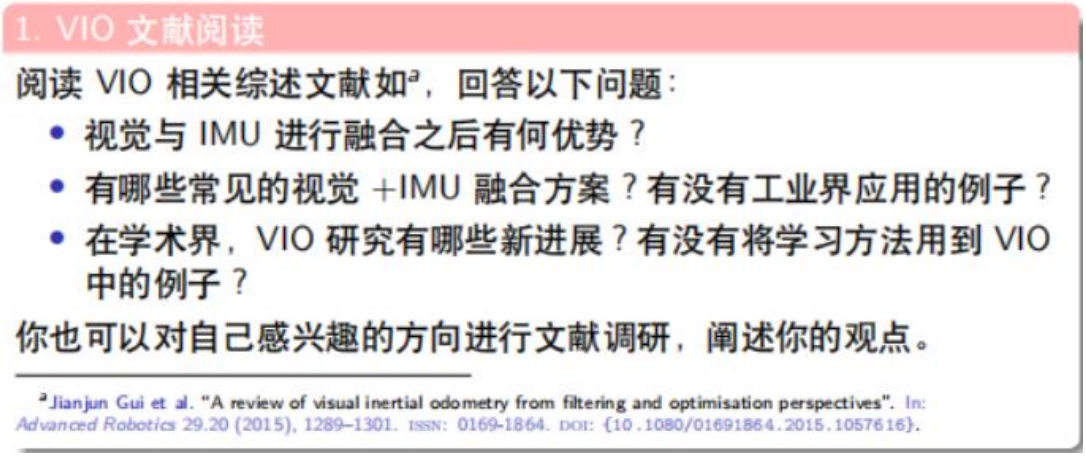


1 视觉与 IMU 融合之后有何优势?

	视觉	IMU
静止状态	初始化后，数据稳定	通常会产生较大偏移
缓慢运动	特征点数量稳定状态下，数据输出稳定	IMU 可以准确计算出运动
快速运动	可能导致特征点丢失或者两帧之间重叠区域太少以至于无法进行特征匹配	IMU 估计较为准确

(1) 当图像发生变化时，本质上我们没法知道是相机自身发生了运动，还是外界条件发生了变化，所以 VSLAM 难以处理动态的障碍物。而 IMU 能够检测到自身运动信息，从某种程度上减轻动态物体的影响。

(2) 对于单目视觉 SLAM，存在尺度不确定性，融合 IMU odmetry 后可以恢复尺度。

(3) 纯视觉 SLAM 在容易受弱纹理场景和光照变化的影响，在定位失败时，可以依靠 IMU 进行短暂的定位。

综上，Visual 与 IMU 融合之后会弥补各自的劣势，可利用 Visual 来估计 IMU 的零偏，减少 IMU 由零偏导致的发散和累积误差。IMU 可以为 Visual 提供快速运动时的定位，以及因为某种因素（场景特征点较少，光照变化较大等）定位失败时的状态估计。

2 有哪些常见的视觉+IMU 融合方案？有没有工业界应用的例子?

（1）常见的视觉+IMU 融合方案

MSCKF(https://github.com/KumarRobotics/msckf_vio.git),
(https://github.com/yuzhou42/MSCKF.git 中文注释版),
MSCKF_mono(https://github.com/daniilidis-group/msckf_mono.git),
OKVIS(https://github.com/ethz-asl/okvis_ros.git),
ROVIO(https://github.com/ethz-asl/rovio.git),
VIORB(https://github.com/jingpang/LearnVIORB.git),
VINS-Mono(https://github.com/HKUST-Aerial-Robotics/VINS-Mono.git),
VINS-Mobile(https://github.com/HKUST-Aerial-Robotics/VINS-Mobile.git),
VINS-Fusion(https://github.com/HKUST-Aerial-Robotics/VINS-Fusion.git)

Visual-Odometry-Review

https://github.com/MichaelBeechan/Visual-Odometry-Review

（2）工业界应用
Google：Tango，ARCore
Apple：ARKit
Microsoft：HoloLens
百度：DuMix AR
3 在学术界，VIO 研究有哪些新进展？有没有将学习方法应用到 VIO 的例子？
Jianjun Gui, Dongbing Gu. A review of visual inertial odometry from filtering and optimization perspectives. [J] Advanced Robotics, 2015
Chang Chen, Hua Zhu. A Review of Visual-Inertial Simultaneous Localization and Mapping from Filtering-Based and Optimization-Based Perspectives. [J] Robotics, 2018
Chen, Changhao, et al. "Selective Sensor Fusion for Neural Visual-Inertial Odometry." arXiv preprint arXiv:1903.01534 (2019).
Lee, Hongyun, Matthew McCrink, and James W. Gregory. "Visual-Inertial Odometry for Unmanned Aerial Vehicle using Deep Learning." AIAA Scitech 2019 Forum. 2019.

2. 四元数和李代数更新

课件提到了可以使用四元数或旋转矩阵存储旋转变量。当我们用计算出来的 ω 对某旋转更新时，有两种不同方式：

$$\begin{aligned} &\mathbf{R} \leftarrow \mathbf{R}\exp\left(\boldsymbol{\omega}^{\wedge}\right) \\ \text{或}\quad &\mathbf{q} \leftarrow \mathbf{q} \otimes \left[1, \tfrac{1}{2}\boldsymbol{\omega}\right]^{\mathrm{T}} \end{aligned} \tag{20}$$

请编程验证对于小量 $\omega = [0.01, 0.02, 0.03]^{\mathrm{T}}$，两种方法得到的结果非常接近，实践当中可视为等同。因此，在后文提到旋转时，我们并不刻意区分旋转本身是 $\mathbf{q}$ 还是 $\mathbf{R}$，也不区分其更新方式为上式的哪一种。

代码如下，库文件引用：

```
#include <iostream>
#include <cmath>
#include <Eigen/Core>
#include <Eigen/Eigen>
#include <Eigen/Dense>
#include <ctime>
#include <sophus/so3.hpp>
```

```cpp
int main(int argv, char ** argc) {
    ///define a minimized vecter
    Eigen::Vector3d w(0.01, 0.02, 0.03);//[0.01, 0.02, 0.03]^T
    Eigen::AngleAxisd rotation_vetor = Eigen::AngleAxisd(M_PI/2, Eigen::Vector3d(0,0,1));
    Eigen::Matrix3d R = rotation_vetor.toRotationMatrix();
    //std::cout << R * R.transpose() << std::endl;
    Sophus::SO3d SO3_R(R);

    // Quaternion update
    // 四元数更新，使用Eigen库
    clock_t time_quaternion = clock();
    std::cout << "================================" << std::endl;
    Eigen::Quaterniond q = Eigen::Quaterniond(R);//build the R_Quaternion
    //std::cout << "quaternion from rotation vector : " << q.coeffs().transpose() << std::endl;
    Eigen::Quaterniond w_quaternion(1, w[0] / 2, w[1] / 2, w[2] / 2);//w 2 quaternion
    Eigen::Quaterniond q_update = (q * w_quaternion).normalized();//update the quaternion & normalized
    std::cout << "Eigen Quaternion updated : " << std::endl << q_update.toRotationMatrix() << std::endl;
    std::cout << "使用Eigen库进行Quaternion更新 use time " << 1000 * (clock() - time_quaternion) / (double) CLOCKS_PER_SEC << "ms" << std::endl;


    // 李代数更新
    // 使用Sophus库
    clock_t time_Sophus_so3 = clock();
    std::cout << "================================" << std::endl;
    Eigen::Vector3d update_so3(0.01, 0.02, 0.03); //更新量
    Sophus::SO3d SO3_updated = SO3_R * Sophus::SO3d::exp(update_so3);
    std::cout<<"Sophus SO3 updated = "<< std::endl << SO3_updated.matrix() <<std::endl;
    std::cout << "使用Sophus库进行SO3更新 use time " << 1000 * (clock() - time_Sophus_so3) / (double) CLOCKS_PER_SEC << "ms" << std::endl;

    // 使用Eigen库
    clock_t time_so3 = clock();
    std::cout << "================================" << std::endl;
    double Theta = sqrt(w[0] * w[0] + w[1] * w[1] + w[2] * w[2]);        // Theta = sqrt(x ^ 2 + y ^ 2 + z ^ 2)
    Eigen::Vector3d a = w.array() / Theta;                               // vecter_a = w / Theta
    Eigen::Matrix3d a_hat,I;                                             //            0    -a3      a2
    a_hat <<          0,       -w[2] / Theta,    w[1] / Theta,           // a_hat <<   a3      0     -a1
            w[2] / Theta,            0,        -w[0] / Theta,          //           -a2     a1      0
           -w[1] / Theta,    w[0] / Theta,          0;
    I << 1, 0, 0,
         0, 1, 0,
         0, 0, 1;
    //Rodrigues's formula: cos(Theta) * I + (1 - cos(Theta)) *  vecter_a * vecter_a ^ T + sin(Theta) * a_hat
    Eigen::Matrix3d exp_w_hat = cos(Theta) * I.array() + (1 - cos(Theta)) * (a * a.transpose()).array() + a_hat.array() * sin(Theta);
    //update:
    Eigen::Matrix3d R_update = R * exp_w_hat;
    std::cout << "Eigen SO3 updated : " << std::endl << R_update << std::endl;
    std::cout << "使用Eigen库进行SO3更新 use time " << 1000 * (clock() - time_so3) / (double) CLOCKS_PER_SEC << "ms" << std::endl;

    return 0;
}
```

结果可以看出，在小量的前提下，不管用旋转矩阵或者四元数进行更新差别不大，且四元数在 Eigen 库中的实现可将速度提升 5 倍左右。

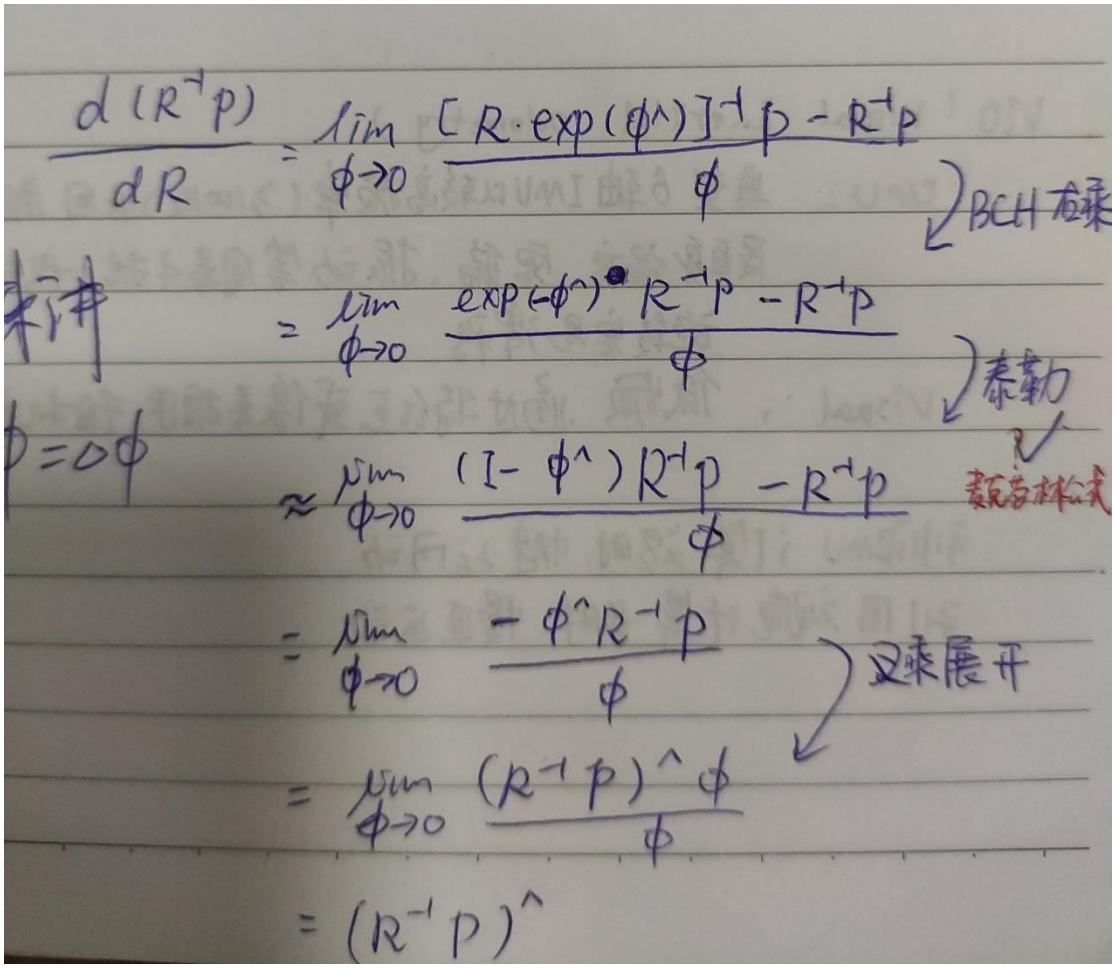
```
duke@duke:~/Documents/VIO$ ./Week0
================================
Eigen Quaternion updated :
-0.0300895     -0.9995 0.00969661
   0.99935 -0.0298895  0.0201429
-0.0198431  0.0102964     0.99975
使用Eigen库进行Quaternion更新 use time 0.232ms
================================
Sophus SO3 updated =
 -0.030093     -0.9995  0.0096977
   0.99935  -0.029893  0.0201453
-0.0198454  0.0102976     0.99975
使用Sophus库进行SO3更新 use time 0.028ms
================================
Eigen SO3 updated :
 -0.030093     -0.9995  0.0096977
   0.99935  -0.029893  0.0201453
-0.0198454  0.0102976     0.99975
使用Eigen库进行SO3更新 use time 0.064ms
```

3. 其他导数

使用右乘 $\mathfrak{so}(3)$，推导以下导数：

$$\frac{\mathrm{d}\left(\mathbf{R}^{-1}\mathbf{p}\right)}{\mathrm{d}\mathbf{R}} \tag{21}$$

$$\frac{\mathrm{d}\ln\left(\mathbf{R}_1\mathbf{R}_2^{-1}\right)^{\vee}}{\mathrm{d}\mathbf{R}_2} \tag{22}$$



$$\frac{d\ln(R_1R_2^{-1})^\vee}{dR_2} = \lim_{\phi\to 0}\frac{\ln[R_1\cdot(R_2\cdot\exp(\phi^\wedge))^{-1}]^\vee - \ln(R_1\cdot R_2^{-1})^\vee}{\phi}$$

$$= \lim_{\phi\to 0}\frac{\ln[R_1\cdot\exp(-\phi^\wedge)R_2^{-1}]^\vee - \ln(R_1\cdot R_2^{-1})^\vee}{\phi}$$

$$= \lim_{\phi\to 0}\frac{\ln[\exp(-R_2\phi)^\wedge R_1R_2^{-1}]^\vee - \ln(R_1R_2^{-1})^\vee}{\phi}$$

$$= \lim_{\phi\to 0}\frac{-J_r^{-1}R_2\phi + \ln(R_1R_2^{-1})^\vee - \ln(R_1R_2^{-1})^\vee}{\phi}$$

$$= \lim_{\phi\to 0}\frac{-J_r^{-1}R_2\phi}{\phi}$$

$$= -J_r^{-1}R_2$$