Track–To–Track 数据结构调整

目录

目标

异步输入五路Radar跟踪结果, 进行track-to-track关联融合, 输出一路融合结果

输入输出

- 输入: 五路Radar数据以异步形式输入, 数据类型为SensorFrameMessage
- 输出: 一路多Radar融合结果输出, 数据类型为SensorFrameMessage, 其中cluster信息以Object中的measurements形式输出

主要步骤

1. 对输入的Radar数据进行软件同步和异常检测. 其采用上一章提到的Time Slice Sysytem技术, 主要包含以下步骤

a. 输入数据异常检测: 同一sensor的时间戳逆序, 同一sensor的时间戳跳变 -> 分低中高三个等级, 分别进行不同程度的报警或功能降级
b. 输入数据插入到Time Slice System, 主要包括: TimeLine更新, 开辟新的slice, 插入数据, 预计算关联距离
c. sensor数据丢失检测: 判断某一路sensor是否故障, 分为低中高三个等级, 进行不同成的报警和功能降低, 同时更新数据到达状态
d. 可关联状态判断: 判断某个slice的数据是否"到齐", 如果到齐则意味着这个slice中的所有数据可以进行关联

2. 数据关联/跟踪. 其是对Track-to-Track Association的具体实现, 主要包含以下步骤
   a. 判断是否可以关联, 如果不能直接返回(不是每一帧数据进来都能关联)
   b. 数据关联: 对slice中多个sensor的数据进行关联, 生成一个个cluster
   c. Cluster ID管理: 对时序上cluster的ID进行关联, 也就是让同一cluster的ID在时序上一致
   d. 释放内存: 这里需要将frame的内存释放, 因为当前slice和之前slice的frame数据不再使用(只使用距离)
   e. 收集结果: 关联得到的cluster存在message中
3. 结果输出. 以SensorFrameMessage的形式输出

伪代码

</> Bash | 收起

```
-- component
    -- multi_radar_fusion_component.h
    -- multi_radar_fusion_component.cc
-- lib
    -- track_fusion
        -- track_fusion.h
        -- track_fusion.cc
        -- typedefs
            -- common.h
        -- slice_manager
            -- slice_manager.h
            -- slice_manager.cc
        -- track_association
            -- track_association.h
            -- track_association.cc
            -- history_distance.h
            -- history_distance.cc
            -- history_distance_manager.h
            -- history_distance_manager.cc
            -- greedy_matcher.h
            -- greedy_matcher.cc
```

```
        -- cluter_tracker
            -- cluster_track.h
            -- cluster_track.cc
            -- cluster_track_manager.h
            -- cluster_track_manager.cc
            -- cluster_association.h
            -- cluster_association.cc
```

1. Component

C++ | 收起

```cpp
class MultiRadarFusionComponent {
    void Proc(SensorFrameMessage in_message) {
        SensorFrameMessage out_message;
        if (track_fusion.Fuse(in_message, out_message)) {
            publish(out_message);
        }
    }
};
```

2. TrackFusion

C++ | 收起

```cpp
class TrackFusion {
    bool Fuse(in_message, out_message) {
        // 1. 输入数据软件同步与异常监测
        slice_manager.UpdateFrame(in_message);

        // 2. 判断是否可以关联并获取相关数据
        RadarSliceFrame::Ptr radar_slice_frame;
        if (!slice_manager.GetSliceFrame(radar_slice_frame)) return false;

        // 3. 数据关联
        RadarClusterFrame::Ptr radar_cluster_frame;
        track_association.Associate(radar_slice_frame, radar_cluster_frame);

        // 4. cluster track 管理
        cluster_track_manager.Track(radar_cluster_frame, out_message);

        // 5. 历史距离计算释放内存
        histoty_distance_manager.DeleteExpiredHistoryDistance();
```

```
    }
};
```

3. SliceManager

C++ | 收起

```
// 第一版可以先做简单一点，不考虑时间戳漂移，以收到的第一帧frame的时间戳作为起始时间，按照给定的传感器频率划分slice，并且按照给定的传感器数量划分sub_slice，
// 当新的frame到来时，要么插入到slice中，要么统计丢帧的次数(这个后面会有用，后面升级时丢帧率就是评价的指标)，如果被插入的slice是full状态，则将数据转为slice_frame，用于TTTA
// 最好也统计一个延迟，也就是数据从发出到可以用于TTTA的耗时，这也是后续升级的评价指标
// 同时，需要加一个Reset模块，如果连续多帧时间戳都发生了跳变，则必须重启silice，并且重启整个fusion

// 后续升级主要是两个方向：一是减少丢帧率(考虑边界情况，考虑时间戳抖动，timeline校准是必须要加的)，二是减少延迟

class SliceManager {
  public
    bool Init() {
        // 从文件中加载配置，初始化以下几个变量
        slices_buffer_.set_capacity();
        cycle_time_;
        sensor_name_2_sub_slice_idx_;
    }
    void Reset() {
        // 这里非常重要，当时间戳跳变时，必须重启
        slices_buffer_.clear();
    }
    void UpdateFrame(SensorFrameMessage in_message) {
        // 1.1 输入数据异常检测：同一sensor的时间戳逆序(忽略这一帧数据，低中高)，同一sensor的时间戳跳变(高级别需要重启slice，低中高)
        // NOTE: 第一版先做时间戳跳变检测，一个是runmode切换时时间戳不连续会跳变，另一个是因为AVP切ANP经常会出现时间戳跳变
        // NOTE: 这个非常重要，一定要将时间戳跳变的frame剔除掉，不然后面插入时一堆坑
        if (!AnomalyDetect(in_message)) {
            return;
        }

        // 1.2 输入数据插入到Time Slice System(需要将前面slice中的状态全部设置为missing)
        UpdateInternal(in_message);

```

```
        // 1.3 sensor数据丢失检测: 连续2帧没有但是其他传感器都是arrived, 也是需要低中高
        // NOTE: 这个目前可以先忽略
        SensorMissingDetect();
    }
    bool GetSliceFrame(RadarSliceFrame::Ptr radar_slice_frame) {
        for (slice : slices_buffer_) {
            if (slice.IsAbleAssociation()) {
                radar_slice_frame->timestamp = 0.5 * (slice.start_timestamp + slice.end_timestamp);
                radar_slice_frame->objects.reserve(40 + 16 * 4); // 最大的目标数量
                radar_slice_frame->sensor_ids.reserve(40 + 16 * 4); // 最大的目标数量
                size_t sensor_id = 0;
                for (sub_slice : slice.sub_slices) {
                    if (sub_slice.frame == nullptr) continue;
                    for (const auto& object : sub_slice.frame->objects) {
                        radar_object = object;
                        radar_slice_frame->objects.push_back(radar_object);
                        radar_slice_frame->sensor_ids.push_back(sensor_id);
                    }
                    sub_slice.frame = nullptr; // NOTE: release frame
                    ++sensor_id;
                }
                slice.has_associated = true; // NOTE: release frame -> 只能调用一次
                return true;
            }
        }
        return false;
    }
  private:
    bool AnomalyDetect(SensorFrameMessage in_message) {
        // slice为空则直接返回true
        if (slices_buffer_.empty()) return true;

        // 时间戳跳变检测, 比最新的大很多, 或者比最旧的小很多
        if (in_message.timestamp - slices_buffer_.back().end_timestamp > 10.0 ||
            slices_buffer_.front().start_timestamp - in_message.timestamp > 10.0) {
            AERROR << "timestamp jump " << in_message.timestamp << ...;
            timestamp_jump_count_++;
```

```
            // 当时间戳跳变的次数大于阈值，则表示系统性跳变，重启slice，否则先丢弃数据
            if (timestamp_jump_count_ > 3) {
                this->Reset();
            }
            return false;
        }

         // NOTE: 时间戳逆序检测放在了更新步骤里面，如果发现已经更新过，则直接返回
    }
    void UpdateInternal(SensorFrameMessage in_message) {
        // case 1: slice buffer为空或者输入时间戳大于buffer的时间戳
        if (slices_buffer_.empty() || in_message->timestamp > slices_buffer_.back().start_timestamp) {
            InsertNewSlices(in_message->timestamp);
        }

        // case 2: 输入数据时间戳小于buffer的时间戳，slice已经释放掉，则这一帧数据无效
        if (in_message->timestamp < slices_buffer_.front().start_timestamp) {
            // TODO: 丢帧统计
            return;
        }

        // 判断应该落在哪个slice中
        int slice_idx = (in_message->timestamp - slices_buffer_.front().start_timestamp) / cycle_time_;
        if (slice_idx < 0 || slice_idx >= slices_buffer_.size()) {
            // NOTE: 这里理论上不能进来，前面要做好保护
            AERROR << "invalid slice " << in_message->timestamp << " " << slice_idx;
        }
        // 判断应该落在哪个sub_slice中
        auto iter = sensor_name_2_sub_slice_idx_.find(in_message->sensor_name);
        if (iter == sensor_name_2_sub_slice_idx_.end()) {
            // NOTE: 这里理论上不能进来，前面要做好保护
            AERROR << "invalid sensor name " << in_message->sensor_name;
            return;
        }
        int sub_slice_idx = iter->second;
        // 获取sub_slice
        auto& sub_slices = slices_buffer_[slice_idx].sub_slices;
        auto& sub_slice = sub_slices[sub_slice_idx];
        // 如果已经更新过，则不再更新
        if (sub_slice.arrival_state >= ArrivalState::MISSING) {
```

```
            // TODO: 统计一下这种重新更新或者逆序的数量
            return;
        }
        // 更新当前sub_slice的状态
        sub_slice.arrival_state = ArrivalState::ARRIVED;
        sub_slice.frame = in_message;
        // NOTE: 提前进行距离计算 -> 输入的frame会和已经存在的所有frame计算距离
        for (i = 0; i < sub_slices.size(); ++i) {
            histoty_distance_manage.UpdateHistotyDistance(sub_slices[i], sub_slices[sub_slice_idx]) {
        }
        // 释放过期的内存
        histoty_distance_manager.DeleteExpiredHistoryDistance(in_message->timestamp);
        // 更新之前sub_slice的状态 -> 从未到达设置为MISSING
        --slice_idx;
        while (slice_idx >= 0) {
            if (slices_buffer_[slice_idx].sub_slices[sub_slice_idx].arrival_state == ArrivalState::NOT_ARRIVED) {

slices_buffer_[slice_idx].sub_slices[sub_slice_idx].arrival_state == ArrivalState::MISSING;
            } else {
                break;
            }
        }
    }
    void InsertNewSlices(timestamp) {
        // 获取需要插入的slice数量和时间戳
        double start_timestamp = 0.0;
        size_t new_slices_count = 0;
        if (slices_buffer_.empty()) {
            // 初始化: 插入固定两个slice, 时间戳位于第一个slice的中心
            start_timestamp = timestamp - 0.5 * cycle_time_;
            new_slices_count = 2;
        } else {
            // Normal: 根据输入的时间戳判断需要插入的slice数量, 但是不能超过阈值(时间戳有跳变), 时间戳跟着之前的slice走
            start_timestamp = slices_buffer_.back().end_timestamp;
            new_slices_count = (timestamp - slices_buffer_.back().end_timestamp) / cycle_time_ + 1;
            if (new_slices_count > 2) {
                // NOTE: 这里理论上不能进来, 前面要做好保护
```

```
                AERROR << "invalid new slice count " << new_slices_count;
                return;
            }
        }

        // 插入新的slice
        while (new_slices_count > 0) {
            Slice slice;
            slice.start_timestamp = start_timestamp;
            slice.end_timestamp = start_timestamp + cycle_time_;
            // NOTE: sub_slice的数量最好和传感器数量一致，这样方便后面的遍历等操作
            slice.sub_slices.resize(sensor_name_2_sub_slice_idx_.size()); //
初版sub_slice使用默认值就可以
            slices_buffer_.push_back(slice);
            start_timestamp = slices_buffer_.back().end_timestamp;
            new_slices_count--;
        }
    }
    void SensorMissingDetect() {
        // slice为空则直接返回
        if (slices_buffer_.empty()) return;

        // 遍历所有的slice，获取每个sensor数据缺失的帧数
        std::vector<int>
sensor_missing_times(slices_buffer_.front().sub_slice.size(), 0);
        for (slice : slices_buffer_) {
            for (i = 0; < slice.sub_slices.size(); ++i) {
                if (slice.sub_slices[i].arrival_state <
ArrivalState::ARRIVED) {
                    sensor_missing_times[i] += 1;
                } else {
                    // 清空
                    sensor_missing_times = 0;
                }
            }
        }

        // 对于缺失帧数大于阈值的sensor，将其状态设置为CANT_ARRIVED，这样在判断是否可
以关联时，就不考虑他了
        for (i = 0; i < sensor_missing_times.size(); ++i) {
            if (sensor_missing_times[i] > 4) {
                // 整个传感器失效了
                for (slice : slices_buffer_) {
```

```
                    if (slice.sub_slices[i].arrival_state < ArrivalState::MISSING) {
                        slice.sub_slices[i].arrival_state = ArrivalState::CANT_ARRIVED;
                    }
                }
            }
        }
    }
    boost::circular_buffer<Slice> slices_buffer_;
    // TODO: timeline可以先不实现，而直接使用固定频率，但是后续必须加，尽量减少前雷达丢帧
    double cycle_time_ = 0.60; // 60ms
    // SensorTimeline slice_timeline_; // 对时间戳做校准，解决时间戳漂移 -> 当前帧时间戳+传感器频率(设定值)
    // std::vector<SensorTimeline> sensors_timeline_;
    // 传感器名称到其索引的map，索引必须从0开始逐次递增
    std::unordered_map<std::string, size_t> sensor_name_2_sub_slice_idx_;
    std::vector<AnomalyDetector> anomaly_detectors_;
};

struct Slice {
    double start_timestamp;
    double end_timestamp;
    std::vector<SubSlice> sub_slices;
    bool has_associated = false;
    bool IsAbleAssociation() {
        // 已经关联过
        if (has_associated) return false;
        // 可关联状态判断: 只要有NOT_ARRIVED就表示不能关联
        for (sub_slice : sub_slices) {
            if (sub_slice.arrival_state == ArrivalState::NOT_ARRIVED) return false;
        }
        return true;
    }
};

struct SubSlice {
    ArrivalState arrival_state = ArrivalState::NOT_ARRIVED;
    SensorFrameMessage frame = nullptr;
};

enum class ArrivalState { NOT_ARRIVED = 0, CANT_ARRIVED, MISSING, ARRIVED };
```

```
/*
struct SliceDistance {
    float GetDistance(sensor1, sensor2, track1, track2) {
        std::string sensor_key = SensorKey(sensor1, sensor2);
        std::string track_key = TrackKey(track1, track2);
        return distances.find(sensor_key).find(track_key);
    }
    // 第一层map是两个sensor之间，第二层map是两个object之间
    // 两个sensor之间计算距离只有6种情况：正前-左前，正前-右前，左前-右前，左前-左后，右前-右后，左后-右后
    // 在查找时需要满足交换不变性，比如正前-左前，左前-正前的查询结果是一样的，建议给传感器编号，编号后将小的放前面
    // 两个track之间计算距离时，也要满足查询时的交换不变性，编号小的sensor的track id在前
    // 如果要将两个string combine成一个，中间一定要加特殊字符，不然'123','21'和'12','321'一样
    std::map<std::string, std::map<std::string, float>> distances;  // 第一层map是两个sensor之间，第二层map是两个object之间
};
*/

/*
SensorTimeline {
    double cycle_time;
    double cycle_noise;
    double control_latest_tstamp;
    double output_latest_tstamp;
    void Predict();
    void Update();
};
*/
```

4. TrackAssociation

TTTA pipeline C++ | 收起

```
// track_association.h 计算关联矩阵，求解关联矩阵，返回关联结果

class TrackAssociation {
    std::vector<std::vector<size_t>> Associate(radar_slice_frame, radar_cluster_frame) {
        // 1. 创建关联矩阵
```

```
        // TODO: 这里的关联矩阵，最好大小都是预先分配好内存的，因为radar最大输出目标数量是已知的
        const auto& objects = radar_slice_frame->objects;
        Eigen::MatrixXf matrix(objects.size(), objects.size());
        for (i = 0; i < objects.size(); ++i) {
            for (j = 0; j < objects.size(); ++j) {
                // 2.1 对角线和上对角元素全为最大值
                if (i <= j) {
                    matrix(i, j) = std::numeric_limits<float>::max();
                }
                // 2.2 + 2.3 获取历史距离 -> 这里在获取时暗含着，如果为同一sensor或没有fov交叠，则距离为最大值
                distance = histoty_distance_manager.GetHistoryDistance(objects[i].sensor_name, objects[i].track_id, objects[j].sensor_name, objects[j].track_id);
                // 2.4 如果距离大于阈值，则直接设为最大值
                if (distance > params_.distance_thresh()) {
                    distance = std::numeric_limits<float>::max();
                }
                // 设置剩余有效的distance
                matrix(i, j) = distance;
            }
        }

        // 2. 求解关联矩阵
        clusters = greedy_matcher.Match(matrix, radar_slice_frame->sensor_ids, params_.distance_thresh());

        // 3. 生成关联结果
        radar_cluster_frame->timestamp = radar_slice_frame->timestamp;
        radar_cluster_frame->clusters.reserve(clusters.size());
        for (cluster : clusters) {
            RadarCluster::ptr radar_cluster;
            for (idx : cluster) {
                radar_cluster->push_back(objects[idx]);
            }
            // fuse state
            radar_cluster->timestamp = radar_slice_frame->timestamp;
            ClusterFusion(radar_cluster);
            // push to frame
            radar_cluster_frame->clusters.push_back(radar_cluster);
        }
    }
    void ClusterFusion(RadarCluster::ptr radar_cluster) {
```

```
        // 这里可以做一个简单的CI融合
        radar_cluster->position = ();
        radar_cluster->velocity = ();
        radar_cluster->position_uncertainty = ();
        radar_cluster->velocity_uncertainty = ();
    }
};

```

关联距离计算模块 C++ | 收起

```
// 整体的设计思路时，每接收到一帧frame后，计算其与其他frame之间的距离，并且更新到历史距离中，后续关联只需在历史距离的map中查找即可，省去了距离计算的耗时

// history_distance.h
class HistotyDistance {
    HistotyDistance(size_t buffer_size, double expired_thresh) {
        distances.set_capacity(buffer_size);
        expired_thresh_  = expired_thresh;
    }
    UpdateDistance(object1, object2) {
        // update buffer
        distance = ComputeObjectDistance(object1, object2);
        distances_.push_back(distance);
        // fuse distances
        min_distance = 0;
        sum_distance = 0;
        for (distance : distances_) {
            min_distance = min(distance, min_distance);
            sum_distance += distance;
        }
        length = distances_.size();
        sum_distance -= min_distance * length;
        fused_distance_ = a * min_distance + b * sum_distance + c * length;
    }
    float GetDistance() { return fused_distance_; }
    bool IsExpired(float timestamp) {
        if (!std::isnan(latest_timestamp_) && std::abs(timestamp - latest_timestamp_) > expired_thresh_) {
            return true;
        }
        return false;
    }
  private:
```

```
    float ComputeObjectDistance() {
        // 计算目标之间的距离: 欧式距离, 马氏距离都行
    }
    boost::circular_buffer<float> distances_;
    float fused_distance_ = std::numeric_limits<float>::max();
    double latest_timestamp_ = NAN;
    double expired_thresh_ = 5.0;
};

// history_distance_manager.h
class HistotyDistanceManager {
  public:
    bool Init() {
        // 读取proto文件, 主要是读取哪两个sensor之间可以计算相似性, 目前来看只需要6种组合, 可以减少计算量
        valid_sesnor_pairs_.insert(SensorToString(sensor_name1, sensor_name2));
    }
    // 输入两帧测量, 计算两两之间的相似性, 并更到到map中  -> 每次drame插入到slice中时调用
    void UpdateHistotyDistance(SensorFrameMessage frame1, SensorFrameMessage frame2) {
        if (frame1 == nullptr || frame2 == nullptr) return;
        std::string sensor_name1 = frame1->sensor_name;
        std::string sensor_name2 = frame2->sensor_name;
        if (!IsAbleToCompute(sensor_name1, sensor_name2)) return;
        for (i = 0; i < frame1.objects.size(); ++i) {
            for (j = 0; j < frame2.objects.size(); ++j) {
                string = SensorTrackToString(sensor_name1, frame1.objects[i]->track_id, sensor_name2, frame2.objects[j]->track_id);
                auto iter = history_distances_.find(string);
                if (iter == history_distances_.end()) {
                    histoty_distance = HistotyDistance(params.buffer_size, params_.expired_thresh);
                    iter = history_distances_.insert(hash, histoty_distance);
                }
                iter->second.UpdateDistance(frame1.objects[i], frame2.objects[i]);
            }
        }
    }
    // 根据输入的传感器名称和track_id, 获取对应的历史距离 -> 计算关联矩阵时调用
    float GetHistoryDistance(sensor_name1, track_id1, sensor_name2, track_id2) {
```

```
        if (!IsAbleToCompute(sensor_name1, sensor_name2)) {
            return std::numeric_limits<float>::max();
        }
        string = SensorTrackToString(sensor_name1, track_id1, sensor_name2,
track_id2);
        auto iter = history_distances_.find(string);
        if (iter == history_distances_.end()) {
            return std::numeric_limits<float>::max();
        } else {
            return iter->second.GetDistance();
        }
    }
    // 删除过期的历史距离  -> 每次算法关联矩阵后调用, 和释放frame内存一起
    void DeleteExpiredHistoryDistance() {
        for (auto iter = history_distances_.begin(); iter !=
history_distances_.end();) {
            if (iter->second.IsExpired()) {
                iter = history_distances_.erase(iter);
            } else {
                ++iter;
            }
        }
    }
  private:
    bool IsAbleToCompute(sensor_name1, sensor_name2) {
        // 如果输入的sensor在列表中, 则返回true, 否则返回false
        string = SensorToString(sensor_name1, sensor_name2);
        return valid_sesnor_pairs_.find(string);
    }
    std::string SensorToString(sensor_name1, sensor_name2) {
        if (sensor_name1 < sensor_name2) {
            return sensor_name1 + '#' + sensor_name2;
        } else {
            return sensor_name2 + '#' + sensor_name1;
        }
    }
    std::string SensorTrackToString(sensor_name1, track_id1, sensor_name2,
track_id2) {
        if (sensor_name1 < sensor_name2) {
            return sensor_name1 + '#' + std::to_string(track_id1) + '#' +
sensor_name2 + '#' + std::to_string(track_id2);
        } else {
            return sensor_name2 + '#' + std::to_string(track_id2) + '#' +
sensor_name1 + '#' + std::to_string(track_id1);
```

```
        }
    }
    std::unordered_map<std::string> valid_sesnor_pairs_;
    std::unordered_map<std::string, HistotyDistance> history_distances_;
};
```

关联矩阵求解 C++ | 收起

```
// greedy_matcher.h 求解关联矩阵 -> 在求解时先对所有可能关联的pair进行排序，然后借助一个已经被分配的flag矩阵(三维矩阵)，避免每次找最小值再设置矩阵

struct AssociationHypothesis {
  size_t raw_idx;
  size_t col_idx;
  float score;

  Hypothesis(int raw_idx, int col_idx, float score) {
    this->raw_idx = raw_idx;
    this->col_idx = col_idx;
    this->score = score;
  }

  bool operator<(const Hypothesis &b) const { return score < b.score; }

  bool operator>(const Hypothesis &b) const { return score > b.score; }
};

class GreedyMatcher {
  public:
    std::vector<std::vector<size_t>> Match(matrix, sensor_ids, distance_thresh) {
        // 1. 获取所有的可能关联并排序
        std::vector<AssociationHypothesis> asso_hypos;
        for (i = 0; i < rows; ++i) {
            for (j = 0; j < cols; ++j) {
                if (matrix[i][j] < distance_thresh) {
                    asso_hypos.emplace_back(i, j, matrix[i][j]);
                }
            }
        }
        std::sort(asso_hypos.begin(), asso_hypos.end(), std::greater<AssociationHypothesis>());

        // 2. 获取关联clusters
```

```
        // 2.1 生成分配flag矩阵 -> TODO: 如何进行简化
        std::vector<size_t> sensor_start_idx;
        for (size_t i = 0; i < sensor_ids; ++i) {
            if (sensor_ids[i] == sensor_start_idx.size()) {
                sensor_start_idx.push_back(i);
            } else if (sensor_ids[i] > sensor_start_idx.size()) {
                for (size_t j = sensor_start_idx.size(); j < sensor_ids[i]; ++j) {

sensor_start_idx.push_back(sensor_start_idx[sensor_start_idx.size() - 1]);
                }
            }
        }
        std::vector<std::vector<bool>> sensor_flags;
        for (size_t i = 1; i < sensor_start_idx.size(); ++i) {
            std::vector<bool> flags(sensor_start_idx(i) - sensor_start_idx(i - 1), false);
            sensor_flags.push_back(flags);
        }
        sensor_flags.push_back(std::vector<bool>(sensor_ids.size() - sensor_start_idx[sensor_start_idx.size() - 1], false));
        std::vector<std::vector<std::vector<bool>>> multi_sensor_flags(sensor_start_idx.size(), sensor_flags);
        // 2.2 遍历所有可能的关联假设，生成关联结果
        std::vector<std::vector<size_t>> clusters; // 关联结果
        std::vector<int> assignments(matrix.rows(), -1); // 用于表示track被分配到第几个cluster，-1表示未分配  -> 用于track的合并
        for (const auto& asso_hypo : asso_hypos) {
            row_sensor_id = sensor_ids[asso_hypo.row];
            row_sensor_id_idx = row - sensor_start_idx[row_sensor_id]
            col_sensor_id = sensor_ids[asso_hypo.col];
            col_sensor_id_idx = col - sensor_start_idx[col_sensor_id]
            // 如果sensor之间关联过，不能再关联
            if (sensor_flags[row_sensor_id][col_sensor_id][col_sensor_id_idx] || sensor_flags[col_sensor_id][row_sensor_id][row_sensor_id_idx]) {
                continue;
            }
            // 可以关联
            sensor_flags[row_sensor_id][col_sensor_id][col_sensor_id_idx] = true;
            sensor_flags[col_sensor_id][row_sensor_id][row_sensor_id_idx] = true;
            // 1. 两个都没有被分配，创建新的cluster
            if (assignments[row] == -1 && assignments[col] == -1) {
```

```
                assignments[row] = clusters.size();
                assignments[col] = clusters.size();
                clusters.push_back({row, col});
            }
            // 2. 其中一个track已经创建cluster
            else if (assignments[row] == -1) {
                clusters[assignments[col]].push_back(row);
                assignments[row] = assignments[col];
            }
            else if (assignments[col] == -1) {
                clusters[assignments[row]].push_back(col);
                assignments[col] = assignments[row];
            }
            // 3. 两个都已经创建cluster
            // do nothing
            // 4. 将不再等关联的目标设置为最大值
        }
        // 2.3 创建单例cluster -> 前雷达静止目标不做关联，本身就是一个cluster；有些目标没有重叠，本身也是一个cluster
        for (i : assignments.size()) {
            if (assignments[i] == -1) {
                assignments[i] = track_clusters.size();
                clusters.push_back({i});
            }
        }

        return clusters;
    }
};

```

5. ClusterTrackManager

ClusterTrack C++ | 收起

```
class ClusterTrack {
  public:
      bool Init(RadarCluster::Ptr cluster, int track_id) {
          track_id_ = track_id;
          UpdateWithMeasurement(cluster);
      }
      void Reset() {
          track_id_ = -1;
```

```
        fused_cluster_.Reset();
    }
    void UpdateWithMeasurement(RadarCluster::Ptr cluster) {
        fused_cluster_ = cluster;
        track_time_info_.Update(cluster.timestamp);
        // maybe more state update
        // 这里可能也可以用一个ci融合，得到状态
    }
    void UpdateWithoutMeasurement(double timestamp) {
        track_time_info_.Propagate(timestamp);
    }
  private:
    int track_id_ = -1;
    RadarCluster::Ptr fused_cluster_;
    TrackTimeInfo track_time_info_;
};
```

ClusterTrackManager C++ | 收起

```
class ClusterTrackManager {
  public:
    bool Init() {
        track_id_manager_.Init(1, 1000000);
        return true;
    }
    void Reset() {
        track_id_manager_.Reset();
    }
    void Track(radar_cluster_frame, out_message) {
        association_results = cluster_association.Associate(cluster_tracks_,
radar_cluster_frame)
        UpdateAssignedClusterTracks(radar_cluster_frame, association_results-
>assignments);
        UpdateUnassignedClusterTracks(radar_cluster_frame->timestamp,
association_results->unassigned_tracks);
        CreateNewClusterTracks(radar_cluster_frame, association_results-
>unassigned_meas);
        TerminateTracks();
        CollectResults(out_message);
    }
  private:
    void UpdateAssignedClusterTracks(radar_cluster_frame, assignments) {
      for (const auto& pair : assignments) {
```

```
        cluster_tracks_[pair.first]->UpdateWithMeasurement(radar_cluster_frame->clusters[pair.second]);
      }
    }
    void UpdateUnassignedClusterTracks(timestamp, unassigned_tracks) {
        for (const auto & idx : unassigned_tracks) {
            cluster_tracks_[idx]->UpdateWithoutMeasurement(timestamp);
        }
    }
    void CreateNewClusterTracks(radar_cluster_frame, unassigned_meas) {
        for (const auto & idx : unassigned_tracks) {
            ClusterTrackPtr cluster_track(new ClusterTrack());
            cluster_track->Init(radar_cluster_frame->clusters[idx], track_id_manager_.Assign());
            cluster_tracks_.push_back(cluster_track);
        }
    }
    void TerminateTracks() {
        auto is_deleted = [&](track) {
            if (track->IsDeleted()) {
                track_id_manager_.Release(track->GetTrackId());
                return true;
            }
            return false;
        };
        auto removed_iter = std::remove_if(cluster_tracks_.begin(), cluster_tracks_.end(), is_deleted);
        cluster_tracks_.erase(removed_iter, cluster_tracks_.end());
    }
    void CollectResults(out_message) {
        for (cluster_track : cluster_tracks_) {
            position = cluster_track->fused_cluster_->position;
            // TOOD
        }
    }
    std::vector<ClusterTrackPtr> cluster_tracks_;
    TrackIDManager track_id_manager_;
};
```

6. ClsuterAssociation

</> C++ | 收起

```
// 这里主要是将cluster在时序上串起来，整体思路是先进行完整id assign，再进行前雷达id assign，最后再计算关联矩阵并求解
class CluterAssociation{
    void Associate(cluster_tracks, radar_cluster_frame, association_result) {
        // NOTE: 这里的实现完全可以参考大融合，包括ID Assign的调用等
        // 1. 完整ID Assign -> 这里track和meas的clsuter的id必须完全对应
        // 2. 指定传感器ID Assign -> 这里只需要指定传感器的id能对应上就行
        // 3. 计算关联矩阵 -> 利用cluster的融合中心计算距离 -> 欧式或马氏都行
        // 4. 求解关联矩阵 -> 初步可以先用贪心求解
    }
}
```

7. Typedefs

</> C++ | 收起

```
enum class MotionState {
  UNKNOWN = 0,
  MOVING = 1,
  STOPPED = 2,  // 相比于base中的motion_state，这里加一个stoped，方便做一些判断
  STATIONARY = 3,
};

struct alignas(16) RadarObject {
    typedef std::shared_ptr<RadarObject> Ptr;
    typedef std::shared_ptr<const RadarObject> ConstPtr;
    // header
    int track_id = -1    ;
    double timestamp = 0.0;
    std::string sensor_name;
    // state
    Eigen::Vector2d position = Eigen::Vector2d::Zero();
    Eigen::Vector2f velocity = Eigen::Vector2f::Zero();
    Eigen::Matrix2f position_uncertainty = Eigen::Matrix2f::Zero();
    Eigen::Matrix2f velocity_uncertainty = Eigen::Matrix2f::Zero();
    base::MotionState motion_state;
    // raw data
    int origin_id;
    float life_time = 0.0f;
    float rcs = -100.0f;
    float exist_prob = 0.0f;
    float mirror_prob = 100.0f;
    float obstacle_prob = 0.0f;
```

```
    radar::MotionState raw_motion_state = RawMotionState::UNKNOWN;
};

struct alignas(16) RadarFrame {
    typedef std::shared_ptr<RadarFrame> Ptr;
    typedef std::shared_ptr<const RadarFrame> ConstPtr;
    double timestamp = 0.0;
    std::string sensor_name;
    std::vector<RadarObject::Ptr> objects;
};

// 这个比较特殊，将slice里面所有的数据都放到一个frame里面，作为TTTA的输入
struct alignas(16) RadarSliceFrame {
    typedef std::shared_ptr<RadarSliceFrame> Ptr;
    typedef std::shared_ptr<const RadarSliceFrame> ConstPtr;
    double timestamp = 0.0;
    std::vector<RadarObject::Ptr> objects;
    std::vector<size_t> sensor_ids; // 每一个object对应的sensor id
};

// TTTA得到的cluster
struct alignas(16) RadarCluster {
    typedef std::shared_ptr<RadarCluster> Ptr;
    typedef std::shared_ptr<const RadarCluster> ConstPtr;
    // header
    int track_id = -1;
    double timestamp = 0.0;
    // state
    Eigen::Vector2d position = Eigen::Vector2d::Zero();
    Eigen::Vector2f velocity = Eigen::Vector2f::Zero();
    Eigen::Matrix2f position_uncertainty = Eigen::Matrix2f::Zero();
    Eigen::Matrix2f velocity_uncertainty = Eigen::Matrix2f::Zero();
    MotionState motion_state;
    // cluster
    std::vector<RadarObject::Ptr> objects;
};

struct alignas(16) RadarClusterFrame {
    typedef std::shared_ptr<RadarClusterFrame> Ptr;
    typedef std::shared_ptr<const RadarClusterFrame> ConstPtr;
    double timestamp = 0.0;
    std::vector<RadarCluster::Ptr> clusters;
};
```

8. Utils

```cpp
// 对输入的多个目标进行编码, 形成sesor_name+track_id的形式
class Encoder {
    std::string Encode(RadarObject::Ptr object) {
        return object->sensor_name + '#' + std::to_string(object->track_id);
    }
    std::string Encode(RadarObject::Ptr object1, RadarObject::Ptr object2) {
       string1 = Encode(object1);
       string2 = Encode(object2);
       if (string1 < string2) {
            return string1 + '#' + string2;
        } else {
            return string2 + '#' + string1;
        }
    }
    std::string Encode(RadarCluster::Ptr cluster) {
        ret = "";
        for (const auto object : cluster->objects) {
            string = Encode(object);
            if (ret < string) {
                ret = ret + string;
            } else {
                ret = string + ret;
            }
        }
        return ret;
    }
};
```

9. RadarSupplement

- 对于一个cluster而言, radar的raw_data尽可能的合并, 在fusion的时候减少supplement的使用

- [x] exist_prob: 数据关联会使用 -> 目前其主要是用于确定特别低置信度目标和特别高置信度目标, 因此在cluster中将其融合为一个置信度等级, 分为低中高三种, 这样在关联时, 只需要判断置信度等级
- [x] mirror_prob: 数据关联会使用 -> 目前其主要是用于确定特别低置信度目标和特别高置信度目标, 因此在cluster中将其融合为一个置信度等级, 分为低中高三种, 这样在关联时, 只需要判断置信度等级

- [ ] sensor_name: 城市motion_fusion会用 -> 这个后面可以在cluster中进行区分, 状态是前雷达还是角雷达, 这样大融合做状态融合的时候只需要拿标志位
- [x] vehicle_yaw_rate: avp, 城市, 高速的motion fusion会用 -> 这个直接用frame里面的egocar_angular_velocity就行, 前面也是这样赋值的
- [ ] underlying_velocity: 数据关联会使用, 城市motion_fusion会用 -> 这个标志为在目标跟踪完之后进行计算, 不要放在
- [ ] is_underlying_moving: 数据关联会使用, 城市motion_fusion会用 -> 对个cluster中的object进行融合, 还是在这个变量里面
- [x] range: gatekeeper会用, 但是该函数没调用 -> 如果是单个目标, 那就用原始的, 如果多个目标直接为0
- [x] angle: gatekeeper会用, 但是该函数没调用 -> 如果是单个目标, 那就用原始的, 如果多个目标, 直接为0
- [x] origin_center: 城市shape融合 -> 这个不变, 只有下游自在用
- [x] move_gap_center: 城市shape融合 -> 这个不变, 只有下游自在用
- [x] refined_center: 城市shape融合 -> 这个不变, 只有下游自在用
- [x] ~~origin_id~~
- [x] ~~longitude_dist~~
- [x] ~~lateral_dist~~
- [x] ~~res~~
- [x] obstacle_prob
- [x] life_time
- [x] ~~raw_motion_state~~
- [x] ~~radar_seg_type~~
- [x] ~~radar_roi_type~~
- [x] ~~debug_origin_vel_x~~
- [x] ~~debug_origin_vel_y~~
- [x] ~~debug_origin_vel_z~~

TODO

- [ ] 系统 @顾恺琦
  - [ ] 可视化
    - [x] V1: radar_detection和radar_fusion结果都发布出来, 并用rviz进行可视化
    - [ ] cluster如何更好的可视化

- [x] dueye可视化更新 -> 图像, 驱动, detect, fusion, 关联, big fusion
- [ ] run mode切换: slice manager需要重启, cluster_track_manager需要重启, HistoryDistanceManager需要重启, 但是id assigner不能重启 -> 能不能radar内部不做城市和高速区分, 等到最后输出给大融合的时候, 统一做一次坐标系转换即可
- [ ] 数据流梳理
  - [ ] radar速度和航向角解耦？-> 速度投影放到最后
  - [ ] 哪些变量有赋值一定要理清楚, 变量之间如何转换也要理清楚, 避免下游拿不到数据, 特别是supplement中的数据
  - [ ] cluster的结果如果传输到下游, 下游如何进行适配
- [ ] slice_manager @顾恺琦
  - [ ] 减少丢帧率: 时间线校准等
  - [ ] 异常检测: 时间戳逆序, 时间戳跳变等
- [ ] 评价速度norm和朝向, fov
- [ ] track_association @王晓亮
  - [x] 相似性计算: 目前采用位置的均值, 后续需要升级到多特征
  - [x] 关联矩阵求解: 目标是贪心算法, 后续看要不要升级, 优先级较低
  - [ ] cluster状态量融合: 现在使用均值或投票, 后续需要升级到概率形式
    - [ ] 有前雷达: 直接用前雷达
    - [ ] 没有前雷达: 位置, 速度, 朝向, 大小 -> 均值, 类型, 动静态 -> 投票
  - [ ] 角雷达位置输出形式确认, 并基于此输出形式优化关联cost
- [ ] cluster_track_manager @王晓亮
  - [x] 相似性计算: 目前两个cluster之间的相似性直接使用距离, 后续需要进行升级
  - [x] 生命周期管理: 目前是基于计数的方法, 后续可以考虑升级, 优先级较低
  - [ ] cluster_track状态量融合: 目前没有融合, 后续需要考虑滤波方法
    - [ ] 没有滤波
- [ ] radar障碍物单报 @王晓亮

排期

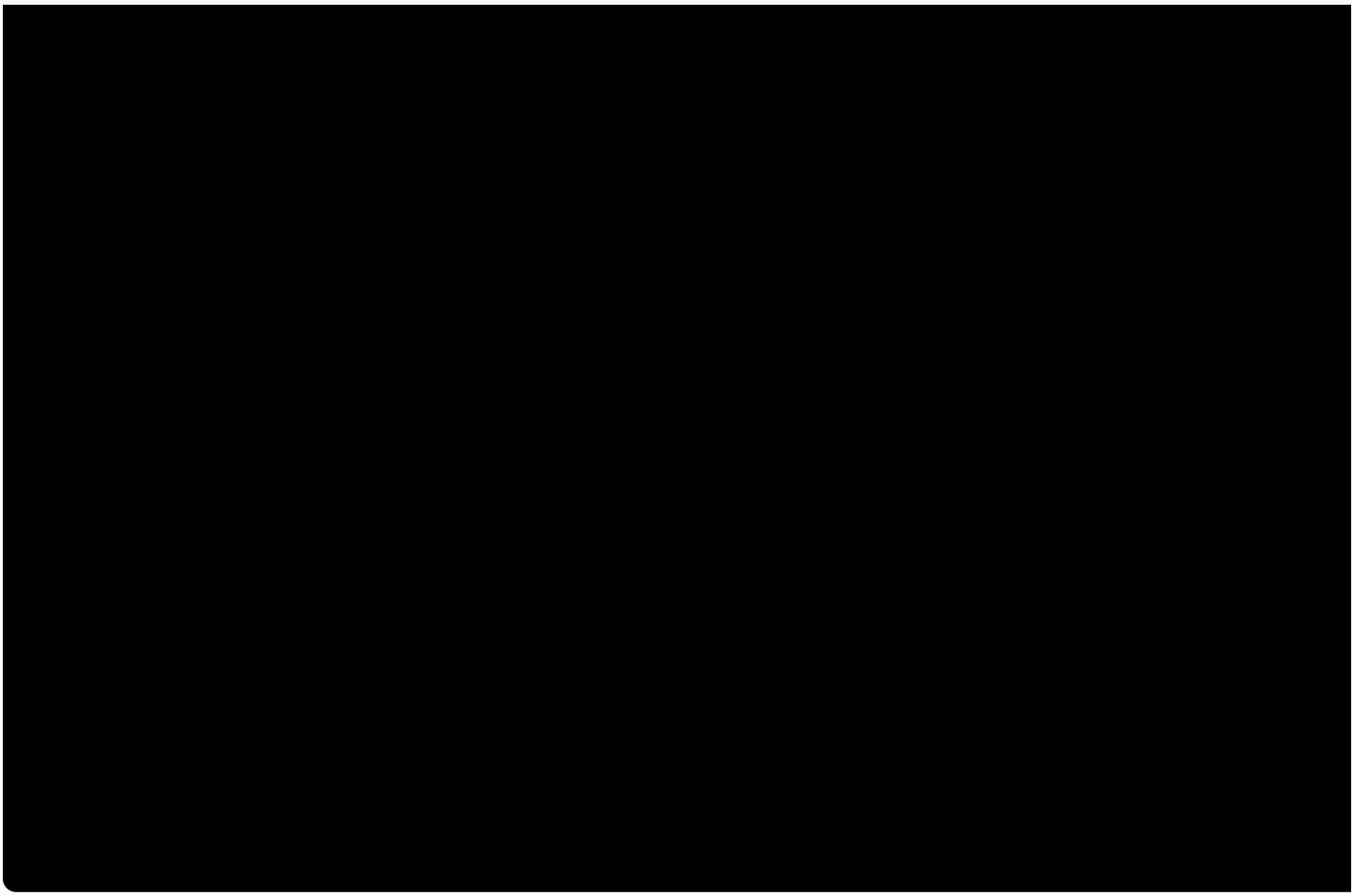
- [ ] 9.15之前: TODO项目搞完
- [ ] 9.22之前: radar单独报出, 下游联调
- [ ] 9.28之前: 路测一周, 合入

效果

V1.0-2023.9.4

- 黄色是track_fusion结果, 后面的字母+数字, 表示关联上的radar名称+id
- 其他颜色为不同radar的感知结果

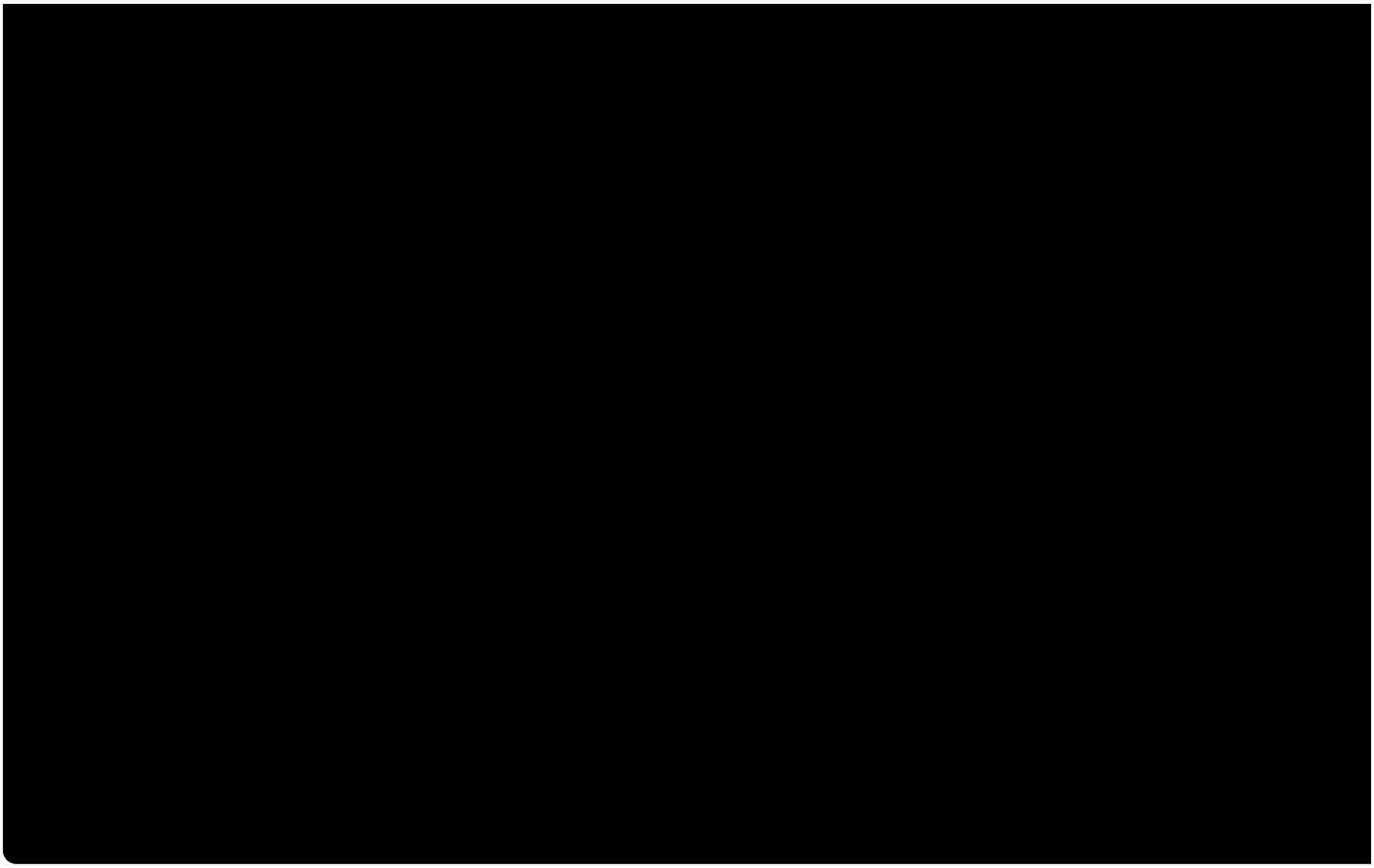
track_fusion_v1.mp4 (10MB)



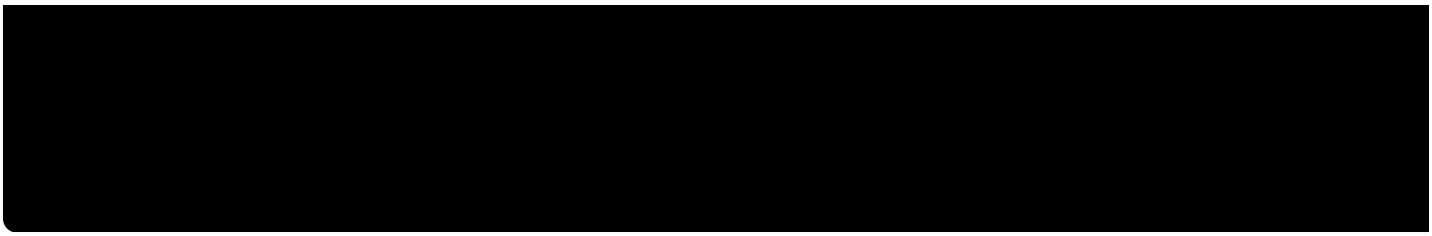
V1.1-2023.9.13

andes_radar.mp4 (78MB)

andes_radar.mp4 (36MB)



andes_radar.mp4 (41MB)



- [ ] 时间戳可视化: 对各路radar的时间戳做可视化: 时间戳 + 抵达时间 -> perception log

多毫米波雷达时间戳-汇总