无人驾驶技术入门（十八）
手把手教你写扩展卡尔曼滤波器

陈 光

《无人驾驶技术入门（十三）| 手把手教你写卡尔曼滤波器》以一个匀速运动小车的模型为例，让读者从感性上认识了卡尔曼滤波器的基本原理，它包含预测(Prediction)和测量值更新(Measurement update)两大过程。预测和测量值更新的交替执行，实现了卡尔曼滤波在状态估计中的闭环。

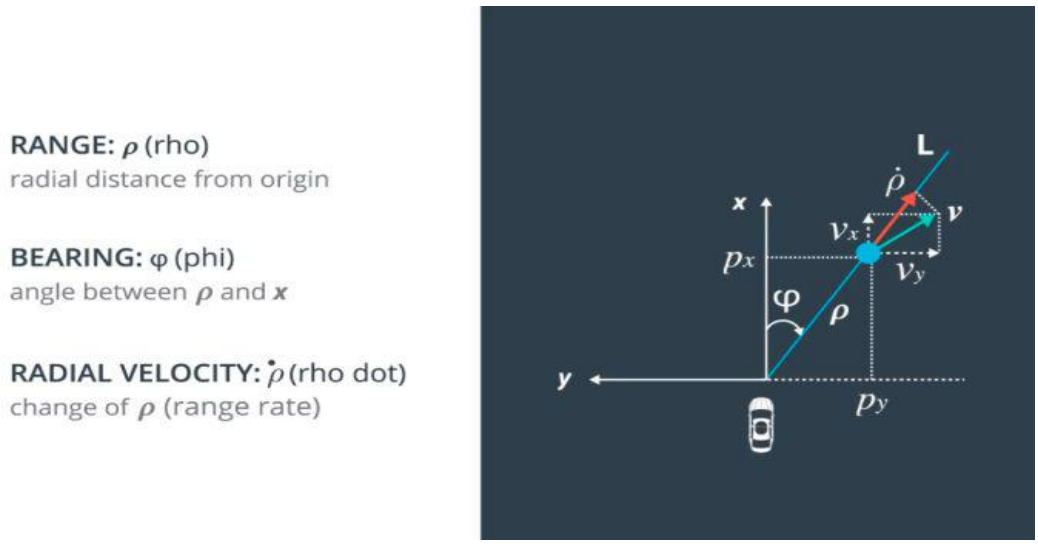
随后，我从理性分析的角度，以无人驾驶中激光雷达测量障碍物位置的数据为例，结合卡尔曼滤波所用到的公式，使用 C++和矩阵运算库 Eigen 一步步实现了卡尔曼滤波器预测和测量值更新这两大过程的代码。

在本次分享中，我将以优达学城(Udacity)无人驾驶工程师学位中提供的传感器融合项目为例，介绍如何利用扩展卡尔曼滤波解决跟踪毫米波雷达目标的问题。由于本次内容为卡尔曼滤波器的进阶内容，推荐读者先阅读《无人驾驶技术入门（十三）| 手把手教你写卡尔曼滤波器》，了解预测和测量值更新这两个步骤所使用到的公式及其含义，再看此篇。

1 毫米波雷达的数据

毫米波雷达观察世界的方式与激光雷达有所不同。激光雷达测量的原理是光的直线传播，因此在测量时能直接获得障碍物在笛卡尔坐标系下 x 方向、y 方向和 z 方向上的距离；而毫米波雷达的原理是多普勒效应，它所测量的数据都是在极坐标系下的。

如下图所示，毫米波雷达能够测量障碍物在极坐标下离雷达的距离ρ、方向角ϕ以及距离的变化率（径向速度）ρ'，如下图所示。

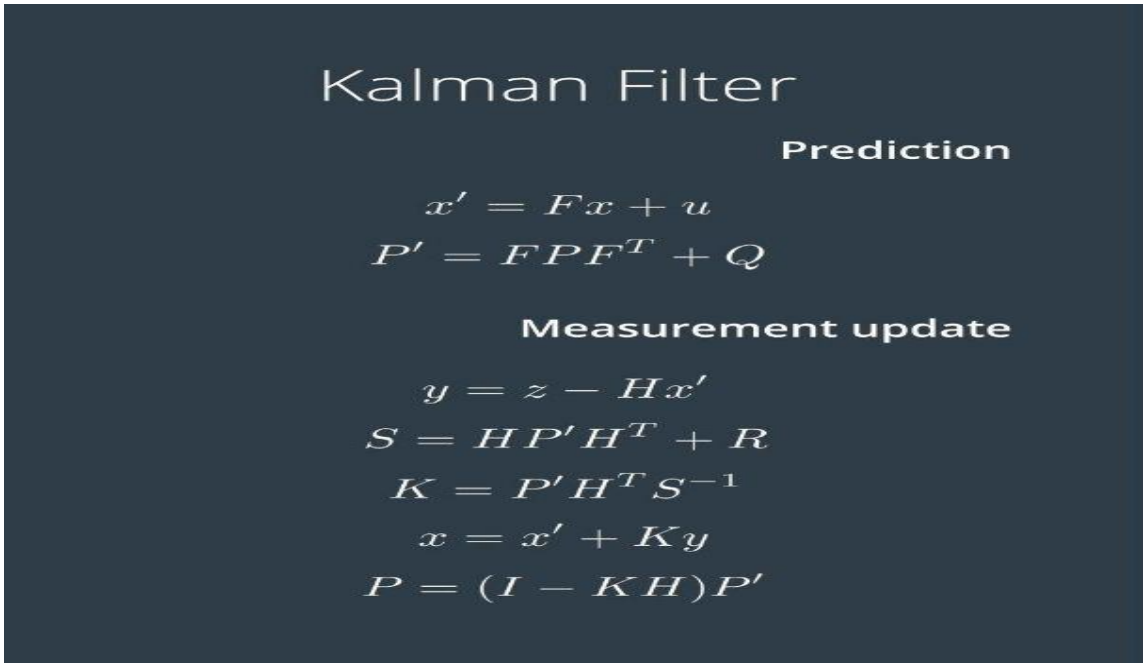


图片出处：优达学城(Udacity)无人驾驶工程师学位

毫米波雷达的测量原理及更为详细的数据介绍可参看《无人驾驶技术入门（七）｜ 量产必备的毫米波雷达》。

2 扩展卡尔曼滤波器理论

我们再次祭出卡尔曼老先生给我们留下的宝贵财富，即状态估计时预测和测量值更新时所用到的 7 个公式，如下图所示。扩展卡尔曼滤波的理论和编程依旧需要使用到这些公式，相比于原生的卡尔曼滤波，只在个别地方有所不同。



图片出处：优达学城(Udacity)无人驾驶工程师学位

完成扩展卡尔曼滤波器 C++代码的过程，实际上就是结合上面的公式，一步步完成初始化、预测、观测的过程。由于公式中涉及大量的矩阵转置和求逆运算，我们使用开源的矩阵运算库 Eigen 辅助我们代码的编写。

2.1 代码：初始化（Initialization）

扩展卡尔曼滤波的初始化，需要将各个变量进行设置，对于不同的运动模型，状态向量是不一样的。为了保证代码对不同状态向量的兼容性，我们使用 Eigen 库中非定长的数据结构。

如下所示，我们新建了一个 ExtendedKalmanFilter 类，定义了一个叫做 x_的状态向量(state vector)。代码中的 VerctorXd 表示 X 维的列向量，元素的数据类型为 double。

```
#pragma once

// @file: extended_kalman_filter.h
// @brief: EKF algorithm for tracking radar obstacle

#include "Eigen/Dense"

class ExtendedKalmanFilter {
public:
    // Constructor
    ExtendedKalmanFilter() {
        is_initialized_ = false;
    }

    // Destructor
    ~ExtendedKalmanFilter() {}

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

private:
    // flag of initialization
    bool is_initialized_;

    // state vector
    Eigen::VectorXd x_;
}
```

初始化扩展卡尔曼滤波器时需要输入一个初始的状态量 x_in，用以表示障碍物最初的位置和速度信息，一般直接使用第一次的测量结果。

2.2 代码：预测（Prediction）

完成初始化后，我们开始写 Prediction 部分的代码。首先是公式：

$$x' = Fx + u$$

这里的 x 为状态向量，通过左乘一个矩阵 F，再加上外部的影响 u，得到预测的状态向量 x'。这里的 F 被称作状态转移矩阵(state transistion matrix)。以 2 维的匀速运动为例，这里的 x 为：

$$x = \begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \end{bmatrix}$$

根据中学物理课本中的公式 s1 = s0 + vt，经过时间△t 后的预测状态向量应该是：

$$x' = \begin{bmatrix} p_x + v_x * \Delta t \\ p_y + v_y * \Delta t \\ v_x \\ v_y \end{bmatrix} + u$$

对于二维的匀速运动模型，加速度为 0，并不会对预测后的状态造成影响，因此：

$$u = 0$$

如果换成加速或减速运动模型，可以引入加速度 ax 和 ay，根据 s1 = s0 + vt + at^2/2 这里的 u 会变成：

$$u = \begin{bmatrix} \frac{1}{2} a_x (\Delta t)^2 \\ \frac{1}{2} a_y (\Delta t)^2 \\ a_x * \Delta t \\ a_y * \Delta t \end{bmatrix}$$

作为入门课程，这里不讨论太复杂的模型，因此公式：

$$x' = Fx + u$$

最终将写成：

$$\begin{bmatrix} p_x + v_x * \Delta t \\ p_y + v_y * \Delta t \\ v_x \\ v_y \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

再看预测模块的第二个公式：

$$P' = FPF^T + Q$$

该公式中 P 表示系统的不确定程度，这个不确定程度，在卡尔曼滤波器初始化时会很大，随着越来越多的数据注入滤波器中，不确定程度会变小，P 的专业术语叫状态协方差矩阵（state covariance matrix）；这里的 Q 表示过程噪声（process covariance matrix），即无法用 x'=Fx+u 表示的噪声，比如车辆运动时突然到了上坡，这个影响是无法用之前的状态转移方程估计的。

毫米波雷达测量障碍物在径向上的位置和速度相对准确，不确定度较低，因此可以对状态协方差矩阵 P 进行如下初始化：

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

由于 Q 对整个系统存在影响，但又不能太确定对系统的影响有多大。对于简单的模型来说，这里可以直接使用单位矩阵或空值进行计算，即：

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

根据以上内容和公式：

$$x' = Fx + u$$

$$P' = FPF^T + Q$$

我们就可以写出预测模块的代码了。

```
22      void SetF(Eigen::MatrixXd F_in) {
23          F_ = F_in;
24      }
25
26      void SetP(Eigen::MatrixXd P_in) {
27          P_ = P_in;
28      }
29
30      void SetQ(Eigen::MatrixXd Q_in) {
31          Q_ = Q_in;
32      }
33
34      void Prediction() {
35          x_ = F_ * x_;
36          Eigen::MatrixXd Ft = F_.transpose();
37          P_ = F_ * P_ * Ft + Q_;
38      }
39
40  private:
41      // flag of initialization
42      bool is_initialized_;
43
44      // state vector
45      Eigen::VectorXd x_;
46
47      // state transistion matrix
48      Eigen::MatrixXd F_;
49
50      // state covariance matrix
51      Eigen::MatrixXd P_;
52
53      // process covariance matrix
54      Eigen::MatrixXd Q_;
55  }
```

实际编程时 x'及 P'不需要申请新的内存去存储，使用原有的 x 和 P 代替即可。

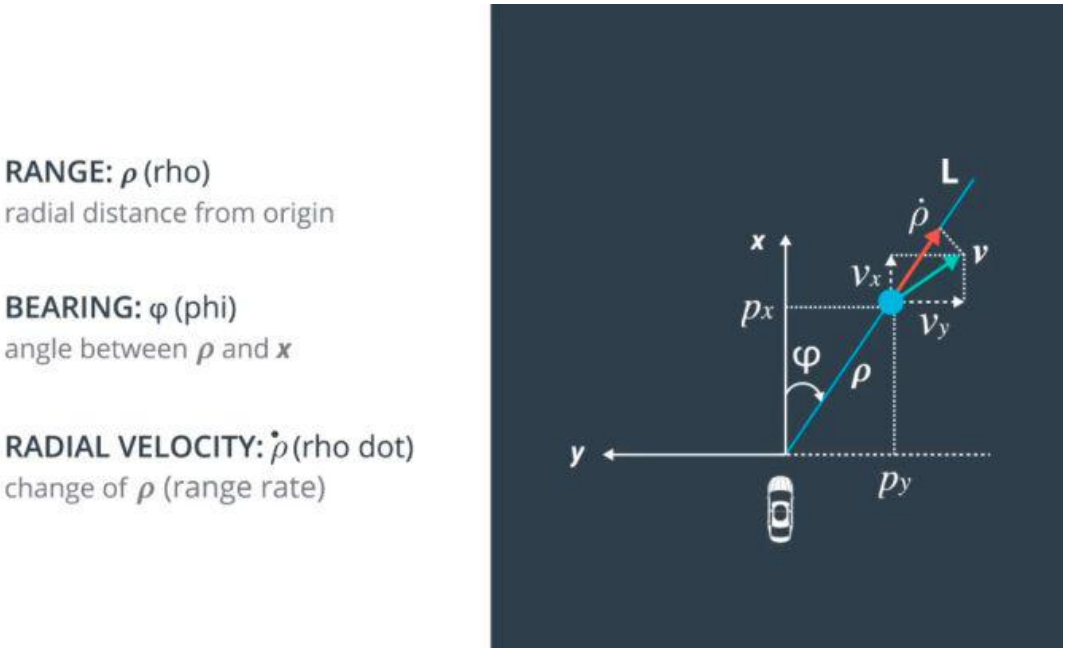
2.3 代码：观测（Measurement）

观测的第一个公式是：

$$y = z - Hx'$$

这个公式的目的是计算观测到的观测值 z 与预测值 x'之间差值 y。

前面提到过毫米波雷达观测值 z 的数据特性，如下图所示：



图片出处：优达学城(Udacity)无人驾驶工程师学位

由图可知观测值 z 的数据维度为 3×1，为了能够实现矩阵运算，y 和 Hx'的数据维度也都为 3×1。

使用基本的数学运算可以完成预测值 x'从笛卡尔坐标系到极坐标系的坐标转换，求得 Hx'，再与观测值 z 进行计算。需要注意的是，在计算差值 y 的这一步中，我们通过坐标转换的方式避开了未知的旋转矩阵 H 的计算，直接得到了 Hx'的值。

为了简化表达，我们用 px，py 以及 vx 和 vy 表示预测后的位置及速度，如下所示：

$$y = \begin{bmatrix} \rho \\ \varphi \\ \dot{\rho} \end{bmatrix} - \begin{bmatrix} \sqrt{p_x^2 + p_y^2} \\ \arctan(\frac{p_y}{p_x}) \\ \frac{p_x v_x + p_y v_y}{\sqrt{p_x^2 + p_y^2}} \end{bmatrix}$$

其中测量值 z 和预测后的位置 x'都是已知量，因此我们很容易计算得到观测与预测的差值 y。

毫米波雷达在转换时涉及到笛卡尔坐标系和极坐标系的位置、速度转换，这个转化过程是非线性的。因而在处理类似毫米波雷达这种非线性的模型时，习惯于将计算差值 y 的公式写成如下，以作线性和非线性模型的区分。

$$y = z - h(x')$$

对应上面的式子，这里的 h(x')为：

$$h(x') = \begin{bmatrix} \sqrt{p_x{}^2 + p_y{}^2} \\ \arctan(\frac{p_y}{p_x}) \\ \frac{p_x v_x + p_y v_y}{\sqrt{p_x{}^2 + p_y{}^2}} \end{bmatrix}$$

再看卡尔曼滤波器接下来的两个公式：

$$S = HP'H^T + R$$

$$K = P'H^T S^{-1}$$

这两个公式求的是卡尔曼滤波器中一个很重要的量——卡尔曼增益 K（Kalman Gain），用人话讲就是求差值 y 的权值。第一个公式中的 R 是测量噪声矩阵（measurement covariance matrix），这个表示的是测量值与真值之间的差值。一般情况下，传感器的厂家会提供。如果厂家未提供，我们也可以通过测试和调试得到。S 只是为了简化公式，写的一个临时变量，不用太在意。

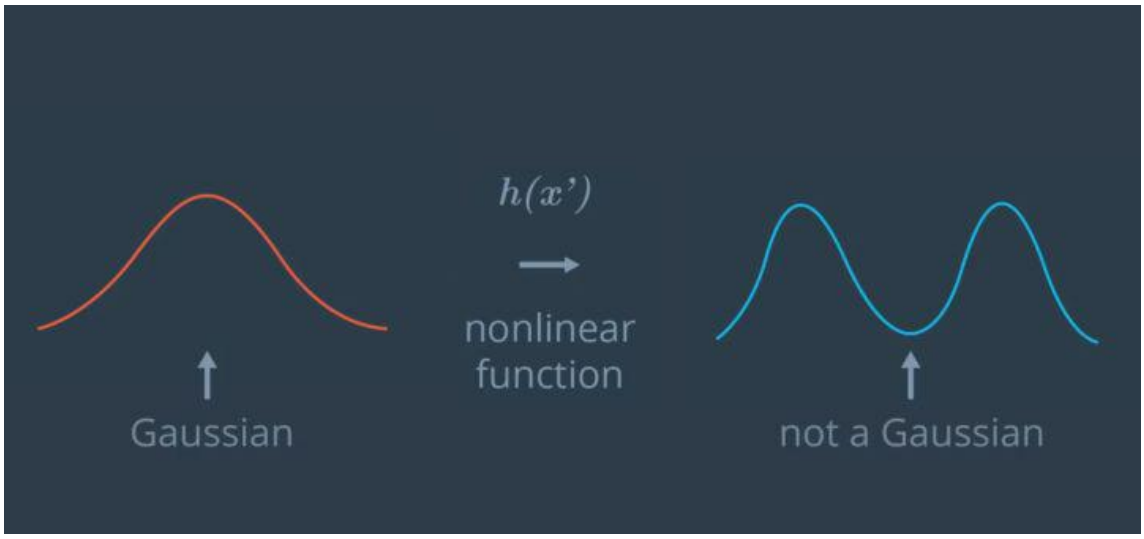
由于求得卡尔曼增益 K 需要使用到测量矩阵 H，因此接下来的任务就是得到 H。

毫米波雷达观测 z 是包含位置、角度和径向速度的 3x1 的列向量，状态向量 x'是包含位置和速度信息的 4x1 的列向量，根据公式 y=z-Hx'可知测量矩阵（Measurement Matrix）H 的维度是 3 行 4 列。即：

$$h(x') = \begin{bmatrix} \sqrt{p_x{}^2 + p_y{}^2} \\ \arctan(\frac{p_y}{p_x}) \\ \frac{p_x v_x + p_y v_y}{\sqrt{p_x{}^2 + p_y{}^2}} \end{bmatrix} = \begin{bmatrix} H_{00} & H_{01} & H_{02} & H_{03} \\ H_{10} & H_{11} & H_{12} & H_{13} \\ H_{20} & H_{21} & H_{22} & H_{23} \end{bmatrix} * \begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \end{bmatrix} = H * x'$$

从上面的公式很容易看出，等式两边的转化是非线性的，并不存在一个常数矩阵 H，能够使得等式两边成立。

如果将高斯分布作为输入，输入到一个非线性函数中，得到的结果将不再符合高斯分布，也就将导致卡尔曼滤波器的公式不再适用。因此我们需要将上面的非线性函数转化为近似的线性函数求解。



图片出处：优达学城(Udacity)无人驾驶工程师学位

在大学课程《高等数学》中我们学过，非线性函数 y=h(x)可通过泰勒公式在点(x0,y0)处展开为泰勒级数：

$$h(x) = h(x_0) + \frac{\dot{h}(x_0)}{1!}(x - x_0) + \frac{\ddot{h}(x_0)}{2!}(x - x_0)^2 + ..$$

忽略二次以上的高阶项，即可得到近似的线性化方程，用以替代非线性函数 h(x)，即：

$$h(x) \approx h(x_0) + \dot{h}(x_0)(x - x_0)$$

将非线性函数 h(x)拓展到多维，即求各个变量的偏导数，即：

$$h(x) \approx h(x_0) + \frac{\partial h(x_0)}{\partial x}(x - x_0)$$

对 x 求偏导数所对应的这一项被称为雅可比(Jacobian)式。

我们将求偏导数的公式与我们的之前推导的公式对应起来看 x 的系数，会发现这里的测量矩阵 H 其实就是泰勒公式中的雅可比式。

$$h(x) \approx h(x_0) + \frac{\partial h(x_0)}{\partial x}(x - x_0)$$

$$H_{jacobian}$$

多维的雅可比式的推导过程有兴趣的同学可以自己研究一下，这里我们直接使用其结论：

$$H_j = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \cdots & \frac{\partial h_1}{\partial x_n} \\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} & \cdots & \frac{\partial h_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \cdots \\ \frac{\partial h_m}{\partial x_1} & \frac{\partial h_m}{\partial x_2} & \cdots & \frac{\partial h_m}{\partial x_n} \end{bmatrix}$$

求得非线性函数 h(x')对 px，py，vx，vy 的一阶偏导数，并排列成的矩阵，最终得到雅克比(Jacobian)矩阵 H：

$$H = \begin{bmatrix} H_{00} & H_{01} & H_{02} & H_{03} \\ H_{10} & H_{11} & H_{12} & H_{13} \\ H_{20} & H_{21} & H_{22} & H_{23} \end{bmatrix} = \begin{bmatrix} \frac{\partial \rho}{\partial p_x} & \frac{\partial \rho}{\partial p_y} & \frac{\partial \rho}{\partial v_x} & \frac{\partial \rho}{\partial v_y} \\ \frac{\partial \varphi}{\partial p_x} & \frac{\partial \varphi}{\partial p_y} & \frac{\partial \varphi}{\partial v_x} & \frac{\partial \varphi}{\partial v_y} \\ \frac{\partial \dot{\rho}}{\partial p_x} & \frac{\partial \dot{\rho}}{\partial p_y} & \frac{\partial \dot{\rho}}{\partial v_x} & \frac{\partial \dot{\rho}}{\partial v_y} \end{bmatrix}$$

其中。

$$\begin{bmatrix}\rho\\ \varphi\\ \dot{\rho}\end{bmatrix}=\begin{bmatrix}\sqrt{{p_x}^2+{p_y}^2}\\ \arctan(\frac{p_y}{p_x})\\ \frac{p_x v_x+p_y v_y}{\sqrt{{p_x}^2+{p_y}^2}}\end{bmatrix}$$

接下来就是考验各位高等数学求偏导数功底的时候了！

经过一系列计算，最终得到测量矩阵 H 为：

$$H=\begin{bmatrix}\frac{p_x}{\sqrt{{p_x}^2+{p_y}^2}} & \frac{p_y}{\sqrt{{p_x}^2+{p_y}^2}} & 0 & 0\\ -\frac{p_y}{{p_x}^2+{p_y}^2} & \frac{p_x}{{p_x}^2+{p_y}^2} & 0 & 0\\ \frac{p_y(v_x p_y-v_y p_x)}{({p_x}^2+{p_y}^2)^{3/2}} & \frac{p_x(v_y p_x-v_x p_y)}{({p_x}^2+{p_y}^2)^{3/2}} & \frac{p_x}{\sqrt{{p_x}^2+{p_y}^2}} & \frac{p_y}{\sqrt{{p_x}^2+{p_y}^2}}\end{bmatrix}$$

根据以上公式可知，在每次预测障碍物的状态后，需要根据预测的位置和速度计算出对应的测量矩阵 H，这个测量矩阵为非线性函数 h(x')在 x'所在位置进行求导的结果。

再看卡尔曼滤波器的最后两个公式：

$$x=x'+Ky$$

$$P=(I-KH)P'$$

这两个公式，实际上完成了卡尔曼滤波器的闭环，第一个公式是完成了当前状态向量 x 的更新，不仅考虑了上一时刻的预测值，也考虑了测量值，和整个系统的噪声，第二个公式根据卡尔曼增益 K，更新了系统的不确定度 P，用于下一个周期的运算，该公式中的 I 为与状态向量同维度的单位矩阵。

完成以上推导后，我们将推导的过程写成代码，如下所示：

```
40      void CalculateJacobianMatrix() {
41          MatrixXd Hj(3,4);
42          // get state parameters
43          float px = x_(0);
44          float py = x_(1);
45          float vx = x_(2);
46          float vy = x_(3);
47          // pre-compute a set of terms to avoid repeated calculation
48          float c1 = px * px + py * py;
49          float c2 = sqrt(c1);
50          float c3 = (c1 * c2);
51          // check division by zero
52          if (fabs(c1) < 0.0001) {
53              H_ = Hj;
54              return;
55          }
56          // compute the Jacobian matrix
57          Hj << (px/c2), (py/c2), 0, 0,
58                -(py/c1), (px/c1), 0, 0,
59                py*(vx*py - vy*px)/c3, px*(px*vy - py*vx)/c3, px/c2, py/c2;
60          H_ = Hj;
61          return;
62      }
63
64      void MeasurementUpdate(const Eigen::VectorXd &z) {
65          double rho = sqrt(x_(0)*x_(0) + x_(1)*x_(1));
66          double theta = atan2(x_(1), x_(0));
67          double rho_dot = (x_(0)*x_(2) + x_(1)*x_(3)) / rho;
68          VectorXd h = VectorXd(3);
69          h << rho, theta, rho_dot;
70
71          VectorXd y = z - h;
72
73          CalculateJacobian();
74
75          MatrixXd S = H_ * P_ * H_.transpose() + R_;
76          MatrixXd K = P_ * H_.transpose() * S.inverse();
77          x_ = x_ + (K * y);
78          MatrixXd I = MatrixXd::Identity(x_.size(), x_.size());
79          P_ = (I - K * H_) * P_;
80      }
```

再补上一些必要的代码，一个扩展卡尔曼滤波器的雏形就出来了，代码如下所示：

```
1   #pragma once
2
3   // @file: extended_kalman_filter.h
4   // @brief: EKF algorithm for tracking radar obstacle
5
6   #include "Eigen/Dense"
7
8   class ExtendedKalmanFilter {
9   public:
10      // Constructor
11      ExtendedKalmanFilter() {
12          is_initialized_ = false;
13      }
14
15      // Destructor
16      ~ExtendedKalmanFilter() {}
17
18      Eigen::VectorXd GetX() {
19          return x_;
20      }
```

```
21
22      bool IsInitialized() {
23          return is_initialized_;
24      }
25
26      void Initialization(Eigen::VectorXd x_in) {
27          x_ = x_in;
28          is_initialized_ = true;
29      }
30
31      void SetF(Eigen::MatrixXd F_in) {
32          F_ = F_in;
33      }
34
35      void SetP(Eigen::MatrixXd P_in) {
36          P_ = P_in;
37      }
38
39      void SetQ(Eigen::MatrixXd Q_in) {
40          Q_ = Q_in;
41      }
42
43      void SetR(Eigen::MatrixXd R_in) {
44          Q_ = Q_in;
45      }
46
47      void Prediction() {
48          x_ = F_ * x_;
49          Eigen::MatrixXd Ft = F_.transpose();
50          P_ = F_ * P_ * Ft + Q_;
51      }
52
53      void CalculateJacobianMatrix() {
54          MatrixXd Hj(3,4);
55          // get state parameters
56          float px = x_(0);
57          float py = x_(1);
58          float vx = x_(2);
59          float vy = x_(3);
60          // pre-compute a set of terms to avoid repeated calculation
61          float c1 = px * px + py * py;
62          float c2 = sqrt(c1);
63          float c3 = (c1 * c2);
64          // check division by zero
65          if (fabs(c1) < 0.0001) {
66              H_ = Hj;
67              return;
68          }
69          // compute the Jacobian matrix
70          Hj << (px/c2), (py/c2), 0, 0,
71                -(py/c1), (px/c1), 0, 0,
72                py*(vx*py - vy*px)/c3, px*(px*vy - py*vx)/c3, px/c2, py/c2;
73          H_ = Hj;
74          return;
75      }
76
77      void MeasurementUpdate(const Eigen::VectorXd &z) {
78          double rho = sqrt(x_(0)*x_(0) + x_(1)*x_(1));
79          double theta = atan2(x_(1), x_(0));
80          double rho_dot = (x_(0)*x_(2) + x_(1)*x_(3)) / rho;
81          VectorXd h = VectorXd(3);
82          h << rho, theta, rho_dot;
83
```

```
        VectorXd y = z - h;

        CalculateJacobian();

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

private:
    // flag of initialization
    bool is_initialized_;

    // state vector
    Eigen::VectorXd x_;

    // state transistion matrix
    Eigen::MatrixXd F_;

    // state covariance matrix
    Eigen::MatrixXd P_;

    // process covariance matrix
    Eigen::MatrixXd Q_;

    // jacobian measurement matrix
    Eigen::MatrixXd H_;

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

2.4 代码：使用扩展卡尔曼滤波器

以毫米波雷达数据为例，使用以上滤波器，代码如下：

```
#include "extended_kalman_filter.h"
#include "math.h"
#include <iostream>
int main() {
    double m_rho = 0.0, m_theta = 0.0, m_rho_dot = 0.0
    double last_timestamp = 0.0, now_timestamp = 0.0;
    ExtendedKalmanFilter ekf;
    while(GetRadarData(&m_rho, &m_theta, &m_rho_dot, &now_timestamp)) {
        if(!ekf.IsInitialized()) {
            last_timestamp = now_timestamp;
            Eigen::VectorXd x_in(4, 1);
            x_in << m_rho*cos(m_theta), m_rho*sin(m_theta),
                    m_rho_dot*cos(m_theta), m_rho_dot*sin(m_theta);
            ekf.Initialization(x_in);
            // state covariance matrix
            Eigen::MatrixXd P_in(4, 4);;
            P_in << 1.0, 0.0, 0.0, 0.0,
                    0.0, 1.0, 0.0, 0.0,
                    0.0, 0.0, 10.0, 0.0,
                    0.0, 0.0, 0.0, 10.0;
            ekf.SetP(P_in);
            // process covariance matrix
            Eigen::MatrixXd Q_in(4, 4);;
            Q_in << 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;
            ekf.SetQ(Q_in);
            // measurement covariance matrix
            // R is provided by Sensor supplier, in datasheet
            Eigen::MatrixRd R_in(3, 3);;
            R_in_ << 0.09, 0, 0,
                     0, 0.0009, 0,
                     0, 0, 0.09;
            ekf.SetR(R_in);
            continue;
        }

        // state transition
        double delta_t = now_timestamp - last_timestamp;
        last_timestamp = now_timestamp;
        Eigen::MatrixXd F_in(4, 4);
        F_in << 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;
        ekf.SetF(F_in);
        ekf.Prediction();

        // measurement value
        Eigen::VectorXd z(3, 1);
        z << m_rho, m_theta, m_rho_dot;
        ekf.MeasurementUpdate(z);

        Eigen::VectorXd x_out = ekf.GetX();
        std::cout << "kalman output x : " << x_out(0) <<
                     " y : " << x_out(1) << std::endl;
    }
}
```

其中 GetRadarData 函数除了获取毫米波雷达障碍物的距离、角度和径向速度外，还获取了信息采集时刻的时间戳，用于计算前后两帧的时间差 delta_t。

以上就是使用扩展卡尔曼滤波器跟踪毫米波雷达障碍物的例子。

扩展卡尔曼(EKF)与经典卡尔曼(KF)的区别在于测量矩阵 H 的计算。EKF 对非线性函数进行泰勒展开后，进行一阶线性化的截断，忽略了其余高阶项，进而完成非线性函数的近似线性化。正是由于忽略了部分高阶项，使得 EKF 的状态估计会损失一些精度。

只要你能够运用卡尔曼滤波器的 7 个经典公式，写出模型的 F、P、Q、H、R 矩阵，任何状态跟踪的问题都将迎刃而解。

3 结语

以上就是整个扩展卡尔曼滤波器的公式推导和代码编写过程。你会发现真正进行工程开发时，除了具备基本的写代码能力外，《高等数学》和《线性代数》中的理论知识也是必不可少的。下一期我会将经典卡尔曼滤波和扩展卡尔曼滤波结合起来实现激光雷达数据和毫米波雷达数据的融合。

文中的公式推导及代码部分参考了优达学城(Udacity)无人驾驶工程师学位中介绍的内容，推荐想要了解更多细节的同学报名学习^_^我这里有一个价值¥320 的邀请码：50C3D1EA。

好了\(^o^)/~，这篇分享就到这啦。

本文原载：知乎号“陈光”，作者授权转载。

临菲信息技术港

临菲信息技术港公众号

临菲学堂