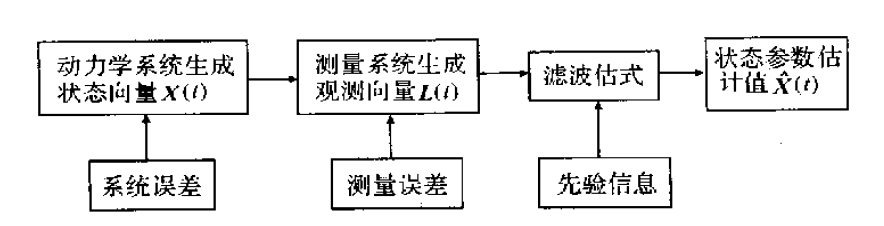
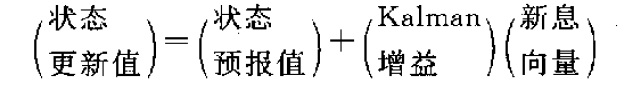
Kalman滤波的显著特点是对状态空间进行估计，而状态空间估计一般是动态估计。Kalman滤波采用递推算法,即由参数的验前估值和新的观测数据进行状态参数的更新。如此Kalman滤波一般只需存储前一个历元的状态参数估值,无须存储所有历史观测信息。显然Kalman滤波具有很高的计算效率,并可进行实时估计。

Kalman滤波考虑了信号与测量值的基本统计特性(一阶、二阶统计特性) ,而且由于采用了状态空间概念,用状态方程描述系统,信号作为状态，所以它既能估计平稳的一维信号随机过程，又能估计非平稳的多维(向量)信号随机过程(张金槐,蔡洪,1995)。

滤波过程：



也可表示为文字框图：



**动力学模型：**

系统的动力学模型一般用微分方程描述：



通过解微分方程，并离散化可得到如下线性向量差分方程：



其中，为时刻的状态向量，为阶的状态转移矩阵，为至时间间隔内取常量的维控制向量，为维矩阵，称为灵敏系数矩阵，为高斯白噪声过程误差向量。



一般情况下，顾及相当困难，的设计也不容易，于是简化为：

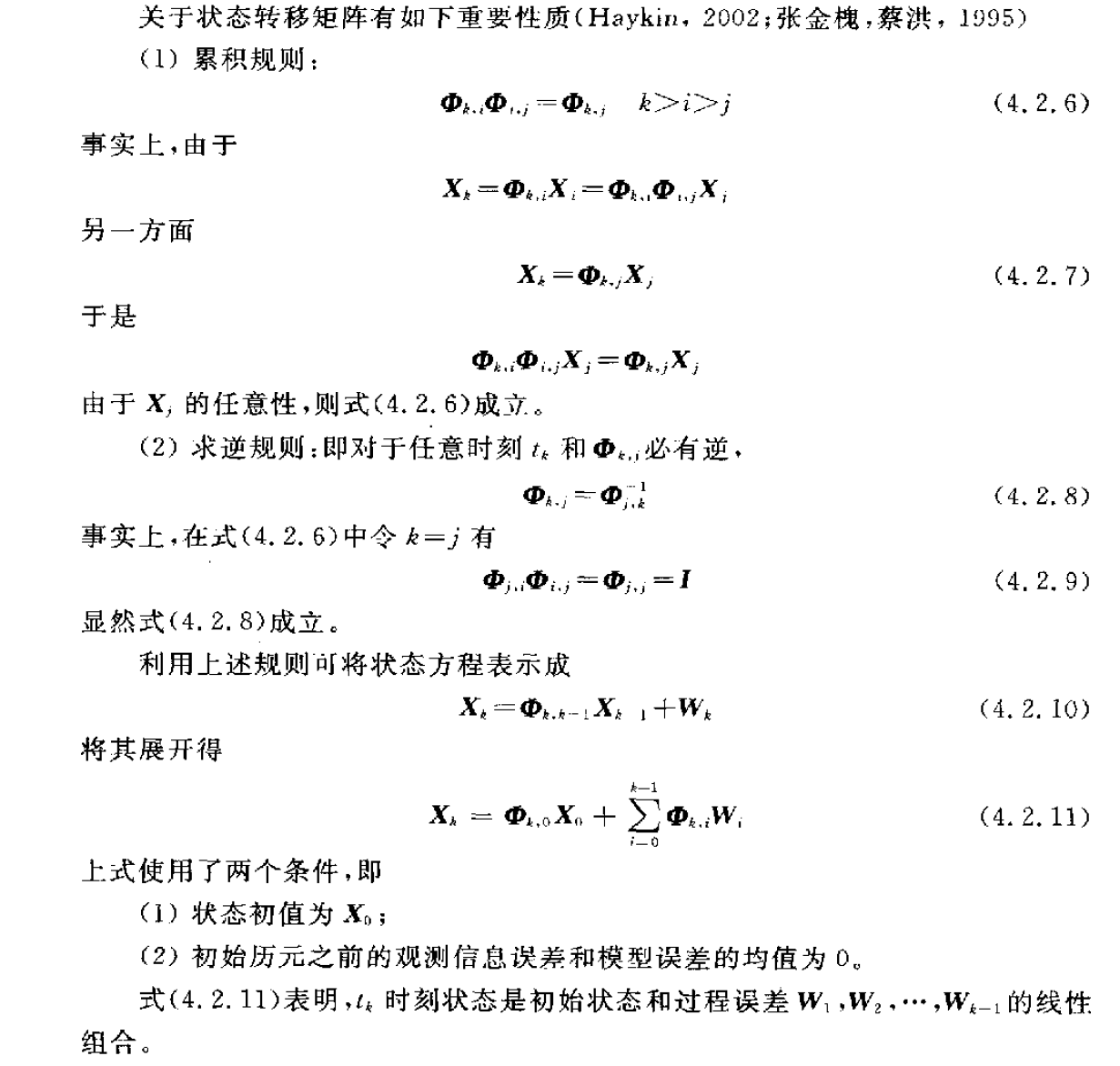


在真值（无误差）情况下，有：



高斯白噪声均值为0

动力学模型有两个重要组成部分：转移矩阵、过程噪声向量



**观测模型**

在讨论Kalman滤波时,除必须知道系统的状态方程(动力学模型)外,还必须有观测序列的支持。假定有随机观测向量L1 ,L2,.,Lx-1,由前k-1步滤波已求得tk-1时刻的状态参数最优估值,又在tk时刻得到观测向量Lk,该观测向量与tk时刻的系统状态参数向量Xk;的关系为



式中:Ak为nXm阶设计矩阵,也称观测矩阵;ek为nXm阶观测噪声向量。其数学期望为0,协方差矩阵为



显然,为高斯白噪声序列。

这里假定状态初始值为,且与及不相关,噪声向量与统计独立,即

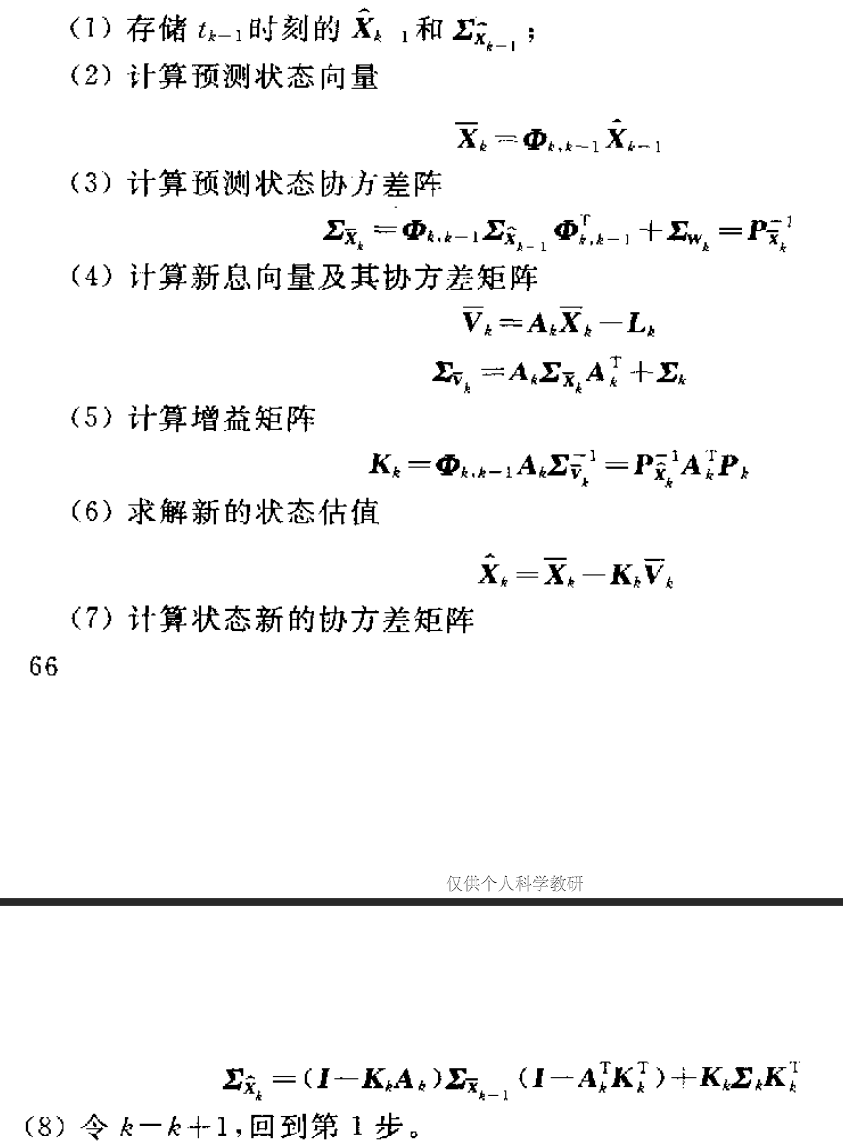


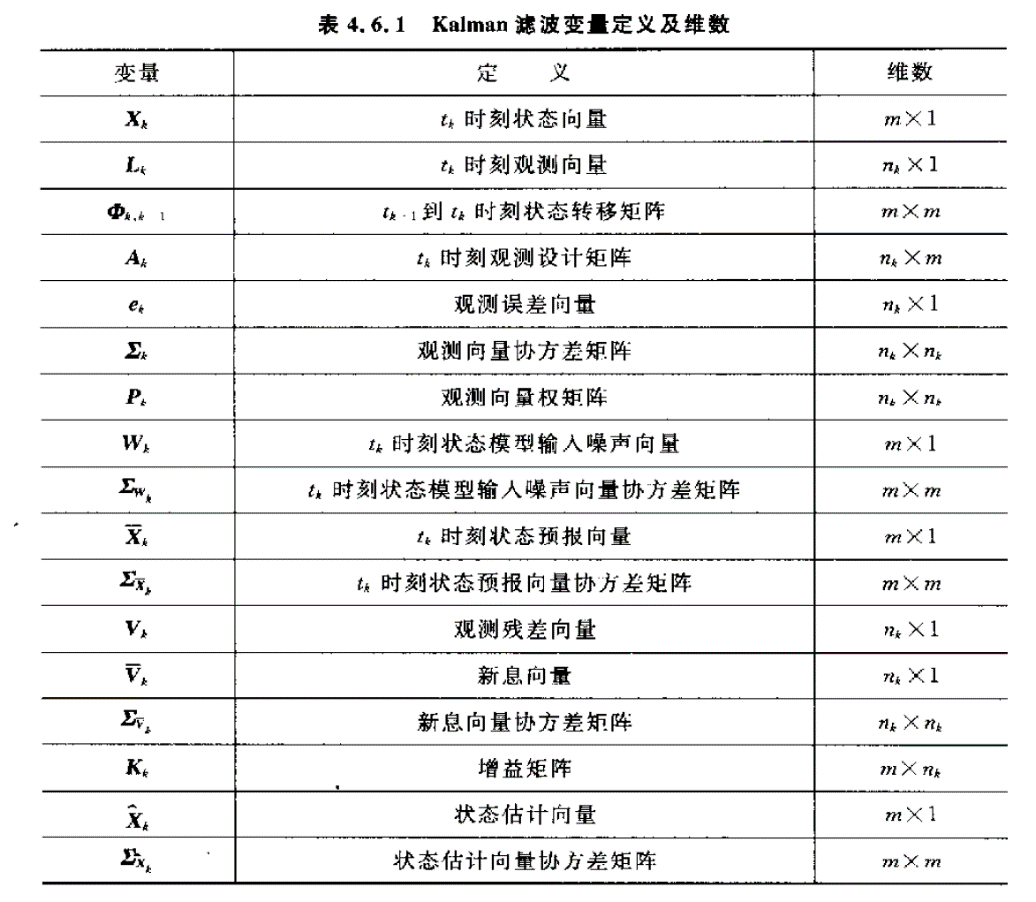
同时使用动力学模型信息和观测信息求解系统状态最优估值的方法一般为动态Kalman滤波法。

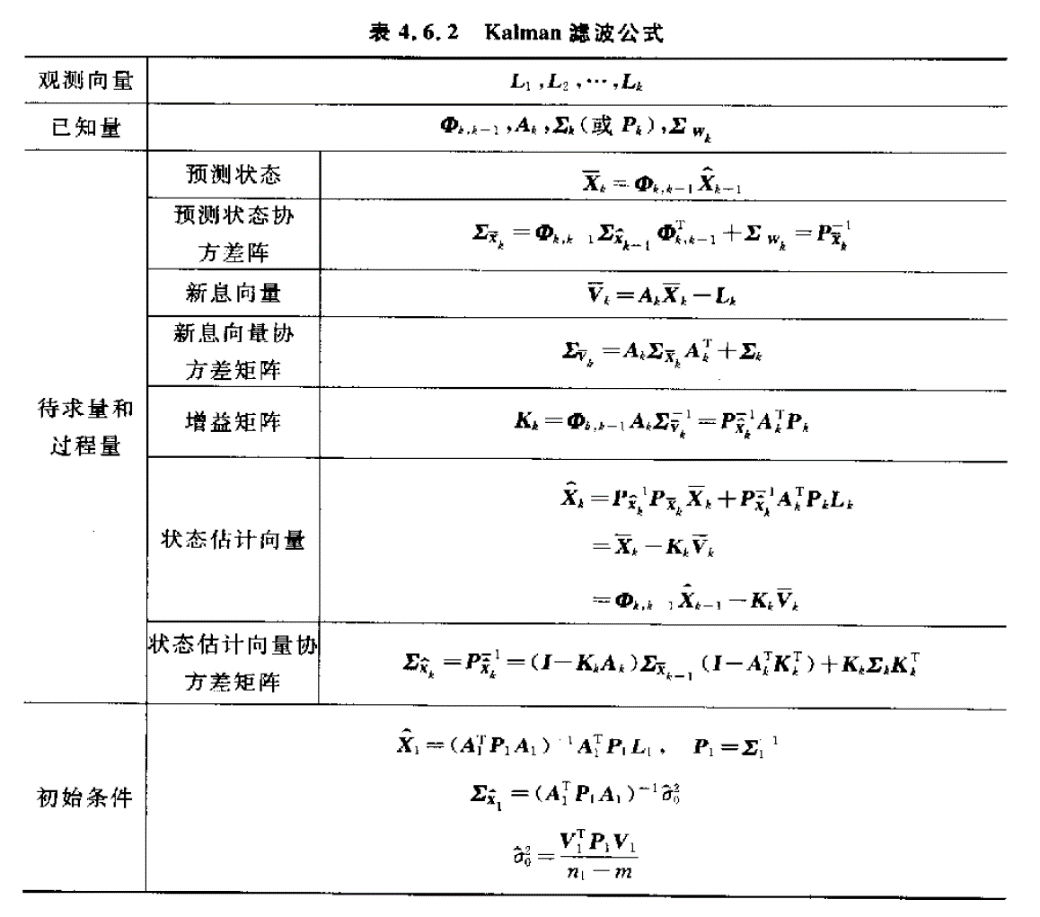
Kalman滤波推导过程：

已在笔记上写了

# Kalman滤波总结：







# 抗差自适应滤波（P200）

实践中,动态载体在全程运行过程中很难满足上述任何一种滤波,例如车辆导航和航空导航,运动载体在开始阶段一般具有加速度，且为非均匀加速度,载体在运动过程中也可能拐弯加速、减速,所有这些运动情形很难用统一的动力学模型加以描述，于是预设的动力学模型只能是载体运动过程的近似描述,即预设的动力学模型与载体的实际运动轨迹之间必然存在误差，而且有时这些误差存在显著异常。在这类情形下,应用标准Kalman滤波、抗差Kalman滤波和Sage开窗自适应滤波都可能导致滤波结果发散。当然利用各历元观测信息进行平差可以排除动力学模型误差对状态参数的影响,但是采用这种平差方法也存在四个问题:

(1)各历元必须有多余观测,否则观测模型及相应的法方程秩亏，从而不可能求得状态参数的解;

(2)当个别观测出现异常时,这类平差方法也不能保证状态参数的解的可靠性;

(3)大多数情况下，当载体运动轨迹与动力学模型预测信息基本相符时,这类平差又因放弃使用动力学模型信息而使状态参数估计蒙受效率方面的损失;

(4)采用历元平差,只能求得离散点的载体状态信息,不能求得连续状态。

抗差自适应Kalman滤波原理

1. 自由极值原理

状态预测信息向量的误差方程为：



在历元观测向量的误差方程为



设历元状态预测向量的协方差矩阵为，相应权阵为，观测向量的协方差矩阵为。

为控制观测异常和状态预测信息异常对状态参数估值的影响，构造极值原则：



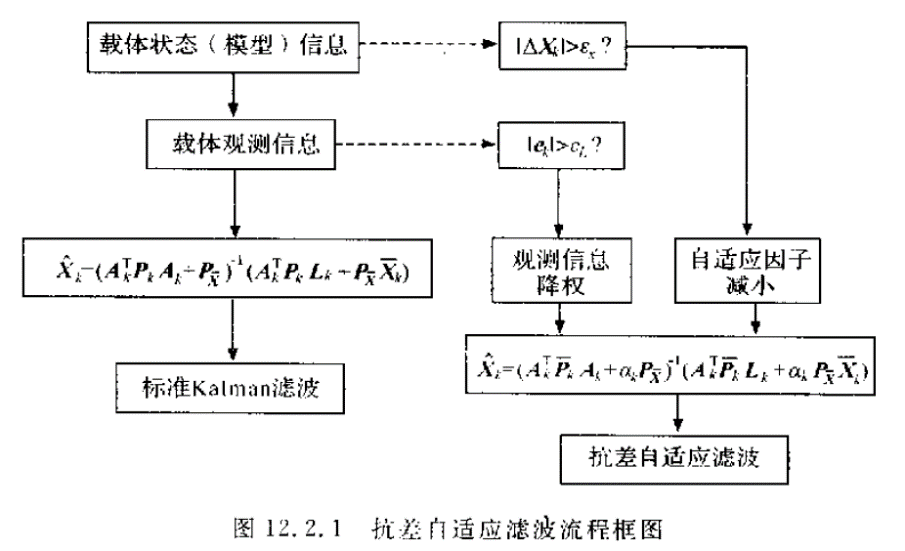
其中，是的抗差等价权矩阵，它是观测向量权矩阵的自适应估值；称为自适应因子，在该模型中，。（对观测向量采用了抗差估计原则，对状态预测信息采用了自适应估计原则，于是上述极值原则成为抗差自适应滤波原则）

对求导后得到：

可以得到状态参数向量的抗差自适应滤波解：



从上述抗差自适应滤波解的表达式中可以看出,若观测信息含有异常,则相应的等价权矩阵元素减小，从而可以控制观测异常对状态参数估值的影响;若动力学模型产生异常扰动.相应的自适应因子减小，从而以控制状态模型预测信息异常对状态参数估计的影响;极端地说，若为0.则状态预测信息对状态参数估计的影响为0。此时抗差自适应滤波自动变成了抗差平差。



1. 条件极值原理

一般推导我们都采用条件极值的方法进行，构造极值函数为



是观测向量的等价协方差矩阵，可通过双因子方差-协方差膨胀模型获得，为拉格朗日乘数向量。

对和求导并等于0得：



由上式可得：



将上式代入和中可得：



利用这个公式，带入可得：



利用上式求解得时候一定要注意，自适应因子不能为0。

对于增益矩阵来说，变成了：



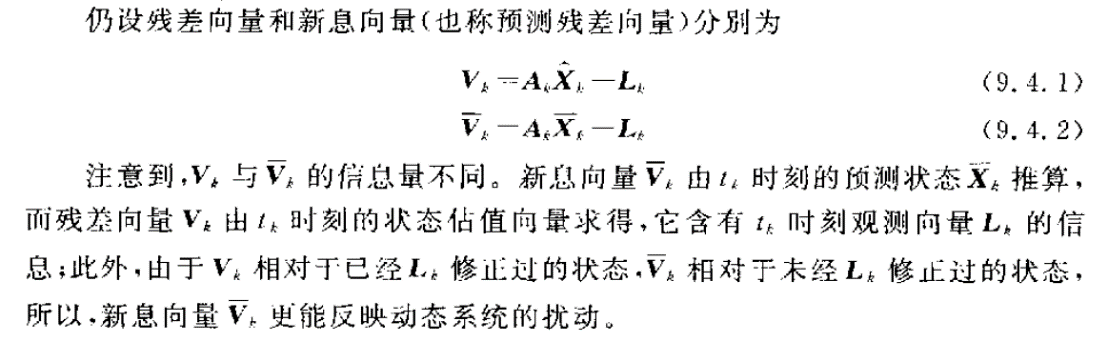
则状态参数向量得抗差自适应滤波解为：



上式也可以写为：



其中称为新息向量。



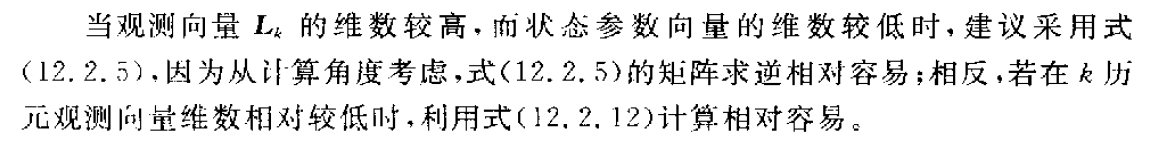
由于和不相关，预测向量和当前历元得观测向量是不相关的，则的协方差矩阵为：



是幂等阵。



注意：（待定）

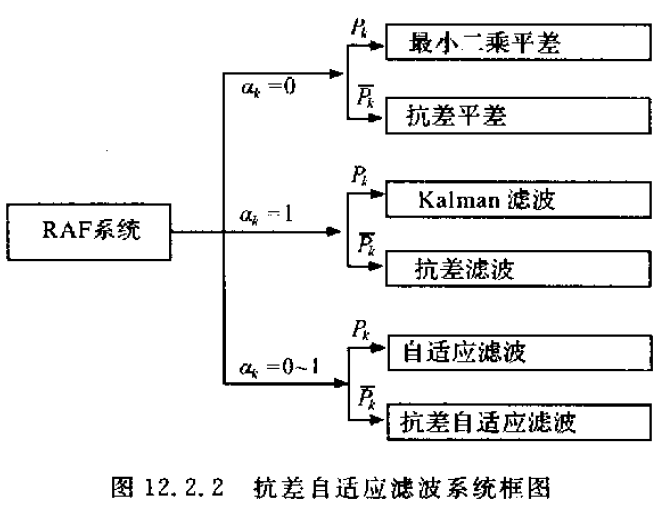


（12.2.5是自由极值推出来的，12.2.12是条件极值推出来的）

特例：

采用自由极值原理推导出来的

是抗差自适应滤波解的通用公式。



# 渐消滤波（P156）

渐消滤波具有控制状态模型误差影响的能力，当动力学模型具有较大误差的时候，可以减小动力学模型的贡献，更多地信赖当前历元地观测信息。

渐消滤波的推导过程：

这里直接给出渐消滤波的解：（推导和之前一样）



增益矩阵变为：





和标准的Kalman滤波区别在于：状态参数预测向量的协方差矩阵增加了一个渐消记忆因子，并且。

如何求取渐消因子？

理论上，若时刻的状态估值误差大（即动力学模型计算出来的不准确），则应增大，即的误差在中的影响应小。

本文采用夏启军、房建成等人所提出的一种渐消因子简化算法



式中，是矩阵求迹符号。





其中称为预测残差，也叫做新息向量。



上式更能敏感地反映当前观测历元动力学模型误差地现状，没有对历史信息取平均，而直接采用了当前历元地信息，而由预测残差地历史信息确定的具有累积效应和平滑效果。

理论分析：

由可以得到：



将上式带入到表达式中，有：



在标准Kalman中有：



带入到表达式中和表达式进行比较，会发现两者相等，即理论上



但是由于的估计式中含有（），



如果动力学模型误差较大，则的值就很大，因此我们采用式1.15计算出来的要大于，所以，所以渐消因子要大于1。

在程序设计中，渐消因子的初值设为1。即：和1.15的表达式相同。

渐消滤波算例分析：

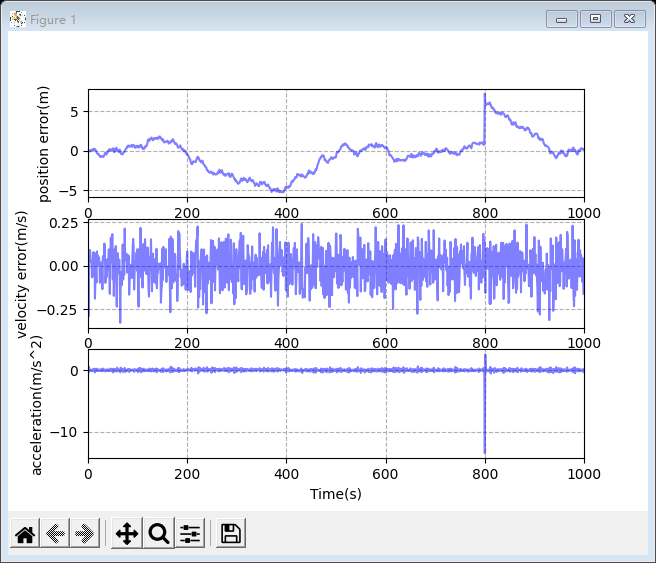
1. 常加速度模型（一维）

历元间隔1s；状态向量维度为3，分别是位置、速度、加速度；每个历元观测向量维度为2，分别是位置和速度。

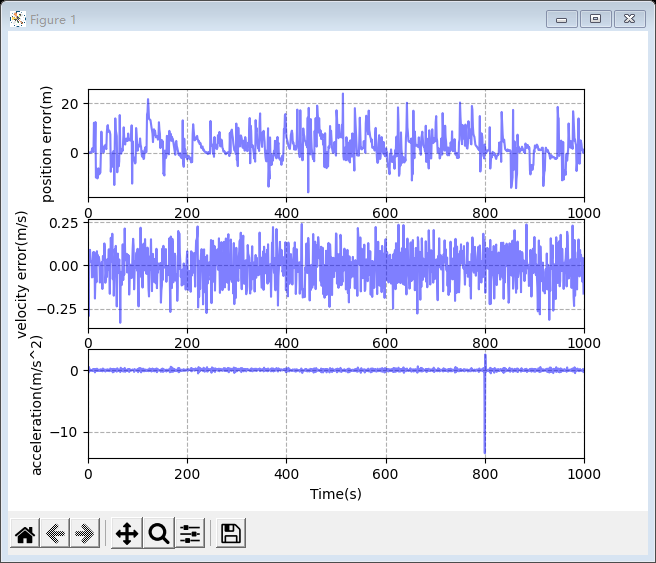
滤波初值设置：

|  |  |
| --- | --- |
| 位置 | 0m |
| 速度 | 5m/s |
| 加速度 | 0m/s2 |
| 状态向量 | [0,5,0] |
| 状态转移矩阵 | [[1,dt,0.5dt2],[0,1,dt],[0,0,1]] |
| 设计矩阵 | [[1,0,0], [0,1,0]] |
| 状态向量协方差阵 | 单位阵（3\*3） |
| 观测噪声协方差阵 | [[100,0], [0,0.01]] |
| 过程噪声矩阵 | [[dt5/20, dt4/8, dt3/6],  [dt4/8, dt3/3, dt2/2],  [dt3/6, dt2/2, dt]] |

假设载体在X轴方向运动，0-200s载体以5m/s的速度匀速运动，200-400s载体以0.05m/s2匀加速运动，400-600s以-0.02 m/s2匀减速运动,600-800s匀速运动，800-1000s停止状态。直接在轨迹发生器的位置和速度输出上叠加白噪声信号来模拟X轴方向上的位置和速度。



标准Kalman滤波

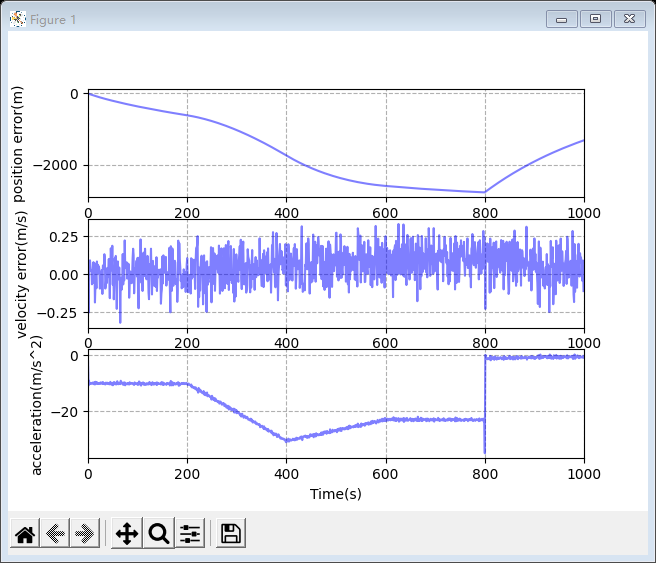


渐消Kalman滤波

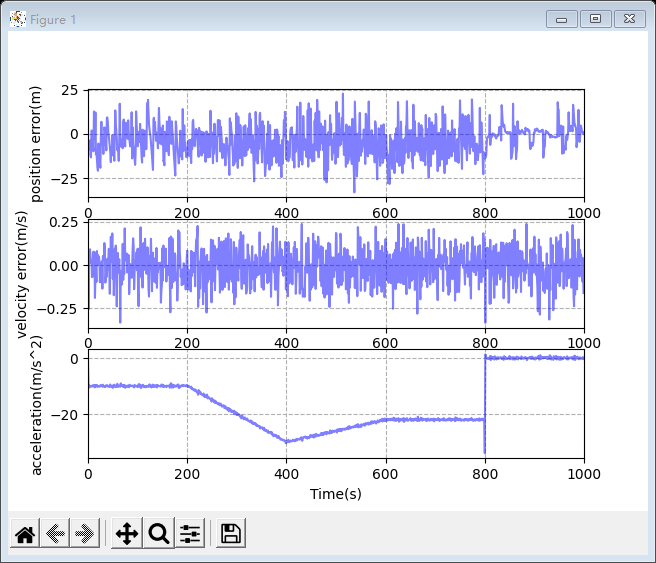
在动力学模型误差较小的情况下，标准Kalman滤波比渐消Kalman要好很多。如果错误地将动力学模型建立错误，会是怎样地结果？

假设动力学模型中状态转移矩阵变为：

[[1,dt,0.5dt2],[0,3,dt],[0,0,1]]



标准Kalman滤波（错误动力学模型）



渐消Kalman滤波（错误动力学模型）

可以看到，采用渐消滤波位置不发散了，位置误差在25m以内。

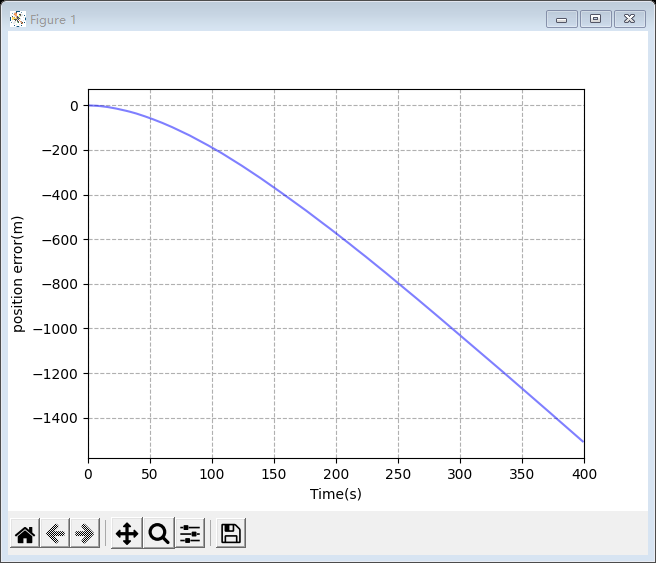
1. 假设载体进行匀加速运动，但设计动力学模型时误以为载体处于静止状态。

历元间隔1s；状态向量维度为1，位置；每个历元观测向量维度为1，位置。

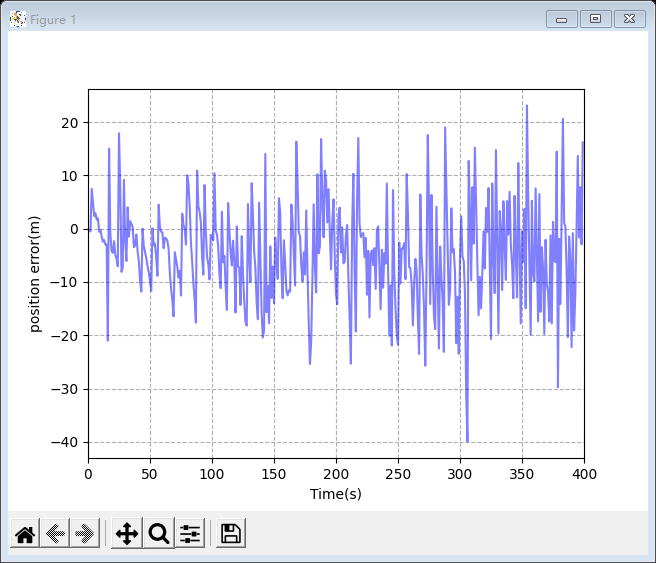
滤波初值设置：

|  |  |
| --- | --- |
| 位置 | 0m |
| 状态转移矩阵 | [1] |
| 设计矩阵 | [1] |
| 状态向量协方差阵 | 单位阵（1\*1） |
| 观测噪声协方差阵 | [100] |
| 过程噪声矩阵 | [0.01] |

0-400s载体以0.05m/s2进行匀加速运动。



标准Kalman滤波



渐消Kalman滤波

采用渐消Kalman滤波，误差大部分历元在20m以内，当动力学模型不准确的时候，采用渐消滤波可以有效抑制发散。

附带程序：

**class** FadingKalmanFilter(object):  
 **def** \_\_init\_\_(self, alpha, dim\_x, dim\_z, dim\_u=0):  
  
 **assert** alpha >= 1  
 **assert** dim\_x > 0  
 **assert** dim\_z > 0  
 **assert** dim\_u >= 0  
  
 self.alpha\_ = alpha *# 渐消因子* self.dim\_x = dim\_x *# 状态值个数* self.dim\_z = dim\_z *# 观测值个数* self.dim\_u = dim\_u *# 控制值个数* self.x = zeros((dim\_x, 1)) *# Current state estimate 当前历元状态参数向量（dim\_x\*1）* self.P = eye(dim\_x) *# Current state covariance matrix 当前状态参数向量协方差阵（dim\_x\*dim\_x）* self.Q = eye(dim\_x) *# Process noise covariance matrix 过程噪声协方差阵（dim\_x\*dim\_x）* self.B = 0. *# control transition matrix 控制矩阵（一般不考虑）* self.F = np.eye(dim\_x) *# state transition matrix 状态转移矩阵（dim\_x\*dim\_x）* self.H = zeros((dim\_z, dim\_x)) *# Measurement function 观测设计矩阵（dim\_z\*dim\_x）* self.R = eye(dim\_z) *# Measurement noise covariance matrix 观测噪声协方差矩阵（dim\_z\*dim\_z）* self.z = np.array([[**None**] \* dim\_z]).T  
  
 *# gain and residual are computed during the innovation step. We  
 # save them so that in case you want to inspect them for various  
 # purposes* self.K = 0 *# kalman gain 增益矩阵* self.y = zeros((dim\_z, 1)) *# residual are computed during the innovation step 新息向量* self.S = np.zeros((dim\_z, dim\_z)) *# system uncertainty (measurement space)* self.SI = np.zeros((dim\_z, dim\_z)) *# inverse system uncertainty  
  
 #self.\_alpha\_sq = 1. # fading memory control  
 # identity matrix. Do not alter this.* self.\_I = np.eye(dim\_x)  
  
 *# Only computed only if requested via property 似然函数* self.\_log\_likelihood = log(sys.float\_info.min)  
 self.\_likelihood = sys.float\_info.min  
 self.\_mahalanobis = **None** *# these will always be a copy of x,P after predict() is called* self.x\_prior = self.x.copy() *# 当前状态向量预测值* self.P\_prior = self.P.copy() *# 当前状态向量预测协方差阵  
  
 # these will always be a copy of x,P after update() is called* self.x\_post = self.x.copy() *# 当前状态向量估值* self.P\_post = self.P.copy() *# 当前状态向量估值的协方差阵* self.inv = np.linalg.inv  
  
 **def** settingalpha\_(self,z, u=**None**, B=**None**, F=**None**, Q=**None**,H=**None**,P=**None**,R=**None**):  
 *#计算预测状态向量* **if** B **is None**:  
 B = self.B  
 **if** P **is None**:  
 P = self.P  
 **if** R **is None**:  
 R = self.R  
 **if** H **is None**:  
 H = self.H  
 **if** F **is None**:  
 F = self.F  
 **if** Q **is None**:  
 Q = self.Q  
 **elif** isscalar(Q): *# 如果Q是标量* Q = eye(self.dim\_x) \* Q  
  
 *# x = Fx + Bu* **if** B **is not None and** u **is not None**:  
 x = dot(F, self.x) + dot(B, u)  
 **else**:  
 x = dot(F, self.x)  
 *#计算预测残差向量* y= z - dot(H, self.x)  
 *#计算预测残差向量的协方差阵* S=self.alpha\_\*dot(y,y.T)/(1+self.alpha\_)  
  
 HF=dot(H,F)  
 HFP=dot(HF,P)  
 M=dot(HFP,HF.T)  
 HQ=dot(H,Q)  
 N=S-dot(HQ,H.T)-R  
  
  
 alpha=np.trace(N)/np.trace(M)  
  
 **if** alpha>=1:  
 self.alpha\_=alpha  
 **else**:  
 self.alpha\_=1.  
  
  
  
  
 **pass** *# 预测* **def** predict(self, u=**None**, B=**None**, F=**None**, Q=**None**):  
 *"""  
 Predict next state (prior) using the Kalman filter state propagation  
 equations.  
 Parameters  
 ----------  
 u : np.array, default 0  
 Optional control vector.控制向量（一般不用）  
  
 B : np.array(dim\_x, dim\_u), or None  
 Optional control transition matrix; a value of None  
 will cause the filter to use `self.B`.  
  
 F : np.array(dim\_x, dim\_x), or None  
 Optional state transition matrix; a value of None  
 will cause the filter to use `self.F`.  
  
 Q : np.array(dim\_x, dim\_x), scalar, or None  
 Optional process noise matrix; a value of None will cause the  
 filter to use `self.Q`.  
 """* **if** B **is None**:  
 B = self.B  
 **if** F **is None**:  
 F = self.F  
 **if** Q **is None**:  
 Q = self.Q  
 **elif** isscalar(Q): *# 如果Q是标量* Q = eye(self.dim\_x) \* Q  
  
 *# x = Fx + Bu* **if** B **is not None and** u **is not None**:  
 self.x = dot(F, self.x) + dot(B, u)  
 **else**:  
 self.x = dot(F, self.x)  
  
 *# P = FPF' + Q P变为状态预测向量的协方差矩阵* self.P = self.alpha\_ \* dot(dot(F, self.P), F.T) + Q  
  
 *# save prior* self.x\_prior = self.x.copy()  
 self.P\_prior = self.P.copy()  
  
 **def** update(self, z, R=**None**, H=**None**):  
 *"""  
 Add a new measurement (z) to the Kalman filter.  
 If z is None, nothing is computed. However, x\_post and P\_post are  
 updated with the prior (x\_prior, P\_prior), and self.z is set to None.  
 Parameters  
 ----------  
 z : (dim\_z, 1): array\_like 当前历元观测向量  
 measurement for this update. z can be a scalar if dim\_z is 1,  
 otherwise it must be convertible to a column vector.  
 If you pass in a value of H, z must be a column vector the  
 of the correct size.  
 R : np.array, scalar, or None 测量噪声矩阵  
 Optionally provide R to override the measurement noise for this  
 one call, otherwise self.R will be used.  
 H : np.array, or None 设计矩阵  
 Optionally provide H to override the measurement function for this  
 one call, otherwise self.H will be used.  
 """  
  
 # set to None to force recompute* self.\_log\_likelihood = **None** self.\_likelihood = **None** self.\_mahalanobis = **None  
  
 if** z **is None**:  
 self.z = np.array([[**None**] \* self.dim\_z]).T  
 self.x\_post = self.x.copy()  
 self.P\_post = self.P.copy()  
 self.y = zeros((self.dim\_z, 1))  
 **return  
  
 if** R **is None**:  
 R = self.R  
 **elif** isscalar(R):  
 R = eye(self.dim\_z) \* R  
  
 **if** H **is None**:  
 *# z = reshape\_z(z, self.dim\_z, self.x.ndim)* H = self.H  
  
 *# y = z - Hx 新息向量y  
 # error (residual) between measurement and prediction* self.y = z - dot(H, self.x)  
  
 *# common subexpression for speed* PHT = dot(self.P, H.T)  
  
 *# S = HPH' + R  
 # project system uncertainty into measurement space  
 # S 新息向量y的协方差矩阵 R观测向量噪声的协方差矩阵* self.S = dot(H, PHT) + R  
 self.SI = self.inv(self.S)  
 *# K = PH'inv(S)  
 # map system uncertainty into kalman gain  
 # K 增益矩阵* self.K = dot(PHT, self.SI)  
  
 *# x = x + Ky  
 # 当前历元的状态向量估值  
 # predict new x with residual scaled by the kalman gain* self.x = self.x + dot(self.K, self.y)  
  
 *# P = (I-KH)P(I-KH)' + KRK'  
 # This is more numerically stable  
 # and works for non-optimal K vs the equation  
 # P = (I-KH)P usually seen in the literature.  
 # 状态向量估值的协方差矩阵* I\_KH = self.\_I - dot(self.K, H)  
 self.P = dot(dot(I\_KH, self.P), I\_KH.T) + dot(dot(self.K, R), self.K.T)  
  
 *# save measurement and posterior state* self.z = deepcopy(z)  
 self.x\_post = self.x.copy()  
 self.P\_post = self.P.copy()

# 先验协方差矩阵的开窗估计法—Sage-Husa滤波

（制约模型误差影响的多种策略，其基本思路不外乎增大模型误差协方差矩阵,以便控制模型误差的影响;或减小当前最新观测向量的协方差矩阵,以便提高最新观测信息的使用效能,间接地削弱模型误差对状态参数向量的影响。然而所有这些方法都需要根据经验确定某些调节参数。）

Sage-Husa滤波是一种移动开窗估计法，即在特定的窗口宽度内，采用样本平均值的方法确定当前历元观测向量的协方差矩阵和模型误差的协方差矩阵。

观测噪声协方差矩阵的自适应开窗估计法

它又分别可以采用新息向量（预测残差向量）【IAE滤波】或观测残差向量【RAE滤波】估计当前观测残差的协方差矩阵。

新息向量

观测残差向量

利用可以得到这两者之间的关系：



两个向量的协方差矩阵分别为：



IAE

设观测误差近于正态分布，取计算窗口为N,则的估值可取为：



然后利用1.13的公式可以得到



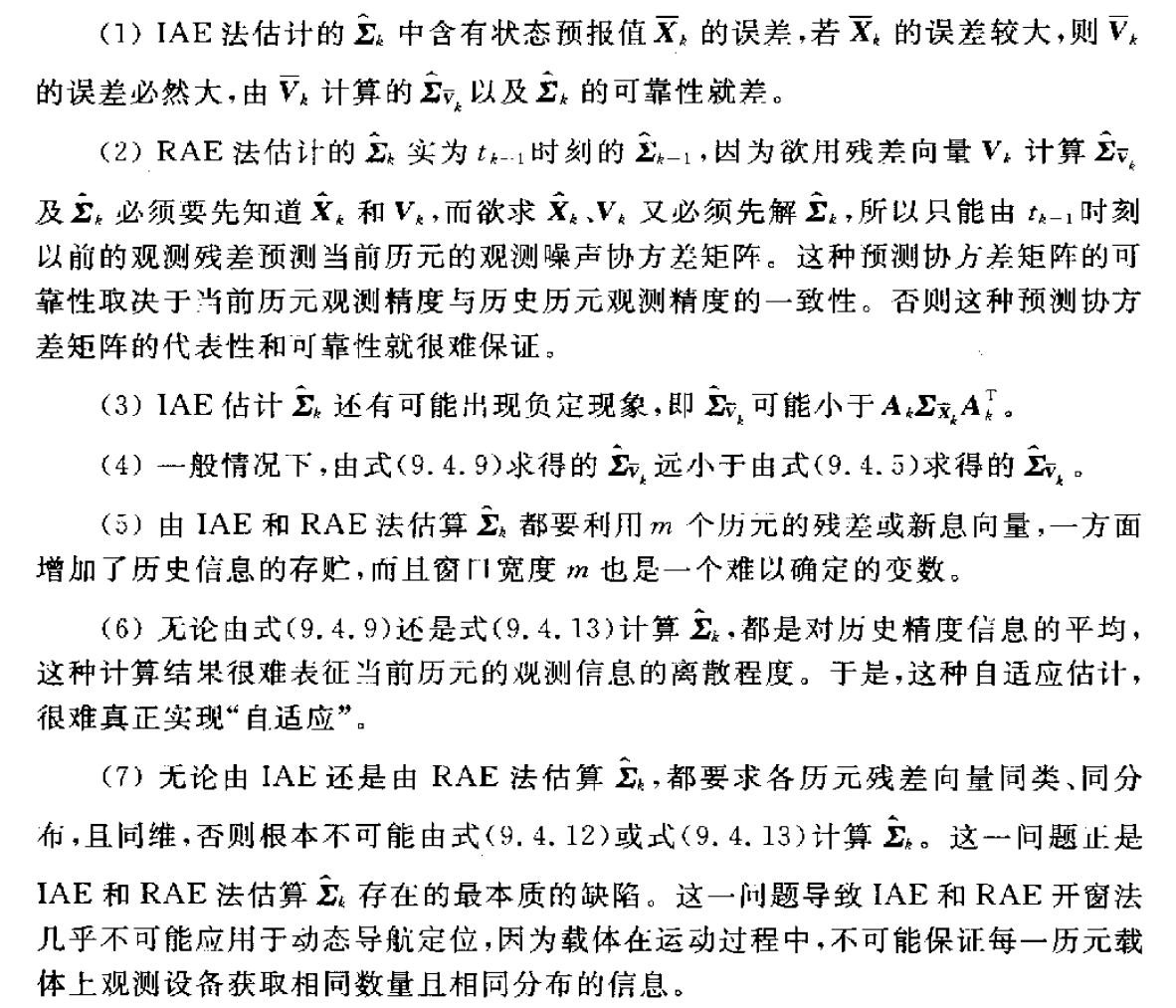
RAE

和上面类似：



然后利用1.13的公式可以得到





动态模型噪声矩阵（过程噪声矩阵）的开窗估计（待定）

状态预报向量的改正数向量为



可以得到：



注意到，的协方差估值可取为：



将1.19改为估值形式：

