**扩展卡尔曼滤波EKF与多传感器融合**

[1 概述 2](#_Toc142957737)

[2 KF与EKF 2](#_Toc142957738)

[3 为什么要用EKF 3](#_Toc142957739)

[4 将非线性系统线性化 3](#_Toc142957740)

[5 多传感器融合 3](#_Toc142957741)

[(1) lidar与radar 3](#_Toc142957742)

[(2) 传感器融合步骤 4](#_Toc142957743)

[(3) 初始化 4](#_Toc142957744)

[(4) 预测未来 4](#_Toc142957745)

[6 修正当下 5](#_Toc142957746)

[(1) lidar 5](#_Toc142957747)

[(2) radar 6](#_Toc142957748)

[7 传感器融合实例 6](#_Toc142957749)

[8 Matlab代码 7](#_Toc142957750)

[9 C\_plus\_plus\_code 9](#_Toc142957751)

[(1) Main\_cpp 9](#_Toc142957752)

[(2) Kalman\_filter\_cpp 13](#_Toc142957753)

[(3) Fusion\_ekf\_cpp 14](#_Toc142957754)

[(4) Fusion\_ekf\_h 16](#_Toc142957755)

[(5) Ground\_truth\_package\_h 16](#_Toc142957756)

[(6) Kalman\_filter\_h 17](#_Toc142957757)

[(7) Measurement\_package\_h 18](#_Toc142957758)

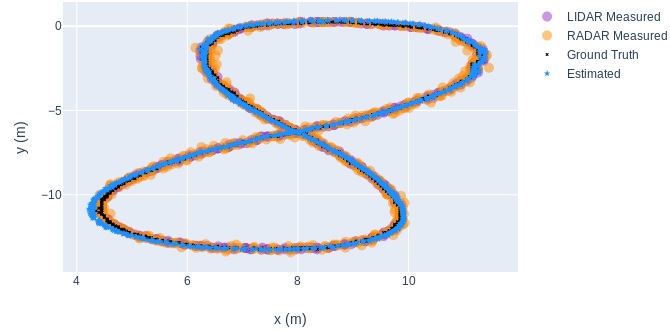
[(8) Utilities\_h 18](#_Toc142957759)

[(9) CMakeLists\_txt 19](#_Toc142957760)

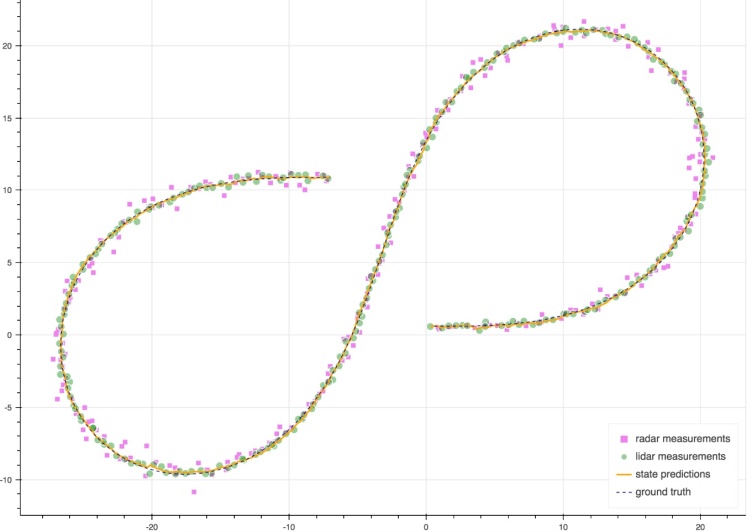
[10 参考文献 20](#_Toc142957761)

# 概述

Extended Kalman Filter（扩展卡尔曼滤波）是卡尔曼滤波的非线性版本。在状态转移方程确定的情况下，EKF已经成为了非线性系统状态估计的事实标准。本文将简要介绍EKF，并介绍其在无人驾驶多传感器融合上的应用。



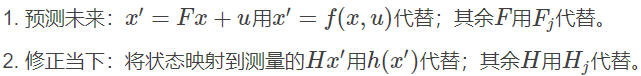
上述基于EKF的Lidar和Rader信号的融合效果不是很理想，左下角阶段，融合后的误差甚至大于融合前Lidar和Radar两个传感器独立测量的误差。可以采用无迹卡尔曼滤波，如文献4所示，获得如下的理想的融合效果。

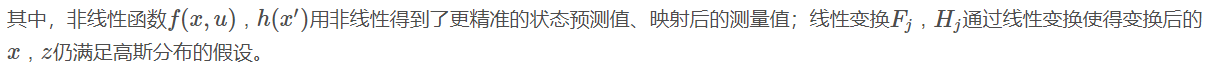


# KF与EKF

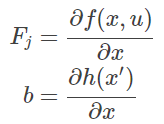
本文假定读者已熟悉KF，若不熟悉请参考卡尔曼滤波简介。

KF与EKF的区别如下：

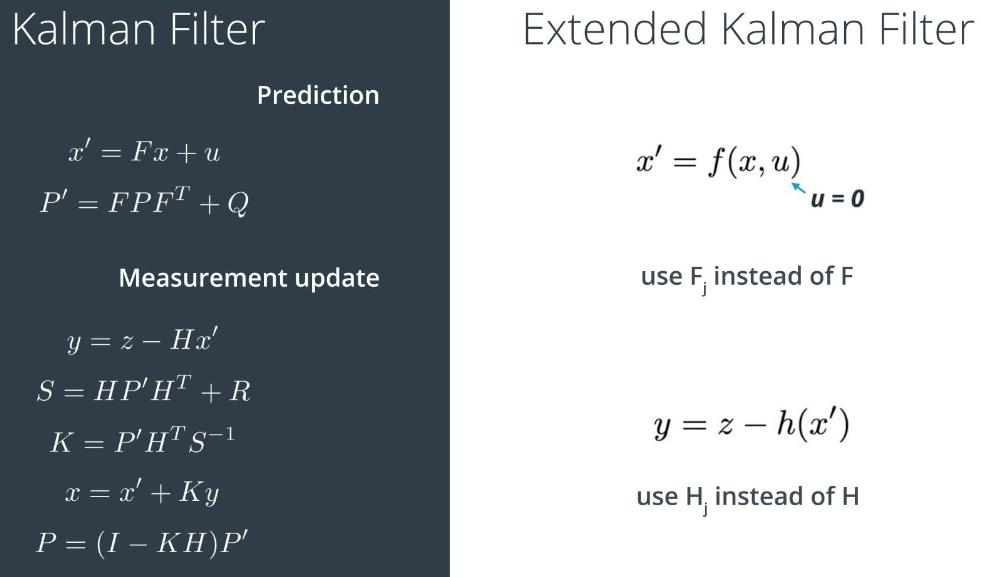








[上述公式参考黄小平著卡尔曼滤波，Eq(4.10),Eq(4.11)]



# 为什么要用EKF

KF的假设之一就是高斯分布的x预测后仍服从高斯分布，高斯分布的x变换到测量空间后仍服从高斯分布。可是，假如F、H是非线性变换，那么上述条件则不成立。

# 将非线性系统线性化

既然非线性系统不行，那么很自然的解决思路就是将非线性系统线性化。对于一维系统，采用泰勒一阶展开即可得到：



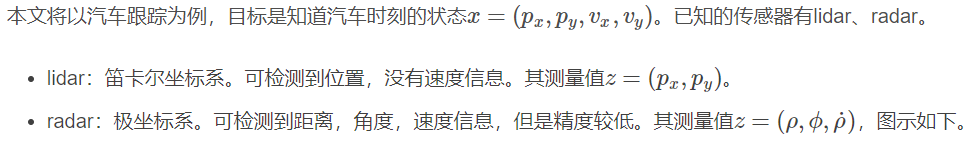
对于多维系统，仍旧采用泰勒一阶展开即可得到：



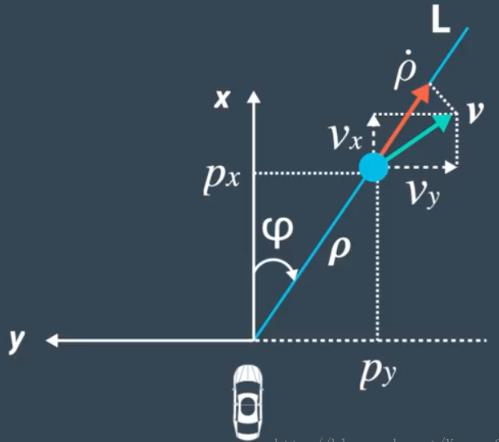
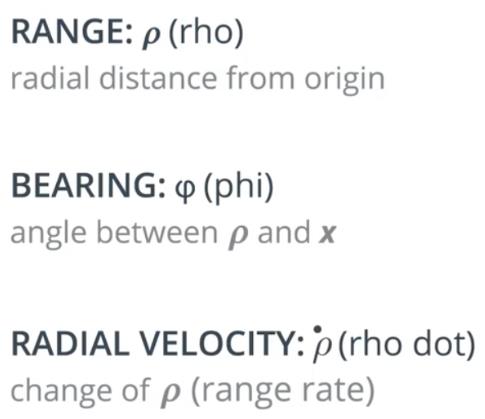


# 多传感器融合

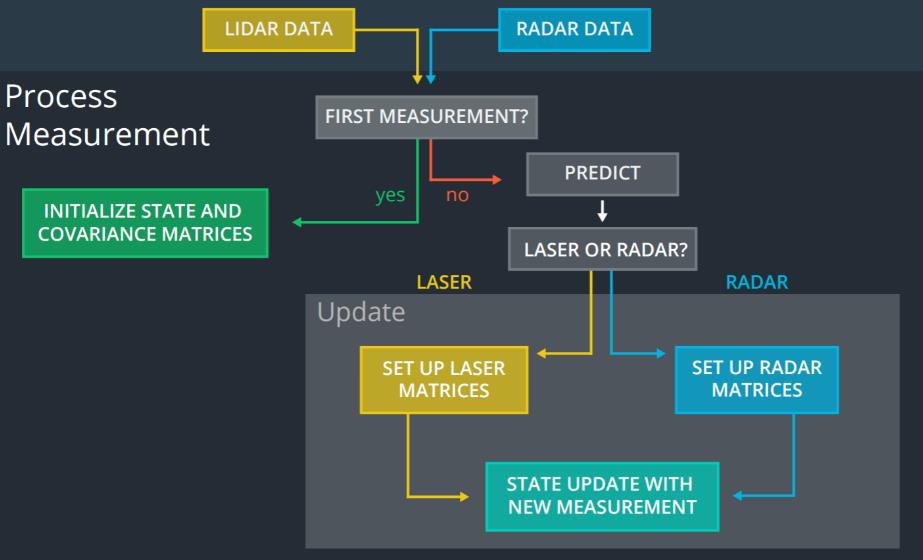
## lidar与radar



其中*ρ*为离原点的半径的距离。



## 传感器融合步骤

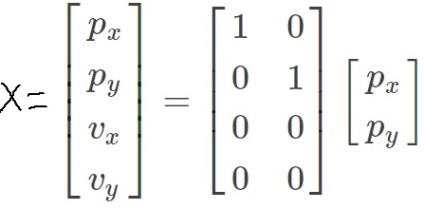


步骤图如上所示，包括：

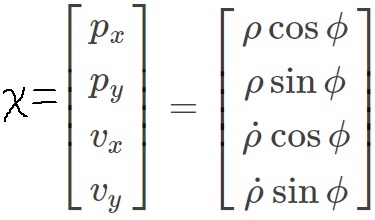
1. 收到第一个测量值，对状态x进行初始化。
2. 预测未来
3. 修正当下

## 初始化

初始化，指在收到第一个测量值后，对状态x进行初始化。初始化如下，同时加上对时间的更新。Lidar和radar的预测方程都一样，但是他们的更新方程不一样。此处需要统一lidar和radar两者的预测方程中的状态x的表达式。

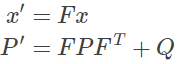
对于lidar来说，  


对于radar来说，



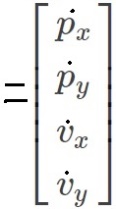
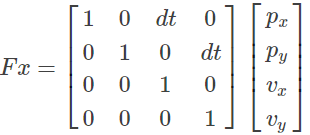
## 预测未来

预测主要涉及的公式是：

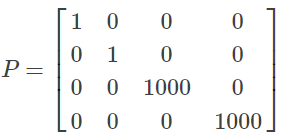


需要求解的有三个变量：F、P、Q。

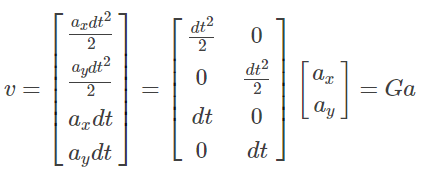
F表明了系统的状态如何改变，这里仅考虑线性系统，F易得：



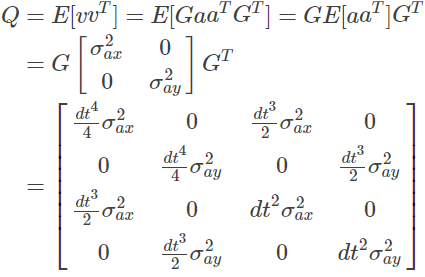
P表明了系统状态的不确定性程度，用x的协方差表示，这里自己指定为：



Q表明了x′=Fx未能刻画的其他外界干扰。本例子使用线性模型，因此加速度变成了干扰项。x′=Fx中未衡量的额外项目v(速度)为：



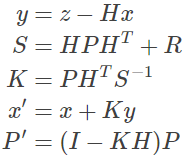
v服从高斯分布N(0,Q)。



# 修正当下

## lidar

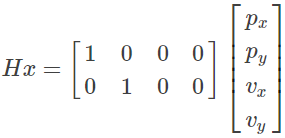
lidar使用了KF。修正当下这里牵涉到的公式主要是：



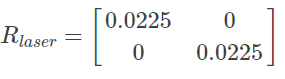
激光雷达的y只包含[px,py]。

需要求解的有两个变量：H、R。

H表示了状态空间到测量空间的映射。

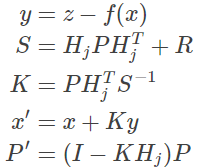


R表示了测量值的不确定度，一般由传感器的厂家提供，这里lidar参考如下：



## radar

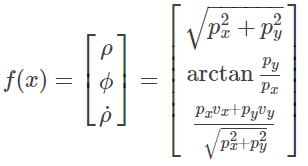
radar使用了EKF。修正当下这里牵涉到的公式主要是：



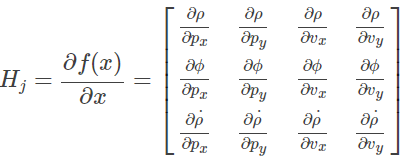
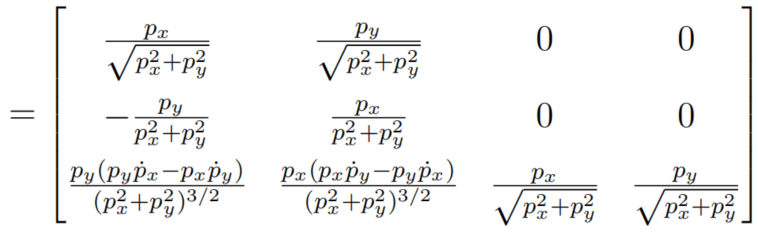
区别与上面lidar的主要有：

1. 状态空间到测量空间的非线性映射f(x)
2. 非线性映射线性化后的Jacob矩阵
3. radar的Rradar
4. y包含

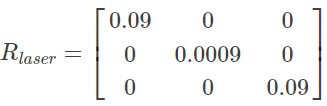
状态空间到测量空间的非线性映射f(x)如下：



非线性映射线性化后的Jacob矩阵Hj

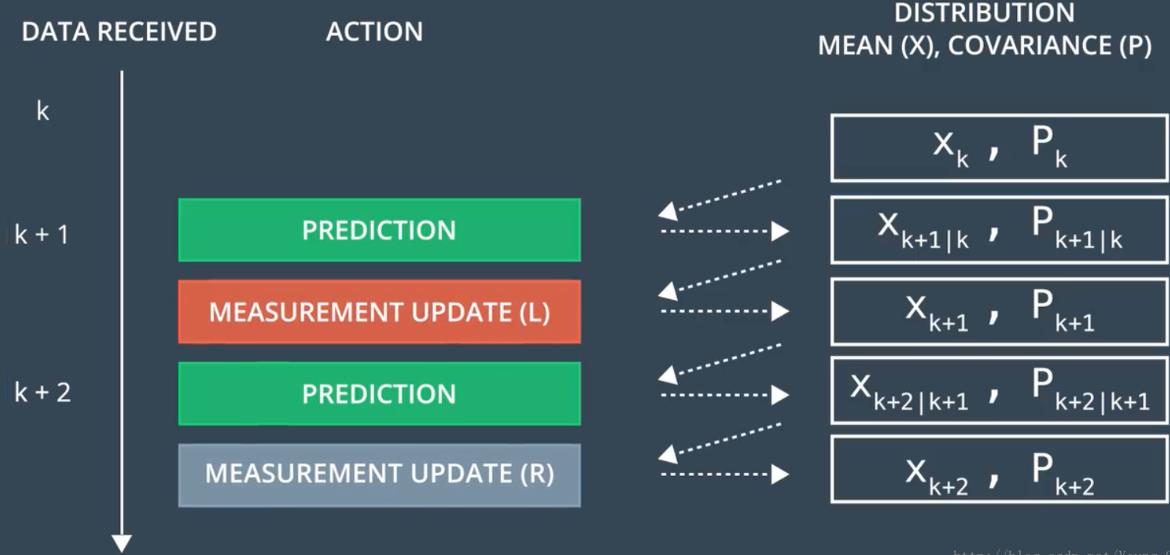
R表示了测量值的不确定度，一般由传感器的厂家提供，这里radar参考如下：

=R\_r

# 传感器融合实例

多传感器融合的示例如下，需要注意的有：

1. lidar和radar的预测部分是完全相同的
2. lidar和radar的参数更新部分是不同的，不同的原因是不同传感器收到的测量值是不同的
3. 当收到lidar或radar的测量值，依次执行预测、更新步骤
4. 当同时收到lidar和radar的测量值，依次执行预测、更新1、更新2步骤



Lidar采集数据和Radar采集数据融合的步骤如下：



多传感器融合的效果如下图所示，红点和蓝点分别表示radar和lidar的测量位置，绿点代表了EKF经过多传感器融合后获取到的测量位置，取得了较低的RMSE。

# Matlab代码

dt = 0.1;

Data = csvread('Radar\_Lidar\_Data1.csv',1,1); % Data = csvread('Radar\_Lidar\_Data2.csv',1,1);

Radar\_Measurement = [];

Lidar\_Measurement = [];

EKF\_Path = [];

F = [[1, 0, dt, 0];

[0, 1, 0, dt];

[0, 0, 1, 0];

[0, 0, 0, 1]];

u = 0;

B = [(dt^2)/2 (dt^2)/2 dt dt]';

P = [[1, 0, 0, 0];

[0, 1, 0, 0];

[0, 0, 1000, 0];

[0, 0, 0, 1000]];

R\_l = [[0.0025, 0];

[0, 0.0025]];

R\_r = [[0.09, 0, 0];

[0, 0.005, 0];

[0, 0, 0.09]];

Q = [(dt^2)/4 0 (dt^3)/2 0;

0 (dt^2)/4 0 (dt^3)/2;

(dt^3/2) 0 (dt^2) 0;

0 (dt^3)/2 0 (dt^2)];

H = [[1, 0, 0, 0];

[0, 1, 0, 0]];

I = eye(4);

if (Data(1,1) == 1)

x = [Data(1,2); Data(1,3); 0; 0];

else

x = [Data(1,2); Data(1,3); Data(1,4); 0];

end

for n = 1:length(Data)

if (Data(n,1) == 2)

% 右边的传感器是毫米波雷达，对应编号2；

%prediction；这里是毫米波雷达的预测部分，和毫米波雷达的预测部分相同

x = F \* x + B\*u; 状态向量x=[px,py,vx,vy]

P = F \* P \* transpose(F) + Q;

%measurement update

%毫米波雷达的直接测量数据为[*ρ, ф,* ]，需要转化为状态向量x=[px,py,vx,vy]

Z = Data(n,2:4);

X = Z(1)\*cos(Z(2));

Y = Z(1)\*sin(Z(2));

VX = Z(3)\*cos(Z(2));

VY = Z(3)\*sin(Z(2));

c1 = X^2 + Y^2;

c2 = sqrt(c1);

c3 = c1 \* c2;

if (c1==0 || c2==0 || c3==0)

H\_Jac = [[0, 0, 0, 0];

[0, 0, 0, 0];

[0, 0, 0, 0]];

else

H\_Jac = [[X/c2, Y/c2, 0, 0];

[-Y/c1, X/c1, 0, 0];

[(Y\*(VX\*Y-VY\*X))/c3, (X\*(X\*VY-Y\*VX))/c3, X/c2, Y/c2]];

end

Z\_Car = [X; Y; VX; VY];

y = transpose(Z) - (H\_Jac \* Z\_Car);

S = H\_Jac \* P \* transpose(H\_Jac) + R\_r;

K = P \* transpose(H\_Jac) \* inv(S);

x = Z\_Car + (K \* y);

P = (I - (K \* H\_Jac)) \* P;

EKF\_Path = [EKF\_Path;[x(1),x(2)]];

Radar\_Measurement = [Radar\_Measurement; Data(n,2:4)];

else %左边的传感器是激光雷达，对应编号1

%prediction；这里是激光雷达的预测部分，和毫米波雷达的预测部分相同

x = (F \* x) + B\*u; 状态向量x=[px,py,vx,vy]

P = F \* P \* transpose(F) + Q;

%measurement update；

% 激光雷达测量值只有px和py，然后通过增益矩阵的运算，生成状态向量x=[px,py,vx,vy]

Z = Data(n,2:3);

y = transpose(Z) - (H \* x);

S = H \* P \* transpose(H) + R\_l;

K = P \* transpose(H) \* inv(S);

x = x + (K \* y);

P = (I - (K \* H)) \* P;

EKF\_Path = [EKF\_Path;[x(1),x(2)]];

Lidar\_Measurement = [Lidar\_Measurement; Data(n,2:3)];

end

end

for i = 1:length(Radar\_Measurement)

Radar\_Measurement\_Cart(i,:) = [[Radar\_Measurement(i,1),0];[0, Radar\_Measurement(i,1)]]\*[cos(Radar\_Measurement(i,2));sin(Radar\_Measurement(i,2))];

end

hold on;

plot(Data(:,6),Data(:,7),'linewidth', 2);

scatter(EKF\_Path(:,1),EKF\_Path(:,2),25,'filled','r');

scatter(Lidar\_Measurement(:,1),Lidar\_Measurement(:,2),5,'filled','blue');

scatter(Radar\_Measurement\_Cart(:,1),Radar\_Measurement\_Cart(:,2),5,'filled','g');

legend('Grundtruth','EKF Path result','Lidar Measurement','Radar Measurement','Location','northwest');

axis square;

hold off;

# C\_plus\_plus\_code

## Main\_cpp

#include <fstream>  
#include <iostream>  
#include <vector>  
#include <Eigen/Dense>  
  
#include "fusion\_ekf.h"  
#include "ground\_truth\_package.h"  
#include "measurement\_package.h"  
#include "utilities.h"  
// Calculate root mean square error.  
inline Eigen::VectorXd calculateRMSE(const std::vector<Eigen::VectorXd>& estimations,  
 const std::vector<Eigen::VectorXd>& ground\_truth) {  
 Eigen::VectorXd rmse(4);  
 rmse << 0, 0, 0, 0;  
 for (std::size\_t i=0; i != estimations.size(); ++i) {  
 Eigen::VectorXd residual = estimations[i] - ground\_truth[i];  
 residual = residual.array()\*residual.array();  
 rmse += residual;  
 }  
 rmse /= estimations.size();  
 rmse = rmse.array().sqrt();  
 return rmse;  
}  
  
int main(int argc, char\* argv[]) {  
 std::string usage\_instructions = "Usage instructions: ";  
 usage\_instructions += argv[0];  
 usage\_instructions += " input output";  
 if (argc != 3) {  
 std::cerr << usage\_instructions << std::endl;  
 exit(EXIT\_FAILURE);  
 }  
  
 std::ifstream ifs(std::string(argv[1]).c\_str(), std::ifstream::in);  
 std::ofstream ofs(std::string(argv[2]).c\_str(), std::ofstream::out);  
  
 if (!ifs.is\_open()) {  
 std::cerr << "Failed to open input file!" << std::endl;  
 exit(EXIT\_FAILURE);  
 }  
  
 if (!ofs.is\_open()) {  
 std::cerr << "Failed to open output file!" << std::endl;  
 exit(EXIT\_FAILURE);  
 }  
 std::vector<MeasurementPackage> m\_hist;  
 std::vector<GroundTruthPackage> gt\_hist;  
 std::string line;  
 // prepare the measurement packages (each line represents a measurement  
 // at a timestamp)  
 while (std::getline(ifs, line)) {  
 std::string sensor\_type;  
 MeasurementPackage m\_pkg;  
 GroundTruthPackage gt\_pkg;  
 std::istringstream iss(line);  
 long long timestamp;  
 // reads first element from the current line  
 iss >> sensor\_type;  
 if (sensor\_type == "L") {  
 // LIDAR measurement  
 m\_pkg.sensor\_type = MeasurementPackage::LIDAR;  
 double x;  
 double y;  
 iss >> x;  
 iss >> y;  
 m\_pkg.values = Eigen::VectorXd(2);  
 m\_pkg.values << x, y;  
 } else if (sensor\_type == "R") {  
 // RADAR measurement  
 m\_pkg.sensor\_type = MeasurementPackage::RADAR;  
 double rho;  
 double phi;  
 double v\_rho;  
 iss >> rho;  
 iss >> phi;  
 iss >> v\_rho;  
  
 // Normalize the angle to (-pi, pi]  
 phi = utilities::normalizeAngle(phi);  
 m\_pkg.values = Eigen::VectorXd(3);  
 m\_pkg.values << rho, phi, v\_rho;  
 } else {  
 std::cerr << "Unknown sensor type: " << m\_pkg.sensor\_type << std::endl;  
 exit(EXIT\_FAILURE);  
 }  
 // read the timestamp for both LIDAR and RADAR  
 iss >> timestamp;  
 m\_pkg.timestamp = timestamp;  
 m\_hist.push\_back(m\_pkg);  
 // read ground truth data to compare later  
 double x\_gt;  
 double y\_gt;  
 double vx\_gt;  
 double vy\_gt;  
 iss >> x\_gt;  
 iss >> y\_gt;  
 iss >> vx\_gt;  
 iss >> vy\_gt;  
 gt\_pkg.values = Eigen::VectorXd(4);  
 gt\_pkg.values << x\_gt, y\_gt, vx\_gt, vy\_gt;  
 gt\_hist.push\_back(gt\_pkg);  
 }  
  
 // process data  
 FusionEKF fusion\_ekf;  
 std::vector<Eigen::VectorXd> ret; // estimated values  
 std::vector<Eigen::VectorXd> gt; // ground truth values  
 // column names in the output file  
 ofs <<"sensor\_type\tpx\tpy\tvx\tvy\tpx\_m\tpy\_m\tpx\_gt\tpy\_gt\tvx\_gt\tvy\_gt\ttimestamp\n";  
 for (std::size\_t k = 0; k < m\_hist.size(); ++k) {  
 fusion\_ekf.processMeasurement(m\_hist[k]);  
 // dump the sensor type  
 if (m\_hist[k].sensor\_type == MeasurementPackage::LIDAR) {  
 ofs << "L\t";  
 } else if (m\_hist[k].sensor\_type == MeasurementPackage::RADAR) {  
 ofs << "R\t";  
 }  
 // dump the estimations  
 ofs << fusion\_ekf.ekf\_.x\_(0) << "\t";  
 ofs << fusion\_ekf.ekf\_.x\_(1) << "\t";  
 ofs << fusion\_ekf.ekf\_.x\_(2) << "\t";  
 ofs << fusion\_ekf.ekf\_.x\_(3) << "\t";  
 // dump the measurements  
 if (m\_hist[k].sensor\_type == MeasurementPackage::LIDAR) {  
 // output the estimation  
 ofs << m\_hist[k].values(0) << "\t";  
 ofs << m\_hist[k].values(1) << "\t";  
 } else if (m\_hist[k].sensor\_type == MeasurementPackage::RADAR) {  
 // output the estimation in the cartesian coordinates  
 double rho = m\_hist[k].values(0);  
 double phi = m\_hist[k].values(1);  
 ofs << rho \* std::cos(phi) << "\t"; // p1\_meas  
 ofs << rho \* std::sin(phi) << "\t"; // ps\_meas  
 }  
 // dump the ground truth values  
 ofs << gt\_hist[k].values(0) << "\t";  
 ofs << gt\_hist[k].values(1) << "\t";  
 ofs << gt\_hist[k].values(2) << "\t";  
 ofs << gt\_hist[k].values(3) << "\t";  
 // dump the timestamp  
 ofs << m\_hist[k].timestamp << "\n";  
 ret.push\_back(fusion\_ekf.ekf\_.x\_);  
 gt.push\_back(gt\_hist[k].values);  
 }  
 Eigen::IOFormat CommaInitFmt(Eigen::StreamPrecision, Eigen::DontAlignCols,  
 ", ", ", ", "", "", " << ", "");  
 std::cout << "RMSE" << calculateRMSE(ret, gt).format(CommaInitFmt) << std::endl;  
 // close files  
 if (ofs.is\_open()) ofs.close();  
 if (ifs.is\_open()) ifs.close();  
 return 0;  
}

## Kalman\_filter\_cpp

#include "kalman\_filter.h"  
#include "utilities.h"  
void KalmanFilter::predict() {  
 x\_ = f\_ \* x\_;  
 Eigen::MatrixXd f\_t = f\_.transpose();  
 p\_ = f\_ \* p\_ \* f\_t + q\_;  
}  
void KalmanFilter::update(const Eigen::VectorXd &z, const Eigen::MatrixXd &r) {  
 predict();  
 // Measurement update  
 Eigen::MatrixXd k;  
 k = updateKalmanGain(p\_, h\_, r);  
 x\_ = x\_ + k \* (z - h\_ \* x\_);  
 long x\_size = x\_.size();  
 Eigen::MatrixXd i = Eigen::MatrixXd::Identity(x\_size, x\_size);  
 p\_ = (i - k \* h\_) \* p\_;  
}  
  
void KalmanFilter::updateEKF(const Eigen::VectorXd &z, const Eigen::MatrixXd &r) {  
 predict();  
 // Measurement update  
 Eigen::MatrixXd h\_j = utilities::calculateJacobian(x\_);  
 Eigen::MatrixXd k;  
 k = updateKalmanGain(p\_, h\_j, r);  
 Eigen::VectorXd x\_polar(4);  
 x\_polar = utilities::cartesian2Polar(x\_);  
 x\_ = x\_ + k \* (z - x\_polar);  
 long x\_size = x\_.size();  
 Eigen::MatrixXd i = Eigen::MatrixXd::Identity(x\_size, x\_size);  
 p\_ = (i - k \* h\_j) \* p\_;  
}  
Eigen::MatrixXd KalmanFilter::updateKalmanGain(const Eigen::MatrixXd &p,  
 const Eigen::MatrixXd &h,  
 const Eigen::MatrixXd &r) {  
 Eigen::MatrixXd h\_t = h.transpose();  
 Eigen::MatrixXd ph\_t = p \* h\_t;  
 Eigen::MatrixXd s = h \* ph\_t + r;  
 Eigen::MatrixXd s\_i = s.inverse();  
 Eigen::MatrixXd k = ph\_t \* s\_i;  
 return k;  
}

## Fusion\_ekf\_cpp

#include <iostream>  
#include "fusion\_ekf.h"  
#include "utilities.h"  
FusionEKF::FusionEKF() {  
 is\_initialized\_ = false;  
 time\_us\_ = 0;  
 use\_radar\_ = true;  
 use\_lidar\_ = true;  
 //state covariance matrix  
 ekf\_.p\_ = Eigen::MatrixXd(4, 4);  
 ekf\_.p\_ << 1, 0, 0, 0,  
 0, 1, 0, 0,  
 0, 0, 1, 0,  
 0, 0, 0, 1;  
 //the initial transition matrix F\_  
 ekf\_.f\_ = Eigen::MatrixXd(4, 4);  
 ekf\_.f\_ << 1, 0, 1, 0,  
 0, 1, 0, 1,  
 0, 0, 1, 0,  
 0, 0, 0, 1;  
 //measurement matrix for the basic Kalman fiter  
 ekf\_.h\_ = Eigen::MatrixXd(2, 4);  
 ekf\_.h\_ << 1, 0, 0, 0,  
 0, 1, 0, 0;  
 //measurement covariance matrices  
 r\_lidar\_ = Eigen::MatrixXd(2, 2);  
 r\_lidar\_ << 0.0225, 0,  
 0, 0.0225;  
 r\_radar\_ = Eigen::MatrixXd(3, 3);  
 r\_radar\_ << 0.09, 0, 0,  
 0, 0.0009, 0,  
 0, 0, 0.09;  
 //set the acceleration noise components  
 noise\_ax\_ = 9.0;  
 noise\_ay\_ = 9.0;  
}  
  
FusionEKF::~FusionEKF() = default;  
void FusionEKF::processMeasurement(const MeasurementPackage &measurement\_pack) {  
 // Initialization  
 if (!is\_initialized\_) {  
 ekf\_.x\_ = Eigen::VectorXd(4);  
 if (measurement\_pack.sensor\_type == MeasurementPackage::RADAR) {  
 Eigen::VectorXd x\_polar(3);  
 x\_polar << measurement\_pack.values[0],  
 measurement\_pack.values[1],  
 measurement\_pack.values[2];  
 // Convert positions and velocities from polar to cartesian coordinates.  
 ekf\_.x\_ = utilities::polar2Cartesian(x\_polar);  
 } else if (measurement\_pack.sensor\_type == MeasurementPackage::LIDAR) {  
 ekf\_.x\_ << measurement\_pack.values[0],measurement\_pack.values[1],0, 0;  
 } else {  
 std::cerr << "Unknown sensor\_type: " << measurement\_pack.sensor\_type << std::endl;  
 exit(EXIT\_FAILURE);  
 }  
 time\_us\_ = measurement\_pack.timestamp;  
 is\_initialized\_ = true;  
 return;  
 }  
  
 // Compute the time elapsed between the current and previous measurements  
 double dt = (measurement\_pack.timestamp - time\_us\_) / 1000000.0; // In second  
 // Update the state transition matrix f according to the new elapsed time.  
 ekf\_.f\_(0, 2) = dt;  
 ekf\_.f\_(1, 3) = dt;  
 // Update the process noise covariance matrix.  
 double dt\_2 = dt \* dt;  
 double dt\_3 = dt\_2 \* dt;  
 double dt\_4 = dt\_3 \* dt;  
 ekf\_.q\_ = Eigen::MatrixXd(4, 4);  
 ekf\_.q\_ << 0.25 \* dt\_4 \* noise\_ax\_, 0, 0.5 \* dt\_3 \* noise\_ax\_, 0,

0, 0.25 \* dt\_4 \* noise\_ay\_, 0, 0.5 \* dt\_3 \* noise\_ay\_,

0.5 \* dt\_3 \* noise\_ax\_, 0, dt\_2 \* noise\_ax\_, 0,

0, 0.5 \* dt\_3 \* noise\_ay\_, 0, dt\_2 \* noise\_ay\_;

// Prediction and Measurement updating  
 if (measurement\_pack.sensor\_type == MeasurementPackage::RADAR && use\_radar\_) {  
 ekf\_.updateEKF(measurement\_pack.values, r\_radar\_);  
 } else if (measurement\_pack.sensor\_type == MeasurementPackage::LIDAR && use\_lidar\_) { ekf\_.update(measurement\_pack.values, r\_lidar\_);  
 } else return;  
 // Update time stamp  
 time\_us\_ = measurement\_pack.timestamp;  
}

## Fusion\_ekf\_h

#ifndef FusionEKF\_H\_  
#define FusionEKF\_H\_  
#include <Eigen/Dense>  
#include "measurement\_package.h"  
#include "kalman\_filter.h"  
class FusionEKF {  
 // Check whether the tracking toolbox was initiallized or not  
 // (first measurement)  
 bool is\_initialized\_;  
 // time when the state is true, in us  
 long long time\_us\_;  
 // Acceleration noise components  
 double noise\_ax\_;  
 double noise\_ay\_;  
 // if this is false, lidar measurements will be ignored (except for init)  
 bool use\_lidar\_;  
 // if this is false, radar measurements will be ignored (except for init)  
 bool use\_radar\_;  
public:  
 FusionEKF();  
 ~FusionEKF();  
 KalmanFilter ekf\_;  
 // Run the whole flow of the Kalman Filter from here.//  
 void processMeasurement(const MeasurementPackage &measurement\_pack);  
 // Measurement covariance matrices for LIDAR and RADAR.  
 Eigen::MatrixXd r\_lidar\_;  
 Eigen::MatrixXd r\_radar\_;  
};  
#endif /\* FusionEKF\_H\_ \*/

## Ground\_truth\_package\_h

#ifndef GROUND\_TRUTH\_PACKAGE\_H\_  
#define GROUND\_TRUTH\_PACKAGE\_H\_  
#include <Eigen/Dense>  
class GroundTruthPackage {  
public:  
 long long timestamp;  
 enum SensorType{  
 LIDAR,  
 RADAR  
 } sensor\_type;  
 Eigen::VectorXd values;  
};  
#endif /\* GROUND\_TRUTH\_PACKAGE\_H\_ \*/

## Kalman\_filter\_h

#ifndef KALMAN\_FILTER\_H\_  
#define KALMAN\_FILTER\_H\_  
#include <Eigen/Dense>  
class KalmanFilter {  
public:  
 // TODO: this is a pool desired and only used for education  
 // state vector  
 Eigen::VectorXd x\_;  
 // state covariance matrix  
 Eigen::MatrixXd p\_;  
 // state transition matrix  
 Eigen::MatrixXd f\_;  
 // process covariance matrix  
 Eigen::MatrixXd q\_;  
 // Linear measurement matrix  
 Eigen::MatrixXd h\_;  
  
private:  
 // Predict the state and the state covariance using the process model  
 void predict();  
 // Update the Kalman gain k. Used in both KF() and EKF()  
 // @param: p: priori error covariance matrix  
 // @param: h: measurement matrix (or the Jacobian)  
 // @param: r: measurement covariance matrix  
 Eigen::MatrixXd updateKalmanGain(  
 const Eigen::MatrixXd &p, const Eigen::MatrixXd &h,  
 const Eigen::MatrixXd &r);  
public:  
 KalmanFilter() = default;  
 ~KalmanFilter() = default;  
 // A full cycle (prediction + measurement update) using  
 // standard Kalman Filter equations  
 // @param z: measurement  
 // @param r: measurement covariance matrix  
 void update(const Eigen::VectorXd &z, const Eigen::MatrixXd &r);  
  
 // A full cycle (prediction + measurement update) using  
 // extended Kalman Filter equations  
 // @param z: measurement  
 // @param r: measurement covariance matrix  
 //  
 void updateEKF(const Eigen::VectorXd &z, const Eigen::MatrixXd &r);  
};  
#endif /\* KALMAN\_FILTER\_H\_ \*/

## Measurement\_package\_h

#ifndef MEASUREMENT\_PACKAGE\_H\_  
#define MEASUREMENT\_PACKAGE\_H\_  
#include <Eigen/Dense>  
class MeasurementPackage {  
public:  
 long long timestamp;  
 enum SensorType{  
 LIDAR,  
 RADAR  
 } sensor\_type;  
 Eigen::VectorXd values;  
};  
#endif /\* MEASUREMENT\_PACKAGE\_H\_ \*/

## Utilities\_h

#ifndef UTILITIES\_H\_  
#define UTILITIES\_H\_  
#include <vector>  
#include <Eigen/Dense>  
namespace utilities {  
const double kPI = std::atan(1.0)\*4;  
  
// Convert Cartesian coordinates to polar coordinates.  
inline Eigen::VectorXd cartesian2Polar(const Eigen::VectorXd &x) {  
 Eigen::VectorXd x\_polar(3);  
 double px = x[0];  
 double py = x[1];  
 double rho = std::sqrt(px\*px + py\*py);  
 double phi = std::atan2(py, px);  
 if (rho < 1e-6 ) rho = 1e-6;  
 double v\_rho = (px\*x[2] + py\*x[3])/rho;  
 x\_polar << rho, phi, v\_rho;  
 return x\_polar;  
}  
  
// Convert polar coordinates to Cartesian coordinates.  
// Only used in the initialization of FusionEKF().  
inline Eigen::VectorXd polar2Cartesian(const Eigen::VectorXd &x) {  
 Eigen::VectorXd x\_cartesian(4);  
 double c = std::cos(x[1]);  
 double s = std::sin(x[1]);  
 double px = x[0] \* c;  
 double py = x[0] \* s;  
 double vx = x[2] \* c;  
 double vy = x[2] \* s;  
 x\_cartesian << px, py, vx, vy;  
 return x\_cartesian;  
}  
  
// Calculate Jacobian.  
inline Eigen::MatrixXd calculateJacobian(const Eigen::VectorXd &x) {  
 double px = x[0];  
 double py = x[1];  
 double vx = x[2];  
 double vy = x[3];  
 double c1 = px\*px + py\*py;  
 if (c1 < 1e-12 ) c1 = 1e-12;  
 double c2 = std::sqrt(c1);  
 double c3 = c1\*c2;  
 Eigen::MatrixXd h\_j(3, 4);  
  
 h\_j << px/c2, py/c2, 0, 0,  
 -py/c1, px/c1, 0, 0,  
 py\*(py\*vx - px\*vy)/c3, px\*(px\*vy - py\*vx)/c3, px/c2, py/c2;  
 return h\_j;  
}  
// Normalize the angle to [-pi, pi]  
inline double normalizeAngle(double phi) {  
 if (phi < -1.0\*kPI || phi > kPI) {  
 double s = std::sin(phi);  
 phi = std::asin(s);  
 }  
 return phi;  
}  
} // utilities  
#endif /\* UTILITIES\_H\_ \*/

## CMakeLists\_txt

cmake\_minimum\_required (VERSION 3.1)

project(ExtendedKF LANGUAGES CXX)

set(CMAKE\_CXX\_STANDARD 11)

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

set(CMAKE\_CXX\_EXTENSIONS OFF)

find\_package(Eigen3 3.3 REQUIRED CONFIG)

if(Eigen3\_FOUND)

message(STATUS "Found Eigen3: ${Eigen3\_VERSION}, ${Eigen3\_INCLUDE\_DIRS}")

endif()

set(sources

src/fusion\_ekf.cpp

src/kalman\_filter.cpp

src/main.cpp)

add\_executable(ekf ${sources})

target\_include\_directories(ekf PRIVATE ${PROJECT\_SOURCE\_DIR}/include)

target\_link\_libraries(ekf PRIVATE Eigen3::Eigen)

# 参考文献

[1] <https://blog.csdn.net/Young_Gy/article/details/78468153>

[2] <https://github.com/zyqdragon/Lidar-and-Radar-sensor-fusion-with-Extended-Kalman-Filter> 数据和matlab代码

[3]<https://github.com/zhujun98/sensor-fusion> 数据及c++代码

[4]<https://github.com/mithi/fusion-ukf> 无迹卡尔曼滤波