开发者说丨手把手教你实现多传感器融合技术

知识点

敲黑板，本文需要学习的知识点有

频率　位置

径向距离　角度

时间戳　观测值

多传感器融合又称多传感器信息融合（multi-sensor information fusion），有时也称多传感器数据融合（multi-sensor date fusion）。就是利用计算机技术将来自**多传感器**或**多源的信息**和**数据**，在一定的准则下加以自动分析和综合，以完成所需要的决策和估计而进行的信息处理过程。

多传感器信息融合技术的基本原理就像人的大脑综合处理信息的过程一样，将各种传感器进行**多层次**、**多空间**的信息互补和优化组合处理，最终产生对观测环境的一致性解释。在这个过程中要充分地利用多源数据进行合理支配与使用，而信息融合的最终目标则是**基于各传感器**获得的分离观测信息，通过对信息多级别、多方面组合导出更多有用信息。这不仅是利用了多个传感器相互协同操作的优势，而且也综合处理了其它信息源的数据来提高整个传感器系统的智能化。

虽然自动驾驶在全球范围内已掀起浪潮，但是从技术方面而言依然存在挑战。目前自动驾驶的痛点在于稳定可靠的感知及认知，包括**清晰的视觉**、**优质的算法**、**多传感器融合**以及高效强大的运算能力。据分析，由自动驾驶引发的安全事故原因中，相关传感器的可能误判也成为了主要原因之一。多个传感器信息融合、综合判断无疑成为提升自动驾驶安全性及赋能车辆环境感知的新趋势。

以下，ENJOY

01

前言

在**《开发者说｜手把手教你写卡尔曼滤波器》**的分享中，以激光雷达的数据为例介绍了卡尔曼滤波器（KF）的七个公式，并用C++代码实现了激光雷达障碍物的跟踪问题；在**《开发者说｜手把手教你写扩展卡尔曼滤波器》**的分享中，以毫米波雷达的数据为例，介绍了扩展卡尔曼滤波器（EKF）是如何处理非线性问题的。

无论是卡尔曼滤波器处理线性问题还是扩展卡尔曼滤波器处理非线性问题，它们都只涉及到单一传感器的障碍物跟踪。**单一传感器**有其局限性，比如激光雷达测量位置更准，但无法测量速度；毫米波雷达测量速度准确，但在位置测量的精度上低于激光雷达。为了充分利用各个传感器的优势，多传感器融合的技术应运而生。

由于多传感器融合为KF和EKF的进阶内容，推荐读者先阅读**《开发者说｜手把手教你写卡尔曼滤波器》**和**《开发者说｜手把手教你写扩展卡尔曼滤波器》**，了解卡尔曼滤波器的基本理论。

本文将以**激光雷达**和**毫米波雷达**检测同一障碍物时的数据为例，进行多传感器融合技术的讲解。同时，会提供**《优达学城（Udacity）无人驾驶工程师学位》**中所使用的多传感器数据，并提供读取数据的代码，方便大家学习。

多传感器融合的输入数据可通过以下链接获取。

链接：

https://pan.baidu.com/s/1zU9_SgZkMfs75_sXt2Odzw

提取码：**umb8**

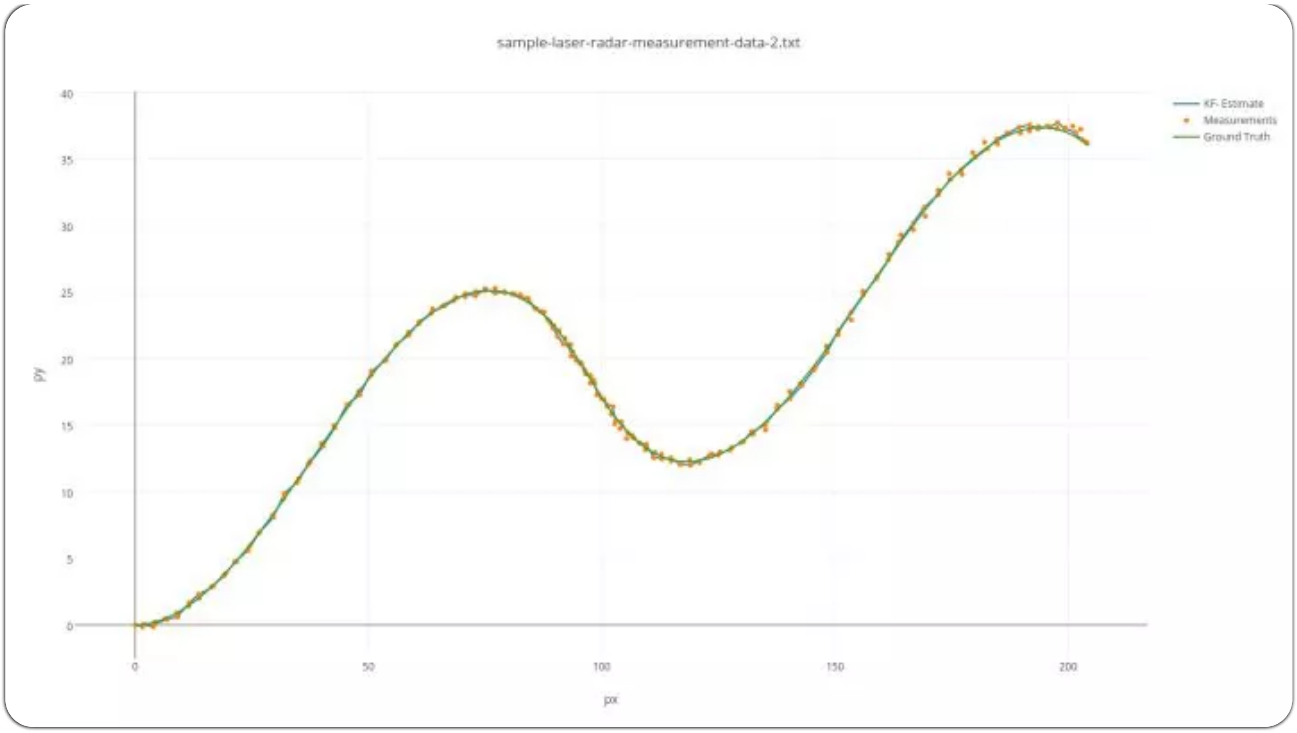
02

正文

传感器融合的输入数据

下载代码并解压后，可在`data`目录下发现一个名为`sample-laser-radar-measurement-data-2.txt`的文件，这就是传感器融合的输入数据。**数据的测量场景**是将检测障碍物频率相同的激光雷达和毫米波雷达固定在原点（0，0）处，随后两个雷达交替触发，对同一运动的障碍物进行检测。

障碍物的运动轨迹如下图所示，图中**绿线**为障碍物运动的真实运动轨迹（Ground Truth），**橙色的点**表示的是多传感器的检测结果。



▲障碍物的运动轨迹

由于激光雷达和毫米波雷达是交替触发检测的，因此系统收到的传感器数据也是交替的。这里的数据除了提供传感器的测量值外，还提供了**障碍物位置**、**速度**的真值，真值可用于评估融合算法的好坏。前十帧（行）数据如下图所示：

每帧数据的第一个字母表示该数据来自于哪一个传感器，**L**是Lidar的缩写，**R**是Radar的缩写。

Lidar只能够测量**位置**，字母L之后的数据依次为障碍物在X方向上的测量值（单位：米），Y方向上的测量值（单位：米），测量时刻的时间戳（单位：微秒）；障碍物位置在X方向上的真值（单位：米），障碍物位置在Y方向上的真值（单位：米）；障碍物速度在X方向上的真值（单位：米/秒），障碍物速度在Y方向上的真值（单位：米/秒）。

Radar能够测量径向**距离**、**角度**和**速度**，字母R之后的数据依次为障碍物在极坐标系下的距离（单位：米），角度（单位：弧度），镜像速度（单位：米/秒），测量时刻的时间戳（单位：微秒）；障碍物位置在X方向上的真值（单位：米），障碍物位置在Y方向上的真值（单位：米）；障碍物速度在X方向上的真值（单位：米/秒），障碍物速度在Y方向上的真值（单位：米/秒）。

使用以上规则对输入数据进行解析，有些类似于无人驾驶技术中的驱动层。`sample-laser-radar-measurement-data-2.txt`就像传感器通过CAN或以太网发来的数据，这里的解析规则就像解析CAN或网络数据时所用的协议。

编程时，我先读取了`sample-laser-radar-measurement-data-2.txt`文件，将每一行数据按照“协议”解析后，将观测值转存在结构体`MeasurementPackage`内，将真值转存在结构体`GroundTruthPackage`内。随后再用一个循环对这些数据进行遍历，将它们一帧帧地输入到算法中，具体代码可以参看`main.cpp`函数。

`MeasurementPackage`的内部构造如下所示：

```
//@ filename: /interface/measurement_package.h
#ifndef MEASUREMENT_PACKAGE_H_
#define MEASUREMENT_PACKAGE_H_

#include "Eigen/Dense"

class MeasurementPackage {
public:
long long timestamp_;

enum SensorType{
LASER,
RADAR
} sensor_type_;

Eigen::VectorXd raw_measurements_;
};
```

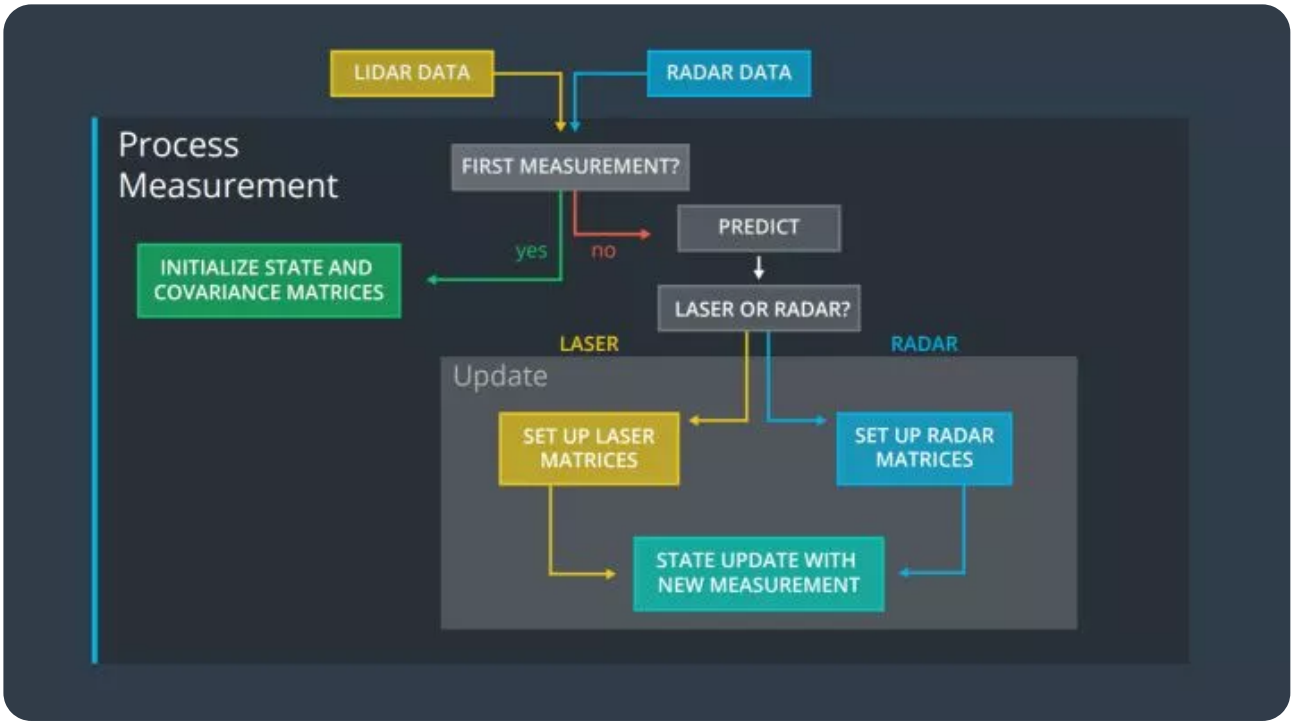
```
#endif /* MEASUREMENT_PACKAGE_H_ */
```

GroundTruthPackage的内部构造如下所示：

```
//@ filename: /interface/ground_truth_package.h
#ifndef GROUND_TRUTH_PACKAGE_H_
#define GROUND_TRUTH_PACKAGE_H_

#include "Eigen/Dense"

class GroundTruthPackage {
public:
long timestamp_;

enum SensorType{
LASER,
RADAR
} sensor_type_;

Eigen::VectorXd gt_values_;

};

#endif /* GROUND_TRUTH_PACKAGE_H_ */
```

III 传感器融合的算法逻辑

由于激光雷达和毫米波雷达的传感器数据有所差异，因此算法处理时也有所不同，先罗列出一个初级的融合算法框架，如下图所示。



▲图片出处：优达学城（Udacity）无人驾驶工程师学位

首先**读入传感器数据**，如果是第一次读入，则需要对卡尔曼滤波器的各个矩阵进行初始化操作；如果不是第一次读入，证明卡尔曼滤波器已完成初始化，直接进行**状态预测**和**状态值**更新的步骤；最后**输出融合后的障碍物位置、速度**。

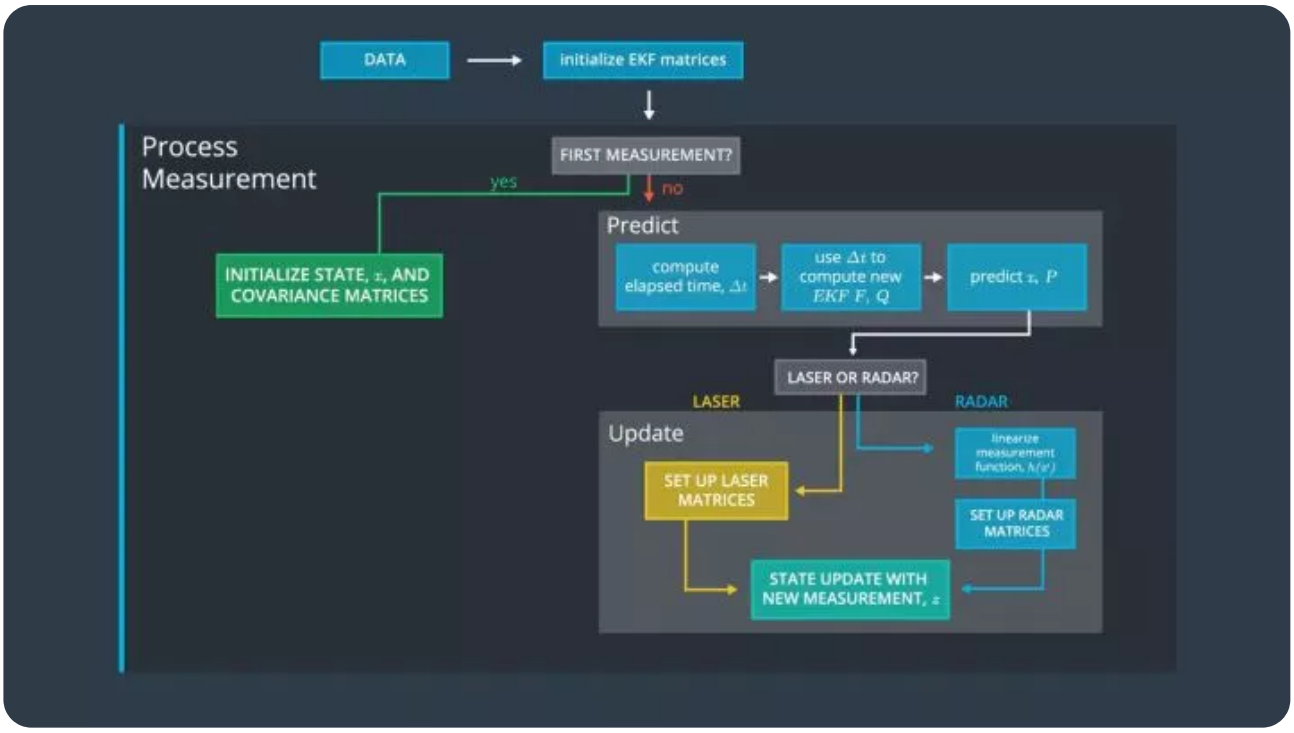
我们将以上过程写成伪代码，方便我们理解随后的开发工作。

▲传感器融合伪代码

通过对比**《开发者说｜手把手教你写卡尔曼滤波器》**和**《开发者说｜手把手教你写扩展卡尔曼滤波器》**中的KF与EKF的代码会发现，对于匀速运动模型，KF和EKF的状态预测（Predict）过程是一样的；KF和EKF唯一区别的地方在于测量值更新（Update）这一步。在测量值更新中，KF使用的测量矩阵H是不变的，而EKF的测量矩阵H是**Jacobian矩阵**。

因此，我们可以将KF和EKF写成同一个类，这个类中有两个`Update`函数，分别命名为`KFUpdate`和`EKFUpdate`。这两个函数对应着伪代码中第12行的“卡尔曼滤波更新”和第15行的“扩展卡尔曼滤波更新”。

根据卡尔曼滤波器的**预测**和**测量值**更新这两大过程，我们将算法逻辑细化，如下图所示。



▲图片出处：优达学城（Udacity）无人驾驶工程师学位

接下来我们合并KF和EKF的跟踪算法的代码，将他们封装在一个名为`KalmanFilter`的类中，方便后续调用，改写后的`KalmanFilter`类如下所示：

```
//@ filename: /algorithims/kalman.h
#pragma once

#include "Eigen/Dense"

class KalmanFilter {
```

```
public:

KalmanFilter();
~KalmanFilter();

void Initialization(Eigen::VectorXd x_in);

bool IsInitialized();

void SetF(Eigen::MatrixXd F_in);

void SetP(Eigen::MatrixXd P_in);

void SetQ(Eigen::MatrixXd Q_in);

void SetH(Eigen::MatrixXd H_in);

void SetR(Eigen::MatrixXd R_in);

void Prediction();

void KFUpdate(Eigen::VectorXd z);

void EKFUpdate(Eigen::VectorXd z);

Eigen::VectorXd GetX();

private:

void CalculateJacobianMatrix();

// flag of initialization
bool is_initialized_;

// state vector
Eigen::VectorXd x_;

// state covariance matrix
Eigen::MatrixXd P_;

// state transistion matrix
Eigen::MatrixXd F_;

// process covariance matrix
Eigen::MatrixXd Q_;

// measurement matrix
Eigen::MatrixXd H_;

// measurement covariance matrix
Eigen::MatrixXd R_;
};
```

```
//@ filename: /algorithims/kalman.cpp
#include "kalmanfilter.h"


KalmanFilter::KalmanFilter()
{
is_initialized_ = false;
}

KalmanFilter::~KalmanFilter()
{

}

void KalmanFilter::Initialization(Eigen::VectorXd x_in)
{
x_ = x_in;
}

bool KalmanFilter::IsInitialized()
{
return is_initialized_;
}

void KalmanFilter::SetF(Eigen::MatrixXd F_in)
{
F_ = F_in;
}

void KalmanFilter::SetP(Eigen::MatrixXd P_in)
{
P_ = P_in;
}

void KalmanFilter::SetQ(Eigen::MatrixXd Q_in)
{
Q_ = Q_in;
}

void KalmanFilter::SetH(Eigen::MatrixXd H_in)
{
H_ = H_in;
}

void KalmanFilter::SetR(Eigen::MatrixXd R_in)
{
R_ = R_in;
}

void KalmanFilter::Prediction()
```

```
{
x_ = F_ * x_;
Eigen::MatrixXd Ft = F_.transpose();
P_ = F_ * P_ * Ft + Q_;
}

void KalmanFilter::KFUpdate(Eigen::VectorXd z)
{
Eigen::VectorXd y = z - H_ * x_;
Eigen::MatrixXd Ht = H_.transpose();
Eigen::MatrixXd S = H_ * P_ * Ht + R_;
Eigen::MatrixXd Si = S.inverse();
Eigen::MatrixXd K =  P_ * Ht * Si;
x_ = x_ + (K * y);
int x_size = x_.size();
Eigen::MatrixXd I = Eigen::MatrixXd::Identity(x_size, x_size);
P_ = (I - K * H_) * P_;
}

void KalmanFilter::EKFUpdate(Eigen::VectorXd z)
{
double rho = sqrt(x_(0)*x_(0) + x_(1)*x_(1));
double theta = atan2(x_(1), x_(0));
double rho_dot = (x_(0)*x_(2) + x_(1)*x_(3)) / rho;
Eigen::VectorXd h = Eigen::VectorXd(3);
h << rho, theta, rho_dot;
Eigen::VectorXd y = z - h;

CalculateJacobianMatrix();

Eigen::MatrixXd Ht = H_.transpose();
Eigen::MatrixXd S = H_ * P_ * Ht + R_;
Eigen::MatrixXd Si = S.inverse();
Eigen::MatrixXd K =  P_ * Ht * Si;
x_ = x_ + (K * y);
int x_size = x_.size();
Eigen::MatrixXd I = Eigen::MatrixXd::Identity(x_size, x_size);
P_ = (I - K * H_) * P_;
}

Eigen::VectorXd KalmanFilter::GetX()
{
return x_;
}

void KalmanFilter::CalculateJacobianMatrix()
{
Eigen::MatrixXd Hj(3, 4);

// get state parameters
float px = x_(0);
float py = x_(1);
```

```
float vx = x_(2);
float vy = x_(3);

// pre-compute a set of terms to avoid repeated calculation
float c1 = px * px + py * py;
float c2 = sqrt(c1);
float c3 = (c1 * c2);

// Check division by zero
if(fabs(c1) < 0.0001){
     H_ = Hj;
     return;
}

Hj << (px/c2), (py/c2), 0, 0,
      -(py/c1), (px/c1), 0, 0,
       py*(vx*py - vy*px)/c3, px*(px*vy - py*vx)/c3, px/c2, py/c2;
H_ = Hj;
}
```

KalmanFilter类封装了卡尔曼滤波的七个公式，专门用于实现障碍物的跟踪。为了使代码尽可能解耦且结构清晰，我们新建一个名为SensorFusion的类。这个类的作用是将数据层和算法层隔离开。即使外部传入的数据结构（接口）发生变化，修改SensorFusion类即可完成数据的适配，而不用修改KalmanFilter算法部分的代码，增强算法的复用性。

在SensorFusion类中添加一个函数Process()用于传入观测值，并在函数Process()中调用算法。如下图所示，在k+1时刻收到激光雷达数据时，根据k时刻的状态完成一次预测，再根据k+1时刻的激光雷达的观测数据实现测量值更新（KFUpdate）；在k+2时刻收到毫米波雷达的数据时，根据k+1时刻的状态完成一次预测，再根据k+2时刻的毫米波雷达的观测数据实现测量值更新（EKFUpdate）。

▲图片出处：优达学城（Udacity）无人驾驶工程师学位

将以上过程转换为代码，在SensorFusion中实现，如下所示：

```
//@ filename: /algorithims/sensorfusion.h
#pragma once

#include "interface/measurement_package.h"
#include "kalmanfilter.h"

class SensorFusion {
public:
SensorFusion();
~SensorFusion();

void Process(MeasurementPackage measurement_pack);
KalmanFilter kf_;

private:
bool is_initialized_;
long last_timestamp_;
Eigen::MatrixXd R_lidar_;
Eigen::MatrixXd R_radar_;
Eigen::MatrixXd H_lidar_;
};


//@ filename: /algorithims/sensorfusion.cpp
```

```
#include "sensorfusion.h"

SensorFusion::SensorFusion()
{
is_initialized_ = false;
last_timestamp_ = 0.0;

// 初始化激光雷达的测量矩阵 H_lidar_
// Set Lidar's measurement matrix H_lidar_
H_lidar_ = Eigen::MatrixXd(2, 4);
H_lidar_ << 1, 0, 0, 0,
            0, 1, 0, 0;

// 设置传感器的测量噪声矩阵，一般由传感器厂商提供，如未提供，也可通过有经验的工程师调试得到
// Set R. R is provided by Sensor supplier, in sensor datasheet
// set measurement covariance matrix
R_lidar_ = Eigen::MatrixXd(2, 2);
R_lidar_ << 0.0225, 0,
            0, 0.0225;

// Measurement covariance matrix - radar
R_radar_ = Eigen::MatrixXd(3, 3);
R_radar_ << 0.09, 0, 0,
            0, 0.0009, 0,
            0, 0, 0.09;
}

SensorFusion::~SensorFusion()
{

}

void SensorFusion::Process(MeasurementPackage measurement_pack)
{
// 第一帧数据用于初始化 Kalman 滤波器
if (!is_initialized_) {
    Eigen::Vector4d x;
    if (measurement_pack.sensor_type_ == MeasurementPackage::LASER) {
        // 如果第一帧数据是激光雷达数据，没有速度信息，因此初始化时只能传入位置，速度设置为0
        x << measurement_pack.raw_measurements_[0], measurement_pack.raw_measuren
    } else if (measurement_pack.sensor_type_ == MeasurementPackage::RADAR) {
        // 如果第一帧数据是毫米波雷达，可以通过三角函数算出x-y坐标系下的位置和速度
        float rho = measurement_pack.raw_measurements_[0];
        float phi = measurement_pack.raw_measurements_[1];
        float rho_dot = measurement_pack.raw_measurements_[2];
        float position_x = rho * cos(phi);
        if (position_x < 0.0001) {
            position_x = 0.0001;
        }
        float position_y = rho * sin(phi);
        if (position_y < 0.0001) {
            position_y = 0.0001;
```

```
        }
        float velocity_x = rho_dot * cos(phi);
        float velocity_y = rho_dot * sin(phi);
        x << position_x, position_y, velocity_x , velocity_y;
    }

    // 避免运算时，0作为被除数
    if (fabs(x(0)) < 0.001) {
        x(0) = 0.001;
    }
    if (fabs(x(1)) < 0.001) {
        x(1) = 0.001;
    }
    // 初始化Kalman滤波器
    kf_.Initialization(x);

    // 设置协方差矩阵P
    Eigen::MatrixXd P = Eigen::MatrixXd(4, 4);
    P << 1.0, 0.0, 0.0, 0.0,
         0.0, 1.0, 0.0, 0.0,
         0.0, 0.0, 1000.0, 0.0,
         0.0, 0.0, 0.0, 1000.0;
    kf_.SetP(P);

    // 设置过程噪声Q
    Eigen::MatrixXd Q = Eigen::MatrixXd(4, 4);
    Q << 1.0, 0.0, 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, 1.0;
    kf_.SetQ(Q);

    // 存储第一帧的时间戳，供下一帧数据使用
    last_timestamp_ = measurement_pack.timestamp_;
    is_initialized_ = true;
    return;
}

// 求前后两帧的时间差，数据包中的时间戳单位为微秒，处以1e6，转换为秒
double delta_t = (measurement_pack.timestamp_ - last_timestamp_) / 1000000.0; //
last_timestamp_ = measurement_pack.timestamp_;

// 设置状态转移矩阵F
Eigen::MatrixXd F = Eigen::MatrixXd(4, 4);
F << 1.0, 0.0, delta_t, 0.0,
     0.0, 1.0, 0.0, delta_t,
     0.0, 0.0, 1.0, 0.0,
     0.0, 0.0, 0.0, 1.0;
kf_.SetF(F);

// 预测
kf_.Prediction();
```

```
// 更新
if (measurement_pack.sensor_type_ == MeasurementPackage::LASER) {
     kf_.SetH(H_lidar_);
     kf_.SetR(R_lidar_);
     kf_.KFUpdate(measurement_pack.raw_measurements_);
} else if (measurement_pack.sensor_type_ == MeasurementPackage::RADAR) {
     kf_.SetR(R_radar_);
     // Jocobian矩阵Hj的运算已包含在EKFUpdate中
     kf_.EKFUpdate(measurement_pack.raw_measurements_);
}
}
```

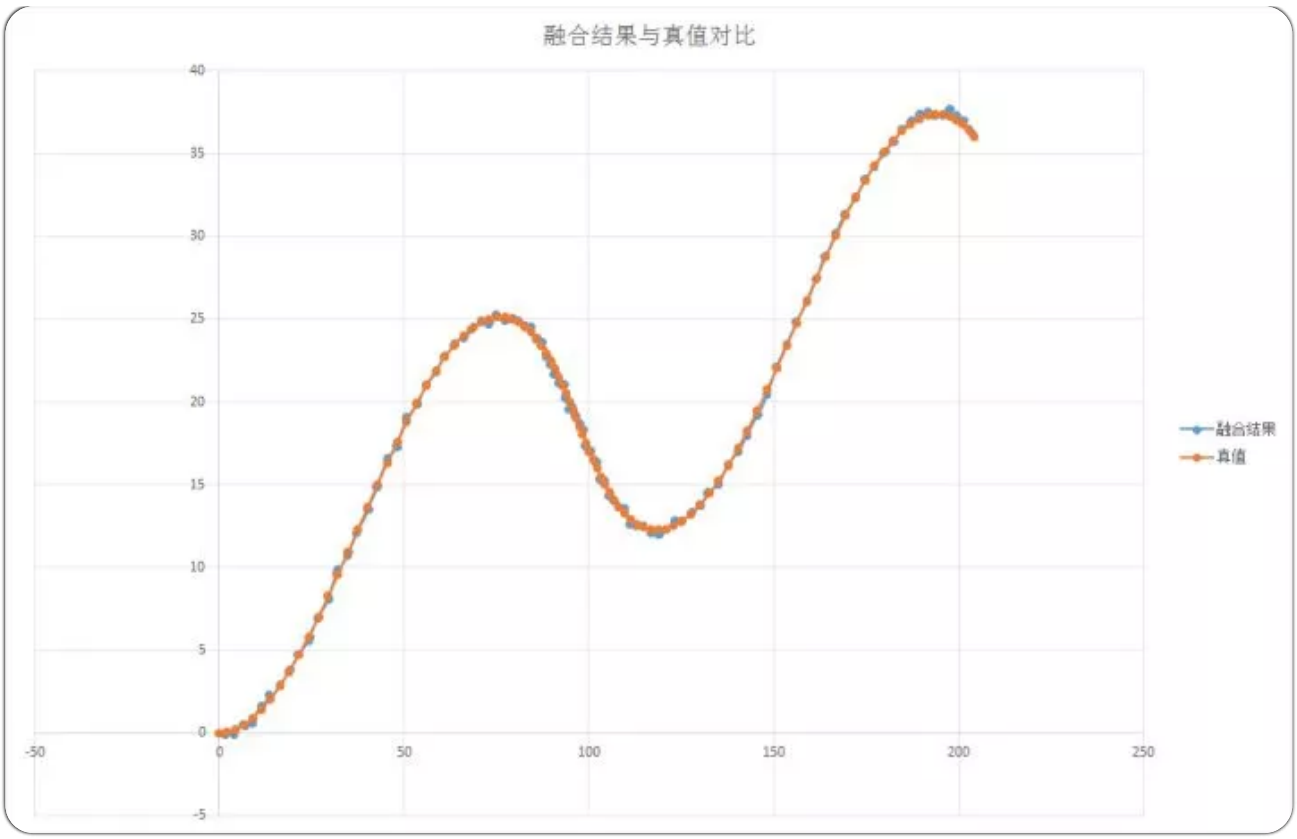
完成传感器数据的读取和融合算法的编写后，我们将两者组合起来，写在main.cpp中，输出每一帧的障碍物融合结果。

```
//@ filename: main.cpp
#include
#include
#include
#include
#include
#include "Eigen/Eigen"
#include "interface/ground_truth_package.h"
#include "interface/measurement_package.h"
#include "algorithims/sensorfusion.h"
```

```
int main(int argc, char* argv[]) {

// 设置毫米波雷达/激光雷达输入数据的路径
// Set radar & lidar data file path
std::string input_file_name = "../data/sample-laser-radar-measurement-data-2.txt";

// 打开数据，若失败则输出失败信息，返回-1，并终止程序
// Open file. if failed return -1 & end program
std::ifstream input_file(input_file_name.c_str(), std::ifstream::in);
if (!input_file.is_open()) {
     std::cout << "Failed to open file named : " << input_file_name << std::endl;
     return -1;
}

// 分配内存
// measurement_pack_list：毫米波雷达/激光雷达实际测得的数据。数据包含测量值和时间戳，即融合算法的
// groundtruth_pack_list：每次测量时，障碍物位置的真值。对比融合算法输出和真值的差别，用于评估融合
std::vector measurement_pack_list;
std::vector groundtruth_pack_list;

// 通过while循环将雷达测量值和真值全部读入内存，存入measurement_pack_list和groundtruth_pack_
// Store radar & lidar data into memory
std::string line;
while (getline(input_file, line)) {
     std::string sensor_type;
     MeasurementPackage meas_package;
     GroundTruthPackage gt_package;
     std::istringstream iss(line);
     long long timestamp;

     // 读取当前行的第一个元素，L代表Lidar数据，R代表Radar数据
     // Reads first element from the current line. L stands for Lidar. R stands f
     iss >> sensor_type;
     if (sensor_type.compare("L") == 0) {
          // 激光雷达数据 Lidar data
          // 该行第二个元素为测量值x，第三个元素为测量值y，第四个元素为时间戳(纳秒)
          // 2nd element is x; 3rd element is y; 4th element is timestamp(nano se
          meas_package.sensor_type_ = MeasurementPackage::LASER;
          meas_package.raw_measurements_ = Eigen::VectorXd(2);
          float x;
          float y;
          iss >> x;
          iss >> y;
          meas_package.raw_measurements_ << x, y;
          iss >> timestamp;
          meas_package.timestamp_ = timestamp;
          measurement_pack_list.push_back(meas_package);
     } else if (sensor_type.compare("R") == 0) {
          // 毫米波雷达数据 Radar data
          // 该行第二个元素为距离pho，第三个元素为角度phi，第四个元素为径向速度pho_dot，第五个元素
          // 2nd element is pho; 3rd element is phi; 4th element is pho_dot; 5th
          meas_package.sensor_type_ = MeasurementPackage::RADAR;
```

```
            meas_package.raw_measurements_ = Eigen::VectorXd(3);
            float rho;
            float phi;
            float rho_dot;
            iss >> rho;
            iss >> phi;
            iss >> rho_dot;
            meas_package.raw_measurements_ << rho, phi, rho_dot;
            iss >> timestamp;
            meas_package.timestamp_ = timestamp;
            measurement_pack_list.push_back(meas_package);
        }

        // 当前行的最后四个元素分别是x方向上的距离真值，y方向上的距离真值，x方向上的速度真值，y方向上的i
        // read ground truth data to compare later
        float x_gt;
        float y_gt;
        float vx_gt;
        float vy_gt;
        iss >> x_gt;
        iss >> y_gt;
        iss >> vx_gt;
        iss >> vy_gt;
        gt_package.gt_values_ = Eigen::VectorXd(4);
        gt_package.gt_values_ << x_gt, y_gt, vx_gt, vy_gt;
        groundtruth_pack_list.push_back(gt_package);
    }

    std::cout << "Success to load data." << std::endl;

    // 部署跟踪算法
    SensorFusion fuser;

    for (size_t i = 0; i < measurement_pack_list.size(); ++i) {
        fuser.Process(measurement_pack_list[i]);
        Eigen::Vector4d x_out = fuser.kf_.GetX();
        // 输出跟踪后的位置
        std::cout << "x " << x_out(0)
                  << " y " << x_out(1)
                  << " vx " << x_out(2)
                  << " vy " << x_out(3)
                  << std::endl;
    }
}
```

将融合结果与真值绘制在同一坐标系下，即可看到融合的实际效果，如下图所示。



▲融合结果与真值对比

由图可以看出，融合结果与真值基本吻合，这只是定性的分析，缺乏定量的描述。随后课程介绍了一种定量分析的方式——**均方根误差**（RMSE，Root Mean Squared Error），其计算方式是预测值与真实值偏差的平方与观测次数n比值的平方根，如下公式所示：

$$RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(x_t^{est} - x_t^{true})^2}$$

▲均方根误差计算公式

分别计算x、y、vx和vy的RMSE，如果RMSE的值越小，则证明结果与真值越接近，跟踪效果越好。

作为入门课程，本次分享仅仅介绍了激光雷达和毫米波雷达对单一障碍物的融合。实际工程开发时，多传感器融合算法工程师除了要掌握融合算法的理论和编码外，还要学习不同传感器（激光雷达、毫米波雷达、摄像机等）的数据特性和相应的障碍物检测算法，这样才能在各种传感器之间游刃有余。