# 视觉惯性导航在连续时间条件下的预积分理论

## 1.准备阶段

### 1.0 介绍

在纯视觉的或者中，由于图像的运动模糊、遮挡、快速运动、纯旋转、尺度不确定性的一系列问题，导致仅靠一个摄像头很难完成我们实际场景的应用需求，而直接可以得到运动主体自身的角速度、加速度的测量数据，从而对运动有一个约束，或者说与视觉形成互补，可实现快速运动的定位和主体纯旋转的处理，从而进一步提高/的可靠性。可以干的事情就是由上个时刻状态可以推算出下个时刻  的状态， 这里的状态就是旋转角度  ，速度 和位置 P 。这里再说明一下预积分的作用。假设在估计过程中有前后两帧图像，它们的位姿分别为，，那么它们的相对位姿为：。在这里我们可以认为是估计项，是有的位姿直接算出来的。如果要构成一个优化问题，我们还需要知道误差项和测量项。于是可以计算在这两个关键帧间的测量项，预积分其实就是在求解这个测量项。并且误差为。

### 1.1系统状态

时刻辅助惯性导航系统的状态为：

 (1)

是对应的单位四元数，表示从全局系到第帧局部系的坐标变换，和分别对应陀螺仪的漂移和加速度计的漂移，和分别对应在全局坐标系下的速度和位置。状态误差表示如下：

 (2)

速度值真值，均值和状态误差值，三者关系为，四元数真值，均值和状态误差值三者关系为：

 (3)

指的是四元数乘法，我们将整个转态的变化表示为，，。这里运算符分别用表示。

### 1.2图优化

给定残差，优化其实是一个状态的最大后验估计问题，

 (4)

我们的估计问题可写作：

 (5)

 (6)

## 2.连续时间下的预积分

会接收角速度和局部坐标系下的线加速度，它们与其真值的关系为：

 (7)

 (8)

这里表示全局坐标系下的重力加速度。表示从全局坐标系到局部惯性系的变换关系。分别指陀螺仪和加速度计的漂移。表示零均值的高斯白噪声。标准的动力学公式为：

 (9)

 (10)

 (11)

 (12)

 (13)

这里，。

## 3.标准过程

我们设时间间的量测数据为，的传播方程可以写作，

 (14)

第时刻的状态由第时刻的状态，量测和量测噪声决定，若噪声为零可得状态的期望值

 (15)

实际中，我们积分肯定是有噪声值的。批优化处理中的残差约束了状态的起始与结束。并且残差表示如下，

 (16)

是离散时间下的线性化的噪声协方差矩阵。

## 4.模型一：零点常数量测

我们令，，结合周期内的量测与初始状态得出最终状态：

 (17)

 (18)

 (19)

 (20)

 (21)

我们令，

 (22)

 (23)

利用这些公式我们也可以求出或。为消除预积分量测对真实漂移的依赖性，我们在处线性化当前的漂移估计，

(24)

 (25)

 (26)

式(24)和式(25)右边分别是的泰勒展开。当前的积分方法均为离散条件下的积分。

### 4.1测量均值

我们来推导的闭式解，我们推导下面的连续时间系统：

 (27)

解这个微分方程组可得，

 (28)

针对状态转移矩阵有，

 (29)

这里有，将式(29)带入式(28)有，

 (30)

 (31)

 (32)

这里。我们有，

 (33)

 (34)

(35)

### 4.2状态转移矩阵

为了使用所推导的连续时间预积分方法，我们必须要有和误差量对应的协方差。我们检查状态误差随时间的变化有，

 (36)

 (37)

 (38)

 (39)

利用,我们可以利用下面的线性系统描述状态误差，

 (40)

我们可以看出上述的系统包含了偏差误差，可以在给定的偏差范围内确定漂移。和描述的是由于随机游走漂移产生的区间上的偏差，而不是当前偏差估计的误差。状态转移矩阵可通过解下面微分方程得到。

 (41)

### 4.2离散协方差传播

结合状态误差矩阵，预积分量测的协方差矩阵传播公式如下：

 (42)

 (43)

 (44)

### 4.4加速度漂移的雅克比

我们需要计算的是更新公式相对于偏差变化的导数。根据式(35)我们给出求导的公式，



 (45)

当很小时有，

 (46)

### 4.5陀螺漂移雅克比

我们考虑让对陀螺漂移的每一项求偏导得，

 (47)

 (48)

针对式(47)与式(48)的每一项有，

 (49)

设是方向上的单位向量，上式的第三部分可写为，

 (50)



对任意有，

 (51)









所以，

 (52)

原式为：

 (53)

同理可得，

 (54)

很小时有，

 (55)

 (56)

同样有，

 (57)



这里

 (58)

 (59)

当很小时有，

 (60)

 (61)

我们接下来推导旋转矩阵对偏差变化量的导数。我们有，设时间间隔内的量测为，我们可得，

 (62)







上述的分解可以用在每一个时间间隔：

 (63)

当我们复合量测，例如区间，有

 (64)







利用性质有，

 (65)







任意时间旋转矩阵的更新公式为：

 (66)

 (67)

增量表达式有，

 (68)

整理旋转矩阵与对陀螺仪偏差的导数有，

 (69)

 (70)

 (71)

整体一次更新下的旋转矩阵为，

 (72)

### 4.6量测误差

(73)

这里，。

### 4.7量测雅克比

预积分残差可以划分为，

 (74)

通过对量测函数中相对应的量添加扰动，可以得到状态向量中每一量测的雅克比矩阵。例如，相对旋转测量残差受当前估计值附近陀螺仪偏差变化的扰动。。

 (75)









因此，考虑偏差扰动的雅克比为，

 (76)

同样考虑的雅克比为，

 (77)







雅克比有，

 (78)

考虑的雅克比有，

 (79)









 (80)

通过上述的方法我们可以求出考虑位置，速度和偏差的雅克比，

 (81)

 (82)

 (83)

 (84)

 (85)

 (86)

 (87)

 (88)

 (89)

 (90)

 (91)

 (92)

 (93)

 (94)

 (95)

### 附录

1.编程实现式(35):

计算。

double delta\_t = t\_1 - t\_0;// 计算采样的间隔时间

// Get estimated imu readings // 得到估计的imu读数

Eigen::Matrix<double, 3, 1> w\_hat = w\_m\_0 - b\_w\_lin;

Eigen::Matrix<double, 3, 1> a\_hat = a\_m\_0 - b\_a\_lin;

计算角度的变化量，我们假定间隔时间内角速度恒定。

// Get angle change w\*dt 求角度的变化量

Eigen::Matrix<double, 3, 1> w\_hatdt = w\_hat \* delta\_t;

计算。

// Get magnitude of w and wdt 这里分别求的是 w 的模， 和模与时间的乘积

double mag\_w = w\_hat.norm();

double w\_dt = mag\_w \* delta\_t;

计算

// Get relative rotation 得到相对旋转矩阵

Eigen::Matrix<double, 3, 3> R\_tau2tau1 = small\_w ? eye3 - delta\_t \* w\_x + (pow(delta\_t, 2) / 2) \* w\_x\_2 :eye3 - (sin\_wt / mag\_w) \* w\_x + ((1.0 - cos\_wt) / (pow(mag\_w, 2.0))) \* w\_x\_2;

计算，，，，，，。

// Get some of the variables used in the preintegration equations

double dt\_2 = pow(delta\_t, 2); // 这些都是积分时要用的

double cos\_wt = cos(w\_dt);

double sin\_wt = sin(w\_dt);

Eigen::Matrix<double, 3, 3> w\_x = skew\_x(w\_hat);

Eigen::Matrix<double, 3, 3> a\_x = skew\_x(a\_hat);

Eigen::Matrix<double, 3, 3> w\_tx = skew\_x(w\_hatdt);

Eigen::Matrix<double, 3, 3> w\_x\_2 = w\_x \* w\_x;

计算。

// Updated rotation and its transpose  更新旋转矩阵

Eigen::Matrix<double, 3, 3> R\_k2tau1 = R\_tau2tau1 \* R\_k2tau;

Eigen::Matrix<double, 3, 3> R\_tau12k = R\_k2tau1.transpose(); // 逆等于转置

计算。

f\_1 = (w\_dt \* cos\_wt - sin\_wt) / (pow(mag\_w, 3));

计算。

f\_2 = (pow(w\_dt, 2) - 2 \* cos\_wt - 2 \* w\_dt \* sin\_wt + 2) / (2 \* pow(mag\_w, 4));

计算。

f\_3 = -(1 - cos\_wt) / pow(mag\_w, 2);

计算。

f\_4 = (w\_dt - sin\_wt) / pow(mag\_w, 3);

计算

Eigen::Matrix<double, 3, 3> alpha\_arg = ((dt\_2 / 2.0) \* eye3 + f\_1 \* w\_x + f\_2 \* w\_x\_2);

计算

Eigen::Matrix<double, 3, 3> Beta\_arg = (delta\_t \* eye3 + f\_3 \* w\_x + f\_4 \* w\_x\_2);

计算alpha\_arg和Beta\_arg和相乘。

Eigen::MatrixXd H\_al = R\_tau12k \* alpha\_arg;

Eigen::MatrixXd H\_be = R\_tau12k \* Beta\_arg;

通过累加的方式计算整个

// Update the measurement means

alpha\_tau += beta\_tau \* delta\_t + H\_al \* a\_hat;

beta\_tau += H\_be \* a\_hat;

计算

// Get right Jacobian

Eigen::Matrix<double, 3, 3> J\_r\_tau1 = small\_w ? eye3 - .5 \* w\_tx + (1.0 / 6.0) \* w\_tx \* w\_tx :eye3 - ((1 - cos\_wt) / (pow((w\_dt), 2.0))) \* w\_tx + ((w\_dt - sin\_wt) / (pow(w\_dt, 3.0))) \* w\_tx \* w\_tx;

在考虑陀螺仪偏差的情况下更新雅克比；

// Update orientation in respect to gyro bias Jacobians

J\_q = R\_tau2tau1 \* J\_q + J\_r\_tau1 \* delta\_t;

更新

公式：



// Update alpha and beta in respect to accel bias Jacobians

/\*

更新位置和速度的积分项

\*/

H\_a -= H\_al;

H\_a += delta\_t \* H\_b;

H\_b -= H\_be;





// Compute the main part of our analytical means

Eigen::Matrix<double, 3, 3> alpha\_arg = ((dt\_2 / 2.0) \* eye3 + f\_1 \* w\_x + f\_2 \* w\_x\_2);

Eigen::Matrix<double, 3, 3> Beta\_arg = (delta\_t \* eye3 + f\_3 \* w\_x + f\_4 \* w\_x\_2);





// Matrices that will multiply the a\_hat in the update expressions

Eigen::MatrixXd H\_al = R\_tau12k \* alpha\_arg;

Eigen::MatrixXd H\_be = R\_tau12k \* Beta\_arg;



// Now compute the gyro bias Jacobian terms

double df\_1\_dbw\_1; double df\_1\_dbw\_2; double df\_1\_dbw\_3;

double df\_2\_dbw\_1; double df\_2\_dbw\_2; double df\_2\_dbw\_3;

double df\_3\_dbw\_1; double df\_3\_dbw\_2; double df\_3\_dbw\_3;

double df\_4\_dbw\_1; double df\_4\_dbw\_2; double df\_4\_dbw\_3;

当很小的时候，

，，，

double df\_1\_dw\_mag = -(pow(delta\_t, 5) / 15);

df\_1\_dbw\_1 = w\_1 \* df\_1\_dw\_mag;

df\_1\_dbw\_2 = w\_2 \* df\_1\_dw\_mag;

df\_1\_dbw\_3 = w\_3 \* df\_1\_dw\_mag;

double df\_2\_dw\_mag = (pow(delta\_t, 6) / 72);

df\_2\_dbw\_1 = w\_1 \* df\_2\_dw\_mag;

df\_2\_dbw\_2 = w\_2 \* df\_2\_dw\_mag;

df\_2\_dbw\_3 = w\_3 \* df\_2\_dw\_mag;

double df\_3\_dw\_mag = -(pow(delta\_t, 4) / 12);

df\_3\_dbw\_1 = w\_1 \* df\_3\_dw\_mag;

df\_3\_dbw\_2 = w\_2 \* df\_3\_dw\_mag;

df\_3\_dbw\_3 = w\_3 \* df\_3\_dw\_mag;

double df\_4\_dw\_mag = (pow(delta\_t, 5) / 60);

df\_4\_dbw\_1 = w\_1 \* df\_4\_dw\_mag;

df\_4\_dbw\_2 = w\_2 \* df\_4\_dw\_mag;

df\_4\_dbw\_3 = w\_3 \* df\_4\_dw\_mag;

若不是小量，









double df\_1\_dw\_mag = (pow(w\_dt, 2) \* sin\_wt - 3 \* sin\_wt + 3 \* w\_dt \* cos\_wt) / pow(mag\_w, 5);

df\_1\_dbw\_1 = w\_1 \* df\_1\_dw\_mag;

df\_1\_dbw\_2 = w\_2 \* df\_1\_dw\_mag;

df\_1\_dbw\_3 = w\_3 \* df\_1\_dw\_mag;

double df\_2\_dw\_mag = (pow(w\_dt, 2) - 4 \* cos\_wt - 4 \* w\_dt \* sin\_wt + pow(w\_dt, 2) \* cos\_wt + 4) / (pow(mag\_w, 6));

df\_2\_dbw\_1 = w\_1 \* df\_2\_dw\_mag;

df\_2\_dbw\_2 = w\_2 \* df\_2\_dw\_mag;

df\_2\_dbw\_3 = w\_3 \* df\_2\_dw\_mag;

double df\_3\_dw\_mag = (2 \* (cos\_wt - 1) + w\_dt \* sin\_wt) / (pow(mag\_w, 4));

df\_3\_dbw\_1 = w\_1 \* df\_3\_dw\_mag;

df\_3\_dbw\_2 = w\_2 \* df\_3\_dw\_mag;

df\_3\_dbw\_3 = w\_3 \* df\_3\_dw\_mag;

double df\_4\_dw\_mag = (2 \* w\_dt + w\_dt \* cos\_wt - 3 \* sin\_wt) / (pow(mag\_w, 5));

df\_4\_dbw\_1 = w\_1 \* df\_4\_dw\_mag;

df\_4\_dbw\_2 = w\_2 \* df\_4\_dw\_mag;

df\_4\_dbw\_3 = w\_3 \* df\_4\_dw\_mag;

，，

，



// Update alpha and beta gyro bias Jacobians

J\_a += J\_b \* delta\_t;

J\_a.block(0, 0, 3, 1) += (d\_R\_bw\_1 \* alpha\_arg + R\_tau12k \* (df\_1\_dbw\_1 \* w\_x - f\_1 \* e\_1x + df\_2\_dbw\_1 \* w\_x\_2 -f\_2 \* (e\_1x \* w\_x + w\_x \* e\_1x))) \* a\_hat;

J\_a.block(0, 1, 3, 1) += (d\_R\_bw\_2 \* alpha\_arg + R\_tau12k \* (df\_1\_dbw\_2 \* w\_x - f\_1 \* e\_2x + df\_2\_dbw\_2 \* w\_x\_2 -

f\_2 \* (e\_2x \* w\_x + w\_x \* e\_2x))) \* a\_hat;

J\_a.block(0, 2, 3, 1) += (d\_R\_bw\_3 \* alpha\_arg + R\_tau12k \* (df\_1\_dbw\_3 \* w\_x - f\_1 \* e\_3x + df\_2\_dbw\_3 \* w\_x\_2 -

f\_2 \* (e\_3x \* w\_x + w\_x \* e\_3x))) \* a\_hat;

J\_b.block(0, 0, 3, 1) += (d\_R\_bw\_1 \* Beta\_arg + R\_tau12k \* (df\_3\_dbw\_1 \* w\_x - f\_3 \* e\_1x + df\_4\_dbw\_1 \* w\_x\_2 -

f\_4 \* (e\_1x \* w\_x + w\_x \* e\_1x))) \* a\_hat;

J\_b.block(0, 1, 3, 1) += (d\_R\_bw\_2 \* Beta\_arg + R\_tau12k \* (df\_3\_dbw\_2 \* w\_x - f\_3 \* e\_2x + df\_4\_dbw\_2 \* w\_x\_2 -

f\_4 \* (e\_2x \* w\_x + w\_x \* e\_2x))) \* a\_hat;

J\_b.block(0, 2, 3, 1) += (d\_R\_bw\_3 \* Beta\_arg + R\_tau12k \* (df\_3\_dbw\_3 \* w\_x - f\_3 \* e\_3x + df\_4\_dbw\_3 \* w\_x\_2 -

f\_4 \* (e\_3x \* w\_x + w\_x \* e\_3x))) \* a\_hat;

由得，

// Going to need orientation at intermediate time i.e. at .5\*dt;

Eigen::Matrix<double, 3, 3> R\_mid = small\_w ? eye3 - .5 \* delta\_t \* w\_x + (pow(.5 \* delta\_t, 2) / 2) \* w\_x\_2 :eye3 - (sin(mag\_w \* .5 \* delta\_t) / mag\_w) \* w\_x + ((1.0 - cos(mag\_w \* .5 \* delta\_t)) / (pow(mag\_w, 2.0))) \* w\_x\_2;

R\_mid = R\_mid \* R\_k2tau;



用四阶龙格库塔法求协方差，分，

//Compute covariance (in this implementation, we use RK4)

//k1-------------------------------------------------------------------------------------------------

// Build state Jacobian

Eigen::Matrix<double, 15, 15> F\_k1 = Eigen::Matrix<double, 15, 15>::Zero();

F\_k1.block(0, 0, 3, 3) = -w\_x;

F\_k1.block(0, 3, 3, 3) = -eye3;

F\_k1.block(6, 0, 3, 3) = -R\_k2tau.transpose() \* a\_x;

F\_k1.block(6, 9, 3, 3) = -R\_k2tau.transpose();

F\_k1.block(12, 6, 3, 3) = eye3;

// Build noise Jacobian

Eigen::Matrix<double, 15, 12> G\_k1 = Eigen::Matrix<double, 15, 12>::Zero();

G\_k1.block(0, 0, 3, 3) = -eye3;

G\_k1.block(3, 3, 3, 3) = eye3;

G\_k1.block(6, 6, 3, 3) = -R\_k2tau.transpose();

G\_k1.block(9, 9, 3, 3) = eye3;

// Get covariance derivative

Eigen::Matrix<double, 15, 15> P\_dot\_k1 = F\_k1 \* P\_meas + P\_meas \* F\_k1.transpose() + G\_k1 \* Q\_c \* G\_k1.transpose();

//k2-------------------------------------------------------------------

// Build state Jacobian

Eigen::Matrix<double, 15, 15> F\_k2 = Eigen::Matrix<double, 15, 15>::Zero();

F\_k2.block(0, 0, 3, 3) = -w\_x;

F\_k2.block(0, 3, 3, 3) = -eye3;

F\_k2.block(6, 0, 3, 3) = -R\_mid.transpose() \* a\_x;

F\_k2.block(6, 9, 3, 3) = -R\_mid.transpose();

F\_k2.block(12, 6, 3, 3) = eye3;

// Build noise Jacobian

Eigen::Matrix<double, 15, 12> G\_k2 = Eigen::Matrix<double, 15, 12>::Zero();

G\_k2.block(0, 0, 3, 3) = -eye3;

G\_k2.block(3, 3, 3, 3) = eye3;

G\_k2.block(6, 6, 3, 3) = -R\_mid.transpose();

G\_k2.block(9, 9, 3, 3) = eye3;

// Get covariance derivative

Eigen::Matrix<double, 15, 15> P\_k2 = P\_meas + P\_dot\_k1 \* delta\_t / 2.0;

Eigen::Matrix<double, 15, 15> P\_dot\_k2 = F\_k2 \* P\_k2 + P\_k2 \* F\_k2.transpose() + G\_k2 \* Q\_c \* G\_k2.transpose();

//k3-------------------------------------------------------------------

// Our state and noise Jacobians are the same as k2

// Since k2 and k3 correspond to the same estimates for the midpoint

Eigen::Matrix<double, 15, 15> F\_k3 = F\_k2;

Eigen::Matrix<double, 15, 12> G\_k3 = G\_k2;

// Get covariance derivative

Eigen::Matrix<double, 15, 15> P\_k3 = P\_meas + P\_dot\_k2 \* delta\_t / 2.0;

Eigen::Matrix<double, 15, 15> P\_dot\_k3 = F\_k3 \* P\_k3 + P\_k3 \* F\_k3.transpose() + G\_k3 \* Q\_c \* G\_k3.transpose();

//k4-------------------------------------------------------------------

// Build state Jacobian

Eigen::Matrix<double, 15, 15> F\_k4 = Eigen::Matrix<double, 15, 15>::Zero();

F\_k4.block(0, 0, 3, 3) = -w\_x;

F\_k4.block(0, 3, 3, 3) = -eye3;

F\_k4.block(6, 0, 3, 3) = -R\_k2tau1.transpose() \* a\_x;

F\_k4.block(6, 9, 3, 3) = -R\_k2tau1.transpose();

F\_k4.block(12, 6, 3, 3) = eye3;

// Build noise Jacobian

Eigen::Matrix<double, 15, 12> G\_k4 = Eigen::Matrix<double, 15, 12>::Zero();

G\_k4.block(0, 0, 3, 3) = -eye3;

G\_k4.block(3, 3, 3, 3) = eye3;

G\_k4.block(6, 6, 3, 3) = -R\_k2tau1.transpose();

G\_k4.block(9, 9, 3, 3) = eye3;

// Get covariance derivative

Eigen::Matrix<double, 15, 15> P\_k4 = P\_meas + P\_dot\_k3 \* delta\_t;

Eigen::Matrix<double, 15, 15> P\_dot\_k4 = F\_k4 \* P\_k4 + P\_k4 \* F\_k4.transpose() + G\_k4 \* Q\_c \* G\_k4.transpose();

求平均协方差和旋转矩阵（四元数表示）。

// Collect covariance solution

// Ensure it is positive definite

P\_meas += (delta\_t / 6.0) \* (P\_dot\_k1 + 2.0 \* P\_dot\_k2 + 2.0 \* P\_dot\_k3 + P\_dot\_k4);

P\_meas = 0.5 \* (P\_meas + P\_meas.transpose());

// Update rotation mean

// Note we had to wait to do this, since we use the old orientation in our covariance calculation

R\_k2tau = R\_k2tau1;

q\_k2tau = rot\_2\_quat(R\_k2tau);