

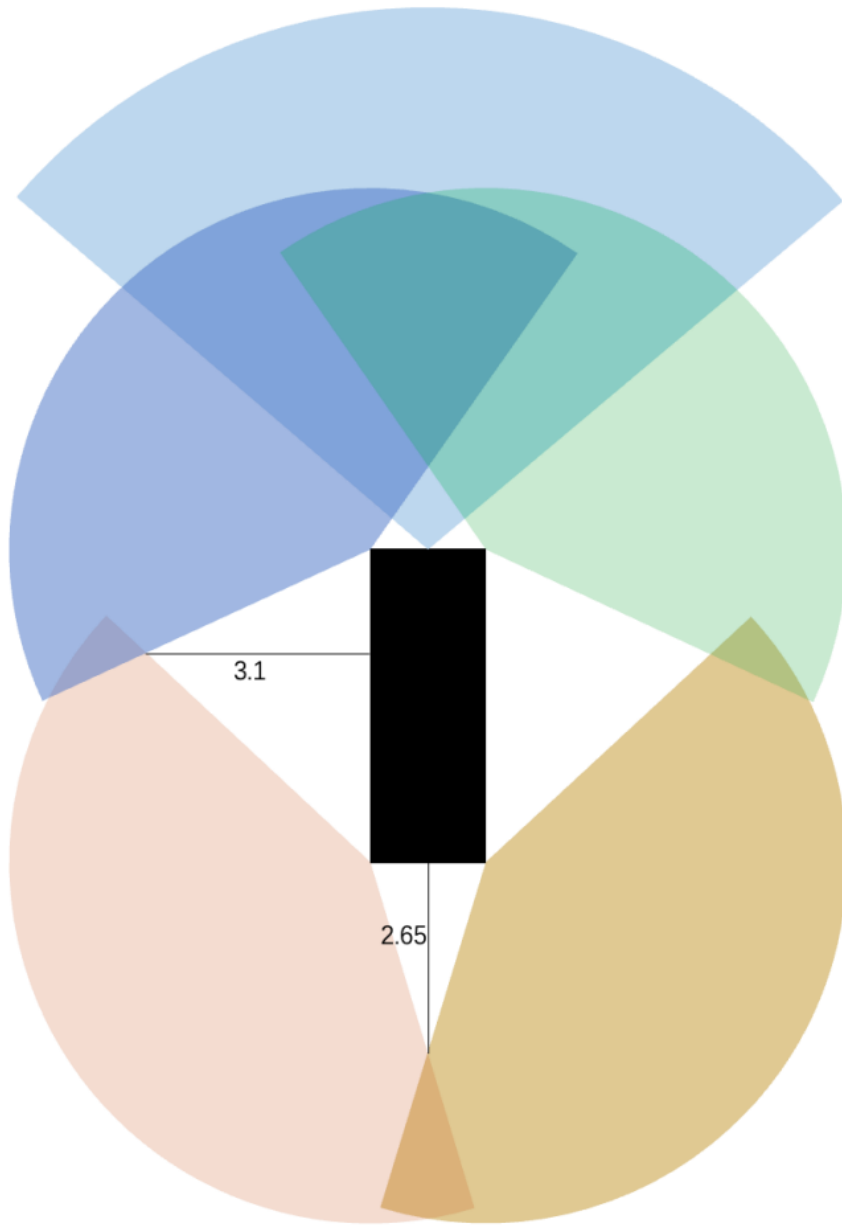
# Multi-Radar Multi-Object Track-To-Track Association and Fusion Algorithm

## 目录

- 背景
- 基础知识
  - 1. 多传感器融合跟踪系统分类
  - 2. Track-to-Track Fusion分类
- 系统架构
- 核心模块与算法
  - 一. 数据处理
  - 二. 数据同步
    - 异步问题处理
    - OOS问题处理
  - 三. track关联
    - 相似度计算
      - 1. 状态相似性
      - 2. 轨迹形状相似性
      - 3. 轨迹长度相似性
      - 4. 相似性融合
    - 关联矩阵求解
      - 1. 算法流程
      - 2. 求解示例
- 参考

## 背景

- 目前, 集度车辆上共装有5个Radar, 其中1个为远距离前向Radar, 4个为中距离角Radar
- 在之前的Radar感知系统中, 只对前向Radar进行了处理, 角Radar没有充分利用
- 因此, 考虑将前向Radar与角Radar进行融合, 实现360度Radar全范围感知



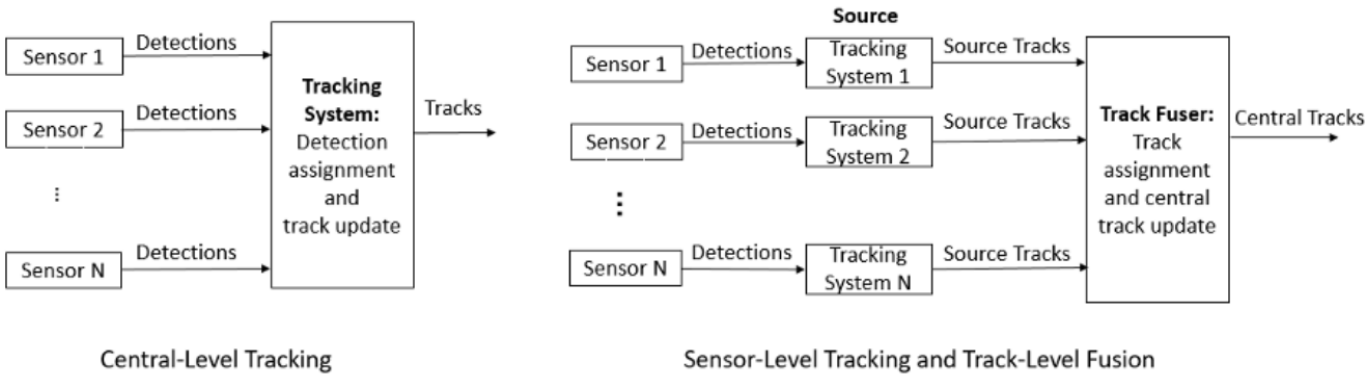
Radar安装与FOV示意图

## 基础知识

### 1. 多传感器融合跟踪系统分类

多传感器融合跟踪系统通常存在两种结构

- 中心化融合跟踪(Centralized Fusion, Central-Level Tracking/Fusion): 所有传感器的检测直接发送到跟踪系统, 其理论上可以实现最佳性能, 因为它可以充分利用检测中包含的所有信息
- 分布式融合跟踪(Distributed Fusion, Track-to-Track Fusion, Track-level Fusion): 将多传感器融合系统拆分为层次结构: 传感器级跟踪和track级融合, 接受到的信息更加集中, 效率更高



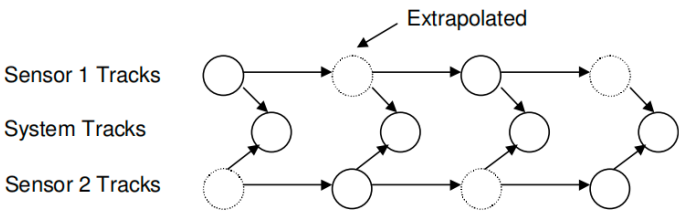
多传感器融合跟踪系统分类 [1]

由于Radar传感器内部已经存在信号处理算法和目标跟踪算法, 因此, 对于多Radar的融合跟踪, 其天然的更适用于分布式融合, 也就是Track-To-Track融合.

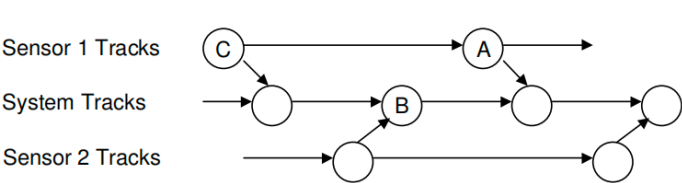
2. Track-to-Track Fusion分类

Track-to-Track Fusion存在两种架构

- Sensor to Sensor Track Fusion: 其在将不同sensor track进行关联融合时不使用system track先前的状态估计, 一般用于超过两个的传感器融合
  - 优点: 是不需要处理track之间的相关性(其没有propagate, 属于无记忆操作)
  - 缺点: 由于过去的状态没有使用, 效果上不如另一种
- Sensor to System Track Fusion: 其将system track外推到sensor track时刻并与其进行关联融合
  - 优点: sensor track到system track的关联问题简化为二分图匹配问题
  - 缺点: 必须处理track之间的相关性, 必须处理oos(out-of-sequence)问题



Sensor to Sensor Track Fusion [2]



Sensor to System Track Fusion [2]

由于多Radar融合存在以下两个特点:

- 传感器数量较多(5个), 传感器数据之间存在oos问题(radar没有做严格时间同步)
- Radar为大融合主要提供速度信息, 对多个Radar的速度进行聚类后, 再结合lidar和camera的速度, 可以为大融合的多假设速度融合提供方便

因此, 采用Sensor to Sensor Track Fusion的形式, 将不同sensor track进行关联聚类, 并将聚类后的簇发送给下游

系统架构

对于本次采用的Sensor to Sensor Track Fusion架构, 将其分为传感器层(Sensor Layer)和融合层(Fusion Layer)[3], 其中:

- 传感器层: 执行特定于传感器的任务, 并为融合层提供与传感器无关的数据. 其核心是每个传感器都有一个独特的数据处理模块(Process Module), 主要包括几个任务:
  - 数据滤波: 去除传感器噪声, 提取有用信息
  - 空间对齐: 将传感器测量从sensor坐标系转到global坐标系
  - 提取元信息: 提取用于数据融合的元信息: 时间戳, 位置, 速度, sensor id, FOV等
- 融合层: 从传感器层异步接收与传感器无关的数据, 并对其进行关联融合, 输出最终的system track. 其主要包括两大步骤, 数据同步和track关联:
  - 数据同步: 对接收到的异步数据进行软件同步, 方便对相邻时间戳的数据进行统一处理
  - track关联: 将来自于同一目标的不同传感器的track进行关联聚类, 生成最终的system track

## 核心模块与算法

### 一. 数据处理

数据处理是传感器层的核心模块, 其对不同传感器数据进行针对性预处理, 并提炼与传感器无关的通用核心信息, 用于后续的融合模块. 目前Radar感知模块提供了两种数据处理算法:

- 通用数据处理: 通用的Radar数据处理算法, 目前主线车辆前向雷达ARS408, 集度车辆前向雷达ARS513, 集度车辆角雷达STA79-2 Pro, 都是采用该模块
- ARS430数据处理: 针对大陆ARS430专用的数据处理模块, 目前未使用

对于目前的集度5R配置: 1个前向雷达ARS513, 4个角雷达STA79-2 Pro. 本次考虑采用不同的数据处理模块, 每个模块参考之前的通用数据处理, 并根据每个传感器的特性进行针对性调整, 主要处理步骤如下:

1. 数据帧有效性验证: 根据Radar返回的信号量, 对标定状态, 工作状态等进行检查, 如果发现异常情况进行报警
2. 目标有效性验证: 去除不在时间范围内的目标, 去除零点目标, 去除重复目标, 去除RCS过低目标, 去除存在性过低目标, 去除质量无效目标(quality\_valid)等
3. 目标ID扩展: 由于雷达使用can总线传输数据, 其数据量比较小, 因此目标跟踪ID也比较小. 这里将其ID扩展到更大范围, 同时也可以进行不同传感器的区分 -> 这里需要注意的是513没有输出跟踪ID, 并且刚出现的目标检测状态可能并不是new, 需要加一个位置保护
4. 提取通用信息: 将Radar提供的数据格式转到用于融合的通用数据格式, 并将目标从sensor坐标系转到global坐标系

 加载中...

**i** 待确认项: 是沿用之前的通用数据处理还是重新写一个单独的数据处理

- 通用数据处理: 工作量小, 但是新加功能可能会影响其他传感器使用
- 专用数据处理: 逻辑清晰, 可扩展和维护性强, 但是工作量较大

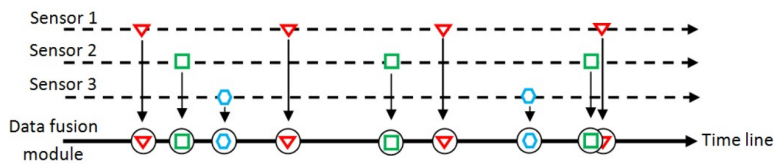
## 二. 数据同步

### 异步问题处理

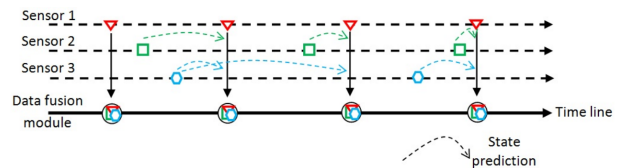
对于多传感器融合, 非常关键的一点是要保证不同传感器数据时间顺序的一致性并且进行同一时刻的数据融合. 对于现有的Radar传感器来说, 无法像lidar或者camera那样进行同步触发, 每个Radar的时间戳都各不相同. 因此, 为了处理传感器数据异步问题, 一般存在两种方法:

- 异步融合: 在每次接收到新传感器数据时执行融合. 其更新频率不稳定, 并且更新质量取决于最新传感器提供信息的准确性和类型, 同时需要融合后的system track作为桥梁进行更新.

- 软件同步: 定义融合率, 然后在每个融合时间计算每个传感器数据的估计值, 并使用所有估计值执行融合. 其更新频率比较稳定, 对system track没有要求, 但是利用预测状态进行更新, 精度上可能会有损失.



异步融合[3]



软件同步[3]

由上文可知, 我们采用Sensor to Sensor Track Fusion架构, 并且只关注不同radar track之间的关联, 不进行system track的融合更新. 同时, 由于前雷达频率(15hz)和角雷达频率(18hz)比较接近, 状态预测的时间比较小, 误差基本可控. 因此, 我们采用软件同步的方式处理异步问题.

## OOS问题处理

对于现有的Radar传感器配置, 时间同步采用ACU时间, 但是前雷达时间戳是最后一次收发波时间, 角雷达是第一次收发波时间, 这中间会差10-20ms. 同时, 由于雷达内部存在信号处理和跟踪算法, 以及数据传输需要时间, 这些总耗时被称为latency, 其根据场景和目标数据具有较大波动(目前前雷达大约在140ms, 角雷达大约在90ms). 基于这两点, 融合算法接受到同一时刻不同传感器数据的时间相差会比较大, 甚至可能出现旧数据比新数据先到的情况, 这种情况称为OOS(out-of-sequence). 为了解决OOS问题, 一般大多都会采用buffer缓存每个融合时刻的数据, 当检测到OOS数据时, 将其插入到缓冲区的对应位置中, 并重新执行对应时间到当前时间的融合. 这种重新执行融合(redo fusion)的方式不是很高效, 当OOS问题比较频繁时(Radar的latency不固定, OOS在复杂场景下可能会比较多), 整个算法的耗时不能满足需求. 因此, 我们提出一种新的处理OOS的数据结构, 称为Time Slice System. 其将整个时间线划分为一个个slice, 并且根据传感器频率预先分配每个传感器的slot, 每当一个传感器数据到来时只需要将其插入对应的slot即可. 其主要有以下优势:

- 解决OOS问题: 由于我们预先知道每个slice对应的传感器数量, 因此可以对传感器数据是否到齐进行判断, 等同一slice的所有数据到齐后进行融合, 可以很好的避免OOS问题.
- 加速track关联: 由于我们将时间线划分为一个个slice, 当我们利用track的历史轨迹计算相似度时, 历史轨迹点的对应关系可以直接获取, 不用再进行相应计算

### 三. track关联

与之前object-to-object的关联类似, track-to-track的关联主要也包括两个核心模块: 相似度计算和关联矩阵求解. 其中:

- 相似度计算: 利用时间, 空间, 跟踪稳定性等信息计算两个track是否为同一目标的相似性, 相似性越大, 为同一目标的概率越大.
- 关联矩阵求解: 当track之间的相似度计算完成后, 就要确定两两之间的对应关系, 一般将其作为二分图匹配问题, 就要进行关联矩阵的求解.

#### 相似度计算

目前采用的2.5D Radar输出目标的点模型, 状态量主要包括目标的位置和速度, 因此之前的object-to-object关联也一般使用位置和速度的距离进行关联. 但是, 对于现在的track-to-track关联, 我们有了目标的历史轨迹, 如何充分利用轨迹信息就成了重点. 由于Radar的观测模型是点模型, 不同Radar对于同一目标的观测位置可能不尽相同, 比如对于自车正前方的目标, 前雷达的目标点可能在目标车尾中心, 左角雷达的目标点可能在目标的车尾左侧, 右角雷达的目标点可能在目标的车尾右侧. 同时, Radar目标分类能力较弱, 很容易将路边的绿化带或者路杆检测成障碍物(这里我们认为是误检), 这会非常影响我们关联的准确性. 这就要求我们能够关联到距离较远目标的同时不会错误关联, 基于此, 我们考虑以下三种相似性:

### 1. 状态相似性

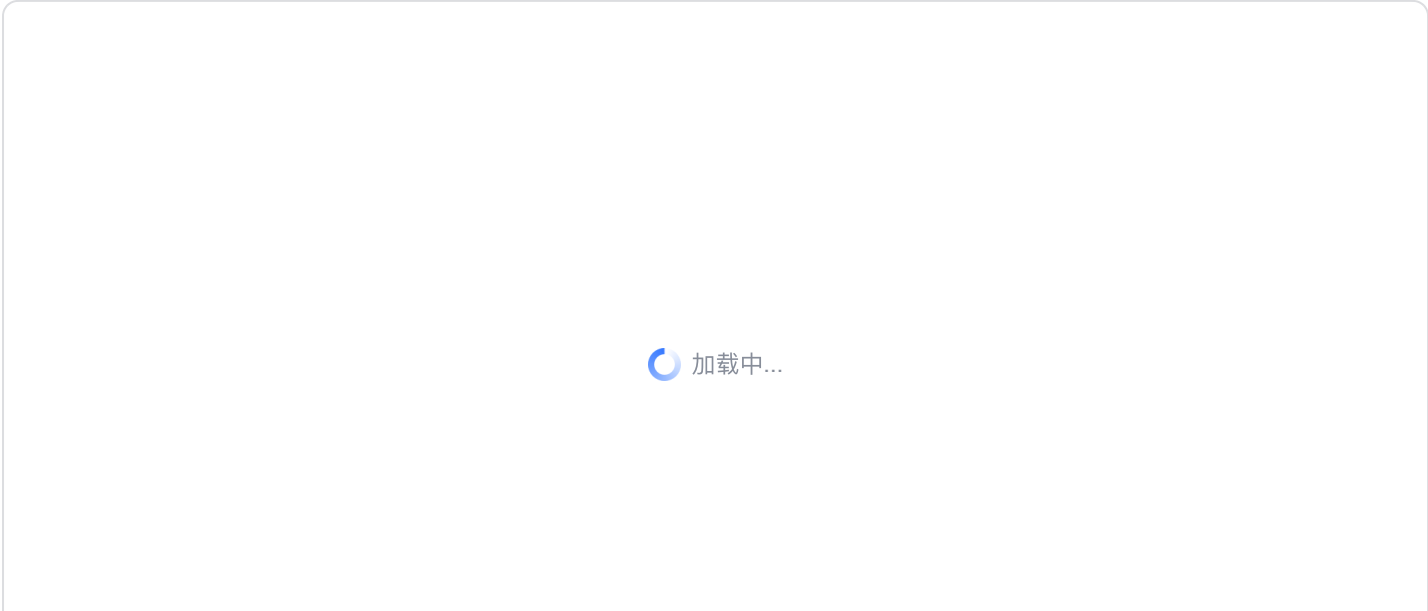
目标状态的相似性是最基础也是最有效的, 对于Radar来说, 状态量主要包含位置和速度, 同时Radar会提供每个状态量的不确定性, 因此状态量的相似性一般采用其马氏距离. 对于整个track轨迹的状态量相似性, 我们采用所有轨迹点中状态量相似性最小的一个, 这样可以去掉轨迹形状的影响, 而轨迹形状的相似性会在下一个进行考虑.



计算两个轨迹之间的最小轨迹

### 2. 轨迹形状相似性

对于轨迹形状相似性的计算, 我们对每一对轨迹点的状态相似性先减去上面计算的track状态相似性, 然后计算所有轨迹点的剩余相似性的均值, 这样可以排除轨迹之间bais的影响





3. 轨迹长度相似性

由于radar误检目标一般轨迹较短, 因此为了防止radar误检目标与正常目标关联, 需要计算轨迹长短的相似性



4. 相似性融合



关联矩阵求解

对于关联矩阵求解, 最常用的方法是贪心算法和匈牙利算法, 其中:

- 贪心算法: 每次取相似性最小的一对认为关联成功, 然后依次取最小知道关联成功. 其是一种局部最优算法, 但是可扩展性较强.
- 匈牙利算法: 其将关联问题看到二分图匹配, 利用最优化方法最小化匹配后的cost. 其是一种全局最优算法, 但是计算复杂度较高, 一般用于两种传感器之间的关联.

由于我们关联的时候存在5个传感器, 传感器数量较多, 匈牙利算法的复杂性将会很高, 因此采用贪心算法, 将其从两个传感器扩展到多个传感器, 具体如下:

## 1. 算法流程

算法流程如下所示, 与两个传感器的关联相比主要有三点不同:

1. 增加了3b步骤: 保证同一sensor之间的track不会关联
2. 增加了4b步骤: 默认track之间的关联具有连通性, 这样可以将多个传感器的track聚为一类
3. 修改了4d步骤: 在将已经关联的行列设为最大值时, 从以前的整个矩阵变成某一传感器

---

**Algorithm 1** Multisensor Track-To-Track Association
 


---

- 1) Collect the tracks of all the sensors
  - 2) Assign a number from 1 to  $N$  to each track,  $N$  being the total number of tracks.
  - 3) Create a  $N \times N$  array for the TTTDs between the tracks
    - a) Set cells over the diagonal to a defined maximal value ( $MaxVal$ ) in order not to compute twice the distance between two same tracks.
    - b) Set cells corresponding to two tracks of the same sensor to  $MaxVal$ , in order not to associate two tracks of a same sensor.
    - c) Set the remaining cells to the distance between the corresponding two tracks.
    - d) Set cells where the distance is greater than a defined threshold to  $MaxVal$ . The threshold symbolizes a gate out of which we assume that the two tracks cannot originate from the same target.
  - 4) Loop: Determine the minimal value ( $MinVal$ ) of the array and its position ( $lin, col$ ) in the array. While  $MinVal$  is smaller than  $MaxVal$ 
    - a) If none of the corresponding two tracks has not been inserted in a cluster yet, then put both of them in a new cluster.
    - b) If only one has already been inserted in a cluster, then add the second to that cluster.
    - c) If both have already been inserted in a cluster (the same or not), then do nothing.
    - d) Set to  $MaxVal$  cells in the line  $lin$  and the ones in the column  $col$ , that correspond to tracks reported by the two concerned sensors.
  - 5) Each track that has not been inserted in a cluster forms a new cluster (singletons).
- 

算法流程

## 2. 求解示例

求解示例如下所示, 一共4个传感器, 4个目标

 A Track-To-Track Association Method for Automotive Perception Systems.pdf (246KB)

## 参考

- [1] [Introduction to Track-To-Track Fusion](#)
- [2] Architectures and Algorithms for Track Association and Fusion
- [3] A Track-To-Track Association Method for Automotive Perception Systems