


HUSTLY26-步兵自瞄技术报告：

 徐天洋

Abstract:

1.介绍：

AutoAim_infantry步兵自瞄体系基于C++开发。该系统通过深度学习目标检测、卡尔曼滤波运动预测、坐标解算和控制算法等技术，实现对敌方装甲板的高精度自动识别与瞄准，同时支持能量机关(能量Buff)的识别与击打，具备完整的硬件驱动、串口通信、UDP调试和录像功能。系统具有良好的扩展性和实时性，能够适应比赛中的复杂环境和快速变化的目标。

2.环境依赖：

支持WSL2，Ubuntu20.04和22.04版本。

依赖库：

- 1 - OpenCV 4.8.0+
- 2 - Eigen3
- 3 - Boost
- 4 - spdlog
- 5 - OpenVINO opset>14.0
- 6 - Ceres

3.使用：

脚本：

```
1 cd AutoAim_infantry
2 ls
3 """"编译""""
4 cd build
5 cmake ..
6 make -j8
7 """"启动""""
8 cd ../bin
9 ./AutoAim
10
11 """"支持网页调试""""
```

```
12 cd utils/scripts
13 python3 server.py
```

4.易食用性与稳定性：

1.采用模块化编程：

无需ROS依赖，易于上手。

各模块间采用不同的命名空间，同时为哨兵的自瞄模块提供了接口。

2.并发处理机制：

系统采用多线程并发处理：

各自涉及mutex_lock，以及用atomic进行修改。

- **主线程**：负责算法逻辑处理 AutoAim.cpp
- **串口线程**：异步处理串口数据收发 serialthread
- **相机线程**：异步获取相机图像数据 camerathread

3.关键技术特点：

3. **数据同步机制**：通过时间戳确保图像和IMU数据同步
4. **预测算法**：补偿系统延迟，提高命中率
5. **双模式支持**：支持常规装甲板打击和大能量机关打击
6. **配置驱动**：通过配置文件灵活调整系统参数

7.为提高准确率，我们在各模块进行一系列的创新处理，见各模块介绍。

Menus：

AutoAim/

- |—— modules/ # 核心功能模块
- | |—— driver/ # 驱动模块
- | |—— buff/ # 新增打符模块
- | |—— detector/ # 检测模块
- | |—— tracker/ # 跟踪模块
- | |—— predictor/ # 预测模块

- | |—— solver/ # 解算模块
- | |—— controller/ # 控制模块
- | |—— replayer/ # 回放模块
- |—— utils/ # 工具库
 - | |—— include/ # 工具头文件目录
 - | |—— log/ # 日志模块头文件
 - | |—— video/ # 视频处理模块头文件
 - | |—— udp/ # UDP 模块头文件（传输协议）
 - | |—— param/ # 参数管理模块头文件（重载operator(),读取config.json的方法）
 - | |—— time/ # 时间戳模块头文件
 - | |—— location/ # 位置信息模块头文件（这里定义了Basesolver:世界，相机，以及角度形式的三种坐标，留虚函数接口，实现在solver中实现）
 - | |—— recorder/ # 录制模块头文件
- |—— scripts/ # 脚本目录，包括可视化工具
 - |—— server.py # 配置工具服务器，在网页127.0.1.8081中启动web调试。
 - |—— client.py # 配置工具客户端
 - |—— solverFit/ # 解算模块拟合工具。python代码，读取x,y,pitch,yaw拟合。
 - |—— process-PitchYawFit.py # 解算模块拟合工具。python代码，读取4个量。
 - |—— visualizer.py # 可视化工具
- |—— bin/ # 编译输出目录（主程序AutoAim的执行程序在这里）
- |—— CMakeLists.txt # CMake 配置文件

readme:

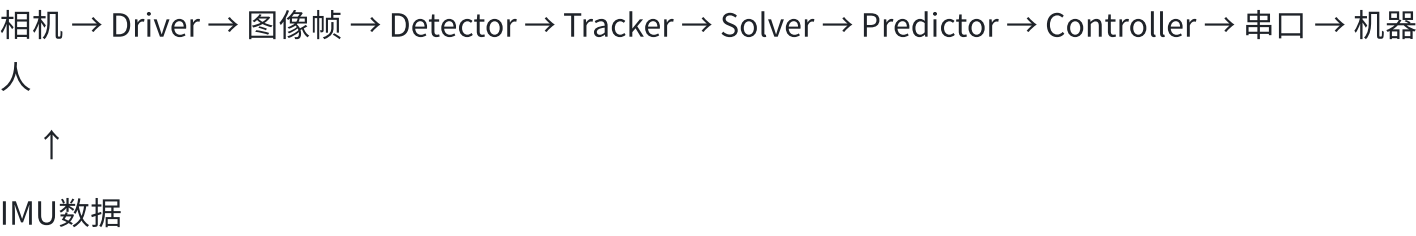
在utils中，定义了各种工具，如时间戳处理，json文件读取与转化，串口数据的传输，日志记录与debug。同时实现了modules中部分类的基类与接口。

在modules中，实现功能的主要模块，从识别到解算等一系列流程。

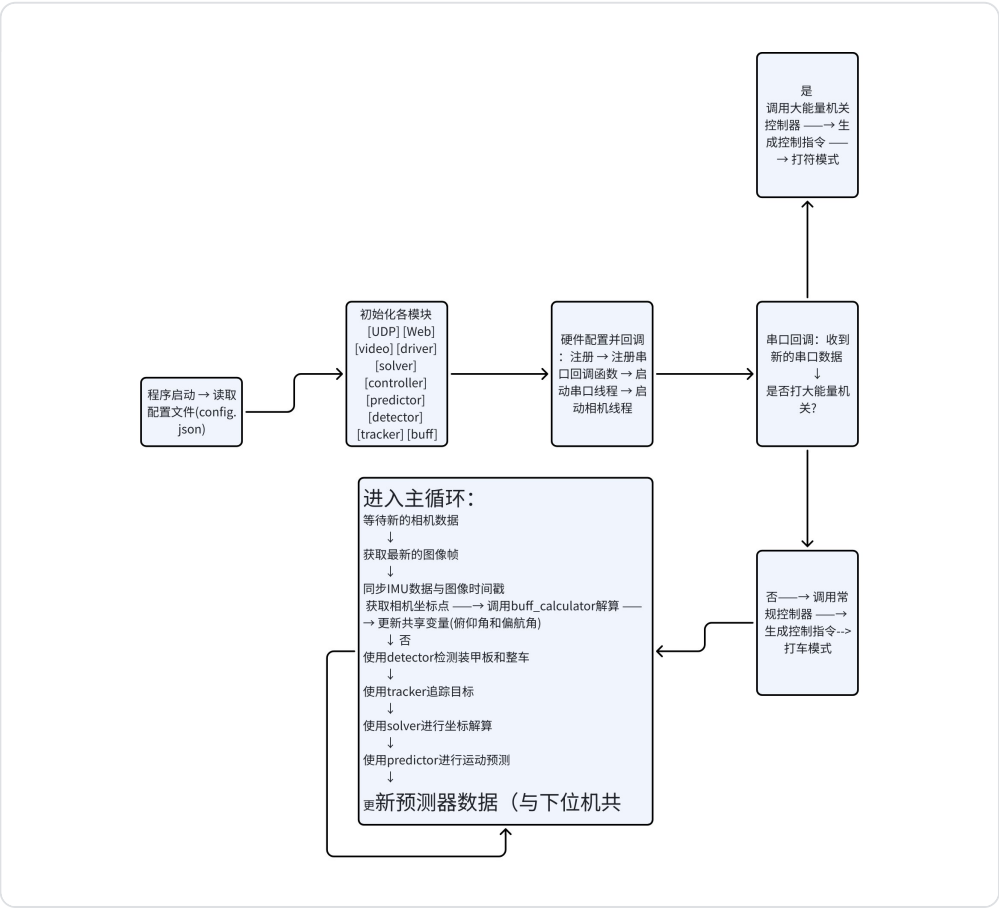
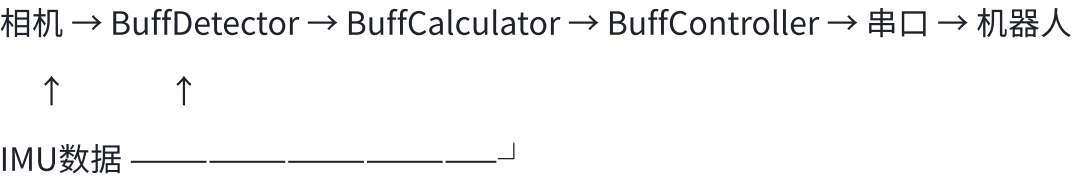
AutoAim:

数据流：可在打车与打符之间自动切换

打车：



打符： 分红蓝模式)



Modules:

```
modules.hpp

1  namespace modules
2  {
3      using namespace driver;
4      using namespace controller;
5      using namespace detector;
```

```

6     using namespace predictor;
7     using namespace tracker;
8     using namespace solver;
9     using namespace replayer;
10    using namespace power_rune;
11    using namespace buff;
12 }

```

1.controller:

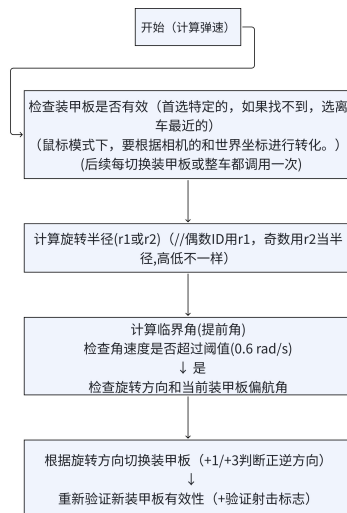
controllertype.hpp

```

1  struct ControlResult{
2      uint8_t shoot_flag; //0,1,2
3      float pitch_setpoint; //
4      float yaw_setpoint; //定义的拟合发射点的中心
5      float pitch_actual_want;
6      float yaw_actual_want; //相机返回值的坐标中心
7      bool valid=true;
8  }

```

1.1 控制流:



1.2 创新点:

1.2.1 提前角补偿:

- 当目标旋转时，子弹飞行期间目标会继续旋转，需要提前瞄准目标将要到达的位置。
- 提前角大小与目标距离、旋转速度、响应速度相关

// - 目标距离越远，需要的提前角越大

// - 旋转速度越快，需要的提前角越小

// - 半径越大，需要的提前角越小

// - 响应速度越快，需要的提前角越大

1.2.2 装甲板预判：

- 根据旋转方向和速度，预判目标旋转到下一个装甲板的时间

动态调整，以及切换的逻辑：

顺时针旋转 ($\omega > 0$) 且 当前装甲板偏航角超过临界角 时：

- 切换到下一个装甲板： $(\text{aim_armor_id.second} + 3) \% 4$

- 例如： $0 \rightarrow 3, 1 \rightarrow 0, 2 \rightarrow 1, 3 \rightarrow 2$ ；

逆时针旋转 ($\omega < 0$) 且 当前装甲板偏航角低于负临界角 时：

- 切换到上一个装甲板： $(\text{aim_armor_id.second} + 1) \% 4$

- 例如： $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 0^*$ /

1.2.3 射击逻辑判断：

- 采用双侧阈值检测，以及车体稳定性的判断。

2.ControllerOptimizer:

`ControlOptimizer` 类是一个测试工具，用于评估和优化云台控制系统的响应特性。通过漂移阈值 (0.01) 的设置，提高系统稳定性。

1.3 主要测试

1. **死区测试**：查找产生可检测运动的最小控制信号
2. **点响应测试**：测量在不同幅度下到达目标位置的时间
3. **正弦波测试**：评估不同频率下的跟踪精度
4. **锯齿波测试**：测试对快速方向变化的响应

1.4 类组件：

- **回调接口**：用于发送控制命令和读取当前位置的函数
- **测试模块**：针对每种响应特性的独立测试函数
- **配置参数**：易于调整的常量，用于调优测试
- **数据管理**：用于跟踪和分析测试结果的简单结构

3. Driver:

驱动模块。

```
drivertype.hpp

1  struct SerialConfig
2  struct CameraConfig//相机和串口
3  struct RawSerialData
4  struct RawSerialWriteData//原串口数据
5  struct TimeImageData{cv::Mat image,Time::TimeStamp timestamp};
6  struct ParsedSerialData//通过串口接受的数据，数据包，通过UPD发送和接受，一次8位
7      {
8          int8_t start_flag;
9          float pitch_now;
10         float yaw_now;
11         float roll_now;
12         float actual_bullet_speed;
13         uint8_t available_shoot_number;
14         uint8_t aim_request;
15         uint8_t mode_want;
16         uint8_t number_want;
17         uint8_t enemy_color; // 0:red 1:blue
18
19         Time::TimeStamp timestamp;
20
21         ParsedSerialData(const RawSerialData& x) : start_flag(x.start_flag),
pitch_now(x.pitch_now), yaw_now(x.yaw_now), roll_now(x.roll_now),
22             actual_bullet_speed(x.actual_bullet_speed),
aim_request(x.aim_request), mode_want(x.mode_want),
available_shoot_number(x.available_shoot_number),
23             number_want(x.number_want), enemy_color(x.enemy_color),
timestamp(Time::TimeStamp()) {};
24         ParsedSerialData() {};
25     };
```

1. camera_driver:

大恒相机的驱动，海康相机驱动。

2. serial_driver:

串口的驱动（parsedserialdata）

CRC中起始位和奇偶校验提高上下位机通信准确性以及同TimeStamp保证数据和图像的同步。开始位 “!” ”

4. Detector:

```
detectortype.hpp

1  struct Detection
2      {
3          cv::Rect2f bounding_rect;
4          cv::Point2f center;
5          int tag_id;
6          float score;
7          std::vector<cv::Point2f> corners;
8          bool isGray = false;
9      };
10 //std::vector<Detection>Detections;//多目标同时
11 struct CarDetection
12     {
13         cv::Rect2f bounding_rect;
14         cv::Point2f center;
15         int tag_id;// will be used for recognize Car's Model
16         //but now it's not used
17         float score;
18     };
19 //std::vector<CarDetection>CarDetections//多个车
20 class BBox://大小装甲板的信息。
```

1. 创新点:

在detector模块中，我们采用了YOLO模型识别整车-->再识别装甲板（四点模型）-->最后根据装甲板进行SVM(new:Resnet)进行数字的分类。整车的BBox和装甲板的BBox实时更新并各自存储，可以方便后续tracker对于多车的跟踪逻辑判断和solver的解算，以判断开火优先级。

2. 检测流程:

ArmorOneStage推理步骤:

- 1 //时间戳记录 : 记录开始时间用于性能统计
- 2 /*初始化模型:
- 3 - 设置CPU为推理设备


```

4  - 读取模型文件*/
5
6  图像预处理：大小转换并分离通道。
7  scaledResize(const cv::Mat&img,Eigen::Matrix<float,3,3>&transform_img)
8
9
10 //模型推理:
11 static void generate_grids_and_stride(const int target_w, const int target_h,
std::vector<int> &strides, std::vector<GridAndStride> &grid_strides)
12 // (生成yolox的检测网络)
13 static void generateYoloxProposals(std::vector<GridAndStride> grid_strides,
const float *feat_ptr,Eigen::Matrix<float, 3, 3> &transform_matrix, float
prob_threshold,BBoxes &bboxes,int& color_flag, bool& allowGray)
14 // (生成候选框)
15 static void qsort_descent_inplace(BBoxes &facebboxes, int left, int right)
16 // (快排获取置信区间下降的)
17 BBoxes decodeOutputs(const float *prob, BBoxes &objects, Eigen::Matrix<float,
3, 3> &transform_matrix, int& color_flag, bool& allowGray)
18 // (解码器)
19
20 //后处理:
21 /*
22 对候选框角点进行平均以降低误差
23 检查角点是否在图像范围内
24 使用number_classifier进行数字识别
25 过滤掉数字识别失败的候选框
26 */
27
28

```

在进行车和装甲板的检测时，各用一个独立的线程。装甲板检测完成后根据是否使用ROI调整检测框和角点坐标的偏移量，从BBox装换成Detections类型；车辆从YoloDetctions转换为CarDetections，并根据置信度更新矩形框。

5. Predictor:

卡尔曼滤波器更新方程，

卡尔曼滤波器（目标跟踪用）

一、状态空间模型

状态向量（位置 + 速度）：

$$\mathbf{x}_k = \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}$$

二、核心公式

1. 状态转移

$$\mathbf{x}_k = \mathbf{F}_k \mathbf{x}_{k-1} + \mathbf{w}_k \quad (\mathbf{w}_k \sim \mathcal{N}(0, \mathbf{Q}_k))$$

$$\text{转移矩阵: } \mathbf{F}_k = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. 观测

$$\mathbf{z}_k = \mathbf{H}_k \mathbf{x}_k + \mathbf{v}_k \quad (\mathbf{v}_k \sim \mathcal{N}(0, \mathbf{R}_k))$$

3. 噪声协方差

$$\bullet \text{ 过程: } \mathbf{Q}_k = \sigma^2 \begin{bmatrix} dt^3/3 & 0 & dt^2/2 & 0 \\ 0 & dt^3/3 & 0 & dt^2/2 \\ dt^2/2 & 0 & dt & 0 \\ 0 & dt^2/2 & 0 & dt \end{bmatrix}$$

$$\bullet \text{ 观测: } \mathbf{R}_k = \begin{bmatrix} r & 0 \\ 0 & r \end{bmatrix} \quad (r = \max(1e-1, \text{sameLabelError} \times 0.5))$$

三、更新步骤

1. 预测

$$\bullet \text{ 状态: } \hat{\mathbf{x}}_{k|k-1} = \mathbf{F}_k \hat{\mathbf{x}}_{k-1|k-1}$$

$$\bullet \text{ 协方差: } \mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_k^T + \mathbf{Q}_k$$

2. 更新

$$\bullet \text{ 卡尔曼增益: } \mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^T (\mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^T + \mathbf{R}_k)^{-1}$$

$$\bullet \text{ 状态修正: } \hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k (\mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_{k|k-1})$$

$$\bullet \text{ 协方差修正: } \mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k|k-1}$$

1. 检测量：

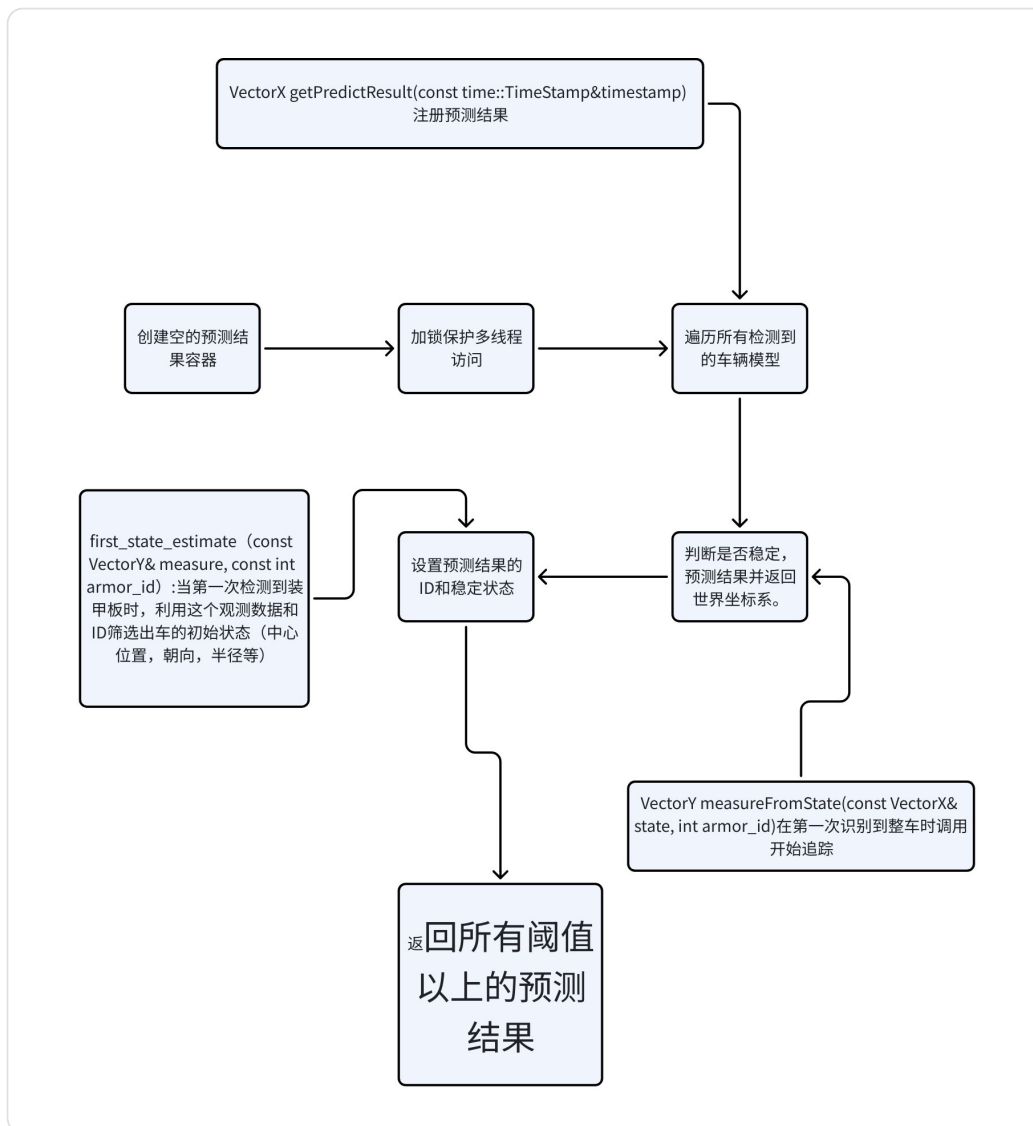
MotionModule.cpp

```
1 state: x,vx,y,vy,theta,omega,r1,r2,z1,z2, ax,ay //加速度
2 measure: ax,ay,az,tangent,angle_left,angle_right //装甲板
3 new measure: armor_pitch, armor_yaw, dist, tangent, armor_left, armor_right
4 //efk估计量:
5 Xe,Xp,H1,H2,P,Q,R1,R2,K1,K2,Yp1,Yp2 (卡尔曼滤波的状态)
6 //N_X->12, N_Y->7对应预测和检测的结果
7 using VectorX = Eigen::Matrix<double, N_X, 1>;
8 using VectorY = Eigen::Matrix<double, N_Y, 1>;
```

2. 创新点：

1. 基于传统的ekf卡尔曼滤波器采用BMEKF的策略，可以处理两种不同的观测模型（例如，可能对应于单个装甲板观测和两个装甲板同时观测的情况），通过template<class &&Func>{ceres::Xe_auto_jet[N_X]和Xp_auto_jet[N_X]}进行自动微分，对Y1,Y2同时进行更新。
2. 采用时间戳和时间微分两种方法进行重载。对于帧数我为10帧以上未检测到的则放弃预测。10FPS以上状态不更新认为静止等逻辑。

3. 流程：



4. 主要函数说明:

predictor.cpp, MotionModel.cpp

```

1  //predictor.cpp
2  //1.检测结果在相机与世界坐标的转化: 参考camera2world的欧拉角变换。
3  VectorY world2model(const VectorY& measure);
4  Prediction model2world(const VectorX& state, std::function<VectorY(const
VectorX&, int)> measureFunc);
5  //2.主函数: 获取所有车辆的运动模型与预测结果, 设置stable和id作为是否瞄准的判定。
6  predict(Time::TimeStamp timestamp)
7  update(const TrackResultPairs& trackResults, const Time::TimeStamp&
timestamp)//更新trackResults - 跟踪结果对
8  getPredictResult()
9
10 //MotionModel
11 initMotionModel()//初始化P,Q,R
12 getPredictResult(const Time::TimeStamp& timestamp)//获取预测状态向量
13 Update(const VectorY& measure_vec, const Time::TimeStamp& timestamp, int
armor_id)//根据预测值更新状态估计
14 //更新卡尔曼滤波器的状态, 根据残差判断车体稳定性。

```

```

15  measureFromState(const VectorX& state, int armor_id) //根据状态向量计算观测向量。
16  update(const VectorY& Y, const Time::TimeStamp& timestamp, int id) //根据观测值更新状态
17
18

```

6. Tracker:

1. 追踪量

```

tracker.type

1  Strcut TrackerResult:{
2      ArmorXYV armor; //装甲板的4个点的坐标和速度。
3      cv::Rect2f rect; //装甲板的矩形框。
4      location::Location location; //装甲板的位置信息。
5      double yaw; //装甲板的yaw角度。
6      bool visible = true; //装甲板是否可见。
7      int armor_id; //装甲板的id。
8      int car_id; //车的id。
9  }
10 Struct CarTrackResult{
11     int car_id;
12     int car_type;
13     cv::Rect2f bounding_rect;
14 }std::vector<CarTrackResult>CarTrackResults //pair,用于solver

```

2. 匹配方法:

1. (class Matcher)

Matcher.hpp:

```

1  track函数: map<int const *T>track(const vector<pair<Point, const
    *T>>&currentPoints, Time::TimeStamp time, Time::timeratio){
2  //步骤:
3  /*
4  a. 计算时间间隔dt并更新卡尔曼滤波器的状态转移矩阵和噪声协方差矩阵
5  b. 使用卡尔曼滤波预测所有轨迹的下一位置
6  c. 生成候选匹配对 (检测点与预测点距离小于阈值的组合)
7  d. 按距离排序并执行匹配 (确保一个检测点只匹配一个轨迹, 一个轨迹只匹配一个检测点)
8  e. 对匹配成功的轨迹进行卡尔曼滤波校正, 更新状态
9  f. 对未匹配到的轨迹, 根据丢失帧数决定是否删除
10 g. 对未匹配到的检测点, 创建新轨迹

```

```

11  h. 根据检测点的x坐标位置确定新轨迹的ID (左侧新点为(leftmost-1+4)%4, 右侧新点为
    (rightmost+1)%4。
12  */}
13

```

2. (class MatcherWithWholeCar)

TrackerMatcherWithWholeCar.hpp:

```

1  //整车与装甲板的匹配。
2  inline double calcTheta(Point armor_point,cv::Rect2f car_rect,int id)
3  map<int,T*> track(const vector<tuple<Point,cv::Rect2f, T*>>& currentPoints,
    Time::TimeStamp time){
4  /*a. 处理空帧情况和丢失帧计数
5  b. 计算时间间隔dt
6  c. 预测当前角度值 (oldtheta + omega * dt)
7  d. 对检测点按x坐标排序,先左后右
8  e. 计算每种可能的匹配方案的总角度差
9  f.
10 //get<0>(sortedPoints[j])- : 获取检测点的坐标
11 //get<1>(sortedPoints[j])- : 获取检测点的矩形框
12 //get<2>(sortedPoints[j])- : 获取检测点关联的数据指针,将检测点按最优方案分配ID并存储到
    结果中
13 //选取角度差总和最小的方案为最优匹配。
14 g. 更新角速度omega和角度oldtheta
15 h. 返回匹配结果*/
16 }

```

3. (tracker.cpp):

tracker.cpp

```

1  //采用两个merge函数处理重叠,根据置信度替换原装甲板。
2  merge(const Detections &detections,double threshold)
3  merge(const CarDetections &detections,double threshold)
4  //检擦输入点,4个再计算外接矩形。
5  //若一定时间内,rect2再rect1内占0.9以上,则更新。
6  //融合IMU数据检测装甲板与整车
7  TrackResults Tracker::getArmorTrackResult(const Time::TimeStamp& time, const
    ImuData& imu)
8  CarTrackResults Tracker::getCarTrackResult(const Time::TimeStamp& time, const
    ImuData& imu, const TrackResults& armor)
9  1. 计算old_car_rects与car_rects间任意两矩形的距离 (采用顶点距离差之和作为损失)
10 // 2. 迭代匹配过程:
11 //     a. 找出当前损失最小的矩形对
12 //     b. 将该对从匹配集合中移除

```

```

13 // c. 将car_rects中的矩形标记为对应old_car_rect的编号
14 // 3. 当最小损失超过阈值时，停止匹配过程
15 // 4. 将car_rects中未匹配的矩形编号标记为-1
16 //
17 // 【第二阶段：装甲板与车辆矩形匹配】
18 // 5. 对每个装甲板armor：
19 // a. 检查是否存在对应car_id的矩形：
20 // - 若存在，判断该矩形是否完整包含armor的rect
21 // * 若完整包含：无需操作
22 // * 若不完整包含：将该矩形标记为未使用(car_id=-1)，执行步骤b
23 // - 若不存在，直接执行步骤b
24 // b. 查找可匹配的矩形：
25 // - 优先在未使用的矩形(car_id=-1)中查找能完整包含armor的矩形
26 // - 若找到，将该矩形的car_id设为armor的car_id
27 // - 若未找到，在已使用的矩形中查找能完整包含armor的矩形
28 // - 若找到，将该矩形的car_id设为armor的car_id
29 // - 若仍未找到，放弃该装甲板的匹配
30 // 最终返回car_id!=-1的car_rects

```

3. 创新点：

1. 动态时间间隔更新：根据实际时间间隔调整卡尔曼滤波器参数
2. 自适应距离门限：同时根据sameLabelError和diffLabelError动态计算匹配阈值，增强平滑性。
3. 平滑滤波更新：使用alpha参数对误差进行平滑更新。根据角速度的预测中加入 α ，避免因旋转过快导致目标丢失出现bug。
4. 匈牙利算法。在TrackerMatcher类中，构建成本矩阵，贪心策略优先选择距离最近的匹配对，并设置为初始的装甲板，可保证只被匹配一次。

7. Solver:

1. 解算量：

```

solver.type

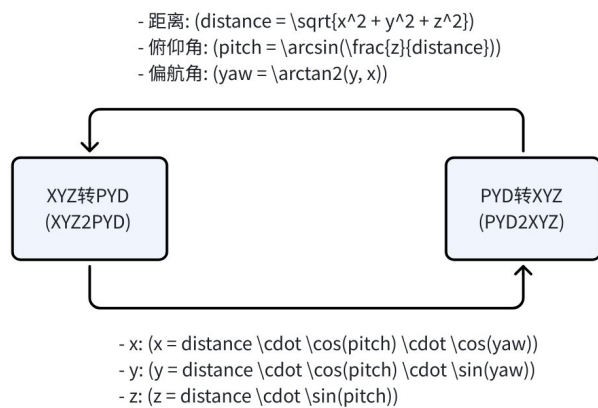
1 //1. 需要融合的IMU数据
2 Struct ImuData:{
3   Pitch,Yaw,Roll
4   operator PYD()
5 }
6 //2. 大小装甲板各自的参数，此处承接Controller与predictor中的

```

```

7 //装甲板检测与切换逻辑，奇偶装甲板各自处理。
8
9 //3.从Location的BaseSolver类中继承坐标转换函数。
10 inline PYD XYZ2PYD (const XYZ& in) const override{}等。和
11 Inline XYZ camera2world(const XYZ &in,const PYD &imuData)
12 Inline XYZ world2camera(const XYZ& in,const PYD &imuData)
13 std::pair<XYZ,double>camera2world(),camera2worldWithWholeCar()

```



其余部分同理

2. 解算逻辑：

1. 此处我们替换了去年根据distance计算坐标的旧版本，改用提取四个点的中心坐标。因为更方便计算与车体中心的距离，判断装甲板是否位于车体内，为有效的打击对象。
2. camera2world和world2camera的实现：两者互为逆矩阵的计算

solver.hpp:

```

1 inline XYZ camera2world(const XYZ& in, const PYD& imuData) const override
2 //a.把输入的XYZ坐标转换成Eigen向量
3 //b.相机旋转矩阵和偏移量将相机坐标系转换到云台坐标系。
4 //c.再根据IMU数据，将云台坐标系转换到世界坐标系。
5 //d.最后，将世界坐标系的坐标转换回XYZ格式。
6 inline XYZ world2camera(const XYZ& in, const PYD& imuData) const override
7 //将输入的XYZ坐标转换为Eigen向量
8 //使用IMU数据 (pitch和-yaw) 进行反向坐标变换，将点从世界坐标系转换到云台坐标系
9 //应用相机旋转矩阵的逆矩阵和偏移量，将点从云台坐标系转换到相机坐标系
10 //返回相机坐标系下的XYZ坐标

```

3. 采用Opencv的PnP方法重载camera2world和camera2worldWithWholeCar的方法。通过逐帧比较角度差选择最优解,解算装甲板姿态。

```

1 std::pair<XYZ,double> Solver::camera2world(const ArmorXYV& trackResult, const
  Solver.cpp:
    ImuData& imuData_deg, bool isLarge) {
2     //1.输入 : 装甲板的四个角点图像坐标( trackResult )、IMU数据( imuData_deg )、装甲板类
      型( isLarge )。
3     //2.预处理;- 将Eigen格式的相机内参和畸变系数转换为OpenCV格式。
4     //- 根据装甲板类型选择对应的3D角点坐标( objectPoints )。
5     //- 将输入的装甲板角点坐标( trackResult )转换为OpenCV的 Point2f 格式( imagePoints )
6     //3. SolverPnP:调用 cv::solvePnPGeneric 函数,使用 SOLVEPNP_IPPE 方法计算所有可能的
      解。该函数会返回旋转旋量( rvecs )和平移向量( tvecs )。
7         int solutions = cv::solvePnPGeneric(objectPoints, imagePoints,
      cameraMatrix, distCoeffs,
8
      rvecs, tvecs, false, cv::SOLVEPNP_IPPE);
9     //4.选择合适解: - 比较当前解的yaw角度与上一帧的yaw角度( prev_armor_yaw ),选择差异最小
      的解。
10    //- 更新 prev_armor_yaw 为当前选择的yaw角度。
11    //5. 调用camera2world 函数,将相机坐标系下的坐标转换为世界坐标系下的坐标。

12    //6. 融合: 返回世界坐标系下的XYZ坐标和装甲板总yaw角度 (装甲板角度+IMU yaw角度)
13
14 }
15
16
17 std::pair<XYZ,double> Solver::camera2worldWithWholeCar(const ArmorXYV&
  trackResult, const ImuData& imuData_deg, const cv::Rect& bounding_rect, bool
  isLarge)
18 {
19     //处理逻辑与上述函数相同,装甲板不同于车体,需要区分大小,
20     //并且额外加入了装甲板中心与车体中心的距离判断,若在车外当舍弃。
21 }

```

8. RecorderSolver:记录解算数据。

9. replayer:回放记录

10. buff:(红蓝模式)being changing