**VINS学习笔记**

**1.最小二乘问题的求解**

**1.1、最速下降法、牛顿法**

最小二乘问题的**基本定义**为找到一个维变量，使得损失函数取局部最小值：

其中是残差函数，比如测量值与预测值之间的差，且有。局部最小值对任意有。

假设损失函数是平滑且可导的，可以得到损失函数的二阶泰勒展开为：

其中和分别为损失函数对变量的一阶导和二阶导矩阵。

由上述的损失函数二阶泰勒展开的形式，忽略高阶项后可将该损失函数看作二次函数，因此可以得到损失函数以下性质

1. 如果在点处有导数为0，则称这个点为稳定点。
2. 在点处对应的Hessian为H：若H为正定矩阵，则它的特征值都大于0，则在处有为局部最小值；若H为负定矩阵，则它的特征值都小于0，则在处为局部最大值；若H为不定矩阵，则它的特征值大于0也有小于0的，则处为鞍点。

最小二乘问题在问题为线性问题时，可以直接通过一步矩阵运算求解变量；若问题为非线性问题时，需要将其使用泰勒展开展开为线性问题之后再进行求解，而泰勒展开的过程存在误差，因此需要经过反复迭代的过程求解变量。迭代法的初衷是寻找一个下降方向使得损失函数随的迭代逐渐减小，知道收敛到。我们通过两个步骤确定迭代的总体过程：

1. 寻找下降方向单位向量；
2. 确定下降步长；

假设足够小，我们可以对损失函数进行一阶泰勒展开只需要寻找下降方向满足，通过line Search方法找到下降的步长：。

以下，将分别讨论最速下降法和牛顿法，作为上述迭代法的展开。

**最速下降法**适用于迭代的开始阶段，从下降的条件可以知道：，表示下降方向和梯度方向的夹角。当时，有，即梯度的负方向为最速下降方向。该方法由于仅仅控制迭代方向而为控制下降步长，导致迭代会在最优质附近产生震荡现象（ZigZag），从而使得收敛变慢。

**牛顿法**适用于最优值附近的优化过程，在局部最优点附近，如果有是最优解，则损失函数对的导数为0，则对损失函数的二阶泰勒展开求导可得：

得到。但是这种方法需要计算损失函数的二阶导，计算较为复杂，效率较低。

* 1. **LM算法的具体实现**

LM方法在高斯牛顿法的基础上进行实现，因此，我们先对高斯牛顿法进行介绍：高斯牛顿法是对牛顿法的改进，牛顿法中由于需要计算损失函数对待优化量的二阶导，导致迭代的效率大大降低，高斯牛顿法通过使用一阶的雅可比矩阵去近似二阶导，提高了迭代的效率，其主要的公式如下，对于残差，其一阶泰勒展开可写为：

注意这里的与上面的定义不同，这里指的是残差函数的一阶导，将上式带入损失函数有：

这样，损失函数就近似成了一个二次函数，并且如果雅可比矩阵是满秩的，则正定，此时损失函数具有最小值，另外，也可以得到、。令损失函数一阶导为0，得到，这就是通常论文中得到迭代方程。

由于高斯牛顿法中，并不能保证是正定矩阵，事实上，该矩阵常常是半正定形式。因此，我们在中添加阻尼因子，使得必定满足正定形式。我们把添加了阻尼因子的高斯牛顿法叫做LM方法，得到。

总结以上内容，可以得到阻尼因子的作用如下：

1. 保证正定，迭代朝着下降方向进行；
2. 非常大，则，接近最速下降法；
3. 比较小，则，接近高斯牛顿法。

阻尼因子是LM方法中的关键，因此，在LM算法之初，我们应该设定一个较为合适的阻尼因子，并且选取一种较为合适的阻尼因子更新策略，保证LM算法较快收敛到变量最优解。阻尼因子是为添加到中的量，因此阻尼因子的大小应该是相对于而言。半正定的信息矩阵特征值和对应的特征向量为。对做特征值分解得到：可得：

所以，一个简单的，通常按需设定

阻尼因子如何随LM算法迭代，以下我们通过定性以及定量分析两种过程详细说明。**定性分析**：若使得损失函数增大，此时应该增大减小迭代步长，加快下降速度并且拒绝本次迭代过程；若使得损失函数减小，此时应该减小增大步长，加快收敛速度并且接受此次迭代。**定量分析：**阻尼因子的更新策略是通过比例因子来确定的：

=

首先比例因子的分母始终大于0，若此时，则上升，应该增大减小步长，以最快的速度使得损失函数下降；若且比较大，此时应该减小，让LM算法接近高斯牛顿法，使得系统更快收敛；反之如果是比较小的正数，依然需要增大阻尼，缩小迭代步长使得损失函数尽可能多地下降。

实际运用中，我们常常使用Niesen策略对阻尼因子进行更新，这种方法被运用于g2o和ceres中。具体如下：

1. 若，
2. 若，
   1. **鲁棒核函数的实现**

当最小二乘函数中遇到outlier时，若不对outlier进行处理，将导致较大偏差。常见outlier处理方法有RANSAC以及鲁棒核函数方法。由于RANSAC方法与非线性优化均需要采用迭代过程，效率不高，因此，此处我们介绍鲁棒核函数的方法对outlier进行处理。

鲁棒核函数直接作用于残差上，最小二乘函数变成了如下形式：

将平方项记作，则鲁棒核函数进行二阶泰勒展开有：

上述函数中的计算稍微复杂一些：

上述公式代入到鲁棒核函数泰勒展开式中有：

对上式求和后，对变量求导，令其为0，得到：

常用的鲁棒核函数有Cauchy函数和Huber函数，下面以Cauchy鲁棒核函数为例，该核函数形式为：

其中为控制参数。对的一阶导和二阶导为：

核函数控制参数的设定可参考论文“At all costs: A comparison of robust cost functions for camera correspondence outliers IEEE”

**2.VIO残差的构建**

* 1. **视觉重投影误差**

**2.2、IMU重投影误差**

**3.VIO残差雅可比的推导**

* 1. **视觉重投影误差的Jacobian**
  2. **IMU预积分残差的Jacobian**

**4.Windows安装evo工具**

**5.VIO系统初始化**

VINS中采用松耦合的方式获取系统的初始值。首先纯视觉SFM求解滑动窗口中所有帧的位姿（以第l帧作为参考帧）和所有路标点的3D坐标。然后将SFM的结果与IMU预积分的值进行对齐实现：陀螺仪偏置的校正；求解每一图像帧对应的速度、重力向量和尺度因子；重力向量的精调。需要注意的是，在初始化过程中，并没有对加速度计的偏置进行校正，这是因为重力是初始化过程中需要求解的量，而加速度计偏置与重力耦合，而且系统的加速度偏置相对于重力加速度很小，所以加速度偏置在初始化过程中很难观测，因此初始化过程中不考虑加速度计偏置的校正。

**5.1、相机与IMU之间相对旋转**

相对旋转是相机与IMU之间外参的一部分。相机与IMU之间的旋转标定非常重要，偏差1~2度系统的精度就会变得很低。

通过视觉SFM获取了相邻两关键帧之间相机的相对旋转，通过IMU预积分获取了相邻两关键帧之间IMU坐标系的相对旋转。设IMU与相机之间的相对旋转，则对于任意，下面等式成立：

四元数形式可写为：

将上面的四元数形式写成矩阵形式，有：

对于N对相对旋转的测量值，可以得到约束的线性方程：

其中，为观测值的权重，计算方式为：

其中定义为观测值的残差：

求解时，取信息矩阵的最小特征值对应的特征向量即为所求的旋转矩阵外参。然而，实际问题中，由于各种噪声的影响，为了得到更加鲁棒的外参参与后续VIO状态估计，常常记录信息矩阵倒数第二小的特征值，并且认为该特征值需要大于一定的阈值，此时认为旋转矩阵外参求解得到的值较为鲁棒，受噪声的影响在可控范围之内。

求解这个方程时，由于没有较好的初始值，因此，我们需要迭代一定次数，待求参数收敛后，结束该求解过程。

**5.2、单目相机初始化（SFM）**

为了获取足够的相机数据参与后续VIO状态估计，需要先求解出一些相机姿态以及路标点将VIO系统固定在三维空间中。我们将滑动窗口中的所有图像帧进行SFM计算，将第l帧作为参考帧确定视觉系统的坐标参考框架。通过逐帧三角化以及PnP的求解，恢复滑窗中所有图像帧的位姿以及路标点的三维坐标。此时求解的位姿和路标点，由于观测误差的存在，将带有一定的累计误差。我们通过非线性优化将这些状态量进行优化。由于单目视觉问题在优化过程中存在7个自由度：分别为三个旋转自由度、三个平移自由度以及一个尺度自由度，这使得优化后的系统参考系会存在整体漂移情况，这不符合我们对参考框架的定义；因此，在SFM优化中，我们固定第ｌ帧以及当前帧的姿态，从而消除单目优化问题中的７个自由度。

SFM中l帧需要预先计算，其计算方法为：在滑动窗口中搜索与当前帧具有一定视差以及一定数量特征匹配点对的图像帧，作为SFM的参考帧，并且通过本质矩阵的求解恢复参考帧与当前帧的位姿变换。

纯视觉SFM获得了以第ｌ帧为参考帧的三维空间（），并且可以得到滑动窗口中所有关键帧的位姿，其中为从坐标系到坐标系的位移向量在坐标系的表示，此坐标无绝对尺度信息。假设已知相机与IMU之间的外参，其中为从本体坐标系到相应时刻相机坐标系的位移向量在本体坐标系的表示。按照下面公式，可完成从本体坐标系到坐标系的转换。

以上的推导较为直接，重点关注的推导。的物理含义为坐标系到坐标系的位移向量在坐标系的表示，的计算公式实际表示的是坐标系下，各个向量的加法运算，如下图所示：

C0

B0

C1

Ck

B1

Bk

由图中内容可知：，即为，因此，我们需要做的是如何将转化为，对于三维空间的位姿变换，我们常常将旋转和平移写在一起形成一个四维的位姿变换矩阵，有：

因此，，代入到公式中，即可的得到。表示给视觉SFM结果赋予的尺度信息。

**5.3、SFM结果与IMU预积分结果对齐**

得到纯视觉SFM以及对应时刻的预积分结果后，我们需要将视觉信息与IMU信息对齐在同一坐标框架之上，尤其是要将重力向量在我们定义的坐标系中的位置以及视觉尺度因子准确估计，其他的信息如陀螺仪偏置的校正，IMU初始化速度的求解。

**5.3.1、陀螺仪偏置校正**

考虑滑动窗口中连续两个关键帧和，通过SFM可以获取两个关键帧时刻本体坐标系相对于参考相机坐标系的旋转和，通过IMU预积分可以获取两个关键帧时刻本体坐标系之间的相对旋转约束，校正陀螺仪的目标函数为：

其中，姿态预积分结果可泰勒展开为：

表示滑窗中的所有关键帧，为姿态预积分量关于陀螺仪偏置误差的一阶导。

校正陀螺仪的目标函数的最小值为单位矩阵对应的单位四元数，因此可以将上式写为：

只考虑上述四元数的虚部，可以得到：

为便于求解，将上述公式信息阵转换为正定矩阵：

此时，直接使用cholesky分解就可以得到陀螺仪偏置校正结果。陀螺仪偏置校正后，需要对IMU测量值重新预积分。

**5.3.2、初始化速度、重力向量和尺度因子**

此处需要求解出滑窗中所有关键帧的初始速度、重力向量以及尺度因子，因此，优化变量为：

IMU预积分项中，位置和速度预积分量分别为：

使用SFM中得到的无尺度位移向量信息，并且将速度转换为相应时刻本体坐标系下的表示，上式可转换为：

将待求的速度、重力向量以及尺度因子写在一边，得到如下形式：

理论上，上述等式应该严格成立，但是由于各种误差的影响与数值运算的误差，等式并不能严格成立，将理论真值、替换为通过IMU测量值计算出的标称值、。

此时令：

此时最小二乘的目标函数变为：

通过求解上述最小二乘问题，就可以得到滑窗中所有关键帧在对应时刻本体坐标系下的速度、重力向量在世界坐标系中的表示以及SFM结果的尺度因子。

**5.3.3、重力向量精调**

重力向量的模值固定，并且数值也是已知的，由于各种误差，上一节中求解得到的重力向量与真实重力向量可能存在一些差距，因此，我们必须对重力向量进行再优化。比较容易想到的是，固定求解得到重力向量的方向并将其单位化，将已知的重力向量模值与其相乘得到新的重力向量作为精调的结果。这种方法潜在地认为重力向量求解过程中误差只存在于重力模值，实际上，在重力方向上也有一定的误差存在。

因此，重力精调时我们必须同时考虑模值与方向，采用的方法为将重力向量在其正切空间重新参数化，并且固定重力模值，此时，重力向量的自由度由3变成了2 ，重力向参数化形式为：

为重力向的方向向量在坐标系下的坐标，、为重力向量正切空间的一对正交基，重新参数化后的重力向量由控制。

将重新参数化后的重力向量代入5.3.2节中，得到：

然后使用上一节中的方法，求解一个最小二乘问题，从而实现了重力的精调。需要注意的是，为了保证求解结果的精确，我们同样需要对迭代该求解过程，直到重力收敛为止。这样我们就得到了在系下的重力向量，通过将其旋转到惯性坐标系的轴方向，可以计算得到相机系到惯性系下的旋转矩阵，此时可以将所有变量都调整至惯性世界系中。

1. **VIO系统代码实现工具**

**6.1、可视化工具**

**6.2、优化库的使用**

**6.3、C++标准库**

**6.3.1、智能指针**

智能指针用于解决C++中存在的两类容易出现的指针错误使用情况，其一为使用new在堆上创建了指针所指向的内存空间，在该指针使用完毕后，忘记使用delete对指针所指向的内存空间进行析构，造成程序内存泄露；其二是当存在两个或多个指针指向同一块内存空间时，对这多个指针的所指向内存空间进行析构，会造成同一块内存被析构多次的现象，这显然非法的。

为了解决以上两种指针使用问题：提出了三种解决方案对其进行稳妥处理：1、定义赋值运算符，使之执行深拷贝，这样多个指针将指向不同的内存空间；2、建立所有权（ownership），对于特定的对象，只能有一个智能指针可以拥有它，这样只有拥有对象的智能指针的构造函数会析构对象对应的内存空间，并且，使用赋值操作转让所有权，这便是unique\_ptr的策略；3、创建智能更高的指针，跟踪所指向特定对象的智能指针数量。这称为引用计数，每多一个智能指针指向特定对象时，引用计数加1，当引用技术为0时，才调用delete析构对象内存空间，这便是shared\_ptr的策略；

1. **图像处理基础**

**7.1、本质矩阵求解方法（Essential Matrix）**

**7.1.1、八点法**

对于本质矩阵，考虑相机归一化平面下的点对：，依据对极几何关系，我们有：



将矩阵E线性展开，写成向量的形式有：



那么对于一对点对，对极几何约束可以写成如下形式：



同理，我们将所有的其他点对也同样表示，然后把所有的点放在同一个方程中，可以得到线性方程组，表示第i个特征点。



如果考虑尺度等价性，即的基础解析是一条直线，那么至少需要8对点就可以计算出。此时系数矩阵的大小为8行9列，如果系数矩阵是满秩的，即秩为8，那么该方程的零空间维度为1，也就是解构成一条直线，与的尺度等价性是一致的，通常，此时可以利用最小二乘方法来求解的最优值。

其最优为系数矩阵最小特征值对应的特征向量，并且为了满足E的奇异值为的性质，在求解出矩阵E后需要进行SVD分解，设定第一和第二个特征值相等，并设定第三个奇异值为0；

ORB\_SLAM中对于E矩阵求解的源代码为：

Cv::Mat ComputeF21(**const** vector**<**cv**::**Point2f**>** **&**vP1,**const** vector**<**cv**::**Point2f**>** **&**vP2) {

**const** **int** N **=** vP1.size();

cv**::**Mat A(N,9,CV\_32F);

**for**(**int** i**=**0; i**<**N; i**++**) {

**const** **float** u1 **=** vP1[i].x;

**const** **float** v1 **=** vP1[i].y;

**const** **float** u2 **=** vP2[i].x;

**const** **float** v2 **=** vP2[i].y;

A.at**<float>**(i,0) **=** u2**\***u1;

A.at**<float>**(i,1) **=** u2**\***v1;

A.at**<float>**(i,2) **=** u2;

A.at**<float>**(i,3) **=** v2**\***u1;

A.at**<float>**(i,4) **=** v2**\***v1;

A.at**<float>**(i,5) **=** v2;

A.at**<float>**(i,6) **=** u1;

A.at**<float>**(i,7) **=** v1;

A.at**<float>**(i,8) **=** 1;

}

cv**::**Mat u,w,vt;

cv**::**SVDecomp(A,w,u,vt,cv**::**SVD**::**MODIFY\_A **|** cv**::**SVD**::**FULL\_UV);

cv**::**Mat Fpre **=** vt.row(8).reshape(0, 3);

cv**::**SVDecomp(Fpre,w,u,vt,cv**::**SVD**::**MODIFY\_A **|** cv**::**SVD**::**FULL\_UV);

w.at**<float>**(2)**=**0;

**return** u**\***cv**::**Mat**::**diag(w)**\***vt;

}

**7.1.2、五点法**

由八点法的推导，可以得到以下方程约束：



由于本质矩阵由平移和旋转构成，具有6个自由度约束，本质矩阵本身没有尺度，因此本质矩阵具有5个自由度约束，理论上最少需要5对点对就可以求解出本质矩阵，设 。

是一个5行9列的矩阵，秩为5，因此零空间的自由度为4，利用SVD分解可以求出零空间的基向量，记为，那么的通解可以表示成的线性组合，其中，为待求的参数。



同时本质矩阵满足以下约束条件：





由于尺度等价性，可以设为1，并分别将带入到上面两个公

式中得到10个约束方程，是一个约束方程，等式左边是一个3\*3的矩阵，每一个元素都等于0，那么每一个元素就对应一个约束方程，由于总共是3个矩阵的乘法，因此最高项是3次，按照以下方式将系数进行排序（这里的排序方式可以有多种选择，此处为了保持与代码表现一致），可以得到10\*20的系数多项式矩阵。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | **9** | **10** | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|  | XXX | XXY | XYY | YYY | XXZ | XYZ | YYZ | XZZ | YZZ | **ZZZ** | **XX** | XY | YY | XZ | YZ | ZZ | X | Y | Z | 1 |
| 0 | \* | \* | \* | \* | \* | \* | \* | \* | \* | **\*** | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 1 | \* | \* | \* | \* | \* | \* | \* | \* | \* | **\*** | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 2 | \* | \* | \* | \* | \* | \* | \* | \* | \* | **\*** | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 3 | \* | \* | \* | \* | \* | \* | \* | \* | \* | **\*** | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 4 | \* | \* | \* | \* | \* | \* | \* | \* | \* | **\*** | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 5 | \* | \* | \* | \* | \* | \* | \* | \* | \* | **\*** | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 6 | \* | \* | \* | \* | \* | \* | \* | \* | \* | **\*** | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 7 | \* | \* | \* | \* | \* | \* | \* | \* | \* | **\*** | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 8 | \* | \* | \* | \* | \* | \* | \* | \* | \* | **\*** | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 9 | \* | \* | \* | \* | \* | \* | \* | \* | \* | **\*** | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |

这是一个关于x,y,z的三元三次方程组，为了求解出x,y,z，先进行高斯消元。由于方程组的秩为10，因此可以高斯约当消元得到以下形式，空白处表示系数为0，\*表示系数是一个已知的具体数值。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | **9** | **10** | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|  | XXX | XXY | XYY | YYY | XXZ | XYZ | YYZ | XZZ | YZZ | **ZZZ** | **XX** | XY | YY | XZ | YZ | ZZ | X | Y | Z | 1 |
| 0 | 1 |  |  |  |  |  |  |  |  |  | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 1 |  | 1 |  |  |  |  |  |  |  |  | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 2 |  |  | 1 |  |  |  |  |  |  |  | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 3 |  |  |  | 1 |  |  |  |  |  |  | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 4 |  |  |  |  | 1 |  |  |  |  |  | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 5 |  |  |  |  |  | 1 |  |  |  |  | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 6 |  |  |  |  |  |  | 1 |  |  |  | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 7 |  |  |  |  |  |  |  | 1 |  |  | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 8 |  |  |  |  |  |  |  |  | 1 |  | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| 9 |  |  |  |  |  |  |  |  |  | **1** | **\*** | \* | \* | \* | \* | \* | \* | \* | \* | \* |

将上述表格写成公式形式可以得到：



此时，第0行的系数可以写成如下形式：



同理，将第1、2、4、5、7行拎出来，并构造剩下的4行，得到如下的action matrix A，A是一个10\*10的矩阵。



A的第6行为，可以得到恒等式，同理可以得到第7、8、9行。

令B等于左边的列向，G等于右边的列向量，于是可以得到B=AG，可以明显看出，向量B是向量G的x倍，于是有B=AG=xG，由于A是一个方阵，那么变量x满足特征值的定义，即G是A的特征值x对应的特征向量，此时特征向量会有10个解，最终需要去挑选出这10个解中最优的那一个，即对称重投影误差最小的解。

此时有；



将得到的x，y，z代入到中，得到本质矩阵，完成算法运算。

以下是对应的代码解析，总共分为五个步骤：

**Step 1、定义多项式系数运算：class Polynomial**

**class** **Polynomial** { *// 定义20\*1的多项式系数*

**public:**

**enum** **GRevLexMonomials** {

XXX **=** 0, XXY **=** 1, XYY **=** 2, YYY **=** 3, XXZ **=** 4, XYZ **=** 5, YYZ **=** 6, XZZ **=** 7,

YZZ **=** 8,ZZZ **=** 9,XX **=** 10, XY **=** 11, YY **=** 12, XZ **=** 13, YZ **=** 14, ZZ **=** 15,

X **=** 16, Y **=** 17,Z **=** 18, I **=** 19

};

Matrix**<double**, 20, 1**>** v;

Polynomial(**const** Matrix**<double**, 20, 1**>** **&**coeffcients) **:** v(coeffcients) {}

**public:**

Polynomial() **:** Polynomial(Matrix**<double**, 20, 1**>::**Zero()) {}

Polynomial(**double** w) {

v.setZero();

v[I] **=** w;

}

**void** **set\_xyzw**(**double** x, **double** y, **double** z, **double** w) { *// 设置初始一次项的值* v.setZero();

v[X] **=** x;

v[Y] **=** y;

v[Z] **=** z;

v[I] **=** w;

}

Polynomial **operator-**() **const** { **return** **Polynomial**(**-**v); }

Polynomial **operator+**(**const** Polynomial **&**b) **const** { **return** **Polynomial**(v **+** b.v); }

Polynomial **operator-**(**const** Polynomial **&**b) **const** { **return** **Polynomial**(v **-** b.v); }

*// 系数乘法，得到每一个阶数所对应的系数*

Polynomial **operator\***(**const** Polynomial **&**b) **const** {

Polynomial r;

r.v[I] **=** v[I] **\*** b.v[I];

r.v[Z] **=** v[I] **\*** b.v[Z] **+** v[Z] **\*** b.v[I];

r.v[Y] **=** v[I] **\*** b.v[Y] **+** v[Y] **\*** b.v[I];

r.v[X] **=** v[I] **\*** b.v[X] **+** v[X] **\*** b.v[I];

r.v[ZZ] **=** v[I] **\*** b.v[ZZ] **+** v[Z] **\*** b.v[Z] **+** v[ZZ] **\*** b.v[I];

r.v[YZ] **=** v[I] **\*** b.v[YZ] **+** v[Z] **\*** b.v[Y] **+** v[Y] **\*** b.v[Z] **+** v[YZ] **\*** b.v[I];

r.v[XZ] **=** v[I] **\*** b.v[XZ] **+** v[Z] **\*** b.v[X] **+** v[X] **\*** b.v[Z] **+** v[XZ] **\*** b.v[I];

r.v[YY] **=** v[I] **\*** b.v[YY] **+** v[Y] **\*** b.v[Y] **+** v[YY] **\*** b.v[I];

r.v[XY] **=** v[I] **\*** b.v[XY] **+** v[Y] **\*** b.v[X] **+** v[X] **\*** b.v[Y] **+** v[XY] **\*** b.v[I];

r.v[XX] **=** v[I] **\*** b.v[XX] **+** v[X] **\*** b.v[X] **+** v[XX] **\*** b.v[I];

r.v[ZZZ] **=** v[I] **\*** b.v[ZZZ] **+** v[Z] **\*** b.v[ZZ] **+** v[ZZ] **\*** b.v[Z] **+** v[ZZZ] **\*** b.v[I];

r.v[YZZ] **=** v[I] **\*** b.v[YZZ] **+** v[Z] **\*** b.v[YZ] **+** v[Y] **\*** b.v[ZZ] **+** v[ZZ] **\*** b.v[Y]

**+** v[YZ] **\*** b.v[Z] **+** v[YZZ] **\*** b.v[I];

r.v[XZZ] **=** v[I] **\*** b.v[XZZ] **+** v[Z] **\*** b.v[XZ] **+** v[X] **\*** b.v[ZZ] **+** v[ZZ] **\*** b.v[X]

**+** v[XZ] **\*** b.v[Z] **+** v[XZZ] **\*** b.v[I];

r.v[YYZ] **=** v[I] **\*** b.v[YYZ] **+** v[Z] **\*** b.v[YY] **+** v[Y] **\*** b.v[YZ] **+** v[YZ] **\*** b.v[Y]

**+** v[YY] **\*** b.v[Z] **+** v[YYZ] **\*** b.v[I];

r.v[XYZ] **=** v[I] **\*** b.v[XYZ] **+** v[Z] **\*** b.v[XY] **+** v[Y] **\*** b.v[XZ] **+** v[X] **\*** b.v[YZ]

**+** v[YZ] **\*** b.v[X] **+** v[XZ] **\*** b.v[Y] **+** v[XY] **\*** b.v[Z] **+** v[XYZ] **\*** b.v[I];

r.v[XXZ] **=** v[I] **\*** b.v[XXZ] **+** v[Z] **\*** b.v[XX] **+** v[X] **\*** b.v[XZ] **+** v[XZ] **\*** b.v[X]

**+** v[XX] **\*** b.v[Z] **+** v[XXZ] **\*** b.v[I];

r.v[YYY] **=** v[I] **\*** b.v[YYY] **+** v[Y] **\*** b.v[YY] **+** v[YY] **\*** b.v[Y] **+** v[YYY] **\*** b.v[I];

r.v[XYY] **=** v[I] **\*** b.v[XYY] **+** v[Y] **\*** b.v[XY] **+** v[X] **\*** b.v[YY] **+** v[YY] **\*** b.v[X]

**+** v[XY] **\*** b.v[Y] **+** v[XYY] **\*** b.v[I];

r.v[XXY] **=** v[I] **\*** b.v[XXY] **+** v[Y] **\*** b.v[XX] **+** v[X] **\*** b.v[XY] **+** v[XY] **\*** b.v[X]

**+** v[XX] **\*** b.v[Y] **+** v[XXY] **\*** b.v[I];

r.v[XXX] **=** v[I] **\*** b.v[XXX] **+** v[X] **\*** b.v[XX] **+** v[XX] **\*** b.v[X] **+** v[XXX] **\*** b.v[I];

**return** r;

}

**const** Matrix**<double**, 20, 1**>** **&**coeffcients() **const** { **return** v; }};

Polynomial **operator\***(**const** **double** **&**scale, **const** Polynomial **&**poly) {

**return** **Polynomial**(scale **\*** poly.coeffcients());}

**inline** Matrix3d **to\_matrix**(**const** Matrix**<double**, 9, 1**>** **&**vec) {

**return** (Matrix3d() **<<** vec.segment**<**3**>**(0),

vec.segment**<**3**>**(3),

vec.segment**<**3**>**(6)).finished();

}

**Step 2、构造约束矩阵，生成零空间：generate\_nullspace\_basis**

**inline** Matrix**<double**, 9, 4**>** generate\_nullspace\_basis(

**const** std**::**array**<**Vector2d, 5**>** **&**points1,

**const** std**::**array**<**Vector2d, 5**>** **&**points2) {

Matrix**<double**, 5, 9**>** A;

**for** (size\_t i **=** 0; i **<** 5; **++**i) {

Matrix3d h **=** Vector3d(points1[i].homogeneous()) **\***

points2[i].homogeneous().transpose();

**for** (size\_t j **=** 0; j **<** 3; **++**j) {

A.block**<**1, 3**>**(i, j **\*** 3) **=** h.row(j);

}

}

**return** A.jacobiSvd(ComputeFullV).matrixV().block**<**9, 4**>**(0, 5);

}

**Step 3、生成多项式矩阵M：generate\_polynomial**

**inline** Matrix**<double**, 10, 20**>** generate\_polynomials(**const** Matrix**<double**, 9, 4**>** **&**basis) {

**typedef** Matrix**<**Polynomial, 3, 3**>** matrix\_poly;

Matrix3d Ex **=** to\_matrix(basis.col(0));

Matrix3d Ey **=** to\_matrix(basis.col(1));

Matrix3d Ez **=** to\_matrix(basis.col(2));

Matrix3d Ew **=** to\_matrix(basis.col(3));

matrix\_poly Epoly;

**for** (size\_t i **=** 0; i **<** 3; **++**i) {

**for** (size\_t j **=** 0; j **<** 3; **++**j) {

Epoly(i, j).set\_xyzw(Ex(i, j), Ey(i, j), Ez(i, j), Ew(i, j));

}

}

Matrix**<double**, 10, 20**>** polynomials;

matrix\_poly EEt **=** Epoly **\*** Epoly.transpose();

matrix\_poly singular\_value\_constraints **=**

(EEt **\*** Epoly) **-** (0.5 **\*** EEt.trace()) **\*** Epoly;

**for** (size\_t i **=** 0; i **<** 3; **++**i) {

**for** (size\_t j **=** 0; j **<** 3; **++**j) {

polynomials.row(i **\*** 3 **+** j) **=**

singular\_value\_constraints(i, j).coeffcients();

}

}

Polynomial detE **=** Epoly.determinant();

polynomials.row(9) **=** detE.coeffcients();

**return** polynomials;

}

**Step 4、矩阵M消元，生成响应矩阵A：generate\_action\_matrix**

**inline** Matrix**<double**, 10, 10**>** generate\_action\_matrix(Matrix**<double**, 10, 20**>** **&**polynomials) {

std**::**array**<**size\_t, 10**>** perm;

*// perm[i] 表示第i列最终主元1所在的行*

**for** (size\_t i **=** 0; i **<** 10; **++**i) {

perm[i] **=** i;

}

**for** (size\_t i **=** 0; i **<** 10; **++**i) {

**for** (size\_t j **=** i **+** 1; j **<** 10; **++**j) {

**if** (abs(polynomials(perm[i], i)) **<** abs(polynomials(perm[j], i))) {

std**::**swap(perm[i], perm[j]);

}

}

*// 先找到第i列绝对值最大元素所在的行，设为主元行*

**if** (polynomials(perm[i], i) **==** 0)

**continue**;

polynomials.row(perm[i]) **/=** polynomials(perm[i], i);

*//利用主元行第一个元素变为1，并利用主元行对其他行消元，保证主元前面都为0*

**for** (size\_t j **=** i **+** 1; j **<** 10; **++**j) {

polynomials.row(perm[j]) **-=** polynomials.row(perm[i]) **\*** polynomials(perm[j], i);

}

}

**for** (size\_t i **=** 9; i **>** 0; **--**i) {

**for** (size\_t j **=** 0; j **<** i; **++**j) {

*//利用主元行第一个元素变为1，并利用主元行对其他行消元，保证主元后面都为0* polynomials.row(perm[j]) **-=** polynomials.row(perm[i]) **\*** polynomials(perm[j], i);

}

}

*// 这里没有显示将M矩阵的左边通过行交换变为单位矩阵，在构造响应矩阵A的时候按顺序进行了手动设置。*

Matrix**<double**, 10, 10**>** action;

action.row(0) **=** **-**polynomials.block**<**1, 10**>**(perm[Polynomial**::**XXX], Polynomial**::**XX);

action.row(1) **=** **-**polynomials.block**<**1, 10**>**(perm[Polynomial**::**XXY], Polynomial**::**XX);

action.row(2) **=** **-**polynomials.block**<**1, 10**>**(perm[Polynomial**::**XYY], Polynomial**::**XX);

action.row(3) **=** **-**polynomials.block**<**1, 10**>**(perm[Polynomial**::**XXZ], Polynomial**::**XX);

action.row(4) **=** **-**polynomials.block**<**1, 10**>**(perm[Polynomial**::**XYZ], Polynomial**::**XX);

action.row(5) **=** **-**polynomials.block**<**1, 10**>**(perm[Polynomial**::**XZZ], Polynomial**::**XX);

action.row(6) **=** Matrix**<double**, 10, 1**>::**Unit(Polynomial**::**XX **-** Polynomial**::**XX).transpose();

action.row(7) **=** Matrix**<double**, 10, 1**>::**Unit(Polynomial**::**XY **-** Polynomial**::**XX).transpose();

action.row(8) **=** Matrix**<double**, 10, 1**>::**Unit(Polynomial**::**XZ **-** Polynomial**::**XX).transpose();

action.row(9) **=** Matrix**<double**, 10, 1**>::**Unit(Polynomial**::**X **-** Polynomial**::**XX).transpose();

**return** action;

}

**Step 4、求解grobner系统：solver\_grobner\_system**

**inline** std**::**vector**<**Vector3d**>** solve\_grobner\_system(**const** Matrix**<double**, 10, 10**>** **&**action) {

EigenSolver**<**Matrix**<double**, 10, 10**>>** eigen(action, true);

Matrix**<**std**::**complex**<double>**, 10, 1**>** xs **=** eigen.eigenvalues();

std**::**vector**<**Vector3d**>** results;

**for** (size\_t i **=** 0; i **<** 10; **++**i) {

**if** (abs(xs[i].imag()) **<** 1.0e-10) {

Matrix**<double**, 10, 1**>** h **=** eigen.eigenvectors().col(i).real();

**double** xw **=** h(Polynomial**::**X **-** Polynomial**::**XX);

**double** yw **=** h(Polynomial**::**Y **-** Polynomial**::**XX);

**double** zw **=** h(Polynomial**::**Z **-** Polynomial**::**XX);

**double** w **=** h(Polynomial**::**I **-** Polynomial**::**XX);

results.emplace\_back(xw **/** w, yw **/** w, zw **/** w);

}

}

**return** results;

}

**Step 5、恢复Essential Matrix：solver\_essential\_5pt**

std**::**vector**<**Matrix3d**>** solve\_essential\_5pt(

**const** std**::**array**<**Vector2d, 5**>** **&**points1, **const** std**::**array**<**Vector2d, 5**>** **&**points2) {

Matrix**<double**, 9, 4**>** basis **=** generate\_nullspace\_basis(points1, points2);

Matrix**<double**, 10, 20**>** polynomials **=** generate\_polynomials(basis);

Matrix**<double**, 10, 10**>** action **=** generate\_action\_matrix(polynomials);

std**::**vector**<**Vector3d**>** solutions **=** solve\_grobner\_system(action);

std**::**vector**<**Matrix3d**>** results(solutions.size());

**for** (size\_t i **=** 0; i **<** solutions.size(); **++**i) {

results[i] **=** to\_matrix(basis **\*** solutions[i].homogeneous());

}

**return** results;

}

**7.2、SVD方法解ICP**

ICP算法用于估计3D-3D点之间的位姿。这个问题可以使用迭代最邻近点方法（ICP）求解。注意到3D-3D位姿估计问题中并没有出现相机模型，也就是说，仅考虑两组3D点之间的变换时，和相机并没有关系。因此，在激光SLAM中也会使用ICP，但是由于激光数据特征不够丰富，我们无从得知两个3D点集之间的匹配关系，只能认为距离最近的两个点为同名点，因此该方法被称为迭代最近点算。在视觉中，特征点为我们提供了较好的匹配关系，所以整个问题变得简单了。下面详细阐述求解ICP数值解的SVD方法。

根据前面描述的ICP问题，我们定义第对点的误差为：



然后，构建最小二乘问题，求使误差平方和达到极小的：



下面来推导上式的求解方法，首先，定义两组点的质心为：



请注意，质心是没有下标的。随后，在误差函数中做如下处理：



注意到交叉项部分中在求和后为零，因此在优化目标函数可以简化为：



仔细观察左右两项，我们发现左边只与旋转矩阵有关，而右边既有也有，但只和质心相关，只要我们获得了，令第二项为零就能得到。于是，ICP可以分为三个步骤求解：

1. 计算两组点的质心位置，然后计算每个点的去质心坐标：



1. 根据以下优化问题计算旋转矩阵：



1. 根据第二步求解的计算：



关注到，只要求出了两组点之间的旋转，平移量是非常容易得到的。所以我们重点关注的计算，展开关于的误差项，得到：



注意到第一项与无关，第二项由于，同样与无关。因此，实际上优化目标函数变为：



接下来，我们介绍如何通过SVD分解求解上述目标函数，参考文献《Least-squares fitting of two 3-D point sets》，其中提到以下引理：

**引理**：对于任意正定矩阵，有任意正交矩阵，满足：



引理的详细证明可以参考以上论文，比较简单。令，对其进行SVD分解可得到,其中表示3x3的正交矩阵，为3x3的特征值对角矩阵。现在，令



为正交矩阵，我们有：



上述矩阵为对称正定矩阵，由引理可知：



因此，对于正交矩阵空间的所有正交矩阵，时可令取最大值。然而当时并不满足旋转矩阵的的性质，表示反射矩阵，此时需要对做特殊处理。如下：

1. 当3D点集不共面时，此时所得必为旋转矩阵；
2. 当3D点集不共面但是共线时，此时秩亏1，此时矩阵的最小特征值为零，，此时改变的符号并不会改变。当时，可令其中一个为负，从而，使满足旋转矩阵的性质；
3. 当3D点集共线时，此时秩亏2，存在无穷多个旋转矩阵和反射矩阵使取最大值。这种情况不做考虑。