cv::KalmanFilter的基本用法

1.相关公式

对于一个动态系统，我们首先定义一组状态空间方程

状态方程：

$$x_k = A_k x_k + B_k u_k + w_k$$

这个式子是一个最终的先验估计更新方程 考虑了许多不确定性 包括

- **目标自身内部的状态更新xk**
- **控制量外部输入干扰uk**
- **以及完全不确定干扰项wk**

注意第二个x是xk-1 表示上一时刻的状态值

测量方程：

$$z_k = H_k x_k + v_k$$

其中， xk是状态向量，zk是测量向量，Ak是状态转移矩阵，uk是控制向量，Bk是控制矩阵，

wk是系统误差（噪声），Hk是测量矩阵，vk是测量误差（噪声）。wk和vk都是高斯噪声，即

$$w_k \sim N(0, Q_k)$$
$$v_k \sim N(0, R_k)$$

就是说这两个噪声符合高斯分布或者正态分布。

整个卡尔曼滤波的过程就是个递推计算的过程，不断的"预测——更新——预测——更新……"

预测

预测状态值：

$$\hat{x}_{k|k-1} = A_k \hat{x}_{k-1|k-1} + B_k u_k$$

当前的状态值 = 上一次的状态值 × 状态转移矩阵 + 控制输入矩阵 × 当次的控制输入量

预测最小均方误差：

$$P_{k|k-1} = A_k P_{k-1|k-1} A_k^T + Q_k$$

这里的 **最小均方误差就是我们所谓的 先验或者后验误差协方差矩阵**

当次的先验估计协方差 = 状态转移矩阵 × 后验估计协方差矩阵 × 状态转移矩阵转置 + 过程/状态噪声矩阵

注意这里面的**后验估计协方差矩阵就是 第k-1次（上一次）的先验估计误差协方差矩阵**

总结一下，预测部分预测两个值 上述这个预测过程可以叫做 **先验估计**

$$\hat{x}_k = F_k \hat{x}_{k-1} + B_k \overrightarrow{u}_k + w_k \quad (2)$$

$$\hat{P}_k = F_k \hat{P}_{k-1} F_k^T + Q_k \quad (3)$$

1. **状态值 由上次的状态值得到本次状态值**
2. **先验估计误差协方差 由后验估计误差协方差也即是上次的先验估计误差得到 这里面考虑过程噪声**
3. **先验估计$\hat{x}_k$取决于如下三部分：一部分是上一次的最优估计值（也就是上一轮卡尔曼滤波的结果），一部分是确定性的外界影响值，另一部分是环境当中不确定的干扰。先验估计协方差矩阵P_k，首先是依据第$k-1$次卡尔曼估计（后验估计)的协方差矩阵进行递推，再与外界在这次更新中可能对系统造成的不确定的影响求和得到。**

更新

测量误差：

$$\tilde{y}_k = z_k - H_k \hat{x}_{k|k-1}$$

测量协方差：

$$S_k = H_k P_{k|k-1} H_k^T + R_k$$

最优卡尔曼增益：

$$K_k = P_{k|k-1} H_k^T S_k^{-1}$$

修正状态值：

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k$$

修正最小均方误差：

$$P_{k|k} = (I - K_k H_k) P_{k|k-1}$$

更新部分重要的是：三个公式

计算卡尔曼增益：

$$K^{'} = P_k H_k^T (H_k P_k H_k^T + R_k)^{-1}$$

其中，

K'是当次流程里的卡尔曼增益值，Pk是当次的先验估计误差协方差矩阵，Hk为观测矩阵，Rk为观测噪声。

最后一步 计算结果

$$\hat{x}_k^{'} = \hat{x}_k + K^{'}(\overrightarrow{z_k} - H_k \hat{x}_k)$$
$$P_k^{'} = P_k - K^{'} H_k P_k$$

- **xk'就是第k次的卡尔曼预测结果，也叫后验状态值。**
- **xk是本次的先验估计值，状态值**
- **Pk'是该结果的协方差矩阵，也就是更新新的估计误差协方差矩阵，也叫后验误差协方差矩阵。**
- **这两个后验值都将作为下一次的先验估计的初始值参与到新的预测中。**
- **zk是传来的测量值 此次测量值**
- **这个第二个式子写错了，应该在Pk-K'Hk这里加括号**

总体上来讲，卡尔曼滤波的步骤大致分为两步，第一步是时间更新，也叫作先验估计，第二步是量测更新，也叫作后验估计，而当前的卡尔曼滤波过程的后验估计结果不仅可以作为本次的最终结果，还能够作为下一次的先验估计的初始值。

2.OpenCV中的KalmanFilter详解

参数讲解

OpenCV中有两个版本的卡尔曼滤波方法**KalmanFilter(C++)**和**CvKalman(C)**，用法差不太多，这里只介绍KalmanFilter。

C++版本中将KalmanFilter封装到一个类中，其结构如下所示：

```
class CV_EXPORTS_W KalmanFilter
{
public:
    CV_WRAP KalmanFilter();
                        //构造默认KalmanFilter对象
    CV_WRAP KalmanFilter(int dynamParams, int measureParams, int
controlParams=0, int type=CV_32F);  //完整构造KalmanFilter对象方法
    void init(int dynamParams, int measureParams, int controlParams=0, int
type=CV_32F);               //初始化KalmanFilter对象，会替换原来的KF对象

    CV_WRAP const Mat& predict(const Mat& control=Mat());           //计算预测的状
态值
```

```
    CV_WRAP const Mat& correct(const Mat& measurement);               //根据测量值更新状态值

    Mat statePre;            //预测值 (x'(k)): x(k)=A*x(k-1)+B*u(k)
    Mat statePost;           //状态值 (x(k)): x(k)=x'(k)+K(k)*(z(k)-H*x'(k))
    Mat transitionMatrix;    //状态转移矩阵 (A)
    Mat controlMatrix;       //控制矩阵 B
    Mat measurementMatrix;   //测量矩阵 H
    Mat processNoiseCov;     //系统误差 Q
    Mat measurementNoiseCov; //测量误差 R
    Mat errorCovPre;         //最小均方误差 (P'(k)): P'(k)=A*P(k-1)*At + Q)
    Mat gain;                //卡尔曼增益    (K(k)): K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)
    Mat errorCovPost;        //修正的最小均方误差 (P(k)): P(k)=(I-K(k)*H)*P'(k)

    // 临时矩阵
    Mat temp1;
    Mat temp2;
    Mat temp3;
    Mat temp4;
    Mat temp5;
};
enum
{
    OPTFLOW_USE_INITIAL_FLOW = CV_LKFLOW_INITIAL_GUESSES,
    OPTFLOW_LK_GET_MIN_EIGENVALS = CV_LKFLOW_GET_MIN_EIGENVALS,
    OPTFLOW_FARNEBACK_GAUSSIAN = 256
};
```

包含cv头文件之后可以直接使用cv库里的KalmanFilter类创建对象，包括其中的任何成员函数。

可以看到这个自带的卡尔曼滤波器是包含所有的kalman滤波过程所需的参数和函数的。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**

  需要注意的是，这里面的修正最小均方误差就是所谓的先验/后验误差协方差矩阵 errorCovPost

  另外重载了几种构造方法

  包含最重要的两个函数预测predict和更新correct

函数原型见：……\OpenCV2\sources\modules\ocl\src\kalman.cpp

只有四个方法：构造KF对象KalmanFilter(DP,MP,CP)、初始化KF对象init(DP,MP,CP)、预测predict( )、更新correct( )。除非你要重新构造KF对象，否则用不到init( )。

KalmanFilter(DP,MP,CP)和init( )就是赋值。

注意：KalmanFilter结构体中并没有测量值，测量值需要自己定义，而且一定要定义，因为后面要用。

编程步骤

step1：定义KalmanFilter类并初始化

```
//构造KF对象
KalmanFilter KF(DP, MP, 0);
//初始化相关参数

KF.transitionMatrix                    转移矩阵 A

KF.measurementMatrix                   测量矩阵     H

KF.processNoiseCov                     过程噪声 Q

KF.measurementNoiseCov                 测量噪声         R

KF.errorCovPost                        最小均方误差 P

KF.statePost                           系统初始状态 x(0)

Mat measurement                        定义初始测量值 z(0)
```

- 这里面转移矩阵A、测量/观测矩阵H、过程噪声Q、观测噪声R初始给定，并且一般为定值，后面不再改变。
- 最小均方误差也叫先验、后验误差协方差，这在更新过程中是会不断改变的。
- 系统的输出状态和初始测量值也要给出，一般其实可以给个随机数，因为实际中是这样不确定性的。

step2：预测

```
KF.predict( )                                                    //返回的是下一时刻的状
态值KF.statePost (k+1)
```

step3：更新

```
更新measurement;                                          //注意measurement不能通过观测方
程进行计算得到，要自己定义!

更新KF   KF.correct(measurement)
```

最终的结果应该是 **更新后的statePost.**

值得注意的是，这样子返回的实际是滤波后的结果，也就是对当前测量值的一个修正，并没有实现对下一个值的预测。

在实验后也发现，在传测量值之后，先进行一次更新，之后再预测-更新这样走，最后会返回一个预测值。

但这个办法在自己写的那个KalmanFilter里面是用不了的，会报错，原因不明，但大致就是传的向量或矩阵在公式运算时出现了行不对列的情况。

相关参数的确定

对于系统状态方程，简记为Y=AX+B，X和Y是表示系统状态的列向量，A是转移矩阵，B是其他项。

状态值（向量）只要能表示系统的状态即可，状态值的维数决定了转移矩阵A的维数，比如X和Y是N×1的，则A是N×N的。

A的确定跟X有关，只要保证方程中不相干项的系数为0即可，看下面例子

X和Y是二维的，

$$\begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} c \\ d \end{bmatrix} + B$$

注意，X和Y描述你的系统状态，他的维数决定了你的状态转移矩阵的维数。 n维对应于 n*n

X和Y是三维的，

$$\begin{bmatrix} c \\ d \\ e \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} c \\ d \\ e \end{bmatrix} + B$$

X和Y是三维的，但c和△ c是相关项

$$\begin{bmatrix} c \\ d \\ \Delta c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} c \\ d \\ \Delta c \end{bmatrix} + B$$

$$\begin{bmatrix} c \\ \Delta c \\ d \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} c \\ \Delta c \\ d \end{bmatrix} + B$$

上面的1也可以是其他值。

预测、更新函数

下面对predict( ) 和correct( )函数介绍下，可以不用看，不影响编程。

```
CV_EXPORTS const oclMat& KalmanFilter::predict(const oclMat& control)
{
    gemm(transitionMatrix, statePost, 1, oclMat(), 0, statePre);
    oclMat temp;

    if(control.data)
        gemm(controlMatrix, control, 1, statePre, 1, statePre);
    gemm(transitionMatrix, errorCovPost, 1, oclMat(), 0, temp1);
    gemm(temp1, transitionMatrix, 1, processNoiseCov, 1, errorCovPre, GEMM_2_T);
    statePre.copyTo(statePost);
    return statePre;
}
```

gemm( )是矩阵的广义乘法

```
void gemm(const GpuMat& src1, constGpuMat& src2, double alpha, const GpuMat&
src3, double beta,GpuMat& dst, int flags=0, Stream& stream=Stream::Null())

    dst = alpha · src1 · src2 +beta· src3
```

上面，oclMat()其实是uk，只不过默认为0，所以没赋值。整个过程就计算了x'和P'。（用x'代表x的预测值，用P'代表P的预测值）。GEMM_2_T表示对第2个参数转置。

x'(k)=1·A·x(k-1)

如果B非空， x'(k) = 1·B·u + 1·x'(k-1)

temp1 = 1·A·P(k-1) + 0·u(k)

P'(k) = 1· temp1·AT + 1· Qk= A·P(k-1)·AT + 1· Qk

> 可见，和第一部分的理论介绍完全一致。

```
CV_EXPORTS const oclMat& KalmanFilter::correct(const oclMat& measurement)
{
    CV_Assert(measurement.empty() == false);
    gemm(measurementMatrix, errorCovPre, 1, oclMat(), 0, temp2);
    gemm(temp2, measurementMatrix, 1, measurementNoiseCov, 1, temp3, GEMM_2_T);
    Mat temp;
    solve(Mat(temp3), Mat(temp2), temp, DECOMP_SVD);
    temp4.upload(temp);
    gain = temp4.t();
    gemm(measurementMatrix, statePre, -1, measurement, 1, temp5);
    gemm(gain, temp5, 1, statePre, 1, statePost);
    gemm(gain, temp2, -1, errorCovPre, 1, errorCovPost);
    return statePost;
}
```

```
bool solve(InputArray src1, InputArray src2, OutputArray dst, int
flags=DECOMP_LU)
```

求解线型最小二乘估计

$$dst = \arg\min \|src1 \cdot X - src2\|$$

temp2 = 1· H·P′ + 0·u(k)

temp3 = 1· temp2·HT + 1·R = H·P′·HT+ 1· R　也就是上面的Sk

temp = argmin||tem2- temp3||

K=temp

temp5 = -1· H·x′ + 1·zk　　就是上面的y′。

x = 1·K·temp5 + 1·x′ = KT·y′ +x′

P =-1·K·temp2 + 1·P′ = -K·H·P′+P′ = (I- K·H) P′
也和第一部分的理论完全一致。

通过深入函数内部，学到了两个实用的函数哦。矩阵广义乘法gemm( )、最小二乘估计solve( )

补充：

1）以例2为例，为什么状态值一般都设置成（x,y,△x,△y）？我们不妨设置成(x,y,△x)，对应的转移矩阵也改成3×3的。可以看到仍能跟上，不过在x方向跟踪速度快，在y方向跟踪速度慢。进一步设置成(x,y)和2×2的转移矩阵，程序的跟踪速度简直是龟速。所以，简单理解，△x和△y严重影响对应方向上的跟踪速度。