测试队内Kalman相关

1.KalmanFilter.h和KalmanFilter.cpp

首先看到队里自写了一份卡尔曼滤波器

详见代码如下，附注释

KalmanFilter.h

```
#ifndef ARMOR_PROCESSOR__KALMANFILTER_H
#define ARMOR_PROCESSOR__KALMANFILTER_H

#include "../include/Eigen/Dense"

namespace angle // 把角度相关放在单独的命名空间
{
    // KalmanFilterMatrices结构体:存储滤波器所需的矩阵F,H,Q,R,P
    struct KalmanFilterMatrices
    {
        Eigen::MatrixXd F; // state transition matrix 状态转移矩阵
        Eigen::MatrixXd H; // measurement matrix 观测矩阵
        Eigen::MatrixXd Q; // process noise covariance matrix 状态/过程噪声协方差
        Eigen::MatrixXd R; // measurement noise covariance matrix 观测噪声协方差
        Eigen::MatrixXd P; // error estimate covariance matrix 先验/后验误差协方差矩
阵
        // 没有定义控制输入矩阵
    };

    class KalmanFilter
    {
    public:
        explicit KalmanFilter(const KalmanFilterMatrices &matrices); // 构造函数

        // Initialize the filter with a guess for initial states.
        // 初始化滤波器状态
        void init(const Eigen::VectorXd &x0);

        // Computes a predicted state
        // 进行预测
        Eigen::MatrixXd predict(const Eigen::MatrixXd &F);
```

```
        // Update the estimated state based on measurement
        // 利用新测量进行更新
        Eigen::MatrixXd update(const Eigen::VectorXd &z);

        // Predicted state
        Eigen::VectorXd x_pre; // 先验状态向量
        // Updated state
        Eigen::VectorXd x_post; // 后验状态向量

    private:
        // Invariant matrices
        Eigen::MatrixXd F, H, Q, R;

        // Priori error estimate covariance matrix
        Eigen::MatrixXd P_pre; // 先验估计误差协方差矩阵,预测步骤中计算
        // Posteriori error estimate covariance matrix
        Eigen::MatrixXd P_post; // 后验估计误差协方差矩阵,更新步骤中计算

        // Kalman gain 卡尔曼增益 反映新测量对状态修正的权重
        Eigen::MatrixXd K;

        // System dimensions
        int n; // 系统状态维度

        // N-size identity
        Eigen::MatrixXd I; // n阶单位矩阵
    };
}
#endif
```

KalmanFilter.cpp

```
#include "../include/kalmanfilter.h"

namespace angle
{

  KalmanFilter::KalmanFilter(const KalmanFilterMatrices &matrices) :
F(matrices.F),

H(matrices.H),

Q(matrices.Q),

R(matrices.R),

P_post(matrices.P),

n(matrices.F.rows()),

I(Eigen::MatrixXd::Identity(n, n)),
                                                                    x_pre(n),
                                                                    x_post(n)
  {
```

```
}
// 构造函数里面初始化了F,H,Q,R,P_post,n,I,x_pre,x_post  其中n是状态向量的维度 I是类里的一个单位矩阵 先验后验状态向量x_pre,x_post是n维的
// 把整数n用于初始化x_pre,x_post，x_pre,x_post是矩阵，就会被赋值为n行1列的矩阵
// 单位阵I初始化为n阶单位阵 用于更新操作里面更新后验估计协方差矩阵P_post

// 用给定的初始状态向量x0初始化状态向量
void KalmanFilter::init(const Eigen::VectorXd &x0) { x_post = x0; } // 是把给定的初始状态向量赋值为后验状态向量

// 预测  传状态控制矩阵进来
Eigen::MatrixXd KalmanFilter::predict(const Eigen::MatrixXd &F)
{
  this->F = F;

  //- 计算先验状态向量x_pre
  x_pre = F * x_post; // x_post事先做初始化了   先验状态值=状态转移矩阵*上一秒的状态值+控制输入矩阵*外部输入的控制量 此处外部输入的控制量为0
  //- 计算先验误差协方差矩阵P_pre
  P_pre = F * P_post * F.transpose() + Q; // 先验估计协方差=状态转移矩阵*上一次的先验估计协方差*状态转移矩阵转置+状态噪声协方差矩阵

  // handle the case when there will be no measurement before the next predict
  x_post = x_pre;
  P_post = P_pre;
  //- 如果没有新的测量,先验结果作为后验结果输出
  return x_pre; // 返回先验状态向量
}
// 更新  传入新的测量值
Eigen::MatrixXd KalmanFilter::update(const Eigen::VectorXd &z)
{
  //- 计算卡尔曼增益K
  K = P_pre * H.transpose() * (H * P_pre * H.transpose() + R).inverse(); // 卡尔曼增益=先验估计协方差*观测矩阵转置*（观测矩阵*先验估计协方差*观测矩阵转置+观测噪声协方差矩阵）的逆
  //- 计算后验状态向量x_post
  x_post = x_pre + K * (z - H * x_pre); // 后验状态向量=先验状态向量+卡尔曼增益*（测量值-观测矩阵*先验状态向量）
  //- 计算后验误差协方差矩阵P_post
  P_post = (I - K * H) * P_pre; // 后验估计协方差=（单位矩阵-卡尔曼增益*观测矩阵）*先验估计协方差

  return x_post; // 返回后验状态向量
}
}
```

可以看到，这里面定义了卡尔曼滤波器的相关参数，包括

- **状态转移矩阵F**
- **观测矩阵H**
- **状态/过程噪声协方差Q**
- **观测噪声协方差R**
- **先验/后验误差协方差矩阵P**

上面的这些参数量保存在结构体种做为一个整体传给KalmanFilter。

值得注意的是，**并没有定义控制输入矩阵**。也就是后面的预测和更新过程都是建立在没有控制输入量的前提下的。也就是只对系统本身的状态做预测，不考虑外部控制输入量。

预测和更新函数由predict和update函数完成，公式严格参照kalman滤波的5个公式。更新后返回后验状态向量，也即是预测值。

下面是针对此KalmanFilter做的测试

```
// 测试kalmanfilter
void test01()
{
    // 1.初始化状态转移矩阵F
    Eigen::MatrixXd F(2, 2);
    F << 1, 1, 0, 1;
    // 2.初始化观测矩阵H
    Eigen::MatrixXd H(1, 2);
    H << 1, 0;
    // 3.初始化状态噪声协方差矩阵Q
    Eigen::MatrixXd Q(2, 2);
    Q << 0.0001, 0, 0, 0.0001;
    // 4.初始化观测噪声协方差矩阵R
    Eigen::MatrixXd R(1, 1);
    R << 0.1;
    // 5.初始化先验估计误差协方差矩阵P
    Eigen::MatrixXd P(2, 2);
    P << 0.1, 0, 0, 0.1;
    // 6.初始化状态向量x
    Eigen::VectorXd x(2);
    x << 0, 0; // 状态向量x的第一个元素表示角度，第二个元素表示角速度
    // 7.初始化测量值
    Eigen::VectorXd measurements(1);
    measurements << 1; // 测量值为角度
    // 8.初始化卡尔曼滤波器
    angle::KalmanFilterMatrices matrices;
    matrices.F = F;
    matrices.H = H;
    matrices.Q = Q;
    matrices.R = R;
    matrices.P = P;
    angle::KalmanFilter kf(matrices);
    // 9.初始化状态向量
    kf.init(x); // 第一次传进来的都是0
    // 10.迭代
   for (int i = 0; i < 100; i++)
    {
        kf.predict(F);
// 预测 每次预测都传进来状态转移矩阵F
        kf.update(measurements);
// 更新 每次更新都传进来测量值
        measurements << i + 1;
// 更新测量值
        std::cout << "x=" << kf.x_post(0) << " v=" << kf.x_post(1) << std::endl;
// 输出后验值
    }
```

```
}
```

每次都是预测、更新。F、H、Q、R是后面的参数，开始时设置为定值。

在循环中手动更新测量值，这里是比较简单的，就是每次加一。

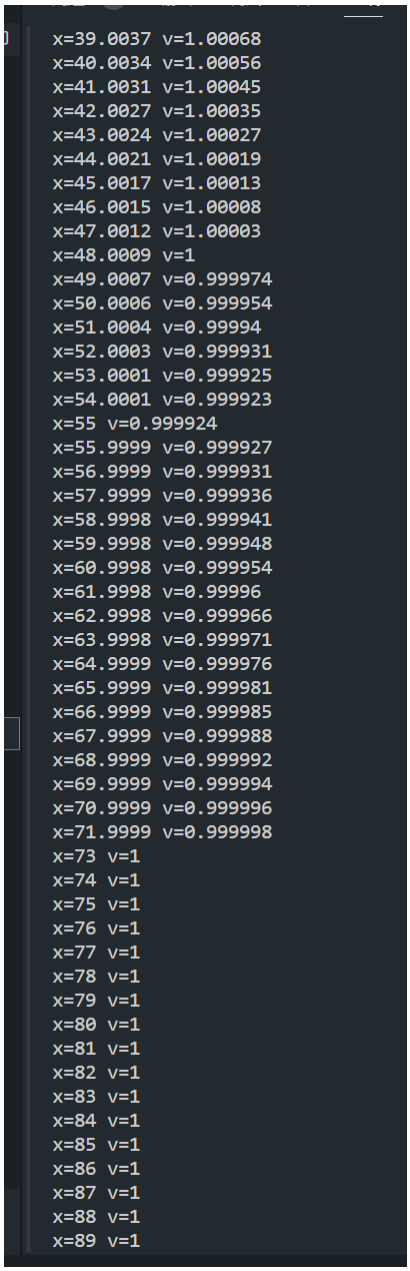
测量值状态向量是二维的 标志值和变化速度

状态转移矩阵是四维的

观测矩阵是二维的

过程噪声和观测噪声与上面的两个矩阵分别对应

测试结果如下：



在73次迭代时彻底收敛。

KalmanFilter是后来用到predict和predictor里面的。并没有用到下面讲的kalman里面，当然这些是后话，因为是我后来才发现的。一开始我还以为这个自己写的KalmanFiter是干啥的？？后来才发现这个是有用的，而下面讲的kalman实际上是没用的，但是也看一下他的工作原理。

2.kalman.h和kalman.cpp

这两个文件定义了新的kalman滤波过程，用到的是cv自带的KalmanFilter。

kalman.h

```
/*
 * @Description: 测试cv自带的卡尔曼滤波器
 * @Author: Hao Zhiyang
 * @Date: 2023-12-15 22:28:05
 * @LastEditors: Hao Zhiyang
 * @LastEditTime: 2023-12-16 11:57:05
 */
#ifndef RM2022_KALMAN_H
#define RM2022_KALMAN_H

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/video/video.hpp>
#include <opencv2/imgproc/types_c.h>
#include <opencv2/core/core.hpp>
#include <opencv2/video/tracking.hpp>

#include "../include/SolvePnP.h" //包含解算pnp的头文件

#include "vgd_standard_content.hpp"
#include <queue>
#include <vector>
#include <iostream>

/**
 * @brief: 卡尔曼类
 */
class kalman
{
private:
    const int m_stateNum = 4;   // 状态量的维数 状态矩阵
    const int m_measureNum = 2; // 测量量的维数 测量矩阵

    cv::Mat m_measurement; // 测量值 测量矩阵
    cv::Mat m_prediction;  // 预测值 矩阵
public:
    cv::KalmanFilter m_KF; // 卡尔曼滤波器对象  cv下自带的
    kalman();
    void reInit(KalmanFilter &KF);                                    // 重新初始化滤
波器状态  传卡尔曼滤波器类
    void predict(vgd_stl::armors &finalarmor, cv::Mat originframe); // 进行卡尔曼预
测
    cv::Point2f kal(float x, float y);                                // 对单个测量进
行滤波
    cv::Point2f kal2(float x, float y);                               // 对单个测量进
行滤波 但是迭代次数设置为1 常规滤波
};
#endif
```

kalman.cpp

```
/*
 * @Description: 卡尔曼预测
 * @Version: 1.0
 * @Author: Barimu
 * @Date: 2022-10-30 00:10:23
 * @LastEditors: Barimu
 * @LastEditTime: 2023-01-15 16:06:46
 */
#ifndef RM2022_KALMAN_CPP
#define RM2022_KALMAN_CPP

#include "../include/kalman.h"

using namespace std;
using namespace cv;

/**
 * @brief: 卡尔曼相关数据初始化
 * @param {KalmanFilter} m_KF
 */
// 卡尔曼初始化
kalman ::kalman()
{
    // 定义KalmanFilter类并初始化
    KalmanFilter KK(m_stateNum, m_measureNum, 0); // 定义KalmanFilter
    // CV自带的KalmanFilter构造 传入状态量维数，测量量维数，控制量维数

    m_KF = KK; // 将定义的KalmanFilter类赋值给m_KF

    // 定义初始测量值    测量矩阵x  2行1列的矩阵
    m_measurement = Mat::zeros(m_measureNum,
                               1,
                               CV_32F); // 用0初始化测量值矩阵m_measurement CV_32F是
32位浮点型

    // 定义状态转移矩阵F  transitionMatrix是cv::KalmanFilter类自带的状态转移矩阵 // 4行
4列的矩阵 状态转移矩阵是提前设置好的
    m_KF.transitionMatrix = (Mat_<float>(m_stateNum,
                                         m_stateNum)
                                 << 1,
                             0, 1, 0,
                             0, 1, 0, 1,
                             0, 0, 1, 0,
                             0, 0, 0, 1);

    // 将下面几个矩阵设置为对角阵  用cv::KalmanFilter类自带的函数setIdentity()
    // 两个参数，第一个是矩阵对象，第二个是对角线元素的值
    setIdentity(m_KF.measurementMatrix);                    // 观测矩阵 H 默认为1
单位阵
    setIdentity(m_KF.processNoiseCov, Scalar::all(1e-3));     // 过程噪声 Q 主对角
线设置为1e-3
    setIdentity(m_KF.measurementNoiseCov, Scalar::all(1e-2)); // 测量噪声 R 主对角
线设置为1e-2
```

```
    setIdentity(m_KF.errorCovPost, Scalar::all(1));             // 最小均方误差 Pt 主对角线设置为1 cv::KalmanFilter类的最小均方误差相当于先验和后验误差协方差矩阵
    //  randn( m_KF.statePost, Scalar::all(0), Scalar::all(0.1) );
    randn(m_KF.statePost, Scalar::all(0), Scalar::all(0.1)); // x(0)初始化 相当于是系统初始状态
    // m_KF.statePost是后验状态向量 储存当前状态值可以表示为x m_KF.statePre是先验状态向量储存预测状态值
    // randn()函数用于生成服从正态分布的随机数 这段是m_KF.statePost初始化为均值为0，标准差为0.1的正态分布随机数。这样做是为了给Kalman滤波器的状态向量一个初始值。在实际应用中，这个初始值通常是不确定的，所以使用随机数来表示这种不确定性
}

// 重新初始化滤波器状态 重新传KalmanFilter进来
void kalman ::reInit(KalmanFilter &KF)
{
    // 将下面几个矩阵设置为对角阵
    setIdentity(m_KF.measurementMatrix);                        // 测量矩阵 H
    setIdentity(m_KF.processNoiseCov, Scalar::all(1e-3));       // 过程噪声 Q
    setIdentity(m_KF.measurementNoiseCov, Scalar::all(1e-2)); // 测量噪声 R
    setIdentity(m_KF.errorCovPost, Scalar::all(1));             // 最小均方误差 Pt
    // randn( m_KF.statePost, Scalar::all(0), Scalar::all(0.1) );
    randn(m_KF.statePost, Scalar::all(0), Scalar::all(0.1)); // x(0)初始化
}

/**
 * @description: 卡尔曼预测装甲板中心点
 * @param {float} x
 * @param {float} y
 * @return {*}
 * @author: Barimu
 */
// 预测装甲板中心点 传点进来 迭代5次 预测间隔为5个时间单位
cv::Point2f kalman::kal(float x, float y)
{
    cv::Point2f center;              // 创建中心点
    m_measurement.at<float>(0) = x; // 将测量值矩阵m_measurement的第一个元素赋值为x
    m_measurement.at<float>(1) = y;
    m_prediction = m_KF.predict();  // 进行一次预测 调用了卡尔曼滤波器的预测函数 自带的predict()函数返回的是下一时刻的后验状态值KF.statePost(k+1)
    for (int i = 1; i <= 5; i++) // 迭代5次
    {
        m_KF.correct(m_measurement);    // 更新 用测量值矩阵m_measurement对当前状态进行更新correct()
        m_prediction = m_KF.predict(); // 再次进行预测predict(),得到新预测值
        m_measurement.at<float>(0) = m_prediction.at<float>(0);
        m_measurement.at<float>(1) = m_prediction.at<float>(1); // 将预测结果写入测量矩阵m_measurement,进行下一次迭代
    }
    // 循环迭代5次后,从预测结果矩阵m_prediction中取出滤波后的值
    center.x = m_prediction.at<float>(0);
    center.y = m_prediction.at<float>(1);
    // 返回滤波预测的中心坐标点
    return center;
}
```

```
// 自写 预测迭代改为1 常规滤波
cv::Point2f kalman::kal2(float x, float y)
{
    cv::Point2f center;              // 创建中心点
    m_measurement.at<float>(0) = x; // 将测量值矩阵m_measurement的第一个元素赋值为x
    m_measurement.at<float>(1) = y;
    m_prediction = m_KF.predict(); // 进行一次预测 调用了卡尔曼滤波器的预测函数 自带的
predict()函数返回的是下一时刻的后验状态值KF.statePost(k+1)
    m_KF.correct(m_measurement);   // 更新 用测量值矩阵m_measurement对当前状态进行更新
correct()
    center.x = m_prediction.at<float>(0);
    center.y = m_prediction.at<float>(1);
    // 返回滤波预测的中心坐标点
    return center;
}

/**
 * @brief 调用卡尔曼预测
 * @param {armors} &finalarmor
 * @param {Mat} originFrame
 * @param {kalman} &m_k
 */
void kalman::predict(vgd_stl::armors &finalarmor, Mat originFrame) // 传入装甲板对
象和原始图像
{
    cv::Point2f centers = finalarmor.center;                                  // 从检
测到的装甲板finalarmor中获取中心坐标centers
    cv::Point predict_pt = kal(centers.x, centers.y);                         // 将
centers作为测量输入,调用kal()函数进行卡尔曼滤波预测,得到预测坐标predict_pt
    cv::circle(originFrame, predict_pt, 3, cv::Scalar(34, 255, 255), -1); // 在原
始图像originFrame上画出预测坐标predict_pt,绘制为一个小圆点
    // 对每一帧的装甲板目标均进行该预测过程
}

#endif
```

在这个kalman类中定义了cv的KalmanFilter。然后对KalmanFilter自带的各种参数进行初始化，包括F、H、Q、R、P等。这些都在构造函数内完成，另外也初始化了测量的状态向量（0）。

在kal函数中调用的是cv对象自带的预测和更新函数，并且可以看到他是对每个测量值在用当次测量值预测更新迭代了5次之后才返回后验状态向量。

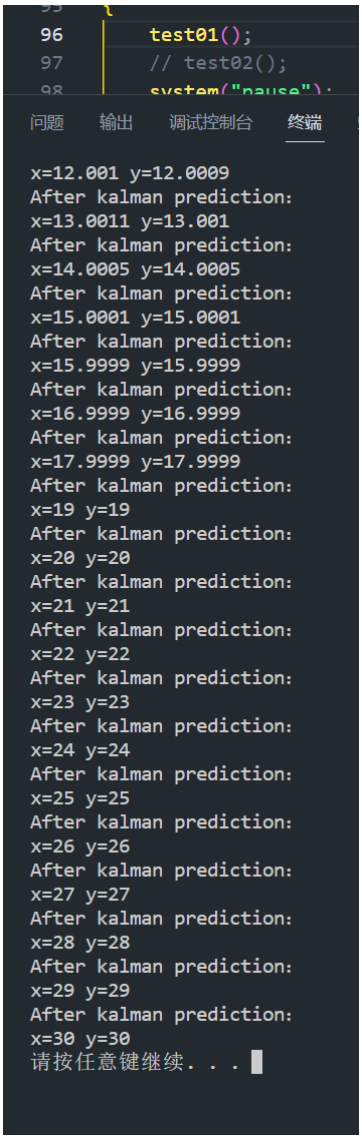
另外在kal函数中做了一些修改，主要是把第一次预测推后了，也就是测量值传进来赋值后再进行预测。虽然类内已经初始化了自己的测量值为0，但实际上还是要等一次测量值更新的。

对kal函数测试如下：

```
// 预测装甲板中心点 传点进来 迭代5次 预测间隔为5个时间单位
cv::Point2f kalman::kal(float x, float y)
{
    cv::Point2f center;              // 创建中心点
    m_measurement.at<float>(0) = x; // 将测量值矩阵m_measurement的第一个元素赋值为x
    m_measurement.at<float>(1) = y;
    // m_prediction = m_KF.predict(); // 进行一次预测 调用了卡尔曼滤波器的预测函数 自带
的predict()函数返回的是下一时刻的后验状态值KF.statePost(k+1)
    for (int i = 1; i <= 5; i++) // 迭代5次
```

```
    {
        m_prediction = m_KF.predict(); // 再次进行预测predict(),得到新预测值
        m_KF.correct(m_measurement);   // 更新 用测量值矩阵m_measurement对当前状态进行更新correct()
        m_measurement.at<float>(0) = m_prediction.at<float>(0);
        m_measurement.at<float>(1) = m_prediction.at<float>(1); // 将预测结果写入测量矩阵m_measurement,进行下一次迭代
    }
    // 循环迭代5次后,从预测结果矩阵m_prediction中取出滤波后的值
    center.x = m_prediction.at<float>(0);
    center.y = m_prediction.at<float>(1);
    // 返回滤波预测的中心坐标点
    return center;
}
```

测试效果：



可以看到仍然是只做了滤波，只收到了滤波值，预测值等于真实值。

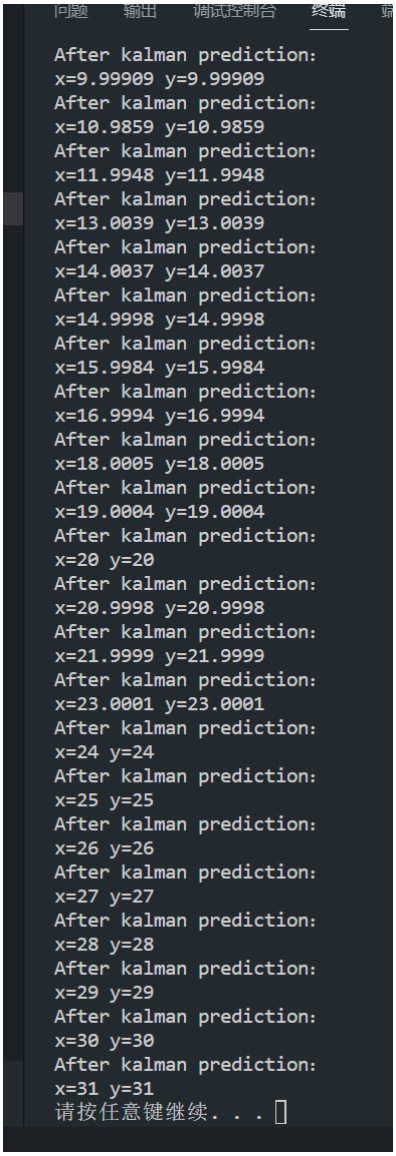
修改：

```
// 预测装甲板中心点 传点进来 迭代5次 预测间隔为5个时间单位
cv::Point2f kalman::kal(float x, float y)
{
    cv::Point2f center;             // 创建中心点
```

```
    m_measurement.at<float>(0) = x; // 将测量值矩阵m_measurement的第一个元素赋值为x
    m_measurement.at<float>(1) = y;
    // m_prediction = m_KF.predict(); // 进行一次预测 调用了卡尔曼滤波器的预测函数 自带的predict()函数返回的是下一时刻的后验状态值KF.statePost(k+1)
    for (int i = 1; i <= 5; i++) // 迭代5次
    {
        m_KF.correct(m_measurement);    // 更新 用测量值矩阵m_measurement对当前状态进行更新correct()
        m_prediction = m_KF.predict(); // 再次进行预测predict(),得到新预测值
        m_measurement.at<float>(0) = m_prediction.at<float>(0);
        m_measurement.at<float>(1) = m_prediction.at<float>(1); // 将预测结果写入测量矩阵m_measurement,进行下一次迭代
    }
    // 循环迭代5次后,从预测结果矩阵m_prediction中取出滤波后的值
    center.x = m_prediction.at<float>(0);
    center.y = m_prediction.at<float>(1);
    // 返回滤波预测的中心坐标点
    return center;
}
```

先更新一次，再预测迭代

测试：



在24次时彻底收敛。

对kal2函数，对每个测量值进行一次迭代预测。测试函数如下：

```
// 测试kal2 单轮迭代1次
void test02()
{
    kalman kl; // 初始化卡尔曼
    // 创建一组点做测试
    std::vector<cv::Point2f> points;
    points.push_back(cv::Point2f(0, 0));
    points.push_back(cv::Point2f(1, 1));
    points.push_back(cv::Point2f(2, 2));
    points.push_back(cv::Point2f(3, 3));
    points.push_back(cv::Point2f(4, 4));
    points.push_back(cv::Point2f(5, 5));
    points.push_back(cv::Point2f(6, 6));
    points.push_back(cv::Point2f(7, 7));
    points.push_back(cv::Point2f(8, 8));
    points.push_back(cv::Point2f(9, 9));
    points.push_back(cv::Point2f(10, 10));
    points.push_back(cv::Point2f(11, 11));
    points.push_back(cv::Point2f(12, 12));
    points.push_back(cv::Point2f(13, 13));
    points.push_back(cv::Point2f(14, 14));
    points.push_back(cv::Point2f(15, 15));
    points.push_back(cv::Point2f(16, 16));
    points.push_back(cv::Point2f(17, 17));
    points.push_back(cv::Point2f(18, 18));
    points.push_back(cv::Point2f(19, 19));
    points.push_back(cv::Point2f(20, 20));
    points.push_back(cv::Point2f(21, 21));
    points.push_back(cv::Point2f(22, 22));
    points.push_back(cv::Point2f(23, 23));
    points.push_back(cv::Point2f(24, 24));
    points.push_back(cv::Point2f(25, 25));
    points.push_back(cv::Point2f(26, 26));
    points.push_back(cv::Point2f(27, 27));
    points.push_back(cv::Point2f(28, 28));
    points.push_back(cv::Point2f(29, 29));
    points.push_back(cv::Point2f(30, 30));
    // 测试kal函数
    for (int i = 0; i < points.size(); i++)
    {
        cv::Point2f center = kl.kal2(points[i].x, points[i].y);
        std::cout << "After kalman prediction: " << std::endl;
        std::cout << "x=" << center.x << " y=" << center.y << std::endl;
    }
}
```

效果

```
90      center.x = m_predic
91      center.y = m_predic
92      // 返回滤波预测的中心
93      return center;
```

问题 输出 调试控制台 终端

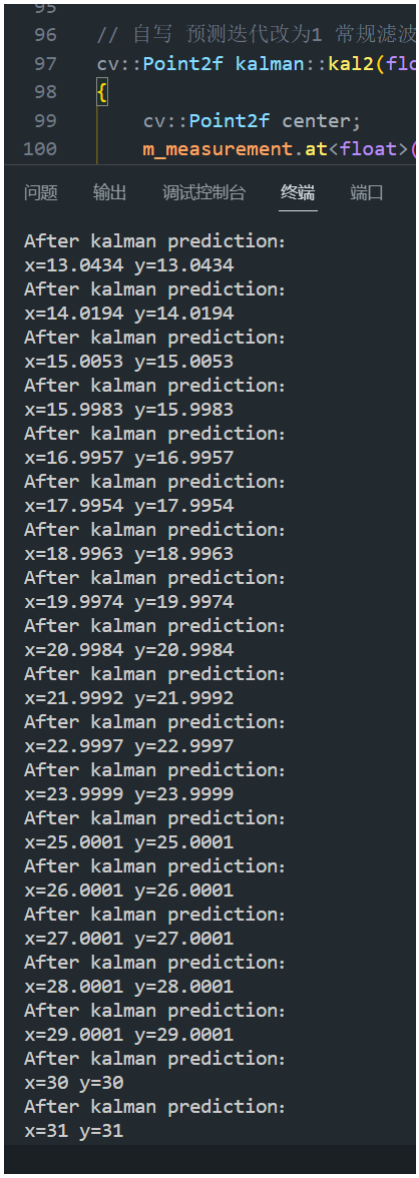
```
x=4.99189 y=4.99187
After kalman prediction:
x=5.99544 y=5.99542
After kalman prediction:
x=6.99768 y=6.99767
After kalman prediction:
x=7.99906 y=7.99905
After kalman prediction:
x=8.99983 y=8.99983
After kalman prediction:
x=10.0002 y=10.0002
After kalman prediction:
x=11.0003 y=11.0003
After kalman prediction:
x=12.0003 y=12.0003
After kalman prediction:
x=13.0002 y=13.0002
After kalman prediction:
x=14.0001 y=14.0001
After kalman prediction:
x=15.0001 y=15.0001
After kalman prediction:
x=16 y=16
After kalman prediction:
x=17 y=17
After kalman prediction:
x=18 y=18
After kalman prediction:
x=19 y=19
After kalman prediction:
x=20 y=20
After kalman prediction:
x=21 y=21
After kalman prediction:
x=22 y=22
After kalman prediction:
x=23 y=23
After kalman prediction:
x=24 y=24
After kalman prediction:
x=25 y=25
After kalman prediction:
```

在16次时彻底收敛。但是没有预测效果，原因不明，就是预测值收敛后等于真实值。

做修改如下，就是在每次开始前先更新一次，这样可以做到预测下一次的值

```
cv::Point2f kalman::kal2(float x, float y)
{
    cv::Point2f center;              // 创建中心点
    m_measurement.at<float>(0) = x; // 将测量值矩阵m_measurement的第一个元素赋值为x
    m_measurement.at<float>(1) = y;
    m_KF.correct(m_measurement);
    m_prediction = m_KF.predict(); // 进行一次预测 调用了卡尔曼滤波器的预测函数 自带的predict()函数返回的是下一时刻的后验状态值KF.statePost(k+1)
    m_KF.correct(m_measurement);    // 更新 用测量值矩阵m_measurement对当前状态进行更新correct()
    center.x = m_prediction.at<float>(0);
    center.y = m_prediction.at<float>(1);
    // 返回滤波预测的中心坐标点
    return center;
}
```

预测效果如下：



在30次时彻底收敛。

可以看到kal函数在内部对每次的测量值进行多次迭代预测是有优势的。对于测量值的传入而言可以更快的达到收敛。

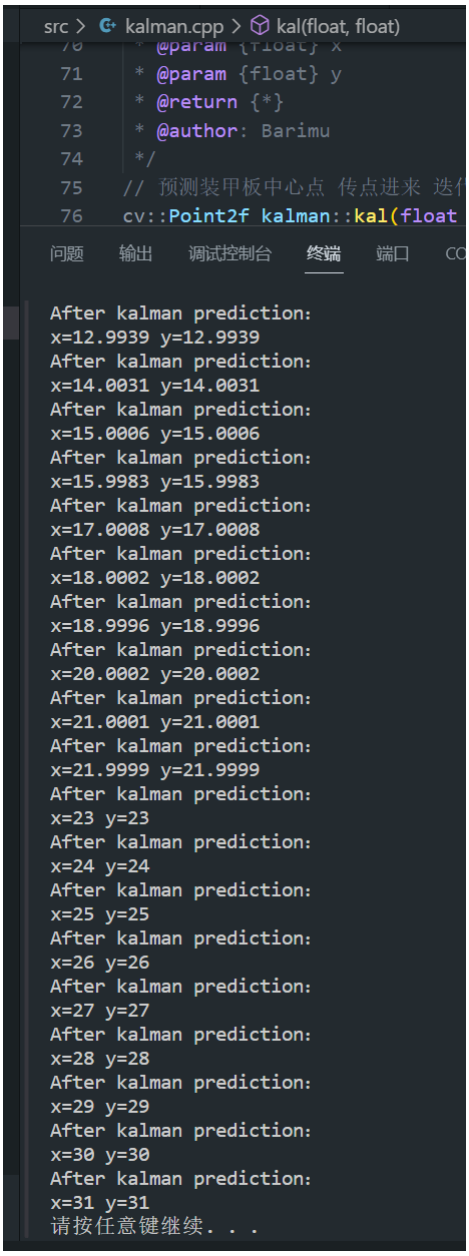
对上述函数做优化,经测试，将迭代次数改为10次后，会更快收敛到预测值。

修改如下：

```
cv::Point2f kalman::kal(float x, float y)
{
    cv::Point2f center;              // 创建中心点
    m_measurement.at<float>(0) = x; // 将测量值矩阵m_measurement的第一个元素赋值为x
    m_measurement.at<float>(1) = y;
    // m_prediction = m_KF.predict(); // 进行一次预测 调用了卡尔曼滤波器的预测函数 自带的predict()函数返回的是下一时刻的后验状态值KF.statePost(k+1)
    for (int i = 1; i <= 10; i++) // 迭代5次
    {
        m_KF.correct(m_measurement);    // 更新 用测量值矩阵m_measurement对当前状态进行更新correct()
        m_prediction = m_KF.predict(); // 再次进行预测predict(),得到新预测值
        m_measurement.at<float>(0) = m_prediction.at<float>(0);
        m_measurement.at<float>(1) = m_prediction.at<float>(1); // 将预测结果写入测量矩阵m_measurement,进行下一次迭代
```

```
    }
    // 循环迭代5次后,从预测结果矩阵m_prediction中取出滤波后的值
    center.x = m_prediction.at<float>(0);
    center.y = m_prediction.at<float>(1);
    // 返回滤波预测的中心坐标点
    return center;
}
```

测试



看到在23次时收敛，并且前面的数据也更接近与预测值。

但是，上面的kalman.cpp和kalman.h实际上目前是没有用的，因为实际的预测代码并没用到这个，而是用了自己的kalmanfilter。妹想到吧，我也妹想到。。。。。

但是目前这个预测效果其实还是可以的，只不过预测的是装甲板中心点，也就是二维的。而实际的预测是返回了三个坐标值。

predict，还需要再研究。

3.问题

首先是KalmanFilter 自己写的这个滤波器。

他的逻辑似乎是没有什么错误，但是目前发现的一个问题就是，我用一组数据去测试，实际上就是开头的那个测试案例。他的结果是"错误"的。所谓的错误就是他似乎只是做到了滤波。

也就是那个测试样例里面，我测试了100次，每次对每个测量值迭代一次预测和更新。测量值从0开始加100次到99结束。也就是相当于是一个很简单的状态方程。但是预测的结果是，收敛的很快，但是值会趋于你的测量值，在这儿也就是所谓的真实值。但是我想要的效果显然是：测量值是99，我应该给出预测是100才对（根据之前的数据变化规律）。事实是并没有这样给出。

在kalman文件中这个问题是有个解决办法的，就是用cv自己的KalmanFilter，并且用它自己的预测更新函数。要避免上述这种问题的简单做法就是，在预测-更新流程之前，首先进行一次更新，然后再预测，再更新。而不是预测、更新。这样子会发现返回下一秒的值。这在上面也是提到了.

但是目前自写的这个kalmanfilter是不能采用这种方法的。目前正在找办法。

2023 12 16 23:21

询问熊学长后得知，自写的KalmanFilter效果就是 **修正真实值**

也就是 预测值会不断接近测量值 起的作用就是滤波

如何算出预测位置呢?

熊永晨 23:12:02
通过滤波估算速度

熊永晨 23:12:17
速度成你的设置时间就是预测位移

熊永晨 23:12:28
当前坐标加预测位移就是预测坐标

所以就求速度就好了

接着研究.....