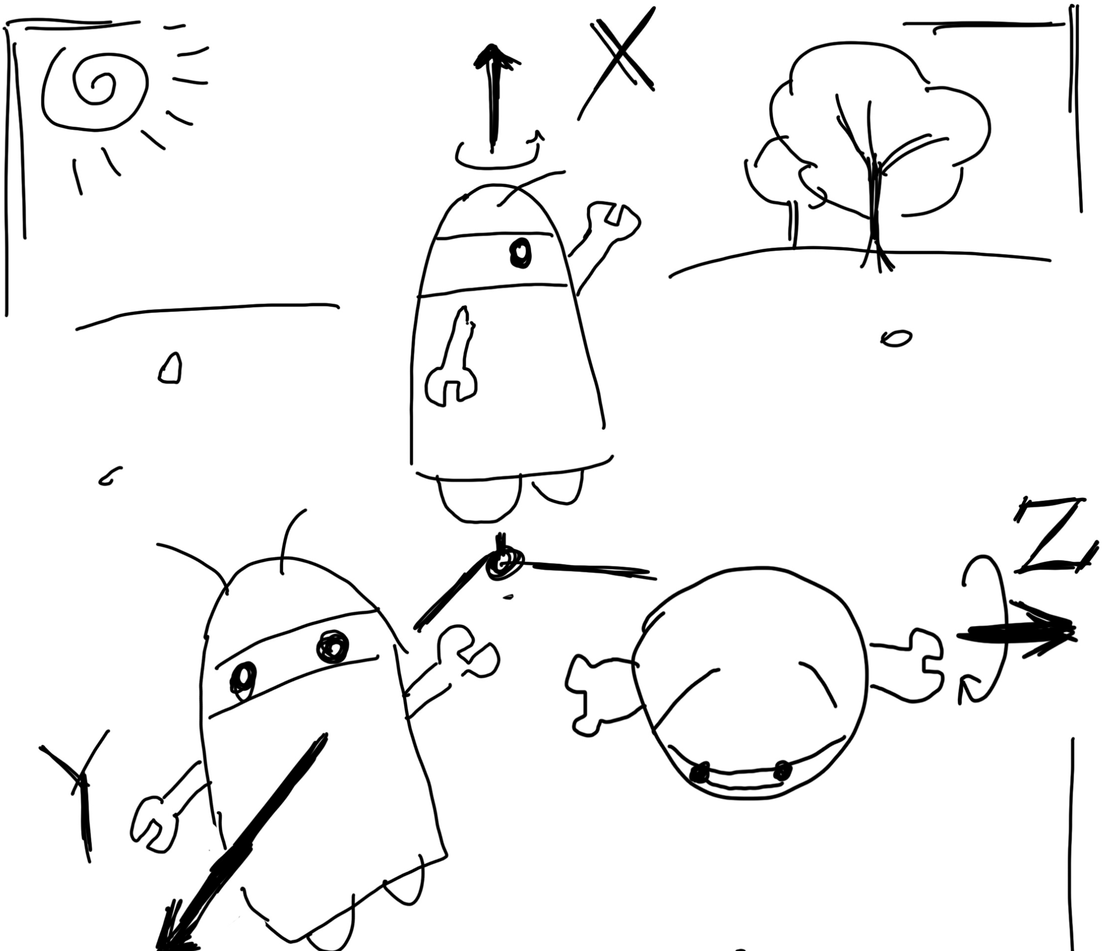
第 3 讲
三维空间刚体运动

本节目标

1. 理解三维空间的刚体运动描述方式：旋转矩阵、变换矩阵、四元数和欧拉角。
2. 掌握 Eigen 库的矩阵、几何模块使用方法。

在上讲中，我们讲解了视觉 SLAM 的框架与内容。本讲将介绍视觉 SLAM 的基本问题之一：**一个刚体在三维空间中的运动是如何描述的**。我们当然知道这由一次旋转加一次平移组成。平移确实没有太大问题，但旋转的处理是件麻烦事。我们将介绍旋转矩阵、四元数、欧拉角的意义，以及它们是如何运算和转换的。在实践部分，我们将介绍线性代数库 Eigen。它提供了 C++ 中的矩阵运算，并且它的 Geometry 模块还提供了四元数等刚体运动的描述。Eigen 的优化非常完善，但是它的使用方法有一些特殊的地方，我们会在程序中介绍。



$\boxed{\text{Rotation}}$

$$SO(3) = \{ R \mid R^T R = I, \det(R) = 1 \}$$

roll, pitch, yaw

$$q = q_0 + q_1 i + q_2 j + q_3 k$$

$$T = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \in SE(3)$$

3.1 旋转矩阵

3.1.1 点和向量，坐标系

我们日常生活的空间是三维的，因此我们生来就习惯于三维空间的运动。三维空间由三个轴组成，所以一个空间点的位置可以由三个坐标指定。不过，我们现在要考虑**刚体**，它不光有位置，还有自身的姿态。相机也可以看成三维空间的刚体，于是位置是指相机在空间中的哪个地方，而姿态则是指相机的朝向。结合起来，我们可以说，"相机正处于空间 $(0,0,0)$ 点处，朝向正前方" 这样的话。但是这种自然语言很繁琐，我们更喜欢用数学语言来描述它。

我们从最基本的开始讲起：**点**和**向量**。点的几何意义很容易理解。向量是什么呢？它是线性空间中的一个元素，可以把它想象成从原点指向某处的一个箭头。需要提醒读者的是，请不要把向量与它的**坐标**两个概念混淆。一个向量是空间当中的一样东西，比如说 $\boldsymbol{a}$。这里 $\boldsymbol{a}$ 并不是和若干个实数相关联的。只有当我们指定这个三维空间中的某个**坐标系**时，才可以谈论该向量在此坐标系下的坐标，也就是找到若干个实数对应这个向量。例如，三维空间中的某个向量的坐标可以用 $\mathbb{R}^3$ 当中的三个数来描述。某个点的坐标也可以用 $\mathbb{R}^3$ 来描述。怎么描述的呢？如果我们确定一个坐标系，也就是一个线性空间的基 $(\boldsymbol{e}_1,\boldsymbol{e}_2,\boldsymbol{e}_3)$，那就可以谈论向量 $\boldsymbol{a}$ 在这组基下的**坐标**了：

$$\boldsymbol{a} = [\boldsymbol{e}_1,\boldsymbol{e}_2,\boldsymbol{e}_3]\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = a_1\boldsymbol{e}_1 + a_2\boldsymbol{e}_2 + a_3\boldsymbol{e}_3. \tag{3.1}$$

所以这个坐标的具体取值，一个是和向量本身有关，第二也和坐标系的选取有关。坐标系通常由三个正交的坐标轴组成（尽管也可以有非正交的，但实际中很少见）。例如，我们给定 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 轴时，$\boldsymbol{z}$ 就可以通过右手（或左手）法则由 $\boldsymbol{x}\times\boldsymbol{y}$ 定义出来。根据定义方式的不同，坐标系又分为左手系和右手系。左手系的第三个轴与右手系相反。就经验来讲，人们更习惯使用右手系，尽管也有一部分程序库仍使用左手系。

根据基本的线性代数知识，我们可以谈论向量与向量，以及向量与数之间的运算，例如数乘、加法，减法，内积，外积等等。数乘和四则运算都是相当基本的内容，我们就不赘述了。内外积对读者来说可能有些陌生，我们给出它们的运算方式。对于 $\boldsymbol{a},\boldsymbol{b}\in\mathbb{R}^3$，内积可以写成：

$$\boldsymbol{a}\cdot\boldsymbol{b} = \boldsymbol{a}^T\boldsymbol{b} = \sum_{i=1}^{3} a_i b_i = |\boldsymbol{a}|\,|\boldsymbol{b}|\cos\langle\boldsymbol{a},\boldsymbol{b}\rangle\,. \tag{3.2}$$

内积可以描述向量间的投影关系。而外积呢是这个样子：

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{bmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \boldsymbol{b} \triangleq \boldsymbol{a}^{\wedge} \boldsymbol{b}. \tag{3.3}$$

外积的方向垂直于这两个向量，大小为 $|\boldsymbol{a}|\,|\boldsymbol{b}| \sin\langle \boldsymbol{a}, \boldsymbol{b} \rangle$，是两个向量张成的四边形的有向面积。对于外积，我们引入了 $^{\wedge}$ 符号，把 $\boldsymbol{a}$ 写成一个矩阵。事实上是一个反对称矩阵（Skew-symmetric），你可以将 $^{\wedge}$ 记成一个反对称符号。这样就把外积 $\boldsymbol{a} \times \boldsymbol{b}$，写成了矩阵与向量的乘法 $\boldsymbol{a}^{\wedge}\boldsymbol{b}$，把它变成了线性运算。这个符号将在后文经常用到，请记住它。外积只对三维向量存在定义，我们还能用外积表示向量的**旋转**。

为什么外积可以表示旋转呢？

考虑两个不平行的向量 $\boldsymbol{a}, \boldsymbol{b}$，我们要描述从 $\boldsymbol{a}$ 到 $\boldsymbol{b}$ 之间是如何旋转的，如图 3-1 所示。我们可以用一个向量来描述三维空间中两个向量的旋转关系。在右手法则下，我们用右手的四个指头从 $\boldsymbol{a}$ 转向 $\boldsymbol{b}$，其大拇指朝向就是旋转向量的方向，事实上也是 $\boldsymbol{a} \times \boldsymbol{b}$ 的方向。它的大小则由 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的夹角决定。通过这种方式，我们构造了从 $\boldsymbol{a}$ 到 $\boldsymbol{b}$ 的一个旋转向量。这个向量同样位于三维空间中，在此坐标系下，可以用三个实数来描述它。

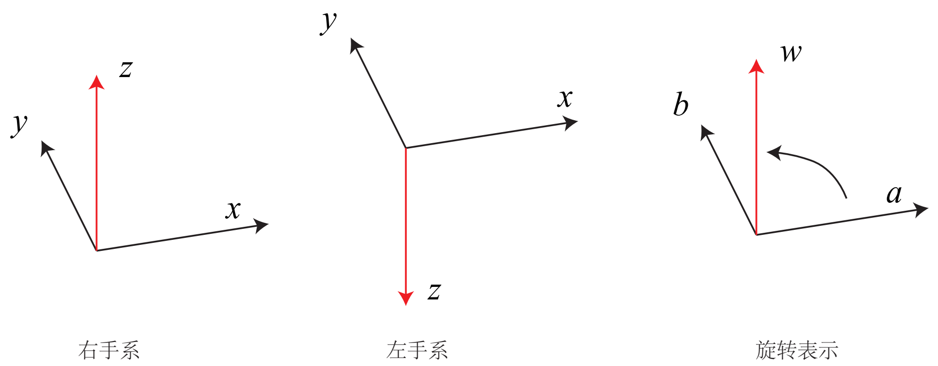


图 3-1　左右手系的区别与向量间的旋转。$\boldsymbol{a}$ 到 $\boldsymbol{b}$ 的旋转可以由向量 $\boldsymbol{w}$ 来描述。

3.1.2　坐标系间的欧氏变换

与向量间的旋转类似，我们同样可以描述两个坐标系之间的旋转关系，再加上平移，统称为坐标系之间的**变换**关系。在机器人的运动过程中，常见的做法是设定一个惯性坐标系

（或者叫世界坐标系），可以认为它是固定不动的，例如图 3-2 中的 x_W, y_W, z_W 定义的坐标系。同时，相机或机器人则是一个移动坐标系，例如 x_C, y_C, z_C 定义的坐标系。我们会问：相机视野中某个向量 $\boldsymbol{p}$，它的坐标为 $\boldsymbol{p}_c$，而从世界坐标系下看，它的坐标 $\boldsymbol{p}_w$。这两个坐标之间是如何转换的呢？这时，就需要先得到该点针对机器人坐标系坐标值，再根据机器人位姿转换到世界坐标系中，这个转换关系由一个矩阵 $\boldsymbol{T}$ 来描述，如图 3-2 所示。

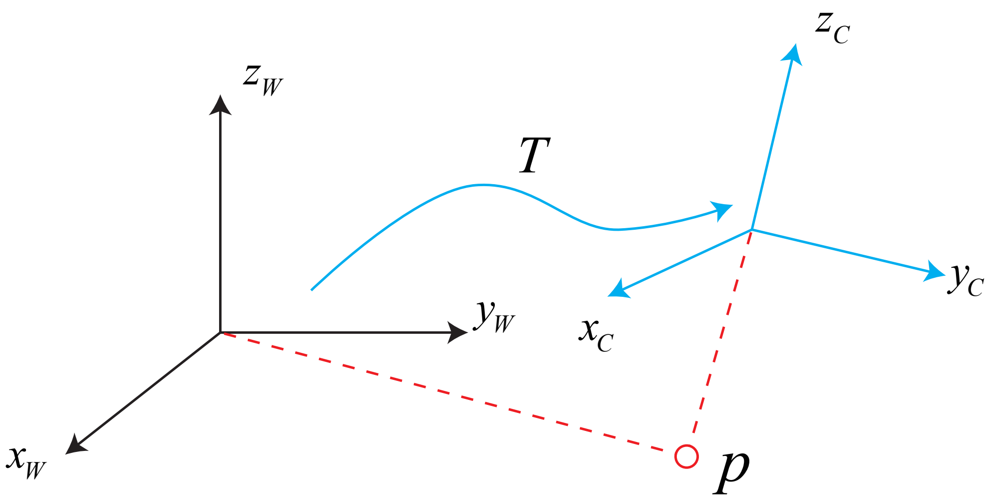


图 3-2 坐标变换。对于同一个向量 $\boldsymbol{p}$，它在世界坐标系下的坐标 $\boldsymbol{p}_w$ 和在相机坐标系下的 $\boldsymbol{p}_c$ 是不同的。这个变换关系由坐标系间的变换矩阵 $\boldsymbol{T}$ 来描述。

相机运动是一个刚体运动，它保证了同一个向量在各个坐标系下的长度和夹角都不会发生变化。这种变换称为**欧氏变换**。想象你把手机抛到空中，在它落地摔碎之前，只可能有空间位置和姿态的不同，而它自己的长度、各个面的角度等性质不会有任何变化。这样一个欧氏变换由一个旋转和一个平移两部分组成。首先来考虑旋转。我们设某个单位正交基 $(\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3)$ 经过一次旋转，变成了 $(\boldsymbol{e}_1', \boldsymbol{e}_2', \boldsymbol{e}_3')$。那么，对于同一个向量 $\boldsymbol{a}$（注意该向量并没有随着坐标系的旋转而发生运动），它在两个坐标系下的坐标为 $[a_1, a_2, a_3]^T$ 和 $[a_1', a_2', a_3']^T$。根据坐标的定义，有：

$$[\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3]\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \left[\boldsymbol{e}_1', \boldsymbol{e}_2', \boldsymbol{e}_3'\right]\begin{bmatrix} a_1' \\ a_2' \\ a_3' \end{bmatrix}. \tag{3.4}$$

为了描述两个坐标之间的关系，我们对上面等式左右同时左乘 $\begin{bmatrix} \boldsymbol{e}_1^T \\ \boldsymbol{e}_2^T \\ \boldsymbol{e}_3^T \end{bmatrix}$，那么左边的系数变成了单位矩阵，所以：

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_1^T \boldsymbol{e}_1^{'} & \boldsymbol{e}_1^T \boldsymbol{e}_2^{'} & \boldsymbol{e}_1^T \boldsymbol{e}_3^{'} \\ \boldsymbol{e}_2^T \boldsymbol{e}_1^{'} & \boldsymbol{e}_2^T \boldsymbol{e}_2^{'} & \boldsymbol{e}_2^T \boldsymbol{e}_3^{'} \\ \boldsymbol{e}_3^T \boldsymbol{e}_1^{'} & \boldsymbol{e}_3^T \boldsymbol{e}_2^{'} & \boldsymbol{e}_3^T \boldsymbol{e}_3^{'} \end{bmatrix} \begin{bmatrix} a_1^{'} \\ a_2^{'} \\ a_3^{'} \end{bmatrix} \triangleq \boldsymbol{R}\boldsymbol{a}^{'}. \tag{3.5}$$

我们把中间的阵拿出来，定义成一个矩阵 $\boldsymbol{R}$。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，那么这个矩阵也是一样的。可以说，矩阵 $\boldsymbol{R}$ 描述了旋转本身。因此它又称为**旋转矩阵**。

旋转矩阵有一些特别的性质。事实上，它是一个行列式为 1 的正交矩阵[1]。反之，行列式为 1 的正交矩阵也是一个旋转矩阵。所以，我们可以把旋转矩阵的集合定义如下：

$$SO(n) = \{\boldsymbol{R} \in \mathbb{R}^{n \times n} | \boldsymbol{R}\boldsymbol{R}^T = \boldsymbol{I}, \det(\boldsymbol{R}) = 1\}. \tag{3.6}$$

$SO(n)$ 是特殊正交群（Special Orthogonal Group）的意思。我们把解释“群”的内容留到下一讲。这个集合由 n 维空间的旋转矩阵组成，特别的，$SO(3)$ 就是三维空间的旋转了。通过旋转矩阵，我们可以直接谈论两个坐标系之间的旋转变换，而不用再从基开始谈起了。换句话说，**旋转矩阵可以描述相机的旋转**。

由于旋转矩阵为正交阵，它的逆（即转置）描述了一个相反的旋转。按照上面的定义方式，有：

$$\boldsymbol{a}^{'} = \boldsymbol{R}^{-1}\boldsymbol{a} = \boldsymbol{R}^T\boldsymbol{a}. \tag{3.7}$$

显然 $\boldsymbol{R}^T$ 刻画了一个相反的旋转。

在欧氏变换中，除了旋转之外还有一个平移。考虑世界坐标系中的向量 $\boldsymbol{a}$，经过一次旋转（用 $\boldsymbol{R}$ 描述）和一次平移 $\boldsymbol{t}$ 后，得到了 $\boldsymbol{a}^{'}$，那么把旋转和平移合到一起，有：

$$\boldsymbol{a}^{'} = \boldsymbol{R}\boldsymbol{a} + \boldsymbol{t}. \tag{3.8}$$

其中，$\boldsymbol{t}$ 称为平移向量。相比于旋转，平移部分只需把这个平移量加到旋转之后的坐标上，显得非常简洁。通过上式，我们用一个旋转矩阵 $\boldsymbol{R}$ 和一个平移向量 $\boldsymbol{t}$ 完整地描述了一个

[1]正交矩阵即逆为自身转置的矩阵。

欧氏空间的坐标变换关系。

3.1.3 变换矩阵与齐次坐标

式（3.8）完整地表达了欧氏空间的旋转与平移，不过还存在一个小问题：这里的变换关系不是一个线性关系。假设我们进行了两次变换：$\boldsymbol{R}_1, \boldsymbol{t}_1$ 和 $\boldsymbol{R}_2, \boldsymbol{t}_2$，满足：

$$\boldsymbol{b} = \boldsymbol{R}_1 \boldsymbol{a} + \boldsymbol{t}_1, \quad \boldsymbol{c} = \boldsymbol{R}_2 \boldsymbol{b} + \boldsymbol{t}_2.$$

但是从 $\boldsymbol{a}$ 到 $\boldsymbol{c}$ 的变换为：

$$\boldsymbol{c} = \boldsymbol{R}_2 \left(\boldsymbol{R}_1 \boldsymbol{a} + \boldsymbol{t}_1 \right) + \boldsymbol{t}_2.$$

这样的形式在变换多次之后会过于复杂。因此，我们要引入齐次坐标和变换矩阵重写式（3.8）：

$$\begin{bmatrix} \boldsymbol{a}^{'} \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{a} \\ 1 \end{bmatrix} \triangleq \boldsymbol{T} \begin{bmatrix} \boldsymbol{a} \\ 1 \end{bmatrix}. \tag{3.9}$$

这是一个数学技巧：我们把一个三维向量的末尾添加 1，变成了四维向量，称为**齐次坐标**。对于这个四维向量，我们可以把旋转和平移写在一个矩阵里面，使得整个关系变成了线性关系。该式中，矩阵 $\boldsymbol{T}$ 称为**变换矩阵（Transform Matrix）**。我们暂时用 $\tilde{\boldsymbol{a}}$ 表示 $\boldsymbol{a}$ 的齐次坐标。

稍微来说一下齐次坐标。它是射影几何里的概念。通过添加最后一维，我们用四个实数描述了一个三维向量，这显然多了一个自由度，但允许我们把变换写成线性的形式。在齐次坐标中，某个点 $\boldsymbol{x}$ 的每个分量同乘一个非零常数 k 后，**仍然表示的是同一个点**。因此，一个点的具体坐标值不是唯一的。如 $[1,1,1,1]^T$ 和 $[2,2,2,2]^T$ 是同一个点。但当最后一项不为零时，我们总可以把所有坐标除以最后一项，强制最后一项为 1，从而得到一个点唯一的坐标表示（也就是转换成非齐次坐标）：

$$\tilde{\boldsymbol{x}} = [x, y, z, w]^T = [x/w, y/w, z/w, 1]^T. \tag{3.10}$$

这时，忽略掉最后一项，这个点的坐标和欧氏空间就是一样的。依靠齐次坐标和变换矩阵，两次变换的累加就可以有很好的形式：

$$\tilde{\boldsymbol{b}} = \boldsymbol{T}_1 \tilde{\boldsymbol{a}}, \ \tilde{\boldsymbol{c}} = \boldsymbol{T}_2 \tilde{\boldsymbol{b}} \quad \Rightarrow \tilde{\boldsymbol{c}} = \boldsymbol{T}_2 \boldsymbol{T}_1 \tilde{\boldsymbol{a}}. \tag{3.11}$$

但是区分齐次和非齐次坐标的符号令我们厌烦。在不引起歧义的情况下，以后我们就

直接把它写成 $\boldsymbol{b} = \boldsymbol{T}\boldsymbol{a}$ 的样子，默认其中是齐次坐标了。

关于变换矩阵 $\boldsymbol{T}$，它具有比较特别的结构：左上角为旋转矩阵，右侧为平移向量，左下角为 $\mathbf{0}$ 向量，右下角为 1。这种矩阵又称为特殊欧氏群（Special Euclidean Group）：

$$SE(3) = \left\{ \boldsymbol{T} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \in \mathbb{R}^{4\times4} | \boldsymbol{R} \in SO(3), \boldsymbol{t} \in \mathbb{R}^3 \right\}. \tag{3.12}$$

与 $SO(3)$ 一样，求解该矩阵的逆表示一个反向的变换：

$$\boldsymbol{T}^{-1} = \begin{bmatrix} \boldsymbol{R}^T & -\boldsymbol{R}^T\boldsymbol{t} \\ \mathbf{0}^T & 1 \end{bmatrix}. \tag{3.13}$$

最后，为了保持符号的简洁，在不引起歧义的情况下，我们以后不区别齐次坐标与普通的坐标的符号，**默认我们使用的是符合运算法则的那一种**。例如，当我们写 $\boldsymbol{T}\boldsymbol{a}$ 时，使用的是齐次坐标（不然没法计算）。而写 $\boldsymbol{R}\boldsymbol{a}$ 时，使用的是非齐次坐标。如果写在一个等式中，我们就假设齐次坐标到普通坐标的转换，是已经做好了的——因为齐次坐标和非齐次坐标之间的转换事实上非常容易。

回顾一下我们介绍的内容：首先，我们说了向量和它的坐标表示，并介绍了向量间的运算；然后，坐标系之间的运动由欧氏变换描述，它由平移和旋转组成。旋转可以由旋转矩阵 $SO(3)$ 描述，而平移直接由一个 $\mathbb{R}^3$ 向量描述。最后，如果将平移和旋转放在一个矩阵中，就形成了变换矩阵 $SE(3)$。

3.2 实践：Eigen

本讲的实践部分有两节。第一部分中，我们将讲解如何使用 Eigen 来表示矩阵、向量，随后引申至旋转矩阵与变换矩阵的计算。本节的代码在 slambook/ch3/useEigen 中。

Eigen[①]是一个 C++ 开源线性代数库。它提供了快速的有关矩阵的线性代数运算，还包括解方程等功能。许多上层的软件库也使用 Eigen 进行矩阵运算，包括 g2o、Sophus 等。照应本讲的理论部分，我们来学习一下 Eigen 的编程。

你的 PC 上可能还没有安装 Eigen。请输入以下命令来安装它：

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

大部分常用的库都在 Ubuntu 软件源中提供。以后，当你想要安装某个库时，不妨先搜索一下 Ubuntu 的软件源是否提供了这样的库。通过 apt 命令，我们能够方便地安装 Eigen。回顾上一讲的知识，我们知道一个库由头文件和库文件组成。Eigen 头文件的默认

[①]官方主页：`http://eigen.tuxfamily.org/index.php?title=Main_Page`

位置在 “/usr/include/eigen3/” 中。如果你不确定，可以输入

```
sudo updatedb
locate eigen3
```

来查找它的位置。相比于其他库，Eigen 特殊之处在于，它是一个纯用头文件搭建起来的库（这非常神奇！）。这意味着你只能找到它的头文件，而没有.so 或.a 那样的二进制文件。我们在使用时，只需引入 Eigen 的头文件即可，不需要链接它的库文件（因为它没有库文件）。下面我们写一段代码，来实际练习一下 Eigen 的使用：

slambook/ch3/useEigen/eigenMatrix.cpp

```
#include <iostream>
#include <ctime>
using namespace std;

// Eigen 部分
#include <Eigen/Core>
// 稠密矩阵的代数运算（逆，特征值等）
#include <Eigen/Dense>

#define MATRIX_SIZE 50

/****************************
* 本程序演示了 Eigen 基本类型的使用
****************************/

int main( int argc, char** argv )
{
    // Eigen 以矩阵为基本数据单元。它是一个模板类。它的前三个参数为：数据类型，行，列
    // 声明一个 2*3 的 float 矩阵
    Eigen::Matrix<float, 2, 3> matrix_23;
    // 同时，Eigen 通过 typedef 提供了许多内置类型，不过底层仍是 Eigen::Matrix
    // 例如 Vector3d 实质上是 Eigen::Matrix<double, 3, 1>
    Eigen::Vector3d v_3d;
    // 还有 Matrix3d 实质上是 Eigen::Matrix<double, 3, 3>
    Eigen::Matrix3d matrix_33 = Eigen::Matrix3d::Zero(); //初始化为零
    // 如果不确定矩阵大小，可以使用动态大小的矩阵
    Eigen::Matrix< double, Eigen::Dynamic, Eigen::Dynamic > matrix_dynamic;
    // 更简单的
    Eigen::MatrixXd matrix_x;
    // 这种类型还有很多，我们不一一列举

    // 下面是对矩阵的操作
    // 输入数据
    matrix_23 << 1, 2, 3, 4, 5, 6;
    // 输出
    cout << matrix_23 << endl;
```

```

    // 用()访问矩阵中的元素
    for (int i=0; i<1; i++)
        for (int j=0; j<2; j++)
            cout<<matrix_23(i,j)<<endl;

    v_3d << 3, 2, 1;
    // 矩阵和向量相乘（实际上仍是矩阵和矩阵）
    // 但是在这里你不能混合两种不同类型的矩阵，像这样是错的
    // Eigen::Matrix<double, 2, 1> result_wrong_type = matrix_23 * v_3d;

    // 应该显式转换
    Eigen::Matrix<double, 2, 1> result = matrix_23.cast<double>() * v_3d;
    cout << result << endl;

    // 同样你不能搞错矩阵的维度
    // 试着取消下面的注释，看看会报什么错
    // Eigen::Matrix<double, 2, 3> result_wrong_dimension = matrix_23.cast<double>() * v_3d;

    // 一些矩阵运算
    // 四则运算就不演示了，直接用对应的运算符即可。
    matrix_33 = Eigen::Matrix3d::Random();
    cout << matrix_33 << endl << endl;

    cout << matrix_33.transpose() << endl; //转置
    cout << matrix_33.sum() << endl; //各元素和
    cout << matrix_33.trace() << endl; //迹
    cout << 10*matrix_33 << endl; //数乘
    cout << matrix_33.inverse() << endl; //逆
    cout << matrix_33.determinant() << endl; //行列式

    // 特征值
    // 实对称矩阵可以保证对角化成功
    Eigen::SelfAdjointEigenSolver<Eigen::Matrix3d> eigen_solver ( matrix_33.transpose()*matrix_33 );
    cout << "Eigen values = " << eigen_solver.eigenvalues() << endl;
    cout << "Eigen vectors = " << eigen_solver.eigenvectors() << endl;

    // 解方程
    // 我们求解 matrix_NN * x = v_Nd 这个方程
    // N 的大小在前边的宏里定义，矩阵由随机数生成
    // 直接求逆自然是最直接的，但是求逆运算量大

    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 time_stt = clock(); // 计时
    // 直接求逆
    Eigen::Matrix<double,MATRIX_SIZE,1> x = matrix_NN.inverse()*v_Nd;
```

```
    cout <<"time use in normal invers is " << 1000* (clock() - time_stt)/(double)CLOCKS_PER_SEC << "ms"
    << endl;

    // 通常用矩阵分解来求，例如 QR 分解，速度会快很多
    time_stt = clock();
    x = matrix_NN.colPivHouseholderQr().solve(v_Nd);
    cout <<"time use in Qr compsition is " <<1000* (clock() - time_stt)/(double)CLOCKS_PER_SEC <<"ms"
    << endl;

    return 0;
}
```

这个例程演示了 Eigen 矩阵的基本操作与运算。要编译它，你需要在 CMakeLists.txt 里指定 Eigen 的头文件目录：

```
# 添加头文件
include_directories( "/usr/include/eigen3" )
```

重复一遍，因为 Eigen 库只有头文件，我们不需要再用 tartget_link_libraries 语句将程序链接到库上。不过，对于其他大部分库，多数时候需要用到链接命令。这里的做法并不见得是最好的，因为他人可能把 Eigen 安装在了不同位置，就必须手动修改这里的头文件目录。在之后的工作中，我们会使用 find_package 命令去搜索库，不过在本讲我们暂时保持这个样子。编译好这个程序后，运行它，看到各矩阵的输出结果。

```
11:42 xiang@virtual /home/xiang/slambook/ch3/useEigen
% build/eigenMatrix
1 2 3
4 5 6
1
2
10
28
0.680375 0.59688 -0.329554
-0.211234 0.823295 0.536459
0.566198 -0.604897 -0.444451
……
```

由于我们在代码中给出了详细的注释，在此就不向读者一一解释每行语句了。在书中，我们仅给出几处重要地方的说明（后面的实践部分亦将保持这个风格）。

1. 读者最好亲手输入一遍上面的代码（不包括注释）。至少要编译运行一遍上面的程序。

2. Kdevelop 可能不会提示 C++ 成员运算，这是它做的不够完善导致的。请你照着上面的内容输入即可，不必理会它是否提示错误。

3. Eigen 提供的矩阵和 MATLAB 很相似，几乎所有的数据都当作矩阵来处理。但是，为了实现更好的效率，在 Eigen 中你需要指定矩阵的大小和类型。对于在编译时期就

知道大小的矩阵，处理起来会比动态变化大小的矩阵更快一些。因此，像旋转矩阵、变换矩阵这样的数据，完全可在编译时期确定它们的大小和数据类型。

4. Eigen 内部的矩阵实现比较复杂，我们不在这里介绍，我们希望你像使用 float、double 那样的内置数据类型那样使用 Eigen 的矩阵。这应该是符合它设计之初衷的。

5. Eigen 矩阵不支持自动类型提升，这和 C++ 的内建数据类型有较大差异。在 C++ 程序中，我们可以把一个 float 数据和 double 数据相加、相乘，**编译器会自动把数据类型转换为最合适的那种**。而在 Eigen 中，出于性能的考虑，必须**显式地**对矩阵类型进行转换。而如果忘了这样做，Eigen 会（不太友好地）提示您一个"YOU MIXED DIFFERENT NUMERIC TYPES ..."的编译错误。你可以尝试找一下这条信息出现错误提示的哪个部分。如果错误信息太长最好保存到一个文件里再找。

6. 同理，在计算过程中你也需要保证矩阵维数的正确性，否则会出现"YOU MIXED MATRICES OF DIFFERENT SIZES"。请你不要抱怨这种错误提示方式，对于 C++ 模板元编程，能够提示出可以阅读的信息已经是很幸运的了。以后，若发现 Eigen 出错，你可以直接寻找大写的部分，推测出了什么问题。

7. 我们的例程只介绍了基本的矩阵运算。你可以阅读 `http://eigen.tuxfamily.org/dox-devel/modules.html` 学习更多的 Eigen 知识。我只演示了最简单的部分，但看懂演示程序不等于你已经能够熟练操作 Eigen 了。

最后一段中我们比较了求逆与求 QR 分解的运行效率，你可以看看自己机器上的时间差异，两种方法是否有明显的差异？

3.3 旋转向量和欧拉角

3.3.1 旋转向量

我们重新回到理论部分。有了旋转矩阵来描述旋转，有了变换矩阵描述一个六自由度的三维刚体运动，是不是已经足够了呢？但是，矩阵表示方式至少有以下几个缺点：

1. $SO(3)$ 的旋转矩阵有九个量，但一次旋转只有三个自由度。因此这种表达方式是冗余的。同理，变换矩阵用十六个量表达了六自由度的变换。那么，是否有更紧凑的表示呢？

2. 旋转矩阵自身带有约束：它必须是个正交矩阵，且行列式为 1。变换矩阵也是如此。当我们想要估计或优化一个旋转矩阵/变换矩阵时，这些约束会使得求解变得更困难。

因此，我们希望有一种方式能够紧凑地描述旋转和平移。例如，用一个三维向量表达旋转，用六维向量表达变换，可行吗？事实上，这件事我们在前面介绍外积的那部分，提到过这件事如何做。我们介绍了如何用外积表达两个向量的旋转关系。对于坐标系的旋转，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。于是，我们可以使用一个向量，其方向与旋转轴一致，而长度等于旋转角。这种向量，称为**旋转向量**（或轴角，Axis-Angle）。这种表示法只需一个三维向量即可描述旋转。同样，对于变换矩阵，我们使用一个旋转向量和一个平移向量即可表达一次变换。这时的维数正好是六维。

事实上，旋转向量就是我们下章准备介绍的李代数。所以我们把它的详细内容留到下一章，本章内读者只需知道旋转可以这样表示即可。剩下的问题是，旋转向量和旋转矩阵之间是如何转换的呢？假设有一个旋转轴为 $\boldsymbol{n}$，角度为 θ 的旋转，显然，它对应的旋转向量为 $\theta\boldsymbol{n}$。由旋转向量到旋转矩阵的过程由**罗德里格斯公式**（Rodrigues's Formula ）表明，由于推导过程比较复杂，我们不作描述，只给出转换的结果[①]：

$$\boldsymbol{R} = \cos\theta \boldsymbol{I} + (1 - \cos\theta)\,\boldsymbol{n}\boldsymbol{n}^T + \sin\theta \boldsymbol{n}^{\wedge}. \tag{3.14}$$

符号 $\wedge$ 是向量到反对称的转换符，见式（3.3）。反之，我们也可以计算从一个旋转矩阵到旋转向量的转换。对于转角 θ，有：

$$\begin{aligned} \mathrm{tr}\left(\boldsymbol{R}\right) &= \cos\theta \mathrm{tr}\left(\boldsymbol{I}\right) + (1 - \cos\theta)\,\mathrm{tr}\left(\boldsymbol{n}\boldsymbol{n}^T\right) + \sin\theta \mathrm{tr}(\boldsymbol{n}^{\wedge}) \\ &= 3\cos\theta + (1 - \cos\theta) \\ &= 1 + 2\cos\theta. \end{aligned} \tag{3.15}$$

因此：

$$\theta = \arccos\left(\frac{\mathrm{tr}(\boldsymbol{R}) - 1}{2}\right). \tag{3.16}$$

关于转轴 $\boldsymbol{n}$，由于旋转轴上的向量在旋转后不发生改变，说明

$$\boldsymbol{R}\boldsymbol{n} = \boldsymbol{n}.$$

因此，转轴 $\boldsymbol{n}$ 是矩阵 $\boldsymbol{R}$ 特征值 1 对应的特征向量。求解此方程，再归一化，就得到了旋转轴。读者也可以从“旋转轴经过旋转之后不变”的几何角度看待这个方程。仍然剧透几句，这里的两个转换公式在下一章仍将出现，你会发现它们正是 $SO(3)$ 上李群与李代数的对应关系。

[①]感兴趣读者请参见`https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula`

3.3.2 欧拉角

下面我们来说说欧拉角。

无论是旋转矩阵、旋转向量，虽然它们能描述旋转，但对我们人类是非常不直观的。当我们看到一个旋转矩阵或旋转向量时，很难想象出来这个旋转究竟是什么样的。当它们变换时，我们也不知道物体是向哪个方向在转动。而欧拉角则提供了一种非常直观的方式来描述旋转——它使用了**三个分离的转角**，把一个旋转分解成三次绕不同轴的旋转。当然，由于分解方式有许多种，所以欧拉角也存在着不同的定义方法。比如说，当我先绕 X 轴旋转，再绕 Y 轴，最后绕 Z 轴，就得到了一个 XYZ 轴的旋转。同理，可以定义 ZYZ、ZYX 等等旋转方式。如果讨论更细一些，还需要区分每次旋转是绕**固定轴**旋转的，还是绕**旋转之后的轴**旋转的，这也会给出不一样的定义方式。

你或许在航空、航模中听说过“俯仰角”、“偏航角”这些词。欧拉角当中比较常用的一种，便是用“偏航-俯仰-滚转”（yaw-pitch-roll）三个角度来描述一个旋转的。由于它等价于 ZYX 轴的旋转，我们就以 ZYX 为例。假设一个刚体的前方（朝向我们的方向）为 X 轴，右侧为 Y 轴，上方为 Z 轴，见图 3-3。那么，ZYX 转角相当于把任意旋转分解成以下三个轴上的转角：

1. 绕物体的 Z 轴旋转，得到偏航角 yaw；
2. 绕**旋转之后**的 Y 轴旋转，得到俯仰角 pitch；
3. 绕**旋转之后**的 X 轴旋转，得到滚转角 roll。

此时，我们可以使用 $[r,p,y]^T$ 这样一个三维的向量描述任意旋转。这个向量十分的直观，我们可以从这个向量想象出旋转的过程。其他的欧拉角亦是通过这种方式，把旋转分解到三个轴上，得到一个三维的向量，只不过选用的轴，以及选用的顺序不一样。这里介绍的 rpy 角是比较常用的一种，只有很少的欧拉角种类会有 rpy 那样脍炙人口的名字。不同的欧拉角是按照旋转轴的顺序来称呼的。例如，rpy 角的旋转顺序是 ZYX。同样，也有 XYZ, ZYZ 这样欧拉角——但是它们就没有专门的名字了。值得一提的是，大部分领域在使用欧拉角时有各自的坐标方向和顺序上的习惯，不一定和我们这里说的相同。

欧拉角的一个重大缺点是会碰到著名的万向锁问题（Gimbal Lock[①]）：在俯仰角为 $\pm 90°$ 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（由三次旋转变成了两次旋转）。这被称为奇异性问题，在其他形式的欧拉角中也同样存在。理论上可以证明，只要我们想用三个实数来表达三维旋转时，都会不可避免地碰到奇异性问题。由于这种原理，欧拉角不适于插值和迭代，往往只用于人机交互中。我们也很少在 SLAM

[①] https://en.wikipedia.org/wiki/Gimbal_lock。

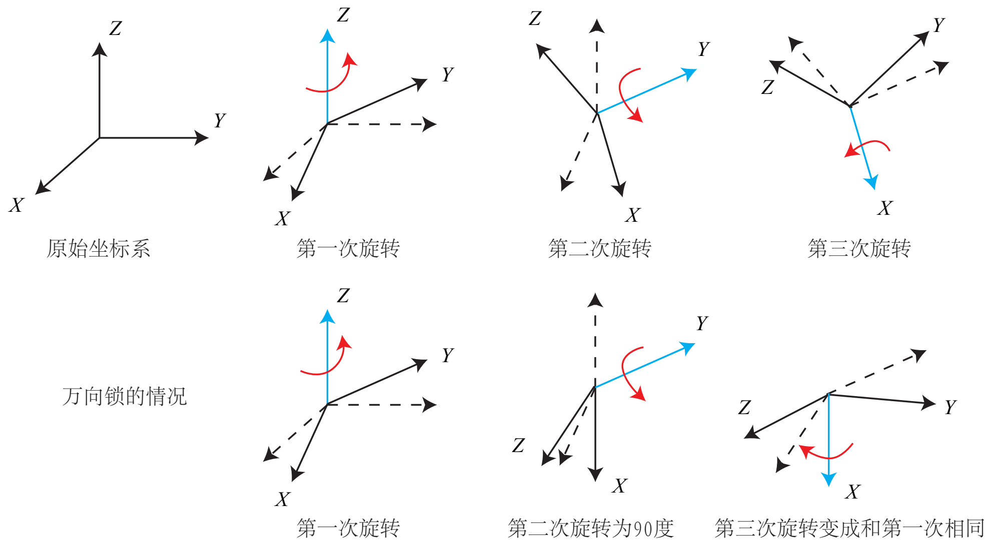


图 3-3 欧拉角的旋转示意图。上方为 ZYX 角定义。下方为 pitch=90 度时，第三次旋转与第一次滚转角相同，使得系统丢失了一个自由度。如果你还没有理解万向锁，可以看看相关视频，理解起来会更方便。

程序中直接使用欧拉角表达姿态，同样不会在滤波或优化中使用欧拉角表达旋转（因为它具有奇异性）。不过，若你想验证自己算法是否有错时，转换成欧拉角能够快速辨认结果的正确与否。

3.4 四元数

3.4.1 四元数的定义

旋转矩阵用九个量描述三自由度的旋转，具有冗余性；欧拉角和旋转向量是紧凑的，但具有奇异性。事实上，我们找不到**不带奇异性的三维向量描述方式** [19]。这有点类似于，当我们想用两个坐标表示地球表面时（如经度和纬度），必定存在奇异性（纬度为 $\pm 90^\circ$ 时经度无意义）。三维旋转是一个三维流形，想要无奇异性地表达它，用三个量是不够的。

回忆我们以前学习过的复数。我们用复数集 $\mathbb{C}$ 表示复平面上的向量，而复数的乘法则能表示复平面上的旋转：例如，乘上复数 i 相当于逆时针把一个复向量旋转 90 度。类似的，在表达三维空间旋转时，也有一种类似于复数的代数：**四元数**（Quaternion）。四元数是 Hamilton 找到的一种扩展的复数. 它**既是紧凑的，也没有奇异性**。如果说缺点的话，四元数不够直观，其运算稍为复杂一些。

一个四元数 $\boldsymbol{q}$ 拥有一个实部和三个虚部。本书把实部写在前面（也有地方把实部写在后面），像这样：

$$\boldsymbol{q} = q_0 + q_1 i + q_2 j + q_3 k, \tag{3.17}$$

其中 i, j, k 为四元数的三个虚部。这三个虚部满足关系式：

$$\begin{cases} i^2 = j^2 = k^2 = -1 \\ ij = k, ji = -k \\ jk = i, kj = -i \\ ki = j, ik = -j \end{cases}. \tag{3.18}$$

由于它的这种特殊表示形式，有时人们也用一个标量和一个向量来表达四元数：

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

这里，s 称为四元数的实部，而 $\boldsymbol{v}$ 称为它的虚部。如果一个四元数虚部为 $\mathbf{0}$，称之为**实四元数**。反之，若它的实部为 0，称之为**虚四元数**。

这和复数非常相似。考虑到三维空间需要三个轴，四元数也有三个虚部，那么，一个虚四元数能不能对应到一个空间点呢？事实上我们就是这样做的。同理，我们知道一个模长为 1 的复数，可以表示复平面上的纯旋转（没有长度的缩放），那么，三维空间中的旋转是否能用单位四元数表达呢？答案也是肯定的。

我们能用**单位四元数**表示三维空间中任意一个旋转，不过这种表达方式和复数有着微妙的不同。在复数中，乘以 i 意味着旋转 90 度。这是否意味着四元数中，乘 i 就是绕 i 轴旋转 90 度？那么，$ij = k$ 是否意味着，先绕 i 转 90 度，再绕 j 转 90 度，就等于绕 k 转 90 度？读者可以找一个手机比划一下——然后你会发现情况并不是这样。正确的事情应该是，乘以 i 应该对应着旋转 180 度，这样才能保证 $ij = k$ 的性质。而 $i^2 = -1$，意味着绕 i 轴旋转 360 度后，你得到了一个相反的东西。这个东西要旋转两周才会和它原先的样子相等。

这似乎有些玄妙了，完整的解释需要引入太多额外的东西，我们还是冷静一下回到眼前。至少，我们知道单位四元数能够表达三维空间的旋转。这种表达方式和旋转矩阵、旋转向量有什么关系呢？我们不妨先来看旋转向量。假设某个旋转是绕单位向量 $\boldsymbol{n} = [n_x, n_y, n_z]^T$ 进行了角度为 θ 的旋转，那么这个旋转的四元数形式为：

$$\boldsymbol{q} = \left[\cos\frac{\theta}{2}, n_x \sin\frac{\theta}{2}, n_y \sin\frac{\theta}{2}, n_z \sin\frac{\theta}{2}\right]^T. \tag{3.19}$$

反之，我们亦可从单位四元数中计算出对应旋转轴与夹角：

$$\begin{cases} \theta = 2\arccos q_0 \\ [n_x, n_y, n_z]^T = [q_1, q_2, q_3]^T / \sin\frac{\theta}{2} \end{cases}. \tag{3.20}$$

这式子给我们一种微妙的“转了一半”的感觉。同样，对式（3.19）的 θ 加上 2π，我们得到一个相同的旋转，但此时对应的四元数变成了 $-\boldsymbol{q}$。因此，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**。同理，取 θ 为 0，则得到一个没有任何旋转的实四元数：

$$\boldsymbol{q}_0 = [\pm 1, 0, 0, 0]^T. \tag{3.21}$$

3.4.2 四元数的运算

四元数和通常复数一样，可以进行一系列的运算。常见的有四则运算、数乘、求逆、共轭等等。我们分别来介绍它们。

现有两个四元数 $\boldsymbol{q}_a, \boldsymbol{q}_b$，它们的向量表示为 $[s_a, \boldsymbol{v}_a], [s_b, \boldsymbol{v}_b]$，或者原始四元数表示为：

$$\boldsymbol{q}_a = s_a + x_a i + y_a j + z_a k, \quad \boldsymbol{q}_b = s_b + x_b i + y_b j + z_b k.$$

那么，它们的运算可表示如下。

1. 加法和减法

   四元数 $\boldsymbol{q}_a, \boldsymbol{q}_b$ 的加减运算为：

$$\boldsymbol{q}_a \pm \boldsymbol{q}_b = [s_a \pm s_b, \boldsymbol{v}_a \pm \boldsymbol{v}_b]. \tag{3.22}$$

2. 乘法

   乘法是把 $\boldsymbol{q}_a$ 的每一项与 $\boldsymbol{q}_b$ 每项相乘，最后相加，虚部要按照式（3.18）进行。整理可得：

$$\begin{aligned} \boldsymbol{q}_a \boldsymbol{q}_b = \; & s_a s_b - x_a x_b - y_a y_b - z_a z_b \\ & + (s_a x_b + x_a s_b + y_a z_b - z_a y_b)\, i \\ & + (s_a y_b - x_a z_b + y_a s_b + z_a x_b)\, j \\ & + (s_a z_b + x_a y_b - y_b x_a + z_a s_b)\, k. \end{aligned} \tag{3.23}$$

虽然稍为复杂，但形式上是整齐有序的。如果写成向量形式并利用内外积运算，该表达会更加简洁：

$$\boldsymbol{q}_a\boldsymbol{q}_b = \left[s_a s_b - \boldsymbol{v}_a^T\boldsymbol{v}_b, s_a\boldsymbol{v}_b + s_b\boldsymbol{v}_a + \boldsymbol{v}_a \times \boldsymbol{v}_b\right]. \tag{3.24}$$

在该乘法定义下，两个实的四元数乘积仍是实的，这与复数也是一致的。然而，注意到，由于最后一项外积的存在，四元数乘法通常是不可交换的，除非 $\boldsymbol{v}_a$ 和 $\boldsymbol{v}_b$ 在 $\mathbb{R}^3$ 中共线，那么外积项为零。

3. 共轭

四元数的共轭是把虚部取成相反数：

$$\boldsymbol{q}_a^* = s_a - x_a i - y_a j - z_a k = [s_a, -\boldsymbol{v}_a]. \tag{3.25}$$

四元数共轭与自己本身相乘，会得到一个实四元数，其实部为模长的平方:

$$\boldsymbol{q}^*\boldsymbol{q} = \boldsymbol{q}\boldsymbol{q}^* = [s_a^2 + \boldsymbol{v}^T\boldsymbol{v}, \boldsymbol{0}]. \tag{3.26}$$

4. 模长

四元数的模长定义为：

$$\|\boldsymbol{q}_a\| = \sqrt{s_a^2 + x_a^2 + y_a^2 + z_a^2}. \tag{3.27}$$

可以验证，两个四元数乘积的模即为模的乘积。这保证单位四元数相乘后仍是单位四元数。

$$\|\boldsymbol{q}_a\boldsymbol{q}_b\| = \|\boldsymbol{q}_a\|\|\boldsymbol{q}_b\|. \tag{3.28}$$

5. 逆

一个四元数的逆为：

$$\boldsymbol{q}^{-1} = \boldsymbol{q}^*/\|\boldsymbol{q}\|^2. \tag{3.29}$$

按此定义，四元数和自己的逆的乘积为实四元数的 $\boldsymbol{1}$：

$$\boldsymbol{q}\boldsymbol{q}^{-1} = \boldsymbol{q}^{-1}\boldsymbol{q} = \boldsymbol{1}. \tag{3.30}$$

如果 $\boldsymbol{q}$ 为单位四元数，逆和共轭就是同一个量。同时，乘积的逆有和矩阵相似的性质：

$$(\boldsymbol{q}_a\boldsymbol{q}_b)^{-1} = \boldsymbol{q}_b^{-1}\boldsymbol{q}_a^{-1}. \tag{3.31}$$

6. 数乘与点乘

和向量相似，四元数可以与数相乘：

$$k\boldsymbol{q} = [ks, k\boldsymbol{v}]\,. \tag{3.32}$$

点乘是指两个四元数每个位置上的数值分别相乘：

$$\boldsymbol{q}_a \cdot \boldsymbol{q}_b = s_a s_b + x_a x_b i + y_a y_b j + z_a z_b k. \tag{3.33}$$

3.4.3 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 $\boldsymbol{p} = [x, y, z] \in \mathbb{R}^3$，以及一个由轴角 $\boldsymbol{n}, \theta$ 指定的旋转。三维点 $\boldsymbol{p}$ 经过旋转之后变成为 $\boldsymbol{p}'$。如果使用矩阵描述，那么有 $\boldsymbol{p}' = \boldsymbol{R}\boldsymbol{p}$。如果用四元数描述旋转，它们的关系如何来表达呢？

首先，把三维空间点用一个虚四元数来描述：

$$\boldsymbol{p} = [0, x, y, z] = [0, \boldsymbol{v}].$$

这相当于我们把四元数的三个虚部与空间中的三个轴相对应。然后，参照式（3.19），用四元数 $\boldsymbol{q}$ 表示这个旋转：

$$\boldsymbol{q} = [\cos\frac{\theta}{2}, \boldsymbol{n}\sin\frac{\theta}{2}].$$

那么，旋转后的点 $\boldsymbol{p}'$ 即可表示为这样的乘积：

$$\boldsymbol{p}' = \boldsymbol{q}\boldsymbol{p}\boldsymbol{q}^{-1}. \tag{3.34}$$

可以验证（留作习题），计算结果的实部为 0，故为纯虚四元数。其虚部的三个分量表示旋转后 3D 点的坐标。

3.4.4 四元数到旋转矩阵的转换

任意单位四元数描述了一个旋转，该旋转亦可用旋转矩阵或旋转向量描述。从旋转向量到四元数的转换方式已在式（3.20）中给出。因此现在看来，把四元数转换为矩阵的最直观方法，是先把四元数 $\boldsymbol{q}$ 转换为轴角 θ 和 $\boldsymbol{n}$，然后再根据罗德里格斯公式转换为矩阵。不过那样要计算一个 arccos 函数，代价较大。实际上这个计算是可以通过一定的技巧绕过的。我们省略过程中的推导，直接给出四元数到旋转矩阵的转换方式。

设四元数 $\boldsymbol{q} = q_0 + q_1 i + q_2 j + q_3 k$，对应的旋转矩阵 $\boldsymbol{R}$ 为：

$$\boldsymbol{R} = \begin{bmatrix} 1 - 2q_2^2 - 2q_3^2 & 2q_1q_2 + 2q_0q_3 & 2q_1q_3 - 2q_0q_2 \\ 2q_1q_2 - 2q_0q_3 & 1 - 2q_1^2 - 2q_3^2 & 2q_2q_3 + 2q_0q_1 \\ 2q_1q_3 + 2q_0q_2 & 2q_2q_3 - 2q_0q_1 & 1 - 2q_1^2 - 2q_2^2. \end{bmatrix} \tag{3.35}$$

反之，由旋转矩阵到四元数的转换如下。假设矩阵为 $\boldsymbol{R} = \{m_{ij}\}, i, j \in [1, 2, 3]$，其对应的四元数 $\boldsymbol{q}$ 由下式给出：

$$q_0 = \frac{\sqrt{\operatorname{tr}(R) + 1}}{2}, q_1 = \frac{m_{23} - m_{32}}{4q_0}, q_2 = \frac{m_{31} - m_{13}}{4q_0}, q_3 = \frac{m_{12} - m_{21}}{4q_0}. \tag{3.36}$$

值得一提的是，由于 $\boldsymbol{q}$ 和 $-\boldsymbol{q}$ 表示同一个旋转，事实上一个 $\boldsymbol{R}$ 对应的四元数表示并不是惟一的。同时，除了上面给出的转换方式之外，还存在其他几种计算方法，而本书都省略了。实际编程中，当 q_0 接近 0 时，其余三个分量会非常大，导致解不稳定，此时我们再考虑使用其他的方式进行转换。

最后，无论是四元数、旋转矩阵还是轴角，它们都可以用来描述同一个旋转。我们应该在实际中选择对我们最为方便的形式，而不必拘泥于某种特定的样子。在随后的实践和习题中，我们会演示各种表达方式之间的转换，加深读者的印象。

3.5 * 相似、仿射、射影变换

3D 空间中的变换，除了欧氏变换之外，还存在其余几种，其中欧氏变换是最简单的。它们一部分和测量几何有关，因为在之后的讲解中可能会提到，所以我们先罗列出来。欧氏变换保持了向量的长度和夹角，相当于我们把一个刚体原封不动地进行了移动或旋转，不改变它自身的样子。而其他几种变换则会改变它的外形。它们都拥有类似的矩阵表示。

1. 相似变换

相似变换比欧氏变换多了一个自由度，它允许物体进行均匀的缩放，其矩阵表示为：

$$\boldsymbol{T}_S = \begin{bmatrix} s\boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix}. \tag{3.37}$$

注意到旋转部分多了一个缩放因子 s，表示我们在对向量旋转之后，可以在 x, y, z 三个坐标上进行均匀的缩放。由于含有缩放，相似变换不再保持图形的面积不变。你可以想象一个边长为 1 的立方体通过相似变换后，变成边长为 10 的样子（但仍然是立方体）。

2. 仿射变换

仿射变换的矩阵形式如下：

$$\boldsymbol{T}_A = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix}. \tag{3.38}$$

与欧氏变换不同的是，仿射变换只要求 $\boldsymbol{A}$ 是一个可逆矩阵，而不必是正交矩阵。仿射变换也叫正交投影。经过仿射变换之后，立方体就不再是方的了，但是各个面仍然是平行四边形。

3. 射影变换

射影变换是最一般的变换，它的矩阵形式为:

$$\boldsymbol{T}_P = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{t} \\ \boldsymbol{a}^T & v \end{bmatrix}. \tag{3.39}$$

它左上角为可逆矩阵 $\boldsymbol{A}$，右上为平移 $\boldsymbol{t}$，左下缩放 $\boldsymbol{a}^T$。由于采用齐坐标，当 $v \neq 0$ 时，我们可以对整个矩阵除以 v 得到一个右下角为 1 的矩阵；否则，则得到右下角为 0 的矩阵。因此，2D 的射影变换一共有 8 个自由度，3D 则共有 15 个自由度。射影变换是现在讲过的变换中，形式最为一般的。从真实世界到相机照片的变换可以看成一个射影变换。读者可以想象一个原本方形的地板砖，在照片当中是什么样子：首先，它不再是方形的。由于近大远小的关系，它甚至不是平行四边形，而是一个不规则的四边形。

表 3.5 总结了目前讲到的几种变换的性质。注意在“不变性质”中，从上到下是有包含关系的。例如，欧氏变换除了保体积之外，也具有保平行、相交等性质。

我们之后会说到，从真实世界到相机照片的变换是一个射影变换。如果相机的焦距为无穷远，那么这个变换则为仿射变换。不过，在详细讲述相机模型之前，我们只要对它们有个大致的印象即可。

3.6 实践：Eigen 几何模块

现在，我们来实际演练一下前面讲到的各种旋转表达方式。我们将在 Eigen 中使用四元数、欧拉角和旋转矩阵，演示它们之间的变换方式。我们还会给出一个可视化程序，帮助读者理解这几个变换的关系。

slambook/ch3/useGeometry/useGeometry.cpp

表 3-1　常见变换性质比较

变换名称	矩阵形式	自由度	不变性质
欧氏变换	$\begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix}$	6 自由度	长度、夹角、体积
相似变换	$\begin{bmatrix} s\boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix}$	7 自由度	体积比
仿射变换	$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix}$	12 自由度	平行性、体积比
射影变换	$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{t} \\ \boldsymbol{a}^T & v \end{bmatrix}$	15 自由度	接触平面的相交和相切

```
#include <iostream>
#include <cmath>
using namespace std;

#include <Eigen/Core>
// Eigen 几何模块
#include <Eigen/Geometry>

/****************************
* 本程序演示了 Eigen 几何模块的使用方法
****************************/

int main( int argc, char** argv )
{
    // Eigen/Geometry 模块提供了各种旋转和平移的表示
    // 3D 旋转矩阵直接使用 Matrix3d 或 Matrix3f
    Eigen::Matrix3d rotation_matrix = Eigen::Matrix3d::Identity();
    // 旋转向量使用 AngleAxis, 它底层不直接是 Matrix ，但运算可以当作矩阵（因为重载了运算符）
    Eigen::AngleAxisd rotation_vector ( M_PI/4, Eigen::Vector3d ( 0,0,1 ) ); // 沿 Z 轴旋转 45 度
    cout .precision(3);
    cout<<"rotation matrix =\n"<<rotation_vector.matrix() <<endl; //用    matrix() 转换成矩阵
    // 也可以直接赋值
    rotation_matrix = rotation_vector.toRotationMatrix();
    // 用 AngleAxis 可以进行坐标变换
    Eigen::Vector3d v ( 1,0,0 );
    Eigen::Vector3d v_rotated = rotation_vector * v;
    cout<<"(1,0,0) after rotation = "<<v_rotated.transpose()<<endl;
    // 或者用旋转矩阵
    v_rotated = rotation_matrix * v;
    cout<<"(1,0,0) after rotation = "<<v_rotated.transpose()<<endl;

    // 欧拉角：可以将旋转矩阵直接转换成欧拉角
```

```
    Eigen::Vector3d euler_angles = rotation_matrix.eulerAngles ( 2,1,0 ); // ZYX 顺序，即 yaw pitch roll 顺序
    cout<<"yaw pitch roll = "<<euler_angles.transpose()<<endl;

    // 欧氏变换矩阵使用 Eigen::Isometry
    Eigen::Isometry3d T=Eigen::Isometry3d::Identity(); //     虽然称为 3d ，实质上是 4*4 的矩阵
    T.rotate ( rotation_vector ); //                          按照 rotation_vector 进行旋转
    T.pretranslate ( Eigen::Vector3d ( 1,3,4 ) ); //          把平移向量设成 (1,3,4)
    cout << "Transform matrix = \n" << T.matrix() <<endl;

    // 用变换矩阵进行坐标变换
    Eigen::Vector3d v_transformed = T*v; //                   相当于 R*v+t
    cout<<"v tranformed = "<<v_transformed.transpose()<<endl;

    // 对于仿射和射影变换，使用 Eigen::Affine3d 和 Eigen::Projective3d 即可，略

    // 四元数
    // 可以直接把 AngleAxis 赋值给四元数，反之亦然
    Eigen::Quaterniond q = Eigen::Quaterniond ( rotation_vector );
    cout<<"quaternion = \n"<<q.coeffs() <<endl; // 请注意 coeffs 的顺序是 (x,y,z,w), w 为实部，前三者为虚部
    // 也可以把旋转矩阵赋给它
    q = Eigen::Quaterniond ( rotation_matrix );
    cout<<"quaternion = \n"<<q.coeffs() <<endl;
    // 使用四元数旋转一个向量，使用重载的乘法即可
    v_rotated = q*v; // 注意数学上是 qvq^{-1}
    cout<<"(1,0,0) after rotation = "<<v_rotated.transpose()<<endl;

    return 0;
}
```

Eigen 中对各种形式的表达方式总结如下。请注意每种类型都有单精度和双精度两种数据类型，而且和之前一样，不能由编译器自动转换。下面以双精度为例，你可以把最后的 d 改成 f，即得到单精度的数据结构。

- 旋转矩阵（3×3）：Eigen::Matrix3d。
- 旋转向量（3×1）：Eigen::AngleAxisd。
- 欧拉角（3×1）：Eigen::Vector3d。
- 四元数（4×1）：Eigen::Quaterniond。
- 欧氏变换矩阵（4×4）：Eigen::Isometry3d。
- 仿射变换（4×4）：Eigen::Affine3d。
- 射影变换（4×4）：Eigen::Projective3d。

我们把如何编译此程序的问题交给读者。在这个程序中，我们演示了如何使用 Eigen 中的旋转矩阵、旋转向量（AngleAxis）、欧拉角和四元数。我们用这几种旋转方式去旋转一个向量 $\boldsymbol{v}$，发现结果是一样的（不一样那真是见鬼了）。同时，也演示了如何在程序中转换这几种表达方式。想进一步了解 Eigen 的几何模块的读者可以参考（`http://eigen.tuxfamily.org/dox/group__TutorialGeometry.html`）。

3.7 可视化演示

最后，我们为读者准备了一个小程序，位于在 slambook/ch3/visualizeGeometry 中。它以可视化的形式演示了各种表达方式的异同。读者可以用鼠标操作一下，看看数据是如何变化的。

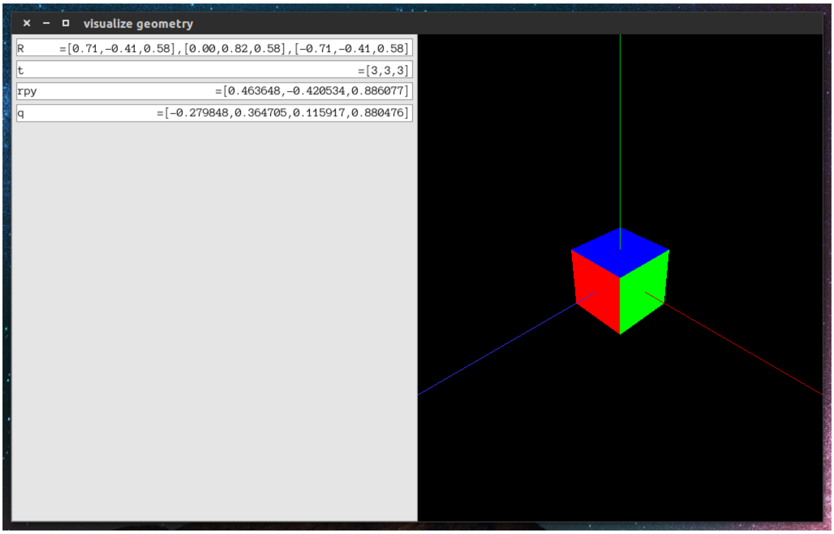


图 3-4 旋转矩阵、欧拉角、四元数的可视化程序。

在写这个小程序中，我们在坐标原点放置一个彩色立方体。用鼠标可以平移/旋转相机。你可以实时地看到相机姿态的变化。我们显示了变换矩阵 $\boldsymbol{R}, \boldsymbol{t}$、欧拉角和四元数的三种姿态，你可以实验体验一下这几个量是如何变化的。然而根据我的经验，除了欧拉角之外，你应该看不出它们直观的含义。

该程序我们就不向读者解释源代码了，如果你感兴趣，可以自行查看。该程序的编译说明请参照它的 Readme.txt，我们在书籍正文中省略了。

值得一提的是，实际当中，我们至少定义两个坐标系：**世界坐标系**和**相机坐标系**。在该定义下，设某个点在世界坐标系中坐标为 $\boldsymbol{p}_w$，在相机坐标系下为 $\boldsymbol{p}_c$，那么：

$$\boldsymbol{p}_c = \boldsymbol{T}_{cw} \boldsymbol{p}_w, \tag{3.40}$$

这里 $\boldsymbol{T}_{cw}$ 表示世界坐标系到相机坐标系间的变换。或者我们可以用反过来的 $\boldsymbol{T}_{wc}$：

$$\boldsymbol{p}_w = \boldsymbol{T}_{wc}\boldsymbol{p}_c = \boldsymbol{T}_{cw}^{-1}\boldsymbol{p}_c. \tag{3.41}$$

原则上，$\boldsymbol{T}_{cw}$ 和 $\boldsymbol{T}_{cw}$ 都可以用来表示相机的位姿，事实上它们也只差一个逆而已。实践当中使用 $\boldsymbol{T}_{cw}$ 更加常见，而 $\boldsymbol{T}_{wc}$ 更为直观。如果把上面两式的 $\boldsymbol{p}_c$ 取成零向量，也就是相机坐标系中的原点，那么，此时的 $\boldsymbol{p}_w$ 就是相机原点在世界坐标系下的坐标：

$$\boldsymbol{p}_w = \boldsymbol{T}_{wc}\boldsymbol{0} = \boldsymbol{t}_{wc}. \tag{3.42}$$

我们发现这正是 $\boldsymbol{T}_{wc}$ 的平移部分。因此，可以从 $\boldsymbol{T}_{wc}$ 中直接看到**相机在何处**，这也是我们说 $\boldsymbol{T}_{wc}$ 更为直观的原因。因此，在可视化程序里，我们显示了 $\boldsymbol{T}_{wc}$ 而不是 $\boldsymbol{T}_{cw}$。

习题

1. 验证旋转矩阵是正交矩阵。
2. * 寻找罗德里格斯公式的推导过程并理解它。
3. 验证四元数旋转某个点后，结果是一个虚四元数（实部为零），所以仍然对应到一个三维空间点（式 3.34）。
4. 画表总结旋转矩阵、轴角、欧拉角、四元数的转换关系。
5. 假设我有一个大的 Eigen 矩阵，我想把它的左上角 3×3 的块取出来，然后赋值为 $\boldsymbol{I}_{3\times3}$。请编程实现此事。
6. * 一般线程方程 $\boldsymbol{Ax} = \boldsymbol{b}$ 有哪几种做法？你能在 Eigen 中实现吗？
7. 设有小萝卜一号和小萝卜二号位于世界坐标系中。小萝卜一号的位姿为：$\boldsymbol{q}_1 = [0.35, 0.2, 0.3, 0.1], \boldsymbol{t}_2 = [0.3, 0.1, 0.1]^T$（$\boldsymbol{q}$ 的第一项为实部。请你把 $\boldsymbol{q}$ 归一化后再进行计算）。这里的 $\boldsymbol{q}$ 和 $\boldsymbol{t}$ 表达的是 $\boldsymbol{T}_{cw}$，也就是世界到相机的变换关系。小萝卜二号的位姿为 $\boldsymbol{q}_2 = [-0.5, 0.4, -0.1, 0.2], \boldsymbol{t} = [-0.1, 0.5, 0.3]^T$。现在，小萝卜一号看到某个点在自身的坐标系下，坐标为 $\boldsymbol{p} = [0.5, 0, 0.2]^T$，求该向量在小萝卜二号坐标系下的坐标。请编程实现此事。