Slam Note 笔记

基于视觉十四讲

ElegantLaTeX Program

Victory won't come to us unless we go to it.

作者：Miao

时间：July 2, 2023

邮箱: chenmiao.ku@gmail.com

版本: 0.10

目　　录

第 1 章 初识 Slam

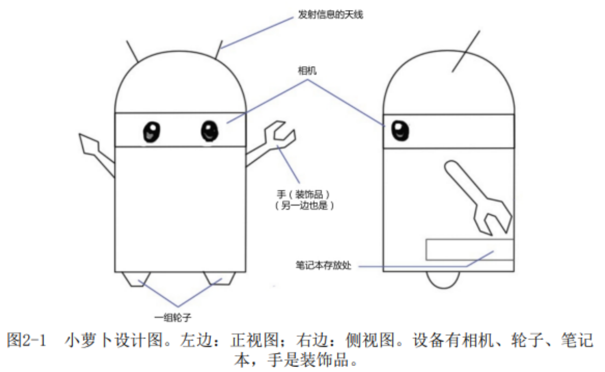

图 1.1: 小萝卜机器人设计图

以小萝卜为例，我们希望小萝卜具有自主运动能力，那么对应的，我们就应该对小萝卜的设计有一定的要求

1) 首先，移动需要有轮子和电机
2) 其次，如果光有移动能力但不知道行动的目标也是不行的。为了避免这种情况，我们需要一个相机以充当眼睛
3) 然后还要有接受信息的天线

那么，为了能够让小萝卜在能够在任意环境中轻松自在地行走，我们至少需要让他知道:

- 定位：我在什么地方？
- 建图：周围环境是什么样？

1.1 传感器

定位和建图可以看作感知的“内外之分”。

一类传感器是携带于机器人本体上的: 携带于本体上的传感器没有对环境提出任何要求，从而使得这种方案可适用于未知环境; 另一类传感器是安装于环境中的：安装于环境中的传感设备，通常能够简单有效地解决定位问题，但是必须要在一定的环境中，从而限制了机器人的适用范围。

值得注意的是: *Slam* 非常强调未知环境，因此通常我们都是通过相机来完成 Slam

相机的种类是多种的，Slam 中使用的相机与平时简单的单反摄像头并不一样。而照片的本质是拍照时的场景 *(Scene)* 在相机的成像平面上留下的一个投影。它以二维

的形式反映了三维的世界。那么显然的，这个过程中丢掉了一个维度：深度 (距离)

1.1.1 单目相机

只使用一个摄像头进行 Slam 的做法称为单目 *Slam*。

这种传感器结构简单，成本较低。但是，由于单目相机拍摄的图像只是三维空间中的二维投影，所以丢失深度的单目需要改变相机的视角，才能估计到运动 (*Motion*) 的发生，同时估计场景中物体的远近和大小，不妨称之为结构 (*Structure*)。通过近处的物体移动快，远处则慢这一原理，使得物体在图像上的运动形成了视差，就能够获取一个相对的深度

因此，单目 Slam 估计的轨迹和地图将与真实的轨迹和地图相差一个因子，被称为尺度 (*Scale*)，又称为尺度不确定性。

图 1.2: 单目相机示意图

1.1.2 双目相机

双目相机由两个单目相机组成，但这两个相机之间的距离 (被称为基线)
时已知的

类似于人眼一样，通过左右眼图像的差异判断物体的远近。计算机上的双目相机需要大量计算才能 (不太可靠的) 估计每一个像素点的深度。双目相机测量到的深度范围与基线相关。基线越大，能够测量到的范围越远。但是，双目或夺目相机的缺点时配置与标定均为复杂，其深度量程和精度受双目相机的基线和分辨率所限，且计算非常消耗计算机资源。

图 1.3: 双目相机示意图

1.1.3 深度相机 (RGB-D)

> 深度相机可以通过红外结构光或者 $Time-of-Flight(TOF)$ 原理，通过主动向物体发射光并接受返回的光，从而测量物体与相机之间的距离

深度相机通过物理的测量手段，所以对比于双目相机可以节省大量的计算。但是，目前的 RGB-D 相机还存在测量范围窄、噪声大、视野小、易受日光干扰、无法测量投射材质等诸多问题。

图 1.4: 深度相机示意图

1.2 经典视觉 Slam 框架

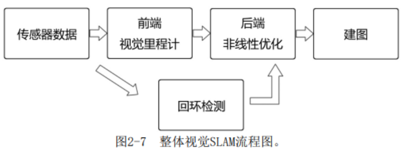

图 1.5: 经典视觉 Slam 框架

经典的视觉 *Slam* 框架本身和所包含的算法基本已经定型，整个视觉 Slam 的流程为：

1) 传感器信息读取：在视觉中主要以相机图像信息的读取和预处理
2) 视觉里程计 ($VisualOdometry, VO$)：视觉里程计的任务时估算相邻图像间相机的运动，以及局部地图的样子，又被称为前端 ($FrontEnd$)
3) 后端优化 ($Optimization$)：后端接受不同时刻视觉里程计测量的相机位姿，以及回环检测的信息，对它们进行优化，得到全局一致的轨迹和地图，又称为后端 ($BackEnd$)
4) 回环检测 ($LoopClosing$)：回环检测判断机器人是否达到过先前的位置，如果检测到回环，把信息提供给后端进行处理
5) 建图 ($Mapping$)：根据估计的轨迹，建立与任务要求对应的地图

1.2.1 视觉里程计

视觉里程计关心的时相邻图像之间的相机运动

如果用人眼来说，我们很轻易就知道我们是否发生移动，是否发生旋转，但是在相机中，我们能够知道发生了旋转，但是我们具体旋转了多少度，平移了多少里面，我们就很难给出确切答案了。

那么，我们需要知道相机与空间点的几何关系以及 *VO* 的实现方法 (在后续介绍)。*VO* 能够通过相邻帧的图像估计相机运动，并恢复场景的空间结构。*VO* 之所以叫做里程计，是因为其和实际的里程计一样，只计算相邻时刻的运动，而和过往的信息没有关联。

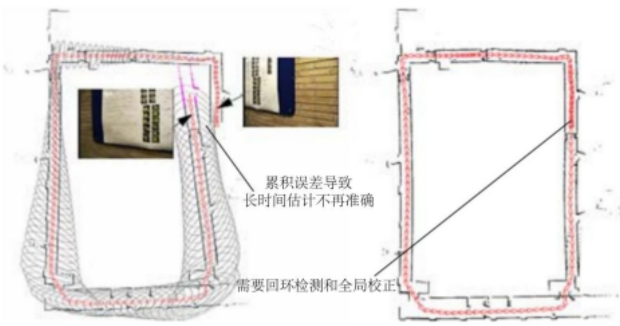

图2-9 累积误差与回环检测的校正结果[10] 。

图 1.6: VO 出现累积漂移

当然，仅仅通过视觉里程计来估计轨迹，将不可避免的出现累计漂移 (*Accumulating Drift*)。这是由于视觉里程计只估计两个图像之间的运动造成的，而每一次的估计都会有一定的误差，误差不断传递并累计造成了漂移 (*Drift*)。为了解决漂移问题，引入后续两种技术：后端优化和回环检测。

1.2.2 后端优化

笼统的说，后端优化主要指处理 Slam 过程中噪声的问题。

对于实际来说，不管多么精确的传感器都有一定的误差，后端要考虑的问题，就是如何从这些带有噪声的数据库中估计整体系统的状态，以及这个状态估计的不确定性有多大，这被称为最大后验概率估计 ($Maximum-a-Posterioru, MAP$)。这里的状态既包括自身的轨迹，也包括地图。在视觉 *Slam* 中，前端和计算机视觉领域研究更为相关。

Slam 问题的本质为：对运动主体自身和周围环境空间不确定性的估计，为了解决 Slam 问题，我们需要状态估计理论把定位和建图的不确定性表达出来，然后采用滤波器或非线性优化，估计状态的均值和不确定性 (在后续介绍)。

1.2.3 回环检测

回环检测又称闭环检测 ($LoopClosureDetection$)，主要解决位置估计随时间漂移的问题

假设实际情况下机器人经过一段时间后回到原点，由于漂移，它的位置估计值却没有回到原点，通过某种手段让机器人知道回到原点。

回环检测与定位和建图有密切的关系，事实上，我们更希望机器人自身判断图像间的相似性。

1.2.4 建图

建图 ($Mapping$) 是指构建地图的过程。地图是对环境的描述，但是该描述是不固定的

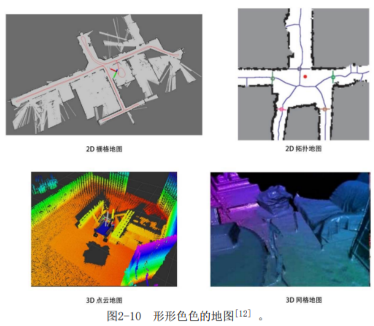

图 1.7: 建图示意图

度量地图 (Metric Map)，度量地图强调精确地表示地图中物体的位置关系，通常用稀疏 ($Sparse$) 与稠密 ($Dense$) 对其分类

稀疏地图进行了一定程度的抽象，并不需要表达所有的物体。我们可以选择一部分具有代表意义的东西，称为路标 (*Landmark*)，然后只记录路标即可。稀疏地图常用于定位，但是对于导航来说，需要使用稠密地图。

拓扑地图 (*TopologicalMap*)，拓扑地图*更强调地图元素之间的关系*。拓扑地图是一个图 (*Graph*)，由节点和边组成，只考虑节点之间的连通性。它放松了地图对精确位置的需要，去掉了地图的细节问题。

1.3 Slam 问题的数学表达

假设小萝卜正携带某种传感器在未知环境里运动，怎么用数学语言描述？

由于相机通常是在某些时刻采集数据的，所以我们只关心这些时刻的位置和地图：

1) 记离散时刻为：$t = 1, \ldots, K$
2) 用 x 表示小萝卜自身的位置，于是：$x_1, \ldots, X_K$ 表示各时刻的位置，构成了小萝卜的轨迹
3) 地图中，假设地图是由多个路标组成的，而每个时刻都会测量到一部分路标，设路标点一共有 N 个，用 $y_1, \ldots, y_N$ 表示

什么是运动，我们要考虑从 $K-1$ 时刻到 K 时刻，小萝卜的位置 x 是如何变化的

通常，机器人会携带一个测量自身运动的传感器，那么我们可以抽离出一个通用数学模型

$$x_k = f(x_{k-1}, u_k, w_k)$$

这里，U_k 是运动传感器的读数 (也叫做输入)，W_k 为噪声。我们使用函数 f 来指代任意的运动传感器，称为运动方程。

什么是观测，假设小萝卜在 k 时刻于 X_k 处探测到某一个路标 y_j，我们需要考虑如何用数学语言描述：

与运动方程对应的，还有一个观测方程。观测方程描述的是，当小萝卜在 X_k 位置上看到的某个路标点 y_j，产生了一个观测数据 $Z_{k,j}$。用抽象函数 h 来描述：

$$Z_{k,j} = h(y_j, x_k, v_{k,j})$$

这里，$V_{k,j}$ 是这次观测里的噪声，为了保证通用性，将其取为抽象形式，则可总结为两个基本方程

$$\begin{cases} x_k = f(x_{k-1}, u_k, w_k) \\ z_{k,j} = h(y_j, x_k, v_{k,j}) \end{cases}$$

第 2 章　三维空间刚体运动

2.1　旋转矩阵

2.1.1　点和向量，坐标系

如何表示一个点在三维空间，假设一个线性空间的基为：(e_1, e_2, e_3)，那么：

$$a = \begin{bmatrix} e_1, & e_2, & e_3 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = a_1 e_1 + a_2 e_2 + a_3 e_3$$

对于向量的内积：

$$a \cdot b = a^T b = \sum_{i=1}^{3} a_i b_i = |a||b| cos < a, b >$$

对于向量的外积：

$$a \times b = \begin{bmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} b \triangleq a^\wedge b$$

外积的方向垂直于这两个方向，大小为 $|a||b| sin < a, b >$。对于外积，此处引入了 ^ 符号，把 a 携程一个矩阵。事实上是一个反对称矩阵 $(Skew - symmetric)$，可以将 ^ 记成一个反对称符号。

外积只对三维向量存在定义，可以用外积表示向量的旋转

2.1.2　坐标系间的欧式变换

描述两个坐标系之间的旋转关系，加上平移统称为坐标系之间的变换关系

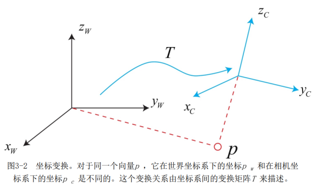

图 **2.1:** 坐标系的旋转

相机运动是一个刚体运动，保证了同一个向量再各个坐标系下的长度和角度不会发生变化，这被称为欧式变换。

一个欧式变换由一个旋转和一个平移两部分组成。首先考虑旋转，我们设某个单位正交基 (e_1, e_2, e_3) 经过一次旋转变为 $(e_1^{'}, e_2^{'}, e_3^{'})$，那么对于同一个向量 a(该向量并没有随着坐标系的旋转而发生运动)，它再这两个坐标系下的坐标分别为：$\begin{bmatrix} a_1, & a_2, & a_3 \end{bmatrix}^T$ 和 $\begin{bmatrix} a_1^{'}, & a_2^{'}, & a_3^{'} \end{bmatrix}^T$，那么就有：

$$\begin{bmatrix} e_1, & e_2, & e_3 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} e_1^{'}, & e_2^{'}, & e_3^{'} \end{bmatrix} \begin{bmatrix} a_1^{'} \\ a_2^{'} \\ a_3^{'} \end{bmatrix}$$

此时将上述等式的左右两边同时乘上 $\begin{bmatrix} e_1^T \\ e_2^T \\ e_3^T \end{bmatrix}$：

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} e_1^T e_1^{'}, & e_1^T e_2^{'}, & e_1^T e_3^{'} \\ e_2^T e_1^{'}, & e_2^T e_2^{'}, & e_2^T e_3^{'} \\ e_3^T e_1^{'}, & e_3^T e_2^{'}, & e_3^T e_3^{'} \end{bmatrix} \begin{bmatrix} a_1^{'} \\ a_2^{'} \\ a_3^{'} \end{bmatrix} \triangleq Ra^{'}$$

我们把中间的矩阵拿出来，定义成一个矩阵 R。这个矩阵由两组基之间的内积组成，刻画了旋转前后同一个向量的坐标变换关系。只要旋转是一样的，这个矩阵也是一样的。可以说，矩阵 R 描述了旋转本身，因此又称为旋转矩阵

旋转矩阵本身有一些特别的性质，比如它是一个行列式为 1 的正交矩阵，反之，行列式为 1 的正交矩阵也是一个旋转矩阵。因此，可以定义：

$$SO(n) = \{R \in \mathbb{R}^{n\times n} | RR^T = I, det(R) = 1\}$$

$SO(n)$ 是特殊正交群 ($SpecialOrthogonalGroup$) 的意思 (下一讲)。旋转矩阵可以描述相机的旋转, 而 R 满足以下性质：

$$\begin{aligned} a^{'} &= R^{-1}a = R^T a \\ a_1 &= R_{12}a_2 \\ a_3 &= R_{32}a_2 = R_{32}R_{21}a_1 = R_{31}a_1 \end{aligned}$$

最后考虑世界坐标系中的向量 a 经过一次旋转和一次平移 t 后，得到 $a^{'}$：

$$a^{'} = Ra + t$$

其中，t 被称为平移向量，相比于旋转，平移部分只需要把这个平移量加到旋转后的坐标上。

2.1.3 变换矩阵与齐次坐标

假定在上述给出的式子上我们进行了两次变换：R_1t_1 和 R_2t_2:

$$b = R_1 a + t_1, \quad c = R_2 b + t_2$$

那么就能够得到从 a 到 c 的变换：

$$c = R_2(R_1 a + t_1) + t_2$$

这样的形式在多次变换后会很复杂，因此引入齐次坐标和变换矩阵重写:

$$\begin{bmatrix} a' \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} a \\ 1 \end{bmatrix} \triangleq T \begin{bmatrix} a \\ 1 \end{bmatrix}$$

用这四个坐标表示三维向量的做法称为齐次坐标，引入齐次坐标后，旋转和平移可以放入同一个矩阵，称为变换矩阵，记作 T 矩阵。那么，经过多次变换后，通过变换矩阵可以得出：

$$\tilde{b} = T_1\tilde{a}, \tilde{c} = T_2\tilde{b} \Rightarrow \tilde{c} = T_2T_1\tilde{a}$$

对于变换矩阵，具有比较特别的结构：左上角为旋转看矩阵，右侧为平移向量，左下角为零向量，右下角为 1。这种矩阵又称为特殊欧式群 ($SpecialEuclideanGroup$)

$$se(3) = \{T = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \in \mathbb{R}^{4\times4} | R \in SO(3), t \in \mathbb{R}^3\}$$

那么可以因此求得该矩阵的一个反向的变换：

$$T^{-1} = \begin{bmatrix} R^T & -R^T t \\ 0^T & 1 \end{bmatrix}$$

例子

在 Slam 中，通常定义世界坐标系 T_W 与机器人坐标系 T_R

一个点的世界坐标系为 p_W，机器人坐标系下为 p_R，则有：$p_R = T_{RW}p_W$。也就是说，机器坐标系的点 p_R 可以由世界坐标系下的点 p_W 通过 T_{RW} 变换得到

2.2 Eigen

对于 Eigen 的安装来说，这是一件非常容易的事 (指在 Linux 操作系统或类 Unix 操作系统上)，我们只需要键入以下命令：

```
sudo apt-get install libeigen3.dev
```

对于 Eigen 第三方库的使用来说，也是较为简便的，因为 Eigen 只有头文件，是的，因此我们可以在 *CMakeLists.txt* 文件中使用以下条件命令来引入：

```
find_package(Eigen3 REQUIRED)
include_directories(${EIGEN3_INCLUDE_DIRS})
```

2.2.1 Eigen 的使用示例

对于 Eigen 来说，其使用是需要了解 Eigen 库的，此处我们使用一个小的例子来说明如何使用 Eigen

```
#include <iostream>
#include <ctime>
#include "eigen3/Eigen/Core"
#include "eigen3/Eigen/Dense"

#define MATRIX_SIZE 50

int main() {
    Eigen::Matrix<float, 2, 3> matrix_23;
    Eigen::Vector3d v_3d;
    Eigen::Matrix3d matrix_33 = Eigen::Matrix3d::Zero();
    Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> matrix_dynamic;
    Eigen::MatrixXd matrix_x;

    matrix_23 << 1, 2, 3, 4, 5, 6;
    std::cout << matrix_23 << std::endl;

    for (int i = 0; i < 1; i++) {
```

```
        for (int j = 0; j < 2; j++)
            std::cout << matrix_23(i, j) << " ";
        std::cout << "\n";
    }

    v_3d << 3, 2, 1;
    // Eigen::Matrix<double, 2, 1> result_wrong_type = matrix_23 * v_3d;
    Eigen::Matrix<double, 2, 1> result = matrix_23.cast<double>() * v_3d;
    std::cout << result << std::endl;

    matrix_33 = Eigen::Matrix3d::Random();
    std::cout << matrix_33 << std::endl;
    std::cout << matrix_33.transpose() << std::endl;
    std::cout << matrix_33.sum() << std::endl;
    std::cout << matrix_33.trace() << std::endl;
    std::cout << 10 * matrix_33 << std::endl;
    std::cout << matrix_33.inverse() << std::endl;
    std::cout << matrix_33.determinant() << std::endl;

    Eigen::SelfAdjointEigenSolver<Eigen::Matrix3d> eigen_solver(matrix_33.transpose() * matrix_33);

    std::cout << "Eigen value: " << eigen_solver.eigenvalues() << std::endl;
    std::cout << "Eigen vectors: " << eigen_solver.eigenvectors() << std::endl;

    Eigen::Matrix<double, MATRIX_SIZE, MATRIX_SIZE> matrix_NN;
    matrix_NN = Eigen::MatrixXd::Random(MATRIX_SIZE, MATRIX_SIZE);
    Eigen::Matrix<double, MATRIX_SIZE, 1> v_Nd;
    v_Nd = Eigen::MatrixXd::Random(MATRIX_SIZE, 1);

    clock_t tiem_stt = clock();
```

```
    Eigen::Matrix<double, MATRIX_SIZE, 1> x = matrix_NN.inverse() * v_Nd;
    std::cout << "tiome use in normal invers is: " << 1000 * (clock() - tiem_stt) / (double)CLOCKS_PER_SEC << "ms" << std::endl;

    tiem_stt = clock();
    x = matrix_NN.colPivHouseholderQr().solve(v_Nd);
    std::cout << "tiome use in Qr compsition invers is: ";

    return 0;
```

2.3 旋转向量和欧拉角

2.3.1 旋转向量

我们回到理论部分，探究矩阵表示方式中的缺点：

1) $SO(3)$ 的旋转矩阵由 9 个量，但一次旋转只有三个自由度，因此是冗余的

2) 旋转矩阵自身带有约束：必须是正交矩阵，且行列式为 1

我们希望有一种方式能够紧凑的描述旋转和平移，我们知道任意旋转都可以用一个旋转轴和一个旋转角来刻画，语句我们使用一个向量，其方向与旋转轴一致，而长度等于旋转角，这种向量被称为旋转向量 (或轴角，$Axis - Angle$)

$$w = \theta n$$

这是下一节中的李代数，因此目前只需要了解这样表示即可。之后，从旋转向量到旋转矩阵的转换过程由罗德里格斯公式 ($Rodrigues'sFormula$) 表明：

$$R = cos\theta I + (1 - cos\theta)nn^T + sin\theta n^\wedge$$

符号 $\wedge$ 是向量到反对称的转换符，反之，我们也可以有一个旋转矩阵到旋转向量的转换：

$$\begin{aligned} tr(R) &= cos\theta tyr(I) + (1 - cos\theta)tr(nn^\wedge) + sin\theta tr(n^\wedge) \\ &= 3cos\theta + (1 - cos\theta) \\ &= 1 + 2cos\theta \end{aligned}$$

对于转轴 n，由于旋转轴上的向量在旋转后不发生改变，说明:

$$Rn = n$$

2.3.2 欧拉角

无论是旋转矩阵、旋转向量对于人来说不是很直观，因此，欧拉角使用了三个分离的转角

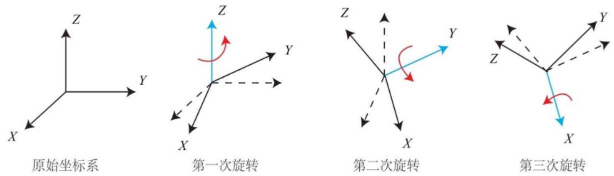

图 2.2: 欧拉角

在欧拉角中，常用的一种是“偏航–俯仰–滚转”($yaw--pitch--roll$) 三个角度来描述一个旋转。

1) 绕物体的 Z 轴旋转，得到偏航角 yaw
2) 绕物体的 Y 轴旋转，得到俯仰角 $pitch$
3) 绕物体的 X 轴旋转，得到滚转角 $roll$

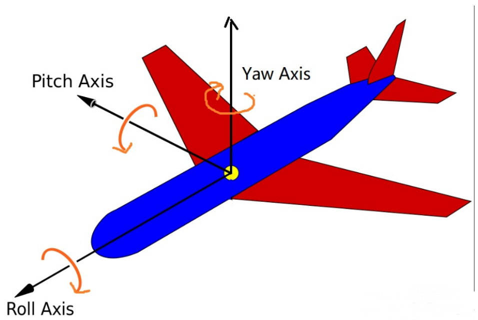

图 2.3: Z-Y-X 欧拉角

此时，可以使用 $\begin{bmatrix} r, & p, & y \end{bmatrix}^T$ 这样的一个三维向量表示任意旋转。

但是，欧拉角的一个重大缺点是会碰见著名的万向锁问题 ($GimbalLock$)：在俯仰角为 ±90° 时，第一次旋转与第三次旋转将使用同一个轴，使得系统丢失了一个自由度，这被称为奇异性问题

由于这种原理，欧拉角不适于插值和迭代，往往只适用于人机交互

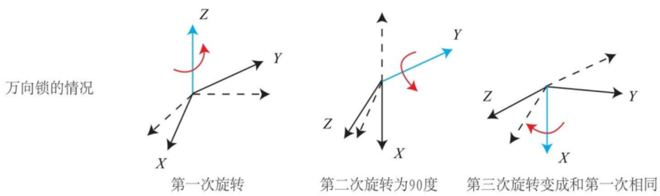

图 **2.4:** 万向锁问题

2.4 四元数

2.4.1 四元数定义

旋转矩阵用 9 个量描述三个自由度的旋转，但具有冗余性；欧拉角和旋转向量是紧凑的，但具有奇异性。因此，我们需要提出一种新的方式来描述旋转

回想一下复数：复数的惩罚表示复平面上的旋转。因此，表达三维空间旋转时，有一种类似复数的代数：四元数 (*Quaternion*)，四元数既是紧凑的，也没有奇异性，唯独不够直观，计算稍微复杂

一个四元数 q 拥有一个实部和三个虚部：$q = q_0 + q_1 i + q_2 j + q_3 k$。其中,$i, j, k$ 为四元数的三个虚部，三个虚部满足：

$$
\begin{cases}
i^2 = j^2 = k^2 = -1 \\
ij = k, ji = -k \\
jk = i, kj = -i \\
ik = j, ki = -j
\end{cases}
$$

我们可以简单的将虚部看作三个转轴向量，但是实际上不是。由于这种特殊的表示形式，我们可以用一个标量和一个向量来表示四元数：

$$
q = [s, v], s = q_0 \in \mathbb{R}, v = [q_1, q_2, q_3]^T \in \mathbb{R}^3
$$

这里，s 称为四元数的实部，v 称为它的虚部。

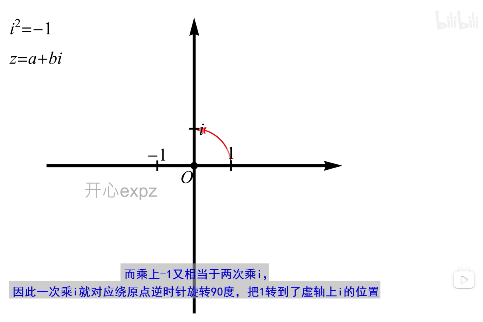

图 **2.5:** 二维向量的乘法

可以看见，从 1 到-1 的过程实际上是旋转了 180°，如果乘以虚数 i，就相当于绕原点逆时针 90°

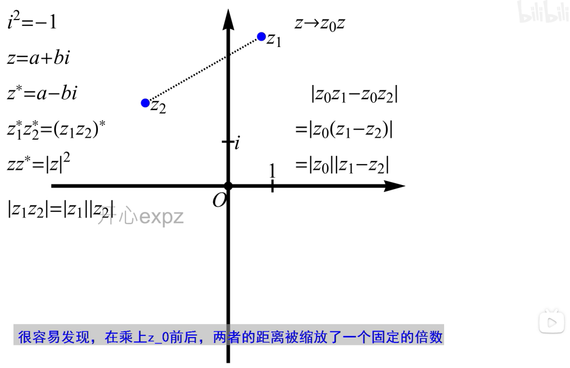

图 2.6: 二维向量的加法

对于加减，相当于对轴上进行扩张和缩减，对于乘除，就相当于原点不变，进行缩放和扩张以及旋转

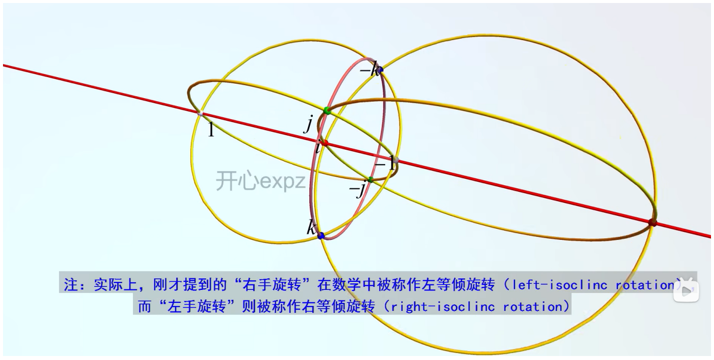

图 2.7: 四元数从四维到三维的变换

因此，对于四元数来说，可以通过四维来表示三维的旋转，$ij = k, ji = -k, i^2 = -1$ 是必须存在的条件，同时也会随着旋转被映射到无穷远最终回到三维

同时，我们发现，旋转两个周期才会回到与原先的样子相等。假设某个旋转是绕单位向量 $n = [n_x, n_y, n_z]^T$ 进行了角度为 θ 的旋转，那么这个旋转可以表示为：

$$q = [cos\frac{\theta}{2}, n_x sin\frac{\theta}{2}, n_y sin\frac{\theta}{2}, n_z sin\frac{\theta}{2}]^T$$

反之，亦可以得到对应旋转轴与夹角：

$$\begin{cases} \theta = 2arccosq_0 \\ [n_x, n_y, n_z]^T = [q_1, q_2, q_3]^T / sin\frac{\theta}{2} \end{cases}$$

在四元数中，任意的旋转都可以由两个互为相反数的四元数表示。取 θ 为 0，则得到一个没有任何旋转的实四元数：

$$q_0 = [\pm 1, 0, 0, 0]^T$$

2.4.2 四元数运算

四元数和通常复数一样，可以进行四则运算

现有两个四元数 q_a, q_b，它们的向量表示为 $[S_a, V_a][S_b, V_b]$ 或者原始四元数 $q_a = s_a + x_a i + y_a j + z_a k q_b = s_b + x_b i + y_b j + z_b k$

- 加法减法：$q_a \pm q_b = [S_a \pm S_b, V_a \pm V_b]$
- 乘法：

$$\begin{aligned} q_a q_b = & s_a s_b - x_a x_b - y_a y_b - z_a z_b \\ & + (s_a x_b + x_a s_b + y_a z_b - z_a y_b) i \\ & + (s_a y_b - x_a z_b + y_a s_b + z_a x_b) j \\ & + (s_a z_b + x_a y_b - y_a x_b + z_a s_b) k \end{aligned}$$

- 使用向量形式并采用内外积运算：$q_a q_b = [s_a s_b - v_a^T v_b, s_a v_b + s_b v_a + v_a \times v_b]$
- 共轭：$q_a^* = s_a - x_a i - y_a j - z_a k = [s_a, -v_a]$
- 模长：$\|q_a\| = \sqrt{s_a^2 + x_a^2 + y_a^2 + z_a^2}$

  两个四元数乘积的模即为模的乘积：$\|q_a q_b\| = \|q_a\| \|q_b\|$
- 逆：$q^{-1} = \frac{q^*}{\|q\|^2}$

  四元数与自己逆的乘积为实四元数：$q q^{-1} = q^{-1} q = 1$

  如果 q 为单位四元数，其逆与共轭相等：$(q_a q_b)^{-1} = q_b^{-1} q_a^{-1}$
- 数乘：$q_a \cdot q_b = s_a s_b + x_a x_b i + y_a y_b j + z_a z_b k$
- 点乘：$q_a \cdot q_b = s_a s_b + x_a x_b i + y_a y_b j + z_a z_b k$

2.4.3 用四元数表示旋转

我们可以用四元数表达对一个点的旋转，假设一个空间三维点 $p = [x, y, z] \in R^3$，以及一个由轴角 n, θ 指定的旋转。如果用矩阵旋转描述，那么有 $p' = Rp$，那么现在用四元数来表示

首先将三维空间点用一个虚四元数描述：

$$p = [0, x, y, z] = [0, v]$$

这相当于将四元数的三个虚部与空间中的三个轴对应，然后：

$$q = [cos\frac{\theta}{2}, nsin\frac{\theta}{2}]$$

那么经过旋转后的 p'，可以表示为：

$$p' = qpq^{-1}$$