tilda.vector.doppler = 3.0f*u_tilda.vector.doppler;
            gtrack_computeMahalanobis(u_tilda.array, inst->gC_inv, &md);
            score_gate = logdet + md;

            if(isGhostPoint) {
                /* We are competing with our own ghost */
                if(score_gate < score_ghost) {
                    score = score_gate;
                    tid = (uint8_t)inst->uid;
                    isGhostPoint = false;

                    if(isDynamicPoint)
                        inst->numAssosiatedPoints++;
                }
            }
            else {
                score = score_gate;
                tid = (uint8_t)inst->uid;

                if(isDynamicPoint)
                    inst->numAssosiatedPoints++;
            }
        }
```

3.6.3. Assignment step

It is possible that a measurement point can be within the gating function of multiple tracks. In the assignment step, a measurement point is assigned to the closest track. This is done by maintaining a best bidding score (*bestScore*) and the corresponding track index **(*bestInd*)** for each measurement.

The initial value of *bestScore* is set to infinity. If the current bidding score (between measurement *k* with track *j*) is smaller than the *bestScore[k]* it is registered as the new *bestScore[k]* and the corresponding track ID is noted down in *bestInd[k].* After the *gtrack_unitScore()* is called for all the active tracks, we will have *bestScore* value and *bestInd* containing the track index of the closest track for the measurement point

```
if((isInsideGate == true) || (isGhostPoint == true))
    {
        if(bestInd[n] > GTRACK_ID_GHOST_POINT_BEHIND)
        {
            /* No competition, we won */
            /* Register our score, and the index */
            bestScore[n] = score;
            bestInd[n] = tid;
            if(isUnitStatic) {
                /* Set the static indication */
                isStatic[n>>3] |= (1<<(n & 0x0007));
            }
            point[n].vector.doppler = rvOut;
        }
        else {
/* We have a competitor. Unless we have an exception, the unique indication will be cleared */
            clearUniqueInd = true;
            /* Read competitor's static indication */
            staticIndication = (isStatic[n>>3] & (1<<(n & 0x0007))) != 0;

            if(score < bestScore[n]) {
/* We won the competition */
/* Check whether we need to clear unique indicator */
/* We don't clear the indicator when dynamic beats static or dynamic beats "futher away" ghost */
                if((isUnitStatic == false) &&              // Dynamic target is a winner
                   ((staticIndication == true) ||          // Dynamic target wins against static one
                    (bestInd[n] == GTRACK_ID_GHOST_POINT_BEHIND)))    // Dynamic target wins agains further away ghost
                {
                    clearUniqueInd = false;
                }

                /* Register our score, and the index */
                bestScore[n] = score;
                bestInd[n] = tid;
                point[n].vector.doppler = rvOut;

            }
            else {
/* We lost */
/* Check whether we need to clear the unqie indication */
/* We don't clear the indicator we lose to dynamic point and we aren't that sure
/* (i.e: we are static, we are ghost point, we are outside of inner gate */
                if((staticIndication == false) &&              // Loss to dynamic target
                   ((isUnitStatic == true) ||                   // Static loses to dynamic
                    (tid == GTRACK_ID_GHOST_POINT_BEHIND)))     // "futher away" ghost looses to dynamic
                {
                    clearUniqueInd = false;
                }
            }

  If required, clear unque indicator */
            if(clearUniqueInd)
                isUnique[n>>3] &= ~(1<<(n & 0x0007));
        }
    }
```

3.7. Track Allocation

Points that are not assigned go through the Allocate function. The function *gtrack_step()* calls MODULE-level allocation function *gtrack_moduleAllocate()* to allocate new targets for the non-associated measurement points. The flow chart of the allocation step is shown in Figure 10
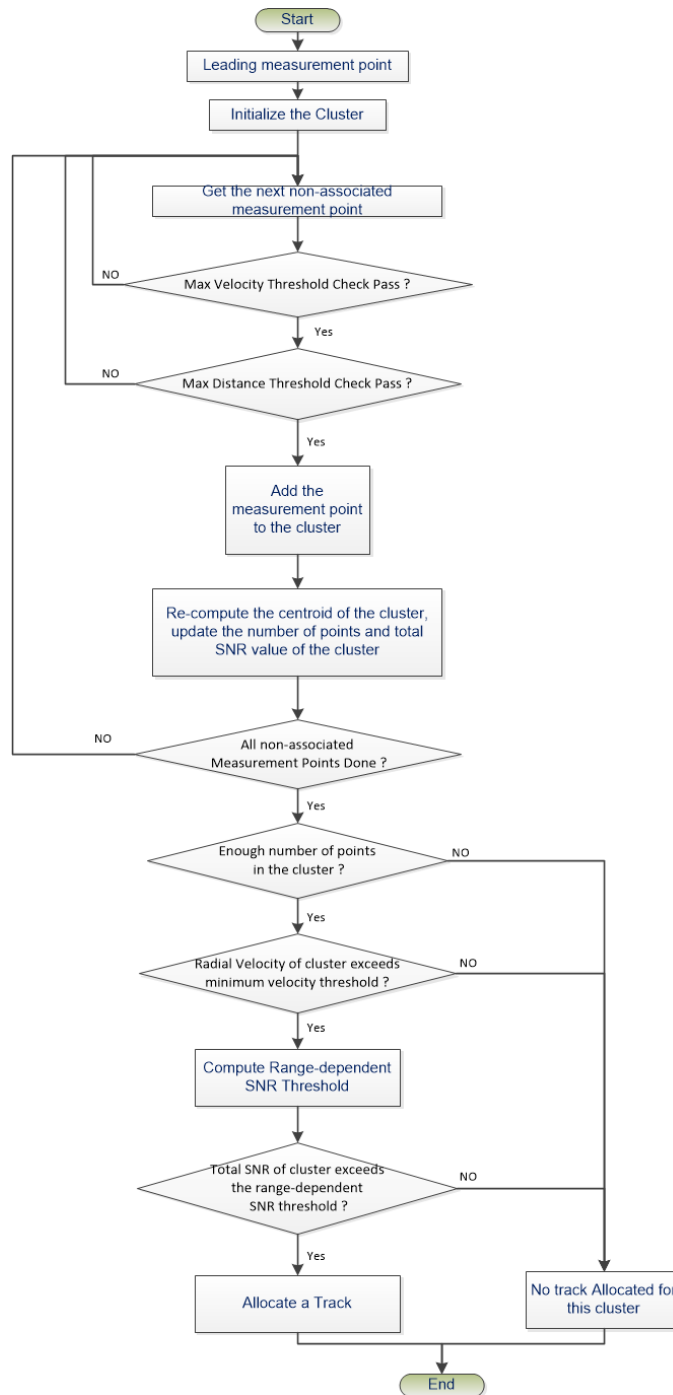

Figure 10: Flow Diagram: Track Allocation Steps

The allocation process can be divided into two main steps

(a) **Building a candidate-set (cluster) of points:** Points are first joined into a set based on their proximity in measurement coordinates. Each set becomes a candidate for allocation decision.

(b) **Check if the candidate-set (cluster) qualifies the track allocation criteria:** A candidate set has to pass multiple tests to become a new track. Once passed, the new tracking unit is allocated.

3.7.1. Building a candidate set

In the **Allocation** step, a leading measurement is used for initialization of a cluster/candidate set. For each of the subsequent non-associated measurements points the following steps are performed

1) **Maximum velocity threshold check :** The radial velocity difference between the current measurement and allocation set is compared with *<allocationParams.maxVelThre>*

2) **Maximum distance threshold check :** If maximum velocity threshold check is passed for the measurement point then a 3D geometric distance between the current measurement point and the mean of cluster is calculated (using *gtrack_calcDistance())* and compared with the *<allocationParams.maxDistanceThre>* in the allocation parameters.

3) **Centroid Update:** If the measurement passes the above checks, then it is added to the cluster and the centroid is recalculated. Number of points and the *totalSNR* value for the cluster is updated.

```
mCurrent.vector = point[k].vector;
mCurrent.vector.doppler = gtrack_unrollRadialVelocity(inst->params.maxRadialVelocity,
mCenter.vector.doppler, mCurrent.vector.doppler);
if(fabsf(mCurrent.vector.doppler - mCenter.vector.doppler) < inst->params.allocationParams.maxVelThre)
   {
   dist = gtrack_calcDistance(&mCenter.vector, &mCurrent.vector);
        if(sqrtf(dist) < inst->params.allocationParams.maxDistanceThre)
            {
            inst->allocIndexCurrent[allocNum] = k;
            allocNum++;
            allocSNR +=point[k].snr;
            // Update the centroid
             gtrack_vectorAdd(GTRACK_MEASUREMENT_VECTOR_SIZE, mCurrent.array, mSum.array, mSum.array);
             gtrack_vectorScalarMul(GTRACK_MEASUREMENT_VECTOR_SIZE, mSum.array, 1.0f/(float)allocNum,
mCenter.array);
            }
   }
```

3.7.2. Qualifying tests to form a track

Once a candidate cluster is formed then a few additional qualifying steps are done for the cluster to qualify it for a track

1) **Comparison with Points Threshold:** The number of points in the cluster is compared with *< allocationParams.pointsThre>*

2) **Comparison with minimum velocity threshold:** The radial velocity of the cluster is compared with *< allocationParams.velocityThre>*

```
if( (set.numAllocatedPoints >= inst->params.allocationParams.pointsThre) &&
            (fabsf(set.mCenter.vector.doppler) >= inst->params.allocationParams.velocityThre) )
```

3) **Computation of Range-Dependent SNR threshold:** Radial range of the current cluster is used to compute the range-dependent SNR threshold based on the *< allocationParams.snrThre>* (if the target is obscured then *< allocationParams.snrThreObscured>*) . A target is said to obscured if the target is outside the static zone(*isInside* Flag) or if it is behind an existing track (*isBehind* Flag). The computation of the range-dependent SNR threshold is explained in Section 3.7.3

4) **Comparison with SNR threshold:** The total SNR is compared with computed range dependent SNR threshold

5) **Allocate a new track:** Once all the above checks are passed a new tracker unit is started by calling the function *gtrack_unitStart().*

```
if(set.totalSNR > snrThreshold)
  {
  /* Associate points with new uid  */
    for(k=0; k < set.numAllocatedPoints; k++)
    inst->bestIndex[inst->allocIndexStored[k]] = (uint8_t)tElemFree->data;

  /* Allocate new tracker */
    inst->targetNumTotal ++;
    inst->targetNumCurrent ++;
    tElemFree = gtrack_listDequeue(&inst->freeList);

gtrack_unitStart(inst->hTrack[tElemFree->data], inst->heartBeat, inst->targetNumTotal,
&set.mCenter.vector, set.numAllocatedPoints, isBehind);
    gtrack_listEnqueue(&inst->activeList, tElemFree);
  }
```

3.7.3. Computation of range-dependent SNR threshold

SNR of targets is range-dependent and scales by a factor of $(1/R)^4$ where R is the radial distance. In other words, larger the radial distance of the target from the sensor, lower is the SNR.

As mentioned in the previous section, one of the qualifying criteria for track allocation is that the total SNR of the cluster needs to exceed a SNR threshold. Given that SNR is range-dependent, it makes sense to also scale the user-defined snr thresholds i.e. *< allocationParams.snrThre>* , *< allocationParams. snrThreObscured>* based on the targets radial distance. The user specifies these values based on the expected total linear SNR from the target at *R = 6* m. If the target is at a different distance than 6 meter, then these values need to be scaled accordingly.

In the current implementation of the group tracker, three radial-zones are defined and SNR-thresholds are computed differently in each of these zones. These zones are hard-coded through *#define* statements

```
#define GTRACK_NOMINAL_ALLOCATION_RANGE        (6.0f) /* Range for allocation SNR scaling */
```

```
#define GTARCK_MAX_SNR_RANGE                 (2.5f) /* Range for SNR maximum */
#define GTARCK_FIXED_SNR_RANGE               (1.0f) /* Range for SNR fixed */
```

Figure 11 shows an example plot of the range-dependent SNR threshold for a target that is not obscured. The different colors highlight the different zones. This function is formulated to have a empirically chosen maxima at R = 2.5 m given by

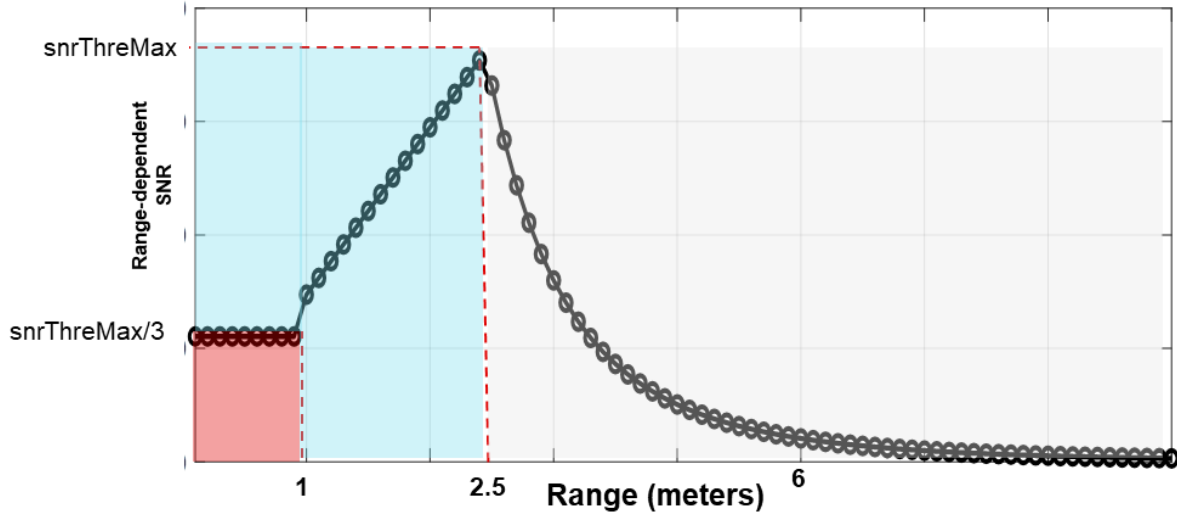$$snrThreMax = \left(\frac{6}{2.5}\right)^4 (allocationParams.snrThre)$$

Figure 11. Plot showing the range-dependent SNR threshold as a function of range. This plot is for a target that is not obscured. SNR threshold has a maxima at 2.5 m, then is linearly scaled to 1/3ʳᵈ of the max value to a range of 1 m and then stays constant

The range-dependent SNR values in different zones are explained below and summarized in Table 5.

- **Zone 1 ($0 < R < 1$)**

  Based on several empirical observations, the SNR values from human targets doesn't significantly increase for radial distances of less than 1 meter. Hence in this region , a constant SNR threshold value set to 1/3ʳᵈ of the Maximum value is used i.e. $snrThreMax/3$

- **Zone 2 ($1 \le R < 2.5$)**

  Range-dependent SNR threshold is linearly scaled from $snrThreMax$ to $snrThreMax/3$ for a target that is not obscured

  $$snrThresh = R * \left(\frac{(snrThreMax - snrThreMax/3)}{(2.5 - 1)}\right)$$

  For an obscured target it is given as

  $$\left(\frac{6}{R}\right)^4 (allocationParams.snrThreObscured)$$

- **Zone 3 ($1 \le R < 2.5$)**

For radial distances greater than 2.5 meter, the range-dependent SNR thresholds are scaled by a factor of $\left(\frac{6}{R}\right)^4$

Radial Zone	Range-Dependent SNR Threshold	
	Target is not obscured	Target is obscured
$0 < R < 1$	$snrThreMax/3$	$snrThreMax/3$
$1 \leq R < 2.5$	$R * \left(\frac{(snrThreMax - snrThreMax/3)}{(2.5-1)}\right)$	$\left(\frac{6}{R}\right)^4 * (allocationParams.snrThreObscured)$
$R \geq 2.5$	$\left(\frac{6}{R}\right)^4 * (allocationParams.snrThre)$	$\left(\frac{6}{R}\right)^4 * (allocationParams.snrThreObscured)$

Table 5. Range-dependent SNR in different radial zones

3.8. Track Update

Tracks are updated based on the set of associated points. We compute the innovation, Kalman gain, a-posteriori state vector, and error covariance. In addition to classic EKF, the error covariance calculation includes group dispersion in measurement noise covariance matrix. If presence detection is enabled, the algorithm determines whether there is any target within the configured occupancy zone.

The function *gtrack_step()* calls MODULE-level allocation function *gtrack_moduleUpdate().* For each active track, the unit-level function *gtrack_unitUpdate ()* is called to update the track based on the set of associated measurements. Tracks that have transitioned to the FREE state are deleted by calling *gtrack_unitStop ()* and the resources returned back to the parent MODULE.

```
gtrack_moduleUpdate(h,….) {
        for (each active unit) {
                state = gtrack_unitUpdate(unit, ….)
                if (state==FREE) {
                gtrack_unitStop(unit,…..)
                }
        }
}
```

A brief description of some of the configuration parameters used within the tracker Update module are described in Table 6

No	Parameters	Description
1	confidenceLevelThre	Each track instance has an associated confidence level. If the track is STATIC AND below the *confidenceLevelThre* then the lifespan of the track is set to *exit2freeThre* value
2	minVelocityStopNoPoints (In Cartesian Co-ordinates)	Velocity threshold to transition to STATIC state: This threshold is used when no measurement points are available
3	minVelocityStopNoDyn (In Cartesian Co-ordinates)	Velocity threshold to transition to STATIC state: This threshold is used when no dynamic points are available

4	minStaticVelocitySlow (In Cartesian Co-ordinates)	Minimal velocity threshold to start slowing down: This threshold is used when no dynamic points are available
5	estSpreadAlpha	The alpha parameter determines how much weight should we give to the current measurement spread vs the running average measurement spread

<div align="center">**Table 6 : Tracker Update Parameters**</div>

3.8.1.   Tracker update flow diagram

The flow diagram of the Tracker Update functional block is shown in Figure 12. The tracker State vector is updated depending on the target status and the associated measurement points as shown in Table 7. Note that the standard Kalman filter update equations are only applied if there are dynamic points associated with a given track (case 4).

CASE	Description	Action
Case 1	Target is dynamic AND there are no measurement points associated with target	1) Check If ($xyVelocity < minVelocityStopNoPoints$) 2) If TRUE, we assume that the target has most likely stopped. Target is static => force zero velocity and zero acceleration 3) If FALSE we assume this is a moving target which is obscured by another target. Target is moving => force zero acceleration
Case 2	Target is dynamic AND only static points associated with the target	1) If ($xyVelocity < minVelocityStopNoDyn$), we assume that the target is moving very slowly. We completely STOP the target and change the target status to STATIC. Target is static => force zero velocity and zero acceleration 2) If ($xyVelocity < minStaticVelocitySlow$), then we SLOW DOWN the target but keep it dynamic. Target is slowing down => decrease the velocity to half and force zero acceleration 3) If the above two conditions do not apply, we keep the target MOVING and decrease the confidence level of the track as this is behavior is not expected
Case 3	Target is static AND only static points associated with the target	This is expected behavior as Static Target gets associated with static point. Update Confidence Metric. No need to update the state vector and covariance matrix as the target is static
Case 4	Target is dynamic AND there are some dynamic points associated with the target	Implement EKF Tracker Update Equations - Compute Measurement Covariance Matrix - Update Group Dispersion Matrix - Update Gating Function

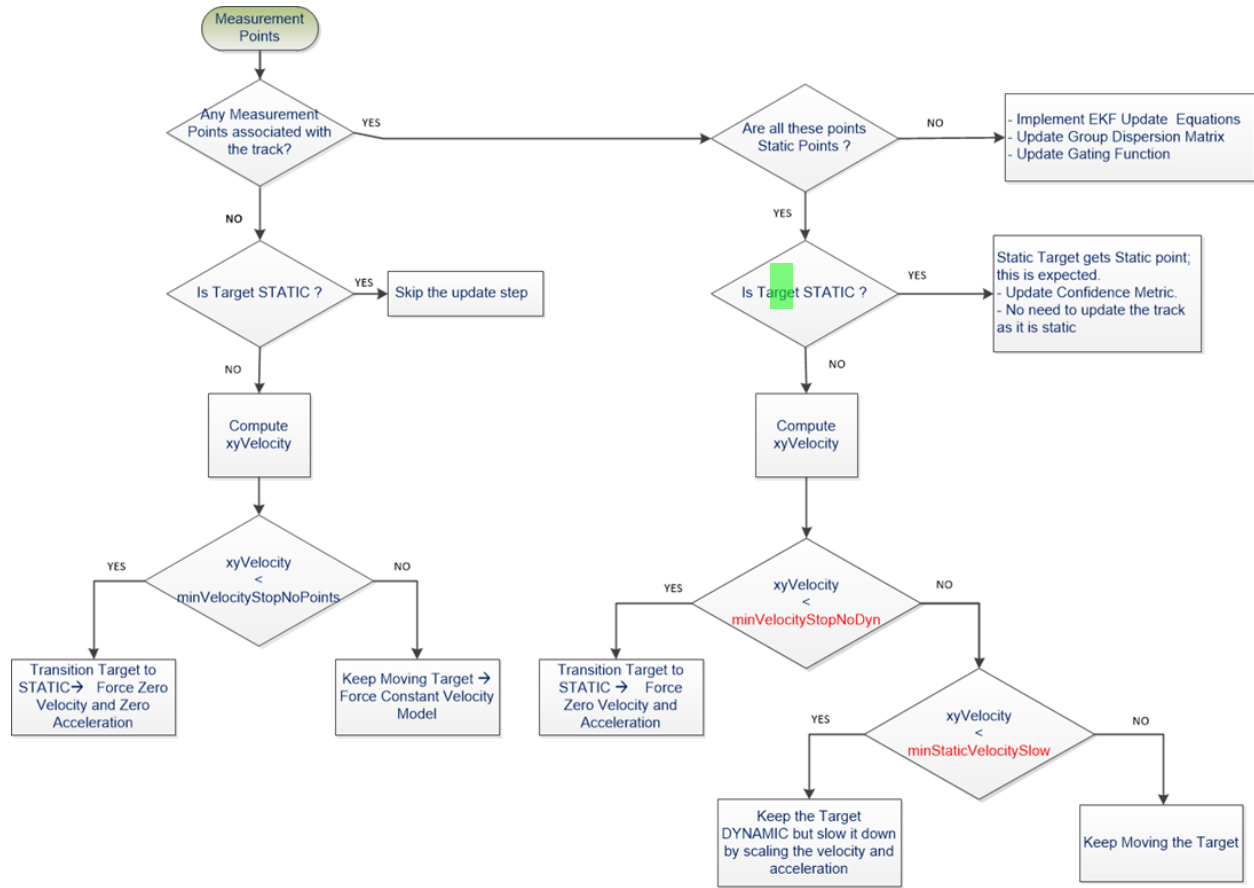<div align="center">**Table 7: Tracker State Vector Update**</div>

Figure 12 : Flow Diagram: Tracker Update Steps

3.8.2. Tracker Update when No dynamic points associated

In this sub-section, we describe the tracker update steps when there are no dynamic points associated with the tracker. This refers to Cases 1 and 2 in Table 7. From Figure 12, it can be seen that in these cases, the target velocity in Cartesian coordinates in the x-y plane (*xyVelocity*) is computed and based on the minimum velocity thresholds (*minVelocityStopNoPoints, minVelocityStopNoDynamic, minVelocitySlowDown)* the state vector is updated.

a) **Determine the velocity of the target in x-y coordinates** (*xyVelocity*)

Using the a-priori state vector $s_{apr}(n)$ of the target the velocity is converted from the sensor-to-World co-ordinates using the function *gtrack_censor2world()* and the magnitude of the velocity in x-y coordinates (*xyVelocity*) is calculated

```
/* Compute cartesian velocity */
if(inst->transormParams->transformationRequired) {
gtrack_censor2world((GTRACK_cartesian_position *)&inst->S_apriori_hat[GTRACK_VEL_DIMENSION*inst-
>stateVectorDimNum], inst->transormParams, &velW);
   velx = velW.posX;
   vely = velW.posY;
```

```
    }
    else
    {
    velx = inst->S_apriori_hat[GTRACK_VEL_DIMENSION*inst->stateVectorDimNum];
    vely = inst->S_apriori_hat[GTRACK_VEL_DIMENSION*inst->stateVectorDimNum + 1];
    }
    xyVelocity = sqrtf(velx*velx + vely*vely);
}
```

b) Updating State Vectors

The state vector of a track is updated based on the *xyVelocity* and the minimum velocity thresholds defined in Table 6. The overall flow diagram is shown in Figure 12 and the implementation code is below

```
if(myPointNum == 0) {
    /* Erasures handling: no measurements available */
    if(inst->isTargetStatic == false) {
    /* Check whether we need to transition to static condition */
        if(xyVelocity < inst->minVelocityStopNoPoints) {
        /* Yes. Force zero velocity/zero acceleration */
            for(n=0; n<inst->stateVectorDimNum; n++) {
                inst->S_apriori_hat[GTRACK_VEL_DIMENSION*inst->stateVectorDimNum + n] = 0.f;
                inst->S_apriori_hat[GTRACK_ACC_DIMENSION*inst->stateVectorDimNum + n] = 0.f;
            }
            /* Transition to static */
            inst->isTargetStatic = true;
        }
        else {
        /* No. Keep moving => force constant velocity model */
        /* Force zero acceleration */
         for(n=0; n<inst->stateVectorDimNum; n++) {
             inst->S_apriori_hat[GTRACK_ACC_DIMENSION*inst->stateVectorDimNum + n] = 0.f;
            }
        * This is not expected */
            inst->confidenceLevel = 0.99f*inst->confidenceLevel;
            }
        }
    }
    else if(myDynamicPointNum == 0) {
    /* We only have static points */
        if(inst->isTargetStatic == true) {
            /* Static target gets static point: this is expected */
            inst->confidenceLevel = 0.99f*inst->confidenceLevel + 0.01f;
        }
        else {
            /* Check whether we need to transition to static condition */
            if(xyVelocity < inst->minVelocityStopNoDyn) {
                /* Yes. Force zero velocity/zero acceleration */
                for(n=0; n<inst->stateVectorDimNum; n++) {
                    inst->S_apriori_hat[GTRACK_VEL_DIMENSION*inst->stateVectorDimNum + n] = 0.f;
                    inst->S_apriori_hat[GTRACK_ACC_DIMENSION*inst->stateVectorDimNum + n] = 0.f;
                }
                /* Transition to static */
                inst->isTargetStatic = true;
                if(myStaticPointNum > 3) {
                    /* Confirmed with a lot of static points: confidence boost */
                    inst->confidenceLevel = 0.95f*inst->confidenceLevel + 0.05f;
                }
            }
            else if(xyVelocity < inst->minStaticVelocitySlow) {
                /* Slow down, keep dynamic */
                for(n=0; n<inst->stateVectorDimNum; n++) {
                    inst->S_apriori_hat[GTRACK_VEL_DIMENSION*inst->stateVectorDimNum + n] *= 0.5f;
                    inst->S_apriori_hat[GTRACK_ACC_DIMENSION*inst->stateVectorDimNum + n] = 0.f;
                }
            }
            else {
                /* Keep moving */
                /* This is not expected */
                inst->confidenceLevel = 0.99f*inst->confidenceLevel;
            }
        }
    }
```

c) **Target transitions (DYNAMIC to STATIC)**

A target/track is transitioned to STATIC when either no points or only static points available.

- If there are NO measurement points associated with a track and `if`
  if(xyVelocity < minVelocityStopNoPoints) then declare the target as STATIC
- If there are NO dynamic points associated with a track and
  if(xyVelocity < minVelocityStopNoDynamic), the target status is changed to STATIC.
- If *myGoodPointNum* is greater than 3, the target is declared as DYNAMIC. The variable *myGoodPointNum* is computed within *gtrack_unitUpdate( )* and is equal to the number of measurement points associated with the track that are both dynamic (i.e. Doppler is greater than zero) AND unique (i.e. point was within a gate of a single target)

3.8.3. Tracker Update when dynamic points associated

This refers to Case 4 in Table 7. The tracker update steps are described in Section 2.5 and their implementation is outlined below.

A. *Updating the Estimated of number of points in a target ($\widehat{N}$)*

As estimate of $\widehat{N}$ (Estimated number of elements in target tracked by the given track) is required when computing the measurement error covariance matrix for the centroid used for Kalman Update. $\widehat{N}$ is estimated as below

(a) **Determine N_A (number of "good" measurement points *myGoodPointNum*):**
    This is defined as the number of measurement points associated with the track that are both dynamic (i.e. Doppler is greater than zero) AND unique (i.e. point is within a gate of a single target)

(b) **Update the estimate of $\widehat{N}$:**
    Depending on the flag *isEnablePointNumberEstimation*, $\widehat{N}$ is estimated as in the table below

isEnablePointNumberEstimation == TRUE	isEnablePointNumberEstimation == FALSE	Description
IF $(N_A > \widehat{N}_{n-1})$ $\widehat{N}_n = N_A$ Else $\widehat{N}_n = (1 - \alpha_N)\widehat{N}_{n-1} + \alpha_N N_A$ End IF $(\widehat{N}_n < pointsThre)$ $\widehat{N}_n = pointsThre$	$\widehat{N}_n = \max(100, N_A)$	- $\alpha_N = 0.9$ in the current implementation - $\widehat{N}_n$ is the estimate at time n - $\widehat{N}_{n-1}$ is the estimate at previous time instance n-1 - N_A is the number of "good" associated measurement points - $pointsThre$ is the number of points set in the Allocation Parameters

B. *Updating Measurement Spread*

The measurement spread (*estSpread*) is required for computing the Measurement Noise Covariance Matrix (Rm). Using the function *gtrack_calcDim()*, the target dimensions are also estimated based on *estSpread* and centroid range. Note that the measurement spread is only updated if N_A >1 (associated Good measurement points is greater than 1).

The measurement spread is updated as below
(a) Find maximums and minimums for each of the dimensions (i.e. range, azimuth, elevation, Doppler) from the measurements points and compute the spread (Max – Min)
(b) Check if the computed spreads are between 1x and 2x of the measurement limits (*H_limit*). If not, constrain these to be within these limits i.e. $spread \in [H_{limits} \quad 2H_{limits}]$
(c) Update the *estSpread* by exponential filtering using the alpha parameter i.e. *estSpreadAlpha*

```
/* Update measurement spread if we have 2 or more good points */
    if(myGoodPointNum > 1) {
        for(m = 0; m < MSIZE; m++) {
            spread = u_max[m] - u_min[m];
            /* Unbiased spread estimation */
            spread = spread*(myGoodPointNum+1)/(myGoodPointNum-1);
            /* computed spread estimation shall be between 1x and 2x of configured limits */
            if(spread > 2*inst->H_limits.array[m])
                spread = 2*inst->H_limits.array[m];
            if(spread < inst->H_limits.array[m])
                spread = inst->H_limits.array[m];
            if(spread > inst->estSpread.array[m])
                inst->estSpread.array[m] = spread;
            else {
                inst->estSpread.array[m] = (1.0f-inst->estSpreadAlpha)*inst->estSpread.array[m] +
inst->estSpreadAlpha*spread;
            }
        }
        gtrack_calcDim(inst->estSpread.array, inst->uCenter.vector.range, inst->estDim);
    }
```

C. *Computation of the Measurement Noise Covariance Matrix (R_m)*

R_m is the diagonal matrix of the measurement variances. These variances are estimated iteratively by computing the estimated spread (*estSpread*) of the associated point values across each measurement dimension. The following assumptions are made (i) Uniform distribution of measurement values within the target dimension. (ii) "2 sigma" spread coverage to compute the variances.

```
if(myGoodPointNum) {
 /* Compute Rm, Measurement Noise covariance matrix */
    if(var == 0) {
       for(m = 0; m < MSIZE; m++) {
          /* spread is covered by 2 sigmas */
          sigma = inst->estSpread.array[m]/2;
          uvar_mean.array[m] = sigma*sigma;
          }
       }
 gtrack_matrixSetDiag(GTRACK_MEASUREMENT_VECTOR_SIZE, uvar_mean.array, Rm);
 }
```

D. _Update Track Confidence Level_

Each track instance has an associated confidence level that has a value between 0 and 1. If the track is declared as STATIC AND its confidence level is below the _confidenceLevelThre_ then the lifespan of the track is set to _<allocationPrams.exit2freeThre>_ value.

```
/* Update target confidence level */
if(myGoodPointNum > 3) {
    if(inst->numAssosiatedPoints)
    confidenceUpdate = (float)(myGoodPointNum + (myDynamicPointNum-myGoodPointNum)/2)/inst-
>numAssosiatedPoints;
        else
    confidenceUpdate = 0;
inst->confidenceLevel = (1.0f-GTRACK_CONFIDENCE_ALPHA)*inst->confidenceLevel +
GTRACK_CONFIDENCE_ALPHA*confidenceUpdate;
}
```

E. _Updating Group Dispersion Matrix_

For the measurement vector, $u(n) = [r(n) \; \theta(n) \; \varphi(n) \; \dot{r}(n) \;]^T$, the dispersion matrix is given by

$$D = \begin{bmatrix} d_{rr}^2 & d_{r\theta}^2 & d_{r\varphi}^2 & d_{r\dot{r}}^2 \\ d_{\theta r}^2 & d_{\theta\theta}^2 & d_{\theta\varphi}^2 & d_{\theta\dot{r}}^2 \\ d_{\varphi r}^2 & d_{\varphi\theta}^2 & d_{\varphi\varphi}^2 & d_{\varphi\dot{r}}^2 \\ d_{\dot{r}r}^2 & d_{\dot{r}\theta}^2 & d_{\dot{r}\varphi}^2 & d_{\dot{r}\dot{r}}^2 \end{bmatrix}$$

Where $d_{r\theta}^2 = \frac{1}{N}\sum_{i=1}^{N}(a_i - \bar{a})(b_i - \bar{b})$ and $a, b \in \{r, \; \theta, \; \varphi, \; \dot{r}\}$ and $\bar{a}, \bar{b}$ are the means computed for a given group of measurements.

The group dispersion Matrix is estimated recursively as $\boldsymbol{C}_D = [1 - \alpha]\boldsymbol{C}_D(n - 1) + \alpha\boldsymbol{D}$ where $\alpha = \frac{N_A}{\hat{N}}$. Note that when the number of measurements (N_A) associated with a track are close to our estimated number of elements $\hat{N}$ then we give a higher weight to the estimated $\boldsymbol{D}$.

```
/* Update Group Dispersion gD matrix */
    if(myGoodPointNum) {
        /* D is the new dispersion matrix, MSIZExMSIZE */
        for(n = 0; n < num; n++) {
            if(pInd[n] == GTRACK_ID_RESERVED_GOOD_POINT) {
                /* restore the good point markings */
                pInd[n] = inst->uid;
                /* Accumulate covariance from all associated points */
                gtrack_matrixCovAcc(GTRACK_MEASUREMENT_VECTOR_SIZE, D, point[n].array, inst-
>uCenter.array);
            }
        }
    }
    if(myGoodPointNum > GTRACK_MIN_POINTS_TO_UPDATE_DISPERSION) {
/* Normalize it */
        gtrack_matrixCovNormalize(GTRACK_MEASUREMENT_VECTOR_SIZE, D, myGoodPointNum);
/* Update persistant group dispersion based on instantaneous D */
/* The factor alpha goes from maximum (1.f) at the first allocation down to minimum of 0.1f once the
target is observed for the long time */
```

```
        alpha = ((float)myGoodPointNum)/inst->estNumOfPoints;
        /*  inst->gD = (1-alpha)*inst->gD + alpha*D */
        gtrack_matrixCovFilter(GTRACK_MEASUREMENT_VECTOR_SIZE, inst->gD, D, alpha);
    }
```

F. _Updating the Gating Function_

In tracking, a gate is formed around the predicted track position. The purpose of gating is to eliminate unlikely measurement to track pairing. The gating function determines the amount of innovations (difference between the measurements and predicted track position) that is acceptable given the current state of uncertainty that exists in the current track.

The group residual covariance matrix is computed to measure this uncertainty. This group covariance matrix is calculated in the _trackUnitUpdate()_ function

```
/* Compute groupCovariance gC (will be used in gating) */
/* We will use ellipsoidal gating, that acounts for the dispersion of the group, target
maneuver, and measurement noise */
/* gC = gD + JPJ + Rm */
    gtrack_matrixAdd(MSIZE, MSIZE, JPJ, Rm, temp1);
    gtrack_matrixAdd(MSIZE, MSIZE, temp1, inst->gD, inst->gC);
/* Compute inverse of group innovation */
    gtrack_matrixInv(inst->gC, &inst->gC_det, inst->gC_inv);
```

3.9. Track Maintenance (State Machine)

The function _gtrack_unitEvent()_ is called within _gtrack_unit_update()_ to update the tracker state. A tracker instance can be in one of three states i.e. DETECT, ACTIVE, or FREE.

```
typedef enum {
    /**  @brief Free (not allocated) */
    TRACK_STATE_FREE = 0,
    /**  @brief INIT */
    TRACK_STATE_INIT,
    /**  @brief DETECTION State */
    TRACK_STATE_DETECTION,
    /**  @brief ACTIVE State */
    TRACK_STATE_ACTIVE
} TrackState;
```

A brief description of the states is below

DETECT: A track is in the DETECT state, when it gets allocated. Note that allocating a track does not imply that it would be counted as a target.

ACTIVE: When a track instance transitions to an ACTIVE state it is counted as a track. An active track can have several associated conditions depending on the dynamic/static status of the track and its location.

Normal Condition: Track is Dynamic AND inside the static boundary box

Static Condition: Track is Static AND inside the static boundary box
Exit Condition: Track with MISS event that is outside the static boundary box
Sleep Condition: Track with MISS event that is inside the static boundary box

FREE: When a track is deleted it is in the FREE state.

3.9.1. State Transitions

The transition from one state to another is determined by the thresholds set in the state transition parameters. The state transition diagram and the associated thresholds are shown Figure 13. Note that these thresholds are user-configurable.
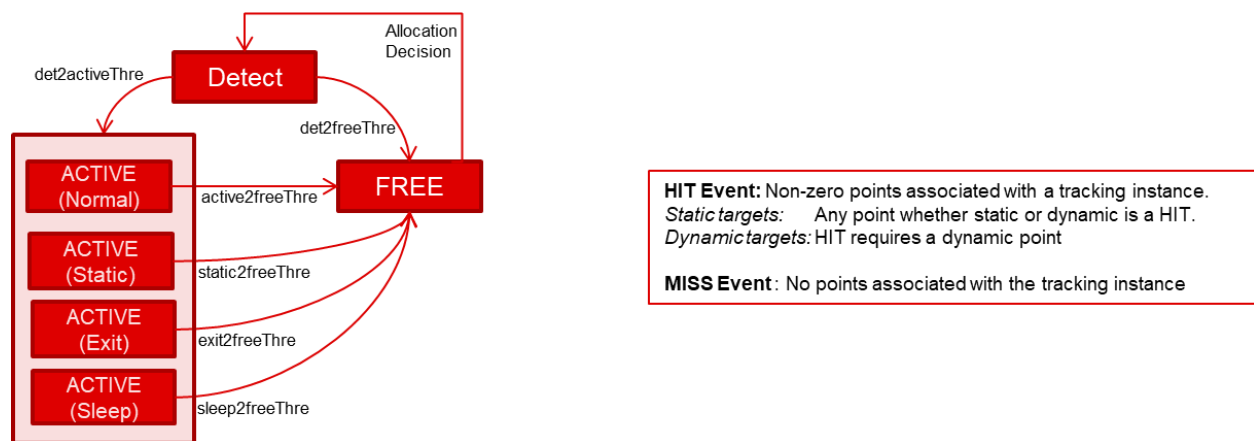
Figure 13: State Transition Diagram

At each time instance, the track gets either a HIT or a MISS event. For each of these state transitions, separate counters are maintained that count the number of HIT or MISS events. The state transition occurs after a counter value exceeds the corresponding threshold values. These transitions are explained below

FREE→ DETECT
Once a track is in the FREE state, the transition to DETECT state is made by the allocation decision. See section 3.7 for the details on the track allocation decision.

DETECT→ ACTIVE
The counter *detect2activeCount* is incremented when a track in DETECT state encounters a HIT event.
A track will transition from DETECT->ACTIVE when a *<stateParams.det2activeThre>* number of consecutive HIT events happen.

DETECT→ FREE
The counter *detect2freeCount* is incremented when a track in DETECT state encounters a MISS event.

A track will transition from DETECT->FREE when a *< stateParams.det2freeThre>* number of consecutive MISS events happen.

ACTIVE→ FREE

Once in ACTIVE state, the target can be in Normal, Exit, Sleep or Static condition. The counter *active2freeCount* is incremented when an ACTIVE track encounters a MISS event. The handling of the MISS event is shown in the flow chart in Figure 14. *active2freeCount* is checked against a threshold value where the threshold value depends on the condition (i.e. Exit, Normal, Static) of the track. Note that on a HIT event, *active2freeCount* is cleared.

- If the target is in the "static zone" AND the target is in STATIC condition then the assumption is made that the reason we don't have detection is because we removed them as "static clutter". In this case we compare the miss count with [larger] *< stateParams.static2freeThre>* threshold to "extend the life expectation" of the static targets. Note that the "static zone" is user-configurable and is set through the scenery parameters described in Section 4.2.1.
- If the target is outside the static zone, then the assumption is made that the reason we didn't get the points is that target is exiting. In this case, we use *< stateParams.exit2freeThre>* threshold to quickly free the exiting targets.
- Otherwise, (meaning target is in the "static zone", but has non-zero motion in radial projection) we assume that the reason of not having detections is that target got obscured by other targets. In this case, we continue target motion according to the model, and use *< stateParams.active2freeThre>* threshold.
- In addition to the *active2freeCount*, a separate counter *sleep2freeCount* is maintained for STATIC condition. This counter is checked against the *< stateParams.sleep2freeThre>.* The flow chart for this is shown in Figure 15
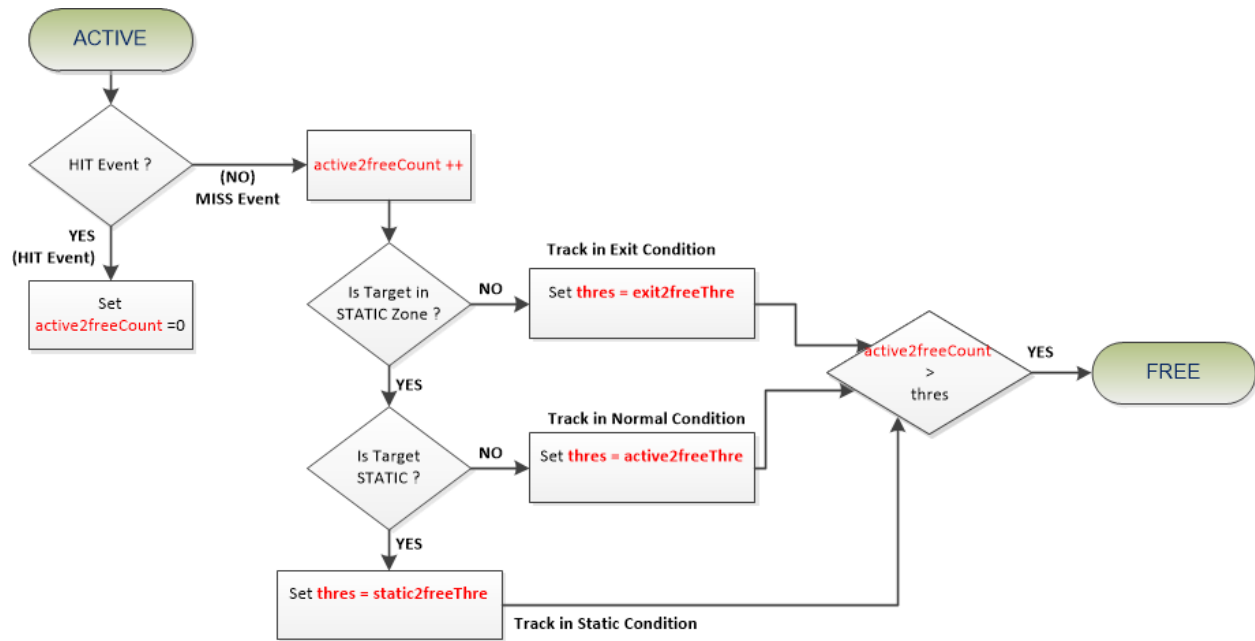
Figure 14 : State Transition from ACTIVE → FREE. The transition is determined if the active2freeCount exceeds a threshold. The threshold value depends on the location and if the target is STATIC.
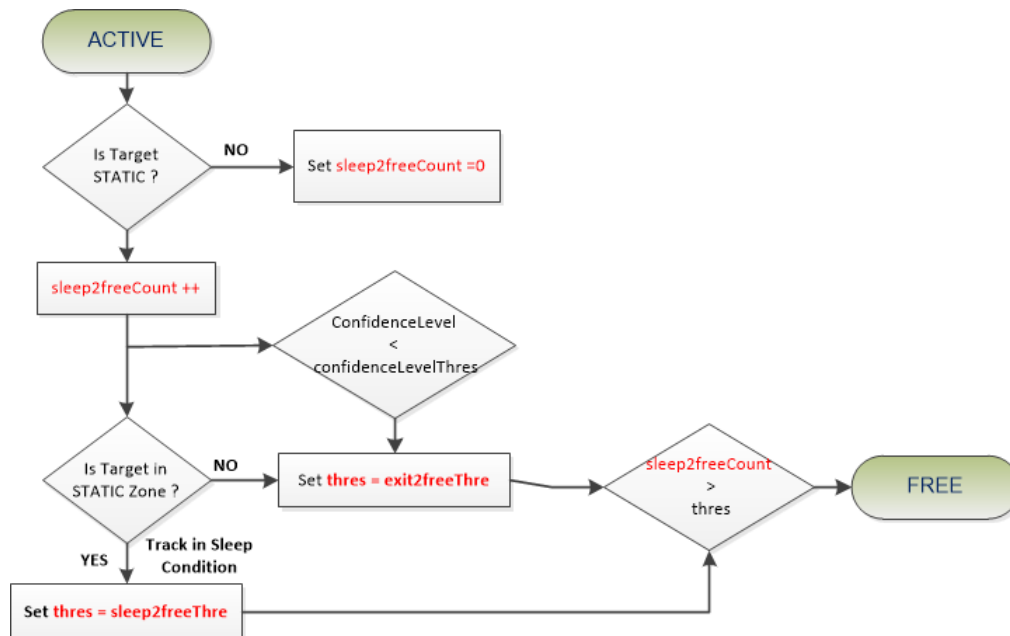
Figure 15 : State Transition from ACTIVE (Sleep) → FREE. The transition is determined if the sleep2freeCount exceeds a threshold. The threshold value depends on the location and the confidence level of the track

3.10. Presence

Presence detection determines if any target exists within the occupancy area. Presence is computed over the combined shape of the occupancy boxes. The occupancy boxes are specified by the user in the configuration parameters. The user can specify up to `GTRACK_MAX_OCCUPANCY_BOXES` occupancy boxes where the maximum value supported is 2.

Presence detection can be performed at the track allocation time within the *gtrack_moduleAllocate()* function or it can be done through the MODULE-level function *gtrack_modulePresence()* which is called by *gtrack_step()*.

If the Presence detection is enabled then the *gtrack_moduleAllocate()* function at track allocation time will re-use the candidate set created by the allocation process and set the *presenceDetectionRaw* flag if the following conditions are satisfied

(a) Number of points in the allocation set is greater than or equal to *<presenseParams.pointsThre>*
(b) Velocity (mean) of points in allocation set is greater than or equal to
     <presenseParams.velocityThre>.
(c) The centroid of the candidate set is within the occupancy area

Based on the *presenceDetectionRaw* flag, the function *gtrack_modulePresence()* will set *presenceDetectionOutput* to 1 if a target is present, otherwise will set to 0.

3.11. Track Report

The function *gtrack_step()* calls the MODULE-level function *gtrack_moduleReport().* The Report function *gtrack_unitReport()* queries each tracking unit and produces the algorithm output.

```
gtrack_moduleReport(h,….) {
        for (each active unit) {
                gtrack_unitReport(unit,…..)
        }
 }
```

The function *gtrack_unitReport()* fills out the following target descriptor structure

```
* @brief
 *  GTRACK target descriptor
 *
 * @details
 *  The structure describes target descriptor format
 */
typedef struct
{
/**  @brief   Tracking Unit Identifier */
uint8_t uid;
/**  @brief   Target Identifier */
uint32_t tid;
/**  @brief   State vector */
float S[GTRACK_STATE_VECTOR_SIZE];
/**  @brief   Group covariance matrix */
float EC[GTRACK_MEASUREMENT_VECTOR_SIZE*GTRACK_MEASUREMENT_VECTOR_SIZE];
```

```
/**  @brief   Gain factor */
float G;
/**  @brief   Estimated target dimensions: depth, width, [height], doppler */
float dim[GTRACK_MEASUREMENT_VECTOR_SIZE];
/**  @brief   Measurement Centroid range/angle/doppler */
float uCenter[GTRACK_MEASUREMENT_VECTOR_SIZE];
/**  @brief   Target confidence level */
float confidenceLevel;
} GTRACK_targetDesc;
```

4. Configuration Parameters

This Section describes the parameters used to configure Tracking algorithm. They shall be adjusted to match customer use case based on particular scenery and targets characteristics. For up to date documentation, please check with the documentation auto-generated by doxygen tools. Below is a snapshot of the documentation based Tracker library version 0.110. For a detailed description on the user-configurable group tracker parameters that can be set through the CLI commands and parameter tuning guidelines please refer to the Group Tracker Tuning Guide [2].

The configuration parameters are passed via *GTRACK_moduleConfig* structure and are described below:

4.1.   Tracker Module Configuration

The *maxAcceleration[3]* parameters are explained in 4.1.1 and more details on the *advParams* is provided in section 4.2.

4.1.1. Max Acceleration Parameters

<maxAcceleration[3]> determines the maximum amount that the acceleration will change between sample periods in the lateral, longitudinal and vertical directions and are used to compute the Processing Noise Matrix **Q.**

As explained in Section 1.2, in 3D constant acceleration model, the assumption is that for each discrete time step the position and velocity of the object can potentially change and the acceleration would remain constant. However, in reality there is no guarantee that the targets acceleration will remain constant as there could be several unknown external un-modeled factors that can alter this assumption. The process noise matrix tries to model this deviation from our assumed model and actual state of the system. [Please refer to Section 6.4.3 in the Appendix for deriving **Q**].
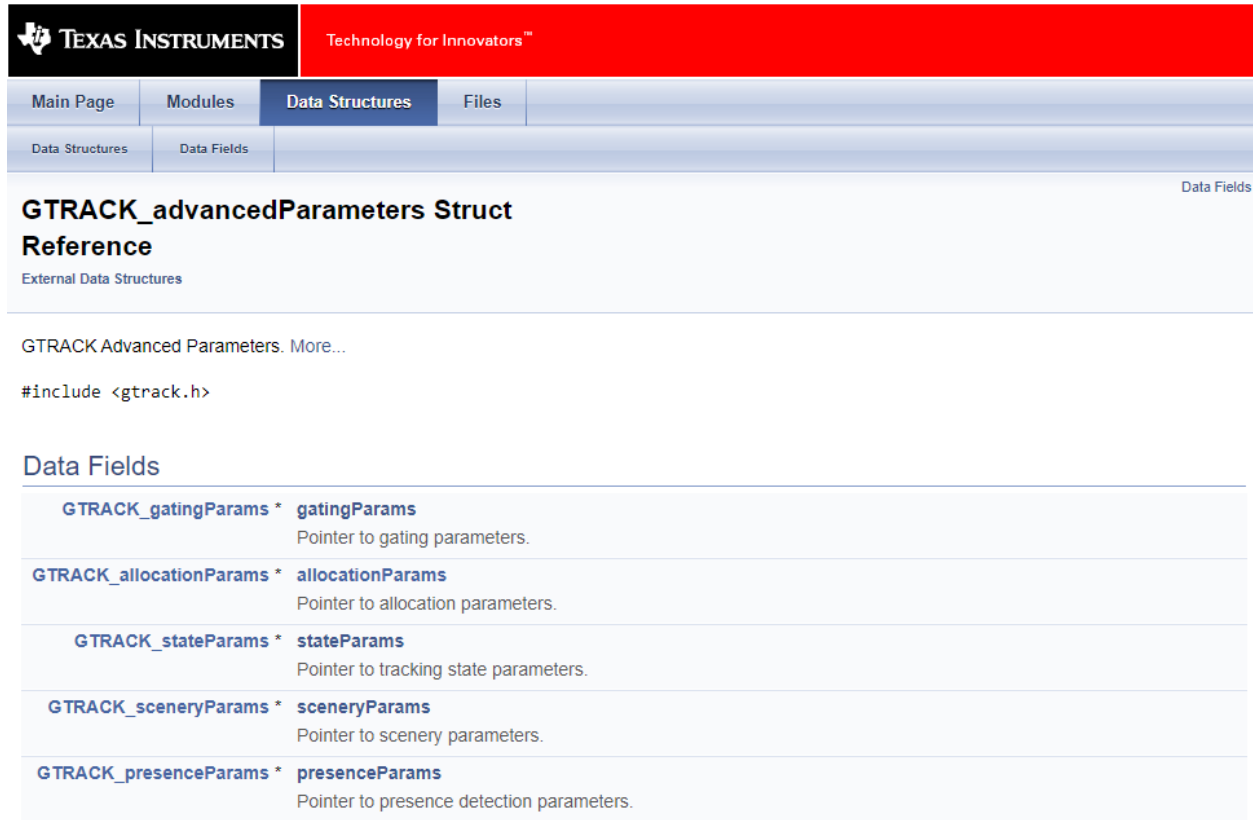
Piecewise White Noise Model is used to model the Process noise matrix. This model assumes that the acceleration is constant for the duration of each time period, but differs from one time period to another, and each of these is uncorrelated with one another. Based on this, the noise can be modeled by a variance which is computed from the *<maxAcceleration[3]>* based on the code below.

```
varX = powf(0.5f*config->maxAcceleration[0],2);
varY = powf(0.5f*config->maxAcceleration[1],2);
varZ = powf(0.5f*config->maxAcceleration[2],2);
```

Essentially the process variance measures the uncertainty we have in our motion model. Setting a large value will imply that we don't trust our motion model and hence the Prediction step is less reliable. Setting a low value implies that our motion model is very accurate and we do not expect much uncertainty.

4.2. Advanced parameters

Advanced parameters are divided into few sets. Each set can be omitted, and defaults will be used by the algorithm. Customers are expected to modify needed parameters to achieve better performance.

4.2.1. Scenery Parameters

These set of parameters define the scenery. It allows the user to configure the tracker with expected boundaries and areas of static behavior. Boxes are defined in meters in the world co-ordinates in Cartesian space (X, Y, Z). The sensor is assumed to be at (0, 0, H) where H is the height of the sensor above the ground.

TEXAS INSTRUMENTS Technology for Innovators™

Main Page	Modules	Data Structures	Files

Data Structures	Data Fields

GTRACK_sceneryParams Struct Reference

External Data Structures

GTRACK Scenery Parameters. More...

`#include <gtrack.h>`

Data Fields

GTRACK_sensorPosition	sensorPosition	
	Sensor position, set to (0.f, 0.f) for 2D, set to (0.f, 0.f, H) for 3D. Where H is sensor height, in m.	
GTRACK_sensorOrientation	sensorOrientation	
	Sensor orientation, set to (0.f, 0.f) for 2D, (AzimTilt, ElevTilt) for 3D. Where AzimTilt and ElevTilt are rotations along Z and X axes correspondily.	
uint8_t	numBoundaryBoxes	
	Number of scene boundary boxes. If defined (numBoundaryBoxes > 0), only points within the boundary box(s) can be associated with tracks.	
GTRACK_boundaryBox	boundaryBox [GTRACK_MAX_BOUNDARY_BOXES]	
	Scene boundary boxes.	
uint8_t	numStaticBoxes	
	Number of scene static boxes. If defined (numStaticBoxes > 0), only targets within the static box(s) can persist as static.	
GTRACK_boundaryBox	staticBox [GTRACK_MAX_STATIC_BOXES]	
	Scene static boxes.	

Detailed Description

GTRACK Scenery Parameters.

Scenery uses 3-dimensional Cartesian coordinate system, defined as W in the picture below
It is expected that the Z=0 plane corresponds to the scene floor
The X coordinates are left (negative)-right; the Y coordinates are near-far.
Origin (O) is typically colocated with senor projection to Z=0 plane

- Sensor Position is 3 dimentioanl coordinate of the sensor
  - For example, (0,0,2) will indicate that sensor is directly above the origin at the height of 2m
- Sensor Orientation is sensor's boresight rotation: down tilt (thetta) and azimuthal tilt (not supported)

User can define up to GTRACK_MAX_BOUNDARY_BOXES boundary boxes, and up to GTRACK_MAX_STATIC_BOXES static boxes.

- Boundary Boxes are used to define area of interest. All reflection points outside the boundary area are ignored.
  - For example, boundary box can be used to ignore the targets in the hallway outside the room, or to ignore potential ghost-behind-the-wall reflections
- Static Boxes defines the zone where targets are expected to become static for a long time. Typically, that area is a smaller (0.5-1.5m) then boundary
  - When not directly configured by customer, the application makes box 0.5m smaller from each side
  - When targets are within the static zone, and miss detection event occurs, the [larger] static threshold will apply for de-allocation. When outside the area, the [smaller] exit threshold will be applied
  - Static reflection points outside the static area are ignored.

4.2.2. Allocation Parameters

The reflection points reported in point cloud are associated with existing tracking instances. Points that don't get associated are subjects for the allocation decision.

The allocation parameters determine if a **candidate point** can be clustered into an **allocation set.** To join the set, each point needs to be within *<maxDistanceThre>* and *<maxVelThre>* from the allocation sets centroid. Moreover, once the set is formed, it has to have more than *<pointsThre>* members and exceed the *<velocityThre>* and *<snrThre>* values.

GTRACK_allocationParams Struct Reference
External Data Structures

GTRACK Allocation Function Parameters. More...

`#include <gtrack.h>`

Data Fields

float	**snrThre**	
	Minimum total SNR of the allocation set-candidate. The threshold shall be roughly equal to the expected linear SNR value of the detection point at 6m range multiplied by pointsThre.	
float	**snrThreObscured**	
	Minimum total SNR when behind another target. Set it larger then snrThre to require stronger SNR for the obscured allocations.	
float	**velocityThre**	
	Minimum initial velocity, m/s.	
uint16_t	**pointsThre**	
	Minimum number of points in a set.	
float	**maxDistanceThre**	
	Maximum squared distance between points in a set.	
float	**maxVelThre**	
	Maximum velocity delta between points in a set.	

Detailed Description

- **snrThre:** The *snrThre* is set to the number of reflections that will lead to allocation decision with very large confidence. The total SNR for the allocation set is obtained by summing SNR values (in linear scale) of all the members of the allocation set. The total SNR is then compared with an internally derived SNR threshold value. The internally derived SNR threshold is a scaled version of *snrThre* where the scaling factor is range-dependent. Setting *snrThre* too low leads to long term errors.

- **snrThreObscured:** The threshold shall be roughly equal to the expected linear SNR value at 6m range multiplied by *pointsThre* Threshold. The usage is similar to that of *snrThre* except that this value is used if the allocation set is obscured by another track. An allocation set is declared obscured if it is behind AND has similar Doppler to an existing track.

- **velocityThre:** Used to ignore allocation sets with very low radial velocity. On several occasions we might observe "spectral leakage" from strong static reflectors into adjacent Doppler bins. Setting *velocityThre* to a non-zero value prevents the static strong reflectors being allocated a track. However, setting a value too high can result in persons that are very stationary to not be allocated a track.

- **pointsThre:** Value is set based on the minimum expected number of points from a target.

- **maxDistanceThre:** This is a measure of how close (geometrically, in meters) the point-candidate shall be to the centroid of the set-under-construction to join the set. It is similar to epsilon in DBSCAN algorithm. Note that the distance measure takes into account only the geometric

difference (range, azimuth, and elevation) between the point and centroid and does not include the velocity and SNR of the measurement points.

— **maxVelThre:** Measure of how close (in radial velocity, m/s) the point-candidate shall be to the centroid of the set-under-construction to join the set. The parameter may be used to filter out obvious Doppler error.

4.2.3. State Transition Parameters

4.2.4. Gating Parameters

Gating parameters set is used in association process to provide a boundary for the points that can be associated with a given track. These parameters are target-specific. For example, for people counting, it is expected that the limits are set based on human body dimensions and dynamicity limits: (ex, 1.5x1.5x2 m in depth x width x height, and 4m/s of Doppler spread). The gain is expected to be around 3 (ex. based on "three sigma rule").

GTRACK_gatingParams Struct Reference

External Data Structures

GTRACK Gating Function Parameters. More...

`#include <gtrack.h>`

Data Fields

float	**gain**	
		Gain of the gating function.
		It is set based on expected tracking errors and uncertainties of detection layer.
GTRACK_gateLimits	**limits**	
		Gating function limits.
		It is based on physical dimensions and agility of the targets.
		Setting it too small will result in allocating multiple tracks for the single object, setting it too big will cause allocating single track for multiple objects.

GTRACK_gateLimits Struct Reference

External Data Structures

GTRACK Gate Limits. More...

`#include <gtrack.h>`

Data Fields

float	**depth**	
		Depth limit, m.
float	**width**	
		Width limit, m.
float	**height**	
		Heigth limit, m.
float	**vel**	
		Radial velocity limit, m/s.

4.2.5. Presence Detection Parameters

Presence detection determines if any target exists within the occupancy area. Presence is computed over the combined shape of the occupancy boxes. The occupancy boxes dimensions are specified by the user in <occupancyBox[2]>. The user can specify up to 2 occupancy boxes through the parameter <numOccupancyBoxes>.

The algorithm combines "raw detection" and "known target in the area" indications. For "raw detection" indication, it re-uses the candidate set created by allocation process. It checks the set against the occupancy thresholds: number of points in the set against the <pointsThre> and set's velocity against the <velocityThre>. For "known target in the area" the algorithm checks whether known target measurement centroid is within the occupancy boxes.

TEXAS INSTRUMENTS Technology for Innovators™

| Main Page | Modules | Data Structures | Files |

| Data Structures | Data Fields |

Data Fields

GTRACK_presenceParams Struct Reference

External Data Structures

GTRACK Presence Detection Parameters. More...

`#include <gtrack.h>`

Data Fields

uint16_t	**pointsThre**
	occupancy threshold, number of points. Setting pointsThre to 0 disables presence detection
float	**velocityThre**
	occupancy threshold, approaching velocity
uint16_t	**on2offThre**
	occupancy on to off threshold
uint8_t	**numOccupancyBoxes**
	Number of occulancy boxes. Presence detection algorithm will determine whether the combined shape is occupied. Setting numOccupancyBoxes to 0 disables presence detection.
GTRACK_boundaryBox	**occupancyBox [GTRACK_MAX_OCCUPANCY_BOXES]**
	Scene occupancy boxes.

Detailed Description

GTRACK Presence Detection Parameters.

This set of parameters describes the presence detection function

Presence is computed over the combined shape of occupancy boxes. Each box is described using 3-dimensional Cartesian coordinate system. It is expected that the Z=0 plane corresponds to the scene floor. The X coordinates are left (negative)-right; the Y ccordinates are near-far. Origin is typically colocated with senor projection to Z=0 plane. User can define up to GTRACK_MAX_OCCUPANCY_BOXES occupancy boxes.

If any target exists with the occupancy area, the alrgorithm returns 1. Othewise, it returns 0.

The algorithm combines "raw detection" and "known target in the area" indications.

For "raw detection" indication re-uses the candidate set created by allocation process. It checks the occupancy thresholds: number of points in the set against the pointsThre and set's velocity against the velocityThre.

For "known target in the area" the algorithm checks whether known target measurment centroid is within the occupancy boxes.

5. Tracker Memory Requirements and Benchmarks

Initial memory footprint for group tracker implementation with 800 measurement points and 20 tracks is 112 KB of Data memory and 20KB of program space on the R4F ARM core.

5.1. Memory

	File Name	Code Size (Bytes)
	gtrack_unit_update.oer4f	4649
	gtrack_math.oer4f	1646
	gtrack_create.oer4f	1760
	gtrack_utilities_3d.oer4f	2350
	gtrack_module.oer4f	1429
	gtrack_utilities.oer4f	1134
	gtrack_unit_score.oer4f	1052
	gtrack_unit_event.oer4f	728
	gtrack_step.oer4f	548
	gtrack_unit_predict.oer4f	518
	gtrack_unit_start.oer4f	416
	gtrack_unit_create.oer4f	348
	gtrack_listlib.oer4f	206
	gtrack_delete.oer4f	200
	gtrack_unit_stop.oer4f	156
	gtrack_unit_report.oer4f	98
	gtrack_unit_get.oer4f	28
	gtrack_unit_delete.oer4f	8
Total		17274

5.2. Tracker Module Execution Time

Please refer to the 3D people counting implementation guide for up-to-date number for the execution time of the tracker [6].

6. Appendix

6.1. Evaluating Partial Derivatives for 2D space tracking

We need to calculate set of partial derivatives to compute Jacobian

$$
J_H(s) = \begin{bmatrix}
\dfrac{\partial r}{\partial x} & \dfrac{\partial r}{\partial y} & \dfrac{\partial r}{\partial \dot{x}} & \dfrac{\partial r}{\partial \dot{y}} & \dfrac{\partial r}{\partial \ddot{x}} & \dfrac{\partial r}{\partial \ddot{y}} \\
\dfrac{\partial \varphi}{\partial x} & \dfrac{\partial \varphi}{\partial y} & \dfrac{\partial \varphi}{\partial \dot{x}} & \dfrac{\partial \varphi}{\partial \dot{y}} & \dfrac{\partial \varphi}{\partial \ddot{x}} & \dfrac{\partial \varphi}{\partial \ddot{y}} \\
\dfrac{\partial \dot{r}}{\partial x} & \dfrac{\partial \dot{r}}{\partial y} & \dfrac{\partial \dot{r}}{\partial \dot{x}} & \dfrac{\partial \dot{r}}{\partial \dot{y}} & \dfrac{\partial \dot{r}}{\partial \ddot{x}} & \dfrac{\partial \dot{r}}{\partial \ddot{y}}
\end{bmatrix}
$$

6.1.1. Evaluating range partial derivatives

$$r = \sqrt{x^2 + y^2}$$

$$\frac{\partial r}{\partial x} = \frac{\partial}{\partial x}\left(\sqrt{x^2 + y^2}\right) = \frac{x}{\sqrt{x^2 + y^2}}$$

$$\frac{\partial r}{\partial y} = \frac{\partial}{\partial y}\left(\sqrt{x^2 + y^2}\right) = \frac{y}{\sqrt{x^2 + y^2}}$$

6.1.2. Evaluating azimuth partial derivatives

$$\frac{\partial \varphi}{\partial x} = \frac{\partial}{\partial x}\left(\tan^{-1}\left(\frac{x}{y}\right)\right)$$

Let $w = \tan^{-1} u$ with $u = \dfrac{x}{y}$

Using the chain rule: $\dfrac{\partial \varphi}{\partial x} = \dfrac{dw}{du}\dfrac{du}{dx} = \left(\dfrac{1}{1+u^2}\right)\left(\dfrac{1}{y}\right) = \dfrac{1}{1+\left(\frac{x}{y}\right)^2}\dfrac{1}{y} = \dfrac{y}{x^2+y^2}$

Using the chain rule: $\dfrac{\partial \varphi}{\partial y} = \dfrac{dw}{du}\dfrac{du}{dy} = \left(\dfrac{1}{1+u^2}\right)\left(\dfrac{-x}{y^2}\right) = \dfrac{1}{1+\left(\frac{x}{y}\right)^2}\dfrac{-x}{y^2} = -\dfrac{x}{x^2+y^2}$

6.1.1. Evaluating doppler partial derivatives

$$\dot{r} = \frac{x\dot{x} + y\dot{y}}{\sqrt{x^2 + y^2}}$$

$$\frac{\partial \dot{r}}{\partial x} = \frac{\partial}{\partial x}\left(\frac{x\dot{x} + y\dot{y}}{\sqrt{x^2 + y^2}}\right) = \dot{x}\frac{\partial}{\partial x}\left(\frac{x}{\sqrt{x^2 + y^2}}\right) + y\dot{y}\frac{\partial}{\partial x}\left(\frac{1}{\sqrt{x^2 + y^2}}\right)$$

Using the quotient rule $\frac{\partial}{\partial x}\left(\frac{f}{g}\right) = \frac{\frac{\partial f}{\partial x}g - \frac{\partial g}{\partial x}f}{g^2}$, solving the first part $\frac{\partial}{\partial x}\left(\frac{x}{\sqrt{x^2+y^2}}\right) = \frac{\sqrt{x^2+y^2} - \frac{x}{\sqrt{x^2+y^2}}x}{\left(\sqrt{x^2+y^2}\right)^2} =$

$\frac{y^2}{(x^2+y^2)^{3/2}}$.

For the second part, $\frac{\partial}{\partial x}\left(\frac{1}{\sqrt{x^2+y^2}}\right) = -\frac{1}{2}\frac{2x}{(x^2+y^2)^{3/2}} = -\frac{x}{(x^2+y^2)^{3/2}}$.

$$\frac{\partial}{\partial x}\left(\frac{x\dot{x}+y\dot{y}}{\sqrt{x^2+y^2}}\right) = \dot{x}\frac{y^2}{(x^2+y^2)^{3/2}} - y\dot{y}\frac{x}{(x^2+y^2)^{3/2}} = \frac{y(\dot{x}y - \dot{y}x)}{(x^2+y^2)^{3/2}}$$

Similarly,

$$\frac{\partial \dot{r}}{\partial y} = \frac{\partial}{\partial y}\left(\frac{x\dot{x}+y\dot{y}}{\sqrt{x^2+y^2}}\right) = \dot{y}\frac{\partial}{\partial y}\left(\frac{y}{\sqrt{x^2+y^2}}\right) + x\dot{x}\frac{\partial}{\partial y}\left(\frac{1}{\sqrt{x^2+y^2}}\right) = \dot{y}\frac{x^2}{(x^2+y^2)^{3/2}} - x\dot{x}\frac{y}{(x^2+y^2)^{3/2}}$$

$$= \frac{x(\dot{y}x - \dot{x}y)}{(x^2+y^2)^{3/2}}$$

$$\frac{\partial \dot{r}}{\partial \dot{x}} = \frac{\partial}{\partial \dot{x}}\left(\frac{x\dot{x}+y\dot{y}}{\sqrt{x^2+y^2}}\right) = \frac{x}{\sqrt{x^2+y^2}}$$

$$\frac{\partial \dot{r}}{\partial \dot{y}} = \frac{\partial}{\partial \dot{y}}\left(\frac{x\dot{x}+y\dot{y}}{\sqrt{x^2+y^2}}\right) = \frac{y}{\sqrt{x^2+y^2}}$$

Putting all together:

$$J_H(s) = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} & 0 & 0 & 0 & 0 \\ \frac{y}{x^2+y^2} & -\frac{x}{x^2+y^2} & 0 & 0 & 0 & 0 \\ \frac{y(\dot{x}y-\dot{y}x)}{(x^2+y^2)^{3/2}} & \frac{x(\dot{y}x-\dot{x}y)}{(x^2+y^2)^{3/2}} & \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} & 0 & 0 \end{bmatrix}$$

6.2. Evaluating Partial Derivatives for 3D space tracking

We need to calculate set of partial derivatives to compute Jacobians for 3DV and 3DA models:

$$J_H(s_{3DV}) = \begin{bmatrix} \dfrac{\partial r}{\partial x} & \dfrac{\partial r}{\partial y} & \dfrac{\partial r}{\partial z} & \dfrac{\partial r}{\partial \dot{x}} & \dfrac{\partial r}{\partial \dot{y}} & \dfrac{\partial r}{\partial \dot{z}} \\[2ex] \dfrac{\partial \varphi}{\partial x} & \dfrac{\partial \varphi}{\partial y} & \dfrac{\partial \varphi}{\partial z} & \dfrac{\partial \varphi}{\partial \dot{x}} & \dfrac{\partial \varphi}{\partial \dot{y}} & \dfrac{\partial \varphi}{\partial \dot{z}} \\[2ex] \dfrac{\partial \theta}{\partial x} & \dfrac{\partial \theta}{\partial y} & \dfrac{\partial \theta}{\partial z} & \dfrac{\partial \theta}{\partial \dot{x}} & \dfrac{\partial \theta}{\partial \dot{y}} & \dfrac{\partial \theta}{\partial \dot{z}} \\[2ex] \dfrac{\partial \dot{r}}{\partial x} & \dfrac{\partial \dot{r}}{\partial y} & \dfrac{\partial \dot{r}}{\partial z} & \dfrac{\partial \dot{r}}{\partial \dot{x}} & \dfrac{\partial \dot{r}}{\partial \dot{y}} & \dfrac{\partial \dot{r}}{\partial \dot{z}} \end{bmatrix}$$

and

$$J_H(s_{3DA}) = \begin{bmatrix} \dfrac{\partial r}{\partial x} & \dfrac{\partial r}{\partial y} & \dfrac{\partial r}{\partial z} & \dfrac{\partial r}{\partial \dot{x}} & \dfrac{\partial r}{\partial \dot{y}} & \dfrac{\partial r}{\partial \dot{z}} & \dfrac{\partial r}{\partial \ddot{x}} & \dfrac{\partial r}{\partial \ddot{y}} & \dfrac{\partial r}{\partial \ddot{z}} \\[2ex] \dfrac{\partial \varphi}{\partial x} & \dfrac{\partial \varphi}{\partial y} & \dfrac{\partial \varphi}{\partial z} & \dfrac{\partial \varphi}{\partial \dot{x}} & \dfrac{\partial \varphi}{\partial \dot{y}} & \dfrac{\partial \varphi}{\partial \dot{z}} & \dfrac{\partial \varphi}{\partial \ddot{x}} & \dfrac{\partial \varphi}{\partial \ddot{x}} & \dfrac{\partial \varphi}{\partial \ddot{x}} \\[2ex] \dfrac{\partial \theta}{\partial x} & \dfrac{\partial \theta}{\partial y} & \dfrac{\partial \theta}{\partial z} & \dfrac{\partial \theta}{\partial \dot{x}} & \dfrac{\partial \theta}{\partial \dot{y}} & \dfrac{\partial \theta}{\partial \dot{z}} & \dfrac{\partial \theta}{\partial \ddot{x}} & \dfrac{\partial \theta}{\partial \ddot{y}} & \dfrac{\partial \theta}{\partial \ddot{z}} \\[2ex] \dfrac{\partial \dot{r}}{\partial x} & \dfrac{\partial \dot{r}}{\partial y} & \dfrac{\partial \dot{r}}{\partial z} & \dfrac{\partial \dot{r}}{\partial \dot{x}} & \dfrac{\partial \dot{r}}{\partial \dot{y}} & \dfrac{\partial \dot{r}}{\partial \dot{z}} & \dfrac{\partial \dot{r}}{\partial \ddot{x}} & \dfrac{\partial \dot{r}}{\partial \ddot{y}} & \dfrac{\partial \dot{r}}{\partial \ddot{z}} \end{bmatrix}$$

6.2.1. Evaluating range partial derivatives

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\frac{\partial r}{\partial x} = \frac{\partial}{\partial x}\left(\sqrt{x^2 + y^2 + z^2}\right) = \frac{x}{\sqrt{x^2 + y^2 + z^2}}$$

$$\frac{\partial r}{\partial y} = \frac{\partial}{\partial y}\left(\sqrt{x^2 + y^2 + z^2}\right) = \frac{y}{\sqrt{x^2 + y^2 + z^2}}$$

$$\frac{\partial r}{\partial z} = \frac{\partial}{\partial z}\left(\sqrt{x^2 + y^2 + z^2}\right) = \frac{z}{\sqrt{x^2 + y^2 + z^2}}$$

6.2.2. Evaluating azimuth partial derivatives

$$\frac{\partial \varphi}{\partial x} = \frac{\partial}{\partial x}\left(\tan^{-1}\left(\frac{x}{y}\right)\right)$$

Let $w = \tan^{-1} u$ with $u = \frac{x}{y}$

Using the chain rule: $\frac{\partial \varphi}{\partial x} = \frac{dw}{du}\frac{du}{dx} = \left(\frac{1}{1+u^2}\right)\left(\frac{1}{y}\right) = \frac{1}{1+\left(\frac{x}{y}\right)^2}\frac{1}{y} = \frac{y}{x^2+y^2}$

Using the chain rule: $\frac{\partial \varphi}{\partial y} = \frac{dw}{du}\frac{du}{dy} = \left(\frac{1}{1+u^2}\right)\left(\frac{-x}{y^2}\right) = \frac{1}{1+\left(\frac{x}{y}\right)^2}\frac{-x}{y^2} = -\frac{x}{x^2+y^2}$

$$\frac{\partial \varphi}{\partial z} = 0$$

6.2.1. Evaluating elevation partial derivatives

$$\theta = \tan^{-1}\left(\frac{z}{\sqrt{x^2+y^2}}\right)$$

Let $w = \tan^{-1} u$ with $u = \frac{z}{\sqrt{x^2+y^2}}$

Using the chain rule: $\frac{\partial \theta}{\partial x} = \frac{dw}{du}\frac{du}{dx} = \left(\frac{1}{1+u^2}\right)\frac{du}{dx} = \frac{1}{1+\frac{z^2}{x^2+y^2}}\left(-\frac{1}{2}\right)2x\frac{z}{(x^2+y^2)^{3/2}} = -\frac{x^2+y^2}{x^2+y^2+z^2}\frac{xz}{(x^2+y^2)^{\frac{3}{2}}} = $

$-\frac{x}{(x^2+y^2+z^2)\sqrt{x^2+y^2}}z = -\frac{x}{r^2}\tan\theta$

Using the chain rule: $\frac{\partial \theta}{\partial y} = \frac{dw}{du}\frac{du}{dy} = \left(\frac{1}{1+u^2}\right)\frac{du}{dy} = \frac{1}{1+\frac{z^2}{x^2+y^2}}\left(-\frac{1}{2}\right)2y\frac{z}{(x^2+y^2)^{3/2}} = -\frac{x^2+y^2}{x^2+y^2+z^2}\frac{yz}{(x^2+y^2)^{\frac{3}{2}}} = $

$-\frac{y}{(x^2+y^2+z^2)\sqrt{x^2+y^2}}z = -\frac{y}{r^2}\tan\theta$

Using the chain rule: $\frac{\partial \theta}{\partial z} = \frac{dw}{du}\frac{du}{dz} = \left(\frac{1}{1+u^2}\right)\frac{du}{dz} = \frac{1}{1+\frac{z^2}{x^2+y^2}}\frac{1}{\sqrt{x^2+y^2}} = \frac{x^2+y^2}{x^2+y^2+z^2}\frac{1}{\sqrt{x^2+y^2}} = \frac{\sqrt{x^2+y^2}}{(x^2+y^2+z^2)} = $

$\frac{\sqrt{x^2+y^2}}{r^2}$

6.2.2. Evaluating Doppler partial derivatives

$$\dot{r} = \frac{x\dot{x} + y\dot{y} + z\dot{z}}{\sqrt{x^2+y^2+z^2}}$$

$$\frac{\partial \dot{r}}{\partial x} = \frac{\partial}{\partial x}\left(\frac{x\dot{x}+y\dot{y}+z\dot{z}}{\sqrt{x^2+y^2+z^2}}\right) = \dot{x}\frac{\partial}{\partial x}\left(\frac{x}{\sqrt{x^2+y^2+z^2}}\right) + (y\dot{y}+z\dot{z})\frac{\partial}{\partial x}\left(\frac{1}{\sqrt{x^2+y^2+z^2}}\right);$$

Using the quotient rule $\frac{\partial}{\partial x}\left(\frac{f}{g}\right) = \frac{\frac{\partial f}{\partial x}g - \frac{\partial g}{\partial x}f}{g^2}$, solving $\frac{\partial}{\partial x}\left(\frac{x}{\sqrt{x^2+y^2+z^2}}\right) = \frac{\sqrt{x^2+y^2+z^2}-\frac{x}{\sqrt{x^2+y^2+z^2}}x}{x^2+y^2+z^2} = $

$\frac{y^2+z^2}{(x^2+y^2+z^2)^{3/2}};$

Using chain rule, $\frac{\partial}{\partial x}\left(\frac{1}{\sqrt{x^2+y^2+z^2}}\right) = -\frac{1}{2}\frac{2x}{(x^2+y^2+z^2)^{3/2}} = -\frac{x}{(x^2+y^2+z^2)^{3/2}};$

$$\frac{\partial}{\partial x}\left(\frac{x\dot{x} + y\dot{y} + z\dot{z}}{\sqrt{x^2 + y^2 + z^2}}\right) = \dot{x}\frac{y^2 + z^2}{(x^2 + y^2 + z^2)^{3/2}} - (y\dot{y} + z\dot{z})\frac{x}{(x^2 + y^2 + z^2)^{3/2}} =$$

$$= \frac{y(\dot{x}y - \dot{y}x) + z(\dot{x}z - \dot{z}x)}{(x^2 + y^2 + z^2)^{3/2}};$$

Due to symmetry:

$$\frac{\partial \dot{r}}{\partial y} = \frac{x(\dot{y}x - \dot{x}y) + z(\dot{y}z - \dot{z}y)}{(x^2 + y^2 + z^2)^{3/2}};$$

$$\frac{\partial \dot{r}}{\partial z} = \frac{x(\dot{z}x - \dot{x}z) + y(\dot{z}y - \dot{y}z)}{(x^2 + y^2 + z^2)^{3/2}};$$

$$\frac{\partial \dot{r}}{\partial \dot{x}} = \frac{\partial}{\partial \dot{x}}\left(\frac{x\dot{x} + y\dot{y} + z\dot{z}}{\sqrt{x^2 + y^2 + z^2}}\right) = \frac{x}{\sqrt{x^2 + y^2 + z^2}};$$

$$\frac{\partial \dot{r}}{\partial \dot{y}} = \frac{\partial}{\partial \dot{y}}\left(\frac{x\dot{x} + y\dot{y} + z\dot{z}}{\sqrt{x^2 + y^2 + z^2}}\right) = \frac{y}{\sqrt{x^2 + y^2 + z^2}};$$

$$\frac{\partial \dot{r}}{\partial \dot{z}} = \frac{\partial}{\partial \dot{z}}\left(\frac{x\dot{x} + y\dot{y} + z\dot{z}}{\sqrt{x^2 + y^2 + z^2}}\right) = \frac{z}{\sqrt{x^2 + y^2 + z^2}};$$

Putting all together:

$$J_H(s_{3DV}) =$$

$$= \begin{bmatrix} \dfrac{x}{r} & \dfrac{y}{r} & \dfrac{z}{r} & 0 & 0 & 0 \\[2mm] \dfrac{y}{x^2 + y^2} & -\dfrac{x}{x^2 + y^2} & 0 & 0 & 0 & 0 \\[2mm] -\dfrac{x}{r^2}\dfrac{z}{\sqrt{x^2 + y^2}} & -\dfrac{y}{r^2}\dfrac{z}{\sqrt{x^2 + y^2}} & \dfrac{\sqrt{x^2 + y^2}}{r^2} & 0 & 0 & 0 \\[2mm] \dfrac{y(\dot{x}y - \dot{y}x) + z(\dot{x}z - \dot{z}x)}{r^3} & \dfrac{x(\dot{y}x - \dot{x}y) + z(\dot{y}z - \dot{z}y)}{r^3} & \dfrac{x(\dot{z}x - \dot{x}z) + y(\dot{z}y - \dot{y}z)}{r^3} & \dfrac{x}{r} & \dfrac{y}{r} & \dfrac{z}{r} \end{bmatrix};$$

$$J_H(s_{3DA}) =$$

$$= \begin{bmatrix} \dfrac{x}{r} & \dfrac{y}{r} & \dfrac{z}{r} & 0 & 0 & 0 & 0 & 0 & 0 \\[2mm] \dfrac{y}{x^2 + y^2} & -\dfrac{x}{x^2 + y^2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\[2mm] -\dfrac{x}{r^2}\dfrac{z}{\sqrt{x^2 + y^2}} & -\dfrac{y}{r^2}\dfrac{z}{\sqrt{x^2 + y^2}} & \dfrac{\sqrt{x^2 + y^2}}{r^2} & 0 & 0 & 0 & 0 & 0 & 0 \\[2mm] \dfrac{y(\dot{x}y - \dot{y}x) + z(\dot{x}z - \dot{z}x)}{r^3} & \dfrac{x(\dot{y}x - \dot{x}y) + z(\dot{y}z - \dot{z}y)}{r^3} & \dfrac{x(\dot{z}x - \dot{x}z) + y(\dot{z}y - \dot{y}z)}{r^3} & \dfrac{x}{r} & \dfrac{y}{r} & \dfrac{z}{r} & 0 & 0 & 0 \end{bmatrix};$$

where $r = \sqrt{x^2 + y^2 + z^2}$.

6.3. Tracking in 2D
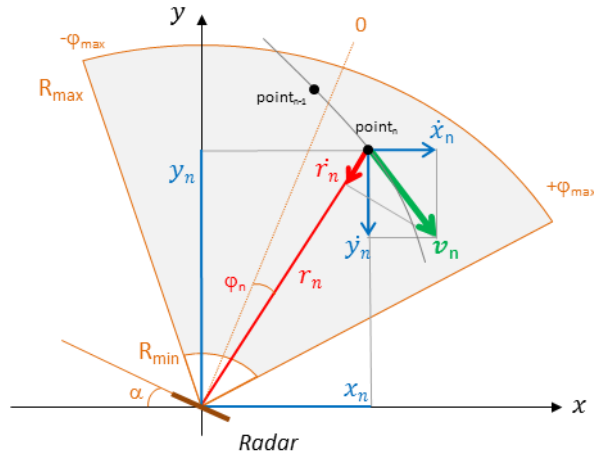
6.3.1. Space Geometry

Figure 16. Tracking in 2D

The angular location coordinates are converted to Cartesian coordinates using

$$x = r\cos\left(\frac{\pi}{2} - (\alpha + \varphi)\right) = r\sin(\alpha + \varphi)$$

$$y = r\sin\left(\frac{\pi}{2} - (\alpha + \varphi)\right) = r\cos(\alpha + \varphi)$$

The objective is to track the location of the objects using the noisy measurements of range, angle, and Doppler (radial velocity)

6.3.2. 2D Space, Constant Velocity Model

We use Kalman filter to "refine" the location estimates. The state of the Kalman filter at time instant n is defined as

$$s(n) = Fs(n-1) + w(n) \tag{6-1}$$

where the state vector $s(n)$ is defined in Cartesian coordinates,

$$s(n) \triangleq [x(n) \quad y(n) \quad \dot{x}(n) \quad \dot{y}(n)]^T, \tag{6-2}$$

F is a transition matrix

$$F = \begin{bmatrix} 1 & 0 & T & 0 \\ 0 & 1 & 0 & T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6-3}$$

and $w(n)$ is the vector of process noise with covariance matrix $Q(n)$ of size 4×4.

The input measurement vector $u(n)$ includes range, angle and radial velocity

$$u(n) = [r(n) \quad \varphi(n) \quad \dot{r}(n)]^T \tag{6-4}$$

The relationship between the state of the Kalman filter and measurement vector is expressed as:

$$u(n) = H(s(n)) + v(n), \tag{6-5}$$

Where, H is a measurement matrix,

$$H(s(n)) = \begin{bmatrix} \sqrt{x^2 + y^2} \\ \tan^{-1}(x,y) - \alpha \\ \frac{x\dot{x} + y\dot{y}}{\sqrt{x^2 + y^2}} \end{bmatrix}, \tag{6-6}$$

the function $\tan^{-1}(x, y)$ is defined as

$$\tan^{-1}(x,y) \triangleq \begin{cases} \tan^{-1}\left(\frac{x}{y}\right), & y > 0, \\ \frac{\pi}{2}, & y = 0, \\ \tan^{-1}\left(\frac{x}{y}\right) + \pi, & y < 0. \end{cases} \tag{6-7}$$

and $v(n)$ is vector of measurement noise with covariance matrix $R(n)$ of size 3×3.

In above formulation, the measurement vector $u(n)$ is related to the state vector $s(n)$ via a non-linear relation. Because of this, we use Extended Kalman filter (EKF), which simplifies the relation between $u(n)$ and $s(n)$ by retaining only the first term in the Taylor series expansion of $H(\cdot)$.

$$u(n) = H(s_{apr}(n)) + J_H(s_{apr}(n))[s(n) - s_{apr}(n)] + v(n), \tag{6-8}$$

where, $s_{apr}(n)$ is a-priori estimation of state vector at time *n* based on *n-1* measurements,

$$J_H(s) = \begin{bmatrix} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} & \frac{\partial r}{\partial \dot{x}} & \frac{\partial r}{\partial \dot{y}} \\ \frac{\partial \varphi}{\partial x} & \frac{\partial \varphi}{\partial y} & \frac{\partial \varphi}{\partial \dot{x}} & \frac{\partial \varphi}{\partial \dot{y}} \\ \frac{\partial \dot{r}}{\partial x} & \frac{\partial \dot{r}}{\partial y} & \frac{\partial \dot{r}}{\partial \dot{x}} & \frac{\partial \dot{r}}{\partial \dot{y}} \end{bmatrix}. \tag{6-9}$$

Calculating partial derivatives (see the Appendix below):

$$J_H(s) = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} & 0 & 0 \\ \frac{y}{x^2+y^2} & -\frac{x}{x^2+y^2} & 0 & 0 \\ \frac{y(\dot{x}y-\dot{y}x)}{(x^2+y^2)^{3/2}} & \frac{x(\dot{y}x-\dot{x}y)}{(x^2+y^2)^{3/2}} & \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} \end{bmatrix} \tag{6-10}$$

6.3.3. 2D Space, Constant Acceleration Model

For constant acceleration model, the state of the Kalman filter at time instant n is defined as

$$s(n) = Fs(n-1) + w(n), \tag{6-11}$$

where the state vector $s(n)$ is defined in Cartesian coordinates,

$$s(n) = [x(n) \quad y(n) \quad \dot{x}(n) \quad \dot{y}(n) \quad \ddot{x}(n) \quad \ddot{y}(n)], \tag{6-12}$$

F is a transition matrix

$$F = \begin{bmatrix} 1 & 0 & T & 0 & 0.5T^2 & 0 \\ 0 & 1 & 0 & T & 0 & 0.5T^2 \\ 0 & 0 & 1 & 0 & T & 0 \\ 0 & 0 & 0 & 1 & 0 & T \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{6-13}$$

and $w(n)$ is the vector of process noise with covariance matrix $Q(n)$ of size 6×6.

The input measurement vector $u(n)$ is the same as in constant velocity model, includes range, angle and radial velocity

$$u(n) = [r(n) \quad \varphi(n) \quad \dot{r}(n)]^T. \tag{6-14}$$

The relationship between the state of the Kalman filter and measurement vector is expressed as:

$$u(n) = H(s(n)) + v(n), \tag{6-15}$$

where

$$H(s(n)) = \begin{bmatrix} \sqrt{x^2 + y^2} \\ \tan^{-1}(x, y) \\ \frac{x\dot{x}+y\dot{y}}{\sqrt{x^2+y^2}} \end{bmatrix} \tag{6-16}$$

To linearize state to measurement relations, we perform partial derivatives

$$J_H(s) = \begin{bmatrix} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} & \frac{\partial r}{\partial \dot{x}} & \frac{\partial r}{\partial \dot{y}} & \frac{\partial r}{\partial \ddot{x}} & \frac{\partial r}{\partial \ddot{y}} \\ \frac{\partial \varphi}{\partial x} & \frac{\partial \varphi}{\partial y} & \frac{\partial \varphi}{\partial \dot{x}} & \frac{\partial \varphi}{\partial \dot{y}} & \frac{\partial \varphi}{\partial \ddot{x}} & \frac{\partial \varphi}{\partial \ddot{y}} \\ \frac{\partial \dot{r}}{\partial x} & \frac{\partial \dot{r}}{\partial y} & \frac{\partial \dot{r}}{\partial \dot{x}} & \frac{\partial \dot{r}}{\partial \dot{y}} & \frac{\partial \dot{r}}{\partial \ddot{x}} & \frac{\partial \dot{r}}{\partial \ddot{y}} \end{bmatrix} \tag{6-17}$$

$$J_H(s) = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} & 0 & 0 & 0 & 0 \\ \frac{y}{x^2+y^2} & -\frac{x}{x^2+y^2} & 0 & 0 & 0 & 0 \\ \frac{y(\dot{x}y-\dot{y}x)}{(x^2+y^2)^{3/2}} & \frac{x(\dot{y}x-\dot{x}y)}{(x^2+y^2)^{3/2}} & \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} & 0 & 0 \end{bmatrix} \tag{6-18}$$

6.4. Standard Kalman Filter Operations

6.4.1. Prediction Step

a) Compute the *a -priori* state estimate

$$s_{apr}(n) = Fs(n-1) \tag{6-19}$$

b) Compute the *a priori* state vector covariance Matrix

$$P_{apr}(n) = FP(n-1)F^T + Q(n-1). \tag{6-20}$$

6.4.2. Update Step

As measurements at time instant n become available, state and error covariance estimates are updated in the following measurement update procedure:

a) Compute innovation (or measurement residual)

$$y(n) = u(n) - H\left(s_{apr}(n)\right) \tag{6-21}$$

b) Compute innovation covariance

$$C(n) = J_H\left(s_{apr}(n)\right)P_{apr}(n)J_H^T\left(s_{apr}(n)\right) + R(n) \tag{6-22}$$

c) Compute Kalman gain

$$K(n) = P_{apr}(n)J_H^T\left(s_{apr}(n)\right)inv[C(n)] \tag{6-23}$$

d) Compute a-posteriori state vector

$$\mathbf{s}(n) = \mathbf{s}_{apr}(n) + \mathbf{K}(n)\mathbf{y}(n) \tag{6-24}$$

e) Compute a-posteriori error covariance

$$\mathbf{P}(n) = \mathbf{P}_{apr}(n) - \mathbf{K}(n)\mathbf{J}_H\left(\mathbf{s}_{apr}(n)\right)\mathbf{P}_{apr}(n) \tag{6-25}$$

6.4.3. Design of Process Noise Matrix

This section is adopted from the works in **Error! Reference source not found.**].

The choice of $\mathbf{Q}(n)$ is important for the behavior of the Kalman filter. If $\mathbf{Q}$ is too small then the filter will be overconfident in its prediction model and will diverge from the actual solution. If $\mathbf{Q}$ is too large than the filter will be too much influenced by the noise in the measurements and perform sub-optimally.

The kinematic system (a system that can be modeled using Newton's equations of motion) is continuous, i.e. their inputs and outputs can vary at any arbitrary point in time. However, Kalman filters used here are discrete. We sample the system at regular intervals. Therefore we must find the discrete representation for the noise term in the equation above. This depends on what assumptions we make about the behavior of the noise. We will consider two different models for the noise.

A. *Continuous White Noise Model*

Let's say that we need to model the position, velocity, and acceleration. We can then assume that acceleration is constant for each discrete time step. Of course, there is process noise in the system and so the acceleration is not actually constant. The tracked object will alter the acceleration over time due to external, un-modeled forces. In this section we will assume that the acceleration changes by a continuous time zero-mean white noise.

Since the noise is changing continuously we will need to integrate to get the discrete noise for the discretization interval that we have chosen. We will not prove it here, but the equation for the discretization of the noise is

$$Q = \int_0^{\Delta t} F(t)Q_c F^T(t)dt, \tag{6-26}$$

where Q_c is the continuous noise. The general reasoning should be clear. $F(t)Q_c F^T(t)$ is a projection of the continuous noise based on our process model $F(t)$ at the instant t. We want to know how much noise is added to the system over a discrete interval Δt, so we integrate this expression over the interval $[0, \Delta t]$.

For the second order Newtonian system, the fundamental matrix is

$$F = \begin{bmatrix} 1 & \Delta t & \Delta t^2/2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix}. \tag{6-27}$$

We now define the continuous noise as

$$Q_c = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \Phi_s \tag{6-28}$$

where Φ_s is the spectral density of the white noise. This can be derived, but is beyond the scope for now. In practice we often do not know the spectral density of the noise, and so this turns into an "engineering" factor - a number we experimentally tune until our filter performs as we expect. We can see that the matrix that Φ_s is multiplied by effectively assigns the power spectral density to the acceleration term. This makes sense; we assume that the system has constant acceleration except for the variations caused by noise. The noise alters the acceleration.

Computing the integral, we obtain

$$Q = \begin{bmatrix} \frac{\Delta t^5}{20} & \frac{\Delta t^4}{8} & \frac{\Delta t^3}{6} \\ \frac{\Delta t^4}{8} & \frac{\Delta t^3}{3} & \frac{\Delta t^2}{2} \\ \frac{\Delta t^3}{6} & \frac{\Delta t^2}{2} & \Delta t \end{bmatrix} \Phi_s \tag{6-29}$$

Extrapolating back to 6 states,

$$Q = \begin{bmatrix} \frac{\Delta t^5}{20} & 0 & \frac{\Delta t^4}{8} & 0 & \frac{\Delta t^3}{6} & 0 \\ 0 & \frac{\Delta t^5}{20} & 0 & \frac{\Delta t^4}{8} & 0 & \frac{\Delta t^3}{6} \\ \frac{\Delta t^4}{8} & 0 & \frac{\Delta t^3}{3} & 0 & \frac{\Delta t^2}{2} & 0 \\ 0 & \frac{\Delta t^4}{8} & 0 & \frac{\Delta t^3}{3} & 0 & \frac{\Delta t^2}{2} \\ \frac{\Delta t^3}{6} & 0 & \frac{\Delta t^2}{2} & 0 & \Delta t & 0 \\ 0 & \frac{\Delta t^3}{6} & 0 & \frac{\Delta t^2}{2} & 0 & \Delta t \end{bmatrix} \Phi_s \tag{6-30}$$

B. *Piecewise White Noise Model*

Another model for the noise assumes that the that highest order term (say, acceleration) is constant for the duration of each time period, but differs for each time period, and each of these is uncorrelated between time periods. In other words there is a discontinuous jump in acceleration at each time step. This is subtly different than the model above, where we assumed that the last term had a continuously varying noisy signal applied to it.

We will model this as

$$f(x) = Fx + \Gamma w \qquad (6\text{-}31)$$

where Γ is the noise gain of the system, and w is the constant piecewise acceleration (or velocity, or jerk, etc).

For the second order system

$$F = \begin{bmatrix} 1 & \Delta t & \frac{\Delta t^2}{2} \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix} \qquad (6\text{-}32)$$

In one time period, the change in acceleration will be $w(t)$, change in velocity will be $w(t)\Delta t$, and change in position will be $w(t)\Delta t^2/2$. This gives us

$$\Gamma = \begin{bmatrix} \Delta t^2/2 \\ \Delta t \\ 1 \end{bmatrix} \qquad (6\text{-}33)$$

The covariance of the process noise is then

$$Q = E[\Gamma w(t)w(t)\Gamma^T] = \Gamma \sigma_v^2 \Gamma^T \qquad (6\text{-}34)$$

$$Q = \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t \\ \frac{\Delta t^2}{2} & \Delta t & 1 \end{bmatrix} \sigma_v^2 \qquad (6\text{-}35)$$

It is not clear whether this model is more or less correct than the continuous model - both are approximations to what is happening to the actual object. Only experience and experiments can guide to the appropriate model. It is expected that either model provides reasonable results, but typically one will perform better than the other.

The advantage of the second model is that we can model the noise in terms of σ^2 which we can describe in terms of the motion and the amount of error we expect. The first model requires us to specify the spectral density, which is not very intuitive, but it handles varying time samples much more easily since the noise is integrated across the time period. However, these are not fixed rules - use whichever model (or a model of your own devising) based on testing how the filter performs and/or your knowledge of the behavior of the physical model.

A good rule of thumb is to set σ somewhere from $\frac{1}{2}\Delta a$ to Δa, where Δa is the maximum amount that the acceleration will change between sample periods. In practice we pick a number, run simulations on data, and choose a value that works well.

7. References

[1]   "mmwave SDK available [online] http://software-dl.ti.com/ra-processors/esd/MMWAVE-SDK/latest/index_FDS.html   GTRACK source code is in <mmwave_sdk_install_location>\packages\ti\alg\gtrack\src>."

[2]   "Group tracker tuning guide, Texas Instruments."

[3]   "https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python."

[4]   " Samuel S. Blackman. Multi-Target Tracking with Radar Applications, Artech House, 1986."

[5]   M. Ester, H. P. Kriegel, J. Sander, and X. Xiaowei, *A density-based algorithm for discovering clusters in large spatial databases with noise*: AAAI Press, Menlo Park, CA (United States), 1996.

[6]   "3D people counting implementation guide, Texas Instruments."