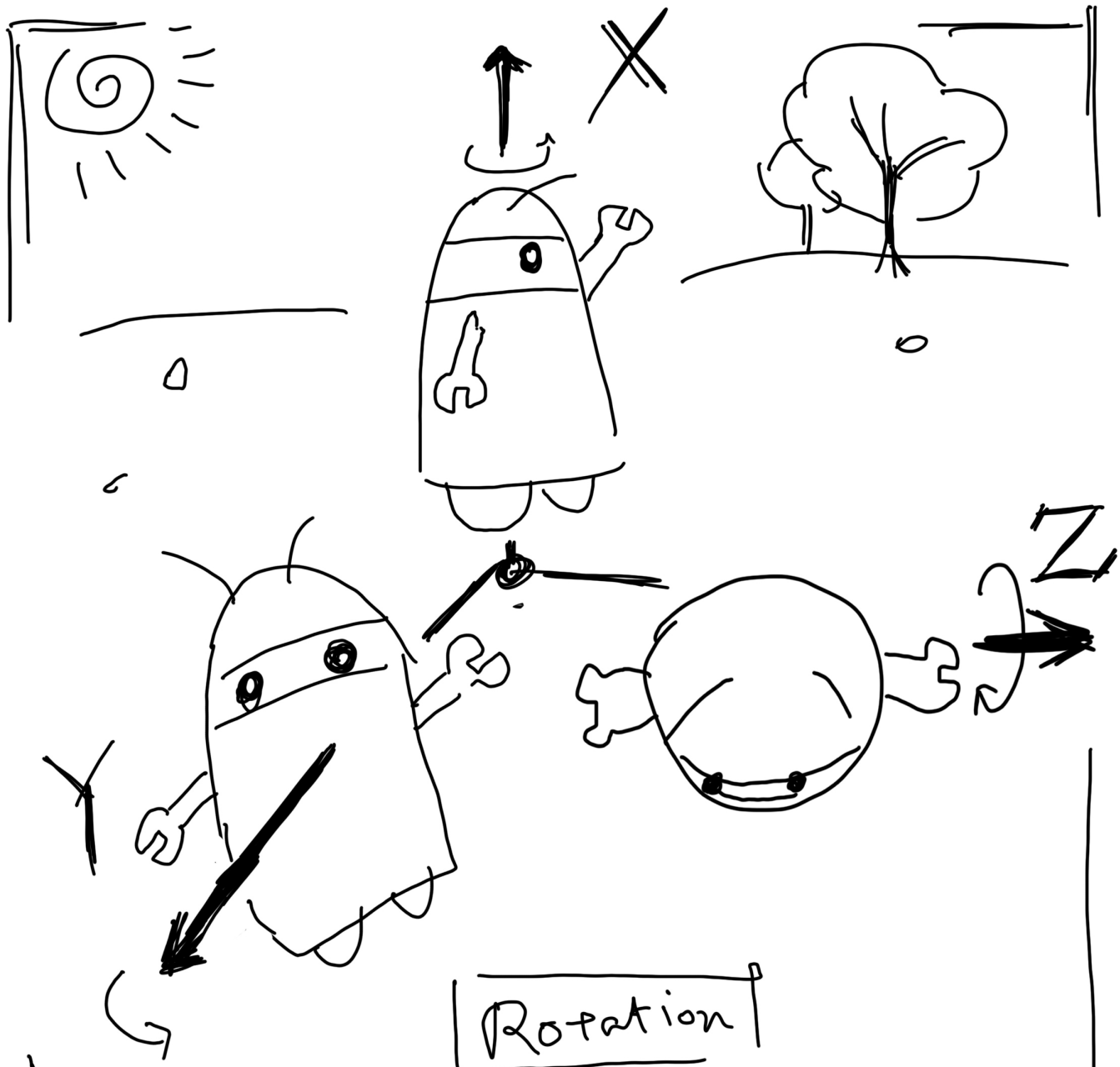
Chapter 1

3D space rigid body movement

main goal

1. understands the description of rigid body motion in three-dimensional space: rotation matrix, transformation matrix, quaternion and Euler angle.
2. grasps the matrix and geometry module usage of the Eigen library.

In the last lecture, we explained the framework and content of visual SLAM. This lecture will introduce one of the basic problems of visual SLAM: **How to describe the motion of a rigid body in three-dimensional space?** Intuitively, we certainly know that this consists of one rotation plus one translation. Translation does not really have much problem, but the processing of rotation is a hassle. We will introduce the meaning of rotation matrices, quaternions, Euler angles, and how they are computed and transformed. In the practice section, we will introduce the linear algebra library Eigen. It provides a C++ matrix calculation, and its Geometry module also provides the structure described quaternion like rigid body motion. Eigen's optimization is perfect, but there are some special places to use it, we will leave it to the program.



Rotation

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

roll, pitch, yaw

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

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

1.1 rotation matrix

1.1.1 point and vector, coordinate system

The space in our daily life is three-dimensional, so we are born to be used to the movement of three-dimensional space. The three-dimensional space consists of three axes, so the position of one spatial point can be specified by three coordinates. However, we should now consider **rigid body** , which has not only its position, but also its own posture. The camera can also be viewed as a rigid body in three dimensions, so the position is where the camera is in space, and the attitude is the orientation of the camera. Combined, we can say, "The camera is in the space $(0, 0, 0)$ point, facing the front". But this natural language is cumbersome, and we prefer to describe it in a mathematical language.

We start with the most basic content: **dots** and **vector**. Points are the basic elements in space, no length, no volume. Connect the two points to form a vector. A vector can be thought of as an arrow pointing from one point to another. Need to remind the reader, please do not confuse the vector with its **coordinate** concept. A vector is one of the things in space, such as $\boldsymbol{a}$. Here $\boldsymbol{a}$ does not need to be associated with several real numbers. Only when we specify a **coordinate system** in this three-dimensional space can we talk about the coordinates of the vector in this coordinate system, that is, find several real numbers corresponding to this vector.

With the knowledge of linear algebra, the coordinates of a point in 3D space can also be described by $\mathbb{R}^3$. How to describe it? Suppose that in this linear space, we find a set of **base** [1] $(\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3)$, then, the arbitrary vector $\boldsymbol{a}$ has a **coordinate** under this set of bases:

$$\boldsymbol{a} = [\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3] \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = a_1\boldsymbol{e}_1 + a_2\boldsymbol{e}_2 + a_3\boldsymbol{e}_3. \tag{1.1}$$

Here $(a_1, a_2, a_3)^{\mathrm{T}}$ is called $\boldsymbol{a}$ under this base coordinates [2]. The specific value of the coordinates, one is related to the vector itself, and the other is related to the selection of the coordinate system (base). The coordinate system usually consists of 3 orthogonal coordinate axes (although it can also be non-orthogonal, it is rare in practice). For example, given $\boldsymbol{x}$ and $\boldsymbol{y}$ axis, the $\boldsymbol{z}$ axis can pass the right-hand (or left-hand) rule by $\boldsymbol{x} \times \boldsymbol{y}$Defined. According to different definitions, the coordinate system is divided into left-handed and right-handed. The third axis of the left hand system is opposite to the right hand system. Most 3D libraries use right-handed (such as OpenGL, 3D Max, etc.), and some libraries use left-handed (such as Unity, Direct3D, etc.).

Based on basic linear algebra knowledge, we can talk about vectors and vectors, and operations between vectors and numbers, such as number multiplication, addition, subtraction, inner product, outer product, and so on. Multi-

[1] of the space in case the reader forgets that the base is a set of linearly independent vectors of Zhang Cheng's space, and some books are also called **Base**.

[2] book vector is column vector, this and general mathematics Books are similar.

plication and addition and subtraction are both fairly basic and intuitive. For example, the result of adding two vectors is to add their respective coordinates, subtraction, and so on. I won't go into details here. Internal and external products may be somewhat unfamiliar to the reader, and their calculations are given here. For $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^3$, in the usual sense [3] can be written as:

$$\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{a}^{\mathrm{T}} \boldsymbol{b} = \sum_{i=1}^{3} a_i b_i = |\boldsymbol{a}| \, |\boldsymbol{b}| \cos \langle \boldsymbol{a}, \boldsymbol{b} \rangle . \tag{1.2}$$

Where $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ refers to the angle between the vector $\boldsymbol{a}, \boldsymbol{b}$. The inner product can also describe the projection relationship between vectors. The outer product is like this:

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{Vmatrix} \boldsymbol{e}_1 & \boldsymbol{e}_2 & \boldsymbol{e}_3 \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{Vmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \boldsymbol{b} \triangleq \boldsymbol{a}^{\wedge} \boldsymbol{b}. \tag{1.3}$$

The result of the outer product is a vector whose direction is perpendicular to the two vectors, and the size is $|\boldsymbol{a}| \, |\boldsymbol{b}| \, \langle \boldsymbol{a}, \boldsymbol{b} \rangle$, is the directed area of the quadrilateral of the two vectors. For outer product operations, we introduce the $^{\wedge}$ symbol and write $\boldsymbol{a}$ as a matrix. In fact, it is a **antisymmetric matrix** (skew-symmetric matrix) [4], you can record $^{\wedge}$ as an antisymmetric symbol. This writes the outer product $\boldsymbol{a} \times \boldsymbol{b}$ as the multiplication of the matrix and the vector $\boldsymbol{a}^{\wedge} \boldsymbol{b}$, which turns it into a linear Operation. This symbol will be used frequently in the following, please remember it, and this symbol is a one-to-one mapping, meaning that any vector corresponds to a unique anti-symmetric matrix, and vice versa:

$$\boldsymbol{a}^{\wedge} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} . \tag{1.4}$$

At the same time, the reader needs to be reminded that the vector and addition and subtraction, internal and external products can be calculated even when they do not talk about their coordinates. For example, although the inner product can be expressed by the sum of the product products of the two vectors when there are coordinates, the inner product of the two can be calculated by the length and the angle even if their coordinates are not known. Therefore, the inner product result of the two vectors is independent of the selection of the coordinate system.

[3] the inner product also has formal rules, but this book only discusses the usual inner product. The inner product of

[4] antisymmetric matrix $\boldsymbol{A}$ meets $\boldsymbol{A}^{\mathrm{T}} = -\boldsymbol{A}$.

1.1.2 Euclidean transformation between coordinate systems

We often define a variety of coordinate systems in the actual scene. In robotics, you define each coordinate system for each link and joint; in 3D mapping, we also define the coordinate system for each cuboid and cylinder. If you consider a moving robot, it is common practice to set an inertial coordinate system (or world coordinate system) that can be considered stationary, such as ***TODO(Hussein)*** defined coordinate system. At the same time, the camera or robot is a moving coordinate system, such as the coordinate system defined by x_C, y_C, z_C. We might ask: a vector $\boldsymbol{p}$ in the camera's field of view, with coordinates in the camera coordinate system of $\boldsymbol{p}_c$, and in the world coordinate system, its coordinates are $\boldsymbol{p}_w$, then how is the conversion between these two coordinates? At this time, it is necessary to first obtain the coordinate value of the point for the robot coordinate system, and then according to the robot pose **transform** into the world coordinate system. We need a mathematical means to describe this transformation. As we will see later, we can describe it with a matrix $\boldsymbol{T}$.

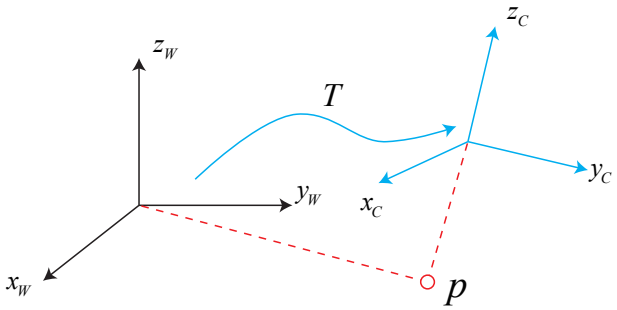


Figure 1.1: Coordinate transformation. For the same vector $thep-$, it coordinates in the world coordinate system ${}_w thep-$ and coordinate in the camera coordinate system ${}_c thep-$ is different. This transformation relationship is described by the transformation matrix $\boldsymbol{T}$.

Intuitively, the motion between two coordinate systems consists of a rotation plus a translation called **rigid body motion**. Camera movement is a rigid body movement. During the rigid body motion, the length and angle of the same vector in each coordinate system will not change. Imagine you throw your phone into the air and [5], there may only be differences in spatial position and posture, and its own length, angle of each face, etc. will not change. The phone will not be squashed like an eraser for a while, and will be stretched for a while. At this point, we say that the phone coordinate system is between the world coordinates, which is a difference of **Euclidean Transform**.

[5] Please don't put it into practice before it falls to the ground .

The Euclidean transformation consists of rotation and translation. We first consider rotation. Let a unit of orthogonal base $(\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3)$ after a rotation becomes $(\boldsymbol{e}'_1, \boldsymbol{e}'_2, \boldsymbol{e}'_3)$. Then, for the same vector $\boldsymbol{a}$ (the vector does not move with the rotation of the coordinate system), its coordinates in two coordinate systems are $[a_1, a_2, a_3]^{\mathrm{T}}$ and $[a'_1, a'_2, a'_3]^{\mathrm{T}}$. Because the vector itself has not changed, according to the definition of coordinates, there are:

$$[\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3] \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = [\boldsymbol{e}'_1, \boldsymbol{e}'_2, \boldsymbol{e}'_3] \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \end{bmatrix}. \tag{1.5}$$

To describe the relationship between the two coordinates, we multiply the left and right sides of the above equation by $\begin{bmatrix} \boldsymbol{e}_1^{\mathrm{T}} \\ \boldsymbol{e}_2^{\mathrm{T}} \\ \boldsymbol{e}_3^{\mathrm{T}} \end{bmatrix}$, then the coefficient on the left becomes the identity matrix, so:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_1^{\mathrm{T}}\boldsymbol{e}'_1 & \boldsymbol{e}_1^{\mathrm{T}}\boldsymbol{e}'_2 & \boldsymbol{e}_1^{\mathrm{T}}\boldsymbol{e}'_3 \\ \boldsymbol{e}_2^{\mathrm{T}}\boldsymbol{e}'_1 & \boldsymbol{e}_2^{\mathrm{T}}\boldsymbol{e}'_2 & \boldsymbol{e}_2^{\mathrm{T}}\boldsymbol{e}'_3 \\ \boldsymbol{e}_3^{\mathrm{T}}\boldsymbol{e}'_1 & \boldsymbol{e}_3^{\mathrm{T}}\boldsymbol{e}'_2 & \boldsymbol{e}_3^{\mathrm{T}}\boldsymbol{e}'_3 \end{bmatrix} \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} \triangleq \boldsymbol{R}\boldsymbol{a}'. \tag{1.6}$$

We take the intermediate matrix out and define it as a matrix $\boldsymbol{R}$. This matrix consists of the inner product between the two sets of bases, characterizing the coordinate transformation relationship of the same vector before and after the rotation. As long as the rotation is the same, then this matrix is the same. It can be said that the matrix $\boldsymbol{R}$ describes the rotation itself. So called **rotation matrix** (Rotation matrix). At the same time, the components of the matrix are the inner product of the two coordinate system bases. Since the length of the base vector is 1, it is actually the cosine of the angle between the base vectors. So this matrix is also called **Direction Cosine Matrix**. We will call it a rotation matrix in the following.

The rotation matrix has some special properties. In fact, it is an orthogonal matrix with a determinant of 1 [6] [7]. Conversely, an orthogonal matrix with a determinant of 1 is also a rotation matrix. So, you can define a collection of n dimensional rotation matrices as follows:

$$\mathrm{SO}(n) = \{\boldsymbol{R} \in \mathbb{R}^{n\times n} | \boldsymbol{R}\boldsymbol{R}^{\mathrm{T}} = \boldsymbol{I}, \det(\boldsymbol{R}) = 1\}. \tag{1.7}$$

$\mathrm{SO}(n)$ isthe meaningof **Special Orthogonal Group**. We leave the contents of the "group" to the next lecture. This collection consists ofa rotation matrix of n dimensional space, in particular, SO(3) refers to the rotation of the three-dimensional space. By rotating the matrix, we can talk directly about the rotation transformation between the two coordinate systems without having to start from the base.

[6] orthogonal matrix that is inversely transposed by itself. The orthogonality of the rotation matrix can be derived directly from the definition.

[7] The determinant is 1 is artificially defined. In fact, only its determinant is ± 1 , but the determinant is -1 is called R rotation, that is, one rotation plus one reflection.

Since the rotation matrix is an orthogonal matrix, its inverse (ie, transpose) describes an opposite rotation. According to the above definition, there are:

$$\boldsymbol{a}' = \boldsymbol{R}^{-1}\boldsymbol{a} = \boldsymbol{R}^{\mathrm{T}}\boldsymbol{a}. \tag{1.8}$$

Obviously $\boldsymbol{R}^{\mathrm{T}}$ portrays an opposite rotation.

In the Euclidean transformation, there is translation in addition to rotation. Consider the vector $\boldsymbol{a}$ in the world coordinate system , after a rotation (depicted by $\boldsymbol{R}$) and a translation of $\boldsymbol{t}$, you get $\boldsymbol{a}'$, then put the rotation and translation together, there are:

$$bma' = \ bmR\ bma + \ bmt. \tag{1.9}$$

Where $\boldsymbol{t}$ is called a translation vector. Compared to rotation, the translation part simply adds the translation vector to the coordinates after the rotation, which is very simple. By the above formula, we completely describe the coordinate transformation relationship of an Euclidean space using a rotation matrix $\boldsymbol{R}$ and a translation vector $\boldsymbol{t}$. In practice, we will define the coordinate system 1, coordinate system 2, then the vector $\boldsymbol{a}$ under the two coordinates is $\boldsymbol{a}_1, \boldsymbol{a}_2$, they are The relationship between the two, in accordance with the complete writing, should be:

$$bma_1 = \boldsymbol{R}_{12}\boldsymbol{a}_2 + \boldsymbol{t}_{12}. \tag{1.10}$$

Here $\boldsymbol{R}_{12}$ means "transform the vector of coordinate system 2 into coordinate system 1". Since the vector is multiplied to the right of this matrix, its subscript is **read from right to left**. This is also the customary way of writing this book. Coordinate transformations are easy to confuse, especially if multiple coordinate systems exist. Similarly, if we want to express "rotation matrix from 1 to 2", we write it as $\boldsymbol{R}_{21}$. The reader must be clear about the notation here, because different books have different writing methods, some will be recorded as the top left/subscript, and the text will be written on the right side.

About panning $\boldsymbol{t}_{12}$, it actually corresponds to the coordinate system 1 origin pointing to the coordinate system 2 origin vector, **coordinates taken under coordinate system**, so I suggest readers to put it It is written as "a vector from 1 to 2." But the reverse $\boldsymbol{t}_{21}$, which is a vector from 2 to 1 **coordinates in coordinate system 2**, is not equal to $-\boldsymbol{t}_{12}$, but related to the rotation of the two systems [8]. Therefore, when beginners ask the question "Where is my coordinates?", we need to clearly explain the meaning of this sentence. Here "my coordinates" actually refers to the vector from the world coordinate system pointing to the origin of the coordinate system of the world, and the coordinates obtained in the world coordinate system. Corresponding to the mathematical symbol, it should be the value of $\boldsymbol{t}_{WC}$. For the same reason, it is not $-\boldsymbol{t}_{CW}$.

1.1.3 transform matrix and homogeneous coordinates

The formula (**??**) fully expresses the rotation and translation of Euclidean space, but there is still a small problem: the transformation relationship here is not a

[8] although from the vector level, they are indeed inverse relations, but the coordinates of the two vectors are not opposite. Can you think about why this is?

linear relationship. Suppose we made two transformations: $\boldsymbol{R}_1, \boldsymbol{t}_1$ and $\boldsymbol{R}_2, \boldsymbol{t}_2$:

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

So, the transformation from $\boldsymbol{a}$ to $\boldsymbol{c}$ is:

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

This form will look awkward after multiple transformations. Therefore, we introduce homogeneous coordinates and transformation matrices, rewriting the form (**??**):

$$\begin{bmatrix} bma' \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{bmatrix} \begin{bmatrix} bma \\ 1 \end{bmatrix} \triangleq \boldsymbol{T} \begin{bmatrix} bma \\ 1 \end{bmatrix}. \tag{1.11}$$

This is a mathematical trick: we add 1 at the end of a 3D vector and turn it into a 4D vector called **homogeneous coordinates**. For this four-dimensional vector, we can write the rotation and translation in a matrix, making the whole relationship a linear relationship. In this formula, the matrix $\boldsymbol{T}$ is called **Transform Matrix**.

We temporarily use $\tilde{\boldsymbol{a}}$ to represent the homogeneous coordinates of $\boldsymbol{a}$. Then, relying on homogeneous coordinates and transformation matrices, the superposition of the two transformations can have a good form:

$$\tilde{\boldsymbol{b}} = \boldsymbol{T}_1 \tilde{\boldsymbol{a}}, \ \tilde{\boldsymbol{c}} = \boldsymbol{T}_2 \tilde{\boldsymbol{b}} \quad \Rightarrow \tilde{\boldsymbol{c}} = \boldsymbol{T}_2 \boldsymbol{T}_1 \tilde{\boldsymbol{a}}. \tag{1.12}$$

But the symbols that distinguish between homogeneous and non-homogeneous coordinates make us annoyed, because here we only need to add 1 at the end of the vector or remove 1 to be [9]. So, without ambiguity, we will write it directly as $\boldsymbol{b} = \boldsymbol{T}\boldsymbol{a}$, and by default we have a homogeneous coordinate conversion [10].

Regarding the transformation matrix $\boldsymbol{T}$, it has a special structure: the upper left corner is the rotation matrix, the right side is the translation vector, the lower left corner is $\boldsymbol{0}$ vector, and the lower right corner is 1. This matrix is also known as the Special Euclidean Group:

$$\mathrm{SE}(3) = \left\{ \boldsymbol{T} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{bmatrix} \in \mathbb{R}^{4\times 4} | \boldsymbol{R} \in \mathrm{SO}(3), \boldsymbol{t} \in \mathbb{R}^3 \right\}. \tag{1.13}$$

Like SO(3) , solving the inverse of the matrix represents an inverse transformation:

$$\boldsymbol{T}^{-1} = \begin{bmatrix} \boldsymbol{R}^{\mathrm{T}} & -\boldsymbol{R}^{\mathrm{T}} \boldsymbol{t} \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{bmatrix}. \tag{1.14}$$

Again, we use the notation of $\boldsymbol{T}_{12}$ to represent a transformation from 2 to 1. Moreover, in order to keep the symbol concise, in the case of no ambiguity,

[9] but the purpose of the homogeneous coordinates is not limited to this, we also in Chapter 7 Will introduce again.

[10] Note that when homogeneous coordinate transformation is not performed, the multiplication here is not true in the matrix dimension.

the symbols of the homogeneous coordinates and the ordinary coordinates are not deliberately distinguished later, and **the default is to use the one that conforms to the algorithm**. For example, when we write $\boldsymbol{Ta}$, we use homogeneous coordinates (otherwise we can't calculate). When you write $\boldsymbol{Ra}$, you use non-homogeneous coordinates. If written in an equation, it is assumed that the conversion from homogeneous coordinates to normal coordinates is already done - because the conversion between homogeneous and non-homogeneous coordinates is actually very easy, but in C++ programs. You can do this with **operator overload** to ensure that the operations you see in the program are uniform.

To review: First, we introduce the vector and its coordinate representation, and introduce the operation between the vectors; then, the motion between the coordinate systems is described by the Euclidean transformation, which consists of translation and rotation. The rotation can be described by the rotation matrix SO(3) , while the translation is directly described by a $\mathbb{R}^3$ vector. Finally, if the translation and rotation are placed in a matrix, the transformation matrix SE(3) is formed .

1.2 Practice: Eigen

The practical part of this lecture has two sections. In the first part, we will explain how to use Eigen to represent matrices and vectors, and then extend to the calculation of rotation matrix and transformation matrix. The code for this section is in **slambook2/ch3/useEigen**.

Eigen [11] is a C++ open source linear algebra library. It provides fast linear algebra operations on matrices, as well as functions such as solving equations. Many upper-level software libraries also use Eigen for matrix operations, including g2o, Sophus, and others. In the theoretical part of this lecture, let's learn about Eigen's programming.

Eigen may not be installed on your PC. Please enter the following command to install:

Listing 1.1: terminal input:

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

Most commonly used libraries are available in the Ubuntu software source. Later, if you want to install a library, you may want to search for the Ubuntu software source. With the apt command, we can easily install Eigen. Looking back at the previous lesson, we know that a library consists of header files and library files. The default location of the Eigen header file is in "/usr/include/eigen3/". If you are not sure, you can find it by entering the following command:

Listing 1.2: terminal input:

```
sudo locate eigen3
```

Compared to other libraries, Eigen is special in that it is a library built with pure header files (this is amazing!). This means you can only find its header files, not binary files like .so or .a. When you use it, you only need to import Eigen's header file, you don't need to link the library file (because it doesn't have a library file). Write a piece of code below to actually practice the use of Eigen:

Listing 1.3: slambook2/ch3/useEigen/eigenMatrix.cpp

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

#include <ctime>
// Eigen core
#include <Eigen/Core>
// Algebraic operations of dense matrices (inverse, eigenvalues, etc.)
#include <Eigen/Dense>
using namespace Eigen;
```

[11] official home page: http://eigen.tuxfamily.org/index.php?title=Main_Page.

```
#define MATRIX_SIZE 50

/**************************
 * This program demonstrates the use of the basic Eigen type
 **************************/

int main(int argc, char **argv) {
  // All vectors and matrices in Eigen are Eigen::Matrix, which is a template
  // class. Its first three parameters are: data type, row, column Declare a 2*3
  // float matrix
  Matrix<float, 2, 3> matrix_23;

  // At the same time, Eigen provides many built-in types via typedef, but the
  // bottom layer is still Eigen::Matrix For example, Vector3d is essentially
  // Eigen::Matrix<double, 3, 1>, which is a three-dimensional vector.
  Vector3d v_3d;
  // This is the same
  Matrix<float, 3, 1> vd_3d;

  // Matrix3d is essentially Eigen::Matrix<double, 3, 3>
  Matrix3d matrix_33 = Matrix3d::Zero(); // initialized to zero
  // If you are not sure about the size of the matrix, you can use a matrix of
  // dynamic size
  Matrix<double, Dynamic, Dynamic> matrix_dynamic;
  // simpler
  MatrixXd matrix_x;
  // There are still many types of this, we doesn't list them one by one.

  // Here is the operation of the Eigen array
  // input data (initialization)
  matrix_23 << 1, 2, 3, 4, 5, 6;
  // output
  cout << "matrix 2x3 from 1 to 6: \n" << matrix_23 << endl;

  // Use () to access elements in the matrix
  cout << "print matrix 2x3: " << endl;
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++)
      cout << matrix_23(i, j) << "\t";
    cout << endl;
  }

  // The matrix and vector are multiplied (actually still matrices and matrices)
  v_3d << 3, 2, 1;
  vd_3d << 4, 5, 6;
```

```
// But in Eigen you can't mix two different types of matrices, like this is
// wrong Matrix<double, 2, 1> result_wrong_type = matrix_23 * v_3d; should be
// explicitly converted
Matrix<double, 2, 1> result = matrix_23.cast<double>() * v_3d;
cout << "[1,2,3;4,5,6]*[3,2,1]=" << result.transpose() << endl;

Matrix<float, 2, 1> result2 = matrix_23 * vd_3d;
cout << "[1,2,3;4,5,6]*[4,5,6]: " << result2.transpose() << endl;

// Also you can't misjudge the dimensions of the matrix
// Try canceling the comments below to see what Eigen will report.
// Eigen::Matrix<double, 2, 3> result_wrong_dimension =
// matrix_23.cast<double>() * v_3d;

// some matrix operations
// Four operations are not demonstrated, just use +-*/.
Matrix_33 = Matrix3d::Random(); // Random Number Matrix
cout << "random matrix: \n" << matrix_33 << endl;
cout << "transpose: \n" << matrix_33.transpose() << endl;
cout << "sum: " << matrix_33.sum() << endl;
cout << "trace: " << matrix_33.trace() << endl;
cout << "times 10: \n" << 10 * matrix_33 << endl;
cout << "inverse: \n" << matrix_33.inverse() << endl;
cout << "det: " << matrix_33.determinant() << endl;

// Eigenvalues
// Real symmetric matrix can guarantee successful diagonalization
SelfAdjointEigenSolver<Matrix3d> eigen_solver(matrix_33.transpose() *
                                              matrix_33);
cout << "Eigen values = \n" << eigen_solver.eigenvalues() << endl;
cout << "Eigen vectors = \n" << eigen_solver.eigenvectors() << endl;

// Solving equations
// We solve the equation of matrix_NN * x = v_Nd
// The size of N is defined in the previous macro, which is generated by a
// random number Direct inversion is the most direct, but the amount of
// inverse operations is large.

Matrix<double, MATRIX_SIZE, MATRIX_SIZE> matrix_NN =
    MatrixXd::Random(MATRIX_SIZE, MATRIX_SIZE);
matrix_NN =
    matrix_NN * matrix_NN.transpose(); // Guarantee semi-positive definite
Matrix<double, MATRIX_SIZE, 1> v_Nd = MatrixXd::Random(MATRIX_SIZE, 1);

Clock_t time_stt = clock(); // timing
// Direct inversion
```

```
    Matrix<double, MATRIX_SIZE, 1> x = matrix_NN.inverse() * v_Nd;
    cout << "time of normal inverse is "
         << 1000 * (clock() - time_stt) / (double)CLOCKS_PER_SEC << "ms" << endl;
    cout << "x = " << x.transpose() << endl;

    // Usually solved by matrix decomposition, such as QR decomposition, the speed
    // will be much faster
    time_stt = clock();
    x = matrix_NN.colPivHouseholderQr().solve(v_Nd);
    cout << "time of Qr decomposition is "
         << 1000 * (clock() - time_stt) / (double)CLOCKS_PER_SEC << "ms" << endl;
    cout << "x = " << x.transpose() << endl;

    // For positive definite matrices, you can also use cholesky decomposition to
    // solve equations.
    time_stt = clock();
    x = matrix_NN.ldlt().solve(v_Nd);
    cout << "time of ldlt decomposition is "
         << 1000 * (clock() - time_stt) / (double)CLOCKS_PER_SEC << "ms" << endl;
    cout << "x = " << x.transpose() << endl;

    return 0;
}
```

This example demonstrates the basic operations and operations of the Eigen matrix. To compile it, you need to specify the header file directory of Eigen in CMakeLists.txt:

Listing 1.4: slambook2/ch3/useEigen/CMakeLists.txt

```
# Add header file
include_directories( "/usr/include/eigen3" )
```

Repeat, because the Eigen library only has header files, so you don't need to link the program to the library with the target _ link _ libraries statement. However, for most other libraries, most of the time you need to use the link command. The approach here is not necessarily the best, because others may have Eigen installed in different locations, then you must manually modify the header file directory here. In the rest of the work, we will use the find _ package command to search the library, but for the time being in this lecture. After compiling this program, run it and you can see the output of each matrix.

Listing 1.5: Terminal input:

```
% build/eigenMatrix
matrix 2x3 from 1 to 6:
1 2 3
4 5 6
print matrix 2x3:
```

```
1         2         3
4         5         6
[1,2,3;4,5,6]*[3,2,1]=10  28
[1,2,3;4,5,6]*[4,5,6]: 32  77
random matrix:
0.680375    0.59688  -0.329554
-0.211234   0.823295   0.536459
0.566198  -0.604897  -0.444451
transpose:
0.680375  -0.211234   0.566198
0.59688   0.823295  -0.604897
-0.329554   0.536459  -0.444451
sum: 1.61307
trace: 1.05922
times 10:
6.80375    5.9688  -3.29554
-2.11234   8.23295   5.36459
5.66198  -6.04897  -4.44451
inverse:
-0.198521    2.22739     2.8357
1.00605  -0.555135   -1.41603
-1.62213    3.59308    3.28973
it: 0.208598
```

Since the detailed comments are given in the code, each line of the statement is not explained here. In this book, we will only give a description of several important places (the latter part will also maintain this style).

1. readers are best to enter the above code (not including comments). At least compile and run the above program.

2. Kdevelop may not prompt C++ member operations, which is caused by its incompleteness. Please follow the above to enter, do not care if it prompts an error. Clion will give you a complete hint.

   The matrix provided by

3. Eigen is very similar to MATLAB, and almost all data is treated as a matrix. However, in order to achieve better efficiency, you need to specify the size and type of the matrix in Eigen. For matrices that know the size at compile time, they are processed faster than dynamically changing matrices. Therefore, data such as rotation matrices and transformation matrices can be determined at compile times by their size and data type.

   The matrix implementation inside

4. Eigen is more complicated. I won't introduce it here. We hope that you can use Eigen's matrix like the built-in data types like float and double. This should be in line with the original intention of its design.

The

5. Eigen matrix does not support automatic type promotion, which is quite different from C++'s built-in data types. In a C++ program, we can add and multiply a float data and double data, and **the compiler will automatically convert the data type to the most appropriate one**. In Eigen, for performance reasons, you must **explicitly** convert the matrix type. And if you forget to do this, Eigen will (not very friendly) prompt you with a "YOU MIXED DIFFERENT NUMERIC TYPES ..." compilation error. You can try to find out which part of the error message this message appears in. If the error message is too long, it is best to save it to a file and find it.

6. is the same, in the calculation process also need to ensure the correctness of the matrix dimension, otherwise there will be "YOU MIXED MATRICES OF DIFFERENT SIZES" error. Please don't complain about this kind of error prompting. For C++ template meta-programming, it is very lucky to be able to prompt the information that can be read. Later, if you find that Eigen is wrong, you can directly look for the uppercase part and figure out what the problem is.

7. Our routines only cover basic matrix operations. You can read more about Eigen by reading the Eigen official website tutorial: http://eigen.tuxfamily.org/dox-devel/modules.html . Only the simplest part is demonstrated here. It is not equal to the fact that you can understand Eigen.

In the last piece of code, the efficiency of inversion and QR decomposition is compared. You can look at the time difference on your own machine. Is there a significant difference between the two methods?

1.3 Rotation vector and Euler angle

1.3.1 rotation vector

We return to the theoretical part. With a rotation matrix to describe the rotation, is there enough a transformation matrix to describe a 6-degree-of-freedom three-dimensional rigid body motion? Matrix representation has at least the following disadvantages:

1. SO(3) has a rotation matrix of 9 quantities, but only 3 degrees of freedom in one rotation. Therefore this expression is redundant. Similarly, the transformation matrix expresses a 6-degree-of-freedom transformation with 16 quantities. So, is there a more compact representation? The

2. rotation matrix itself has constraints: it must be an orthogonal matrix with a determinant of 1. The same is true for the transformation matrix. These constraints make the solution more difficult when you want to estimate or optimize a rotation matrix/transform matrix.

Therefore, we hope that there is a way to describe rotation and translation in a compact manner. For example, is it feasible to express rotation with a three-dimensional vector and express transformation with a six-dimensional vector? In fact, any rotation can be characterized by **a rotation axis and a rotation angle**. Thus, we can use a vector whose direction is consistent with the axis of rotation and the length is equal to the angle of rotation. This vector is called **rotation vector** (or axis/angle axis, Axis-Angle), and only a three-dimensional vector is needed to describe the rotation. Similarly, for a transformation matrix, we use a rotation vector and a translation vector to express a transformation. The variable dimension at this time is exactly six dimensions.

Consider a rotation represented by $\boldsymbol{R}$. If described by a rotation vector, assuming that the rotation axis is a unit length vector $\boldsymbol{n}$ and the angle is θ, then the vector $\theta\boldsymbol{n}$ can also describe this rotation. So, we have to ask, what is the connection between the two expressions? In fact, it is not difficult to derive their conversion relationship. The conversion process from the rotation vector to the rotation matrix is shown by **Rodrigues's Formula**. Since the derivation process is more complicated, it is not described here. Only the result of the conversion is given. [12]:

$$bmR = \cos\theta\boldsymbol{I} + (1 - \cos\theta)\,\boldsymbol{n}\boldsymbol{n}^{\mathrm{T}} + \sin\theta\boldsymbol{n}^{\wedge}. \tag{1.15}$$

The symbol $^{\wedge}$ is a vector to anti-symmetric conversion, see the formula (**??**). Conversely, we can also calculate the conversion from a rotation matrix to a rotation vector. For the corner θ, take the **track** [13],Have:

[12]For interested readers, please refer to https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula, in fact the next chapter will give a proof from the Lie algebra level.

[13]see **trace** on both sides to find the sum of the diagonal elements of the matrix.

$$\begin{aligned}\operatorname{tr}(\boldsymbol{R}) &= \cos\theta \operatorname{tr}(\boldsymbol{I}) + (1-\cos\theta)\operatorname{tr}\left(\boldsymbol{n}\boldsymbol{n}^{\mathrm{T}}\right) + \sin\theta \operatorname{tr}(\boldsymbol{n}^{\wedge}) \\ &= 3\cos\theta + (1-\cos\theta) \\ &= 1 + 2\cos\theta.\end{aligned} \tag{1.16}$$

therefore:

$$\theta = \arccos(\frac{\operatorname{tr}(\boldsymbol{R})-1}{2}). \tag{1.17}$$

Regarding the recursive axis $\boldsymbol{n}$, since the vector on the rotation axis does not change after the rotation, it means:

$$\boldsymbol{R}\boldsymbol{n} = \boldsymbol{n}. \tag{1.18}$$

Therefore, the recursive $\boldsymbol{n}$ is the eigen vector corresponding to the matrix $\boldsymbol{R}$ eigenvalue 1. Solving this equation and normalizing it gives the axis of rotation. The reader can also look at this equation from the geometrical point of "the axis of rotation is unchanged after rotation". By the way, the two conversion formulas here will still appear in the next lecture, and you will find that they are exactly the correspondence between Lie group and Lie algebra on SO(3) .

1.4 Quadra

1.4.1 definition of quaternion

The rotation matrix describes the rotation of 3 degrees of freedom with 9 quantities, with redundancy; the Euler angles and the rotation vectors are compact but singular. In fact, we **cannot find a three-dimensional vector description without singularity** [?]. This is somewhat similar to using two coordinates to represent the Earth's surface (such as longitude and latitude), and there will be singularity (latitude is meaningless when latitude is $\pm 90^\circ$).

Recall the plurals that I have studied before. We use the complex set $\mathbb{C}$ to represent the vector on the complex plane, and the complex multiplication represents the rotation on the complex plane: for example, multiplying the complex i is equivalent to rotating a complex vector counterclockwise by 90° . Similarly, when expressing a three-dimensional space rotation, there is also an algebra similar to a complex number: **quaternary**. The quaternion is an extended complex number found by Hamilton. It **is both compact and singular**. If you say the shortcomings, the quaternion is not intuitive enough, and its operation is a bit more complicated.

Comparing quaternions to complex numbers can help you understand quaternions faster. For example, when we want to rotate the vector of a complex plane by θ , we can multiply this complex vector by $\mathrm{e}^{i\theta}$. This is a complex number represented by polar coordinates. It can also be written in the usual form, as long as the Euler formula is used:

$$\mathrm{e}^{i\theta} = \cos\theta + i\sin\theta. \tag{1.19}$$

This is a plural of unit length. Therefore, in the case of two dimensions, the rotation can be described by **unit plural**. Similarly, we will see that 3D rotation can be described by **unit quaternion**.

A quaternion $\boldsymbol{q}$ has a real part and three imaginary parts. The book writes the real part in front (and there are places where the real part is written later), like this:

$$\boldsymbol{q} = q_0 + q_1 i + q_2 j + q_3 k, \tag{1.20}$$

Where i, j, k are the three imaginary parts of the quaternion. These three imaginary parts satisfy the following relationship:

$$\begin{cases} i^2 = j^2 = k^2 = -1 \\ ij = k, ji = -k \\ jk = i, kj = -i \\ ki = j, i = -j \end{cases}. \tag{1.21}$$

If we look at i, j, k as three axes, they are the same as their own multiplications and complex numbers, and the multiplication and outer product are the same. Sometimes people also use a scalar and a vector to express quaternions:

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

Here, s is called the real part of the quaternion, and $\boldsymbol{v}$ is called its imaginary part. If the imaginary part of a quaternion is $\mathbf{0}$, it is called **real quaternion**. Conversely, if its real part is 0 , it is called **imaginary quaternion**.

Considering that 3D space requires 3 axes and Quaternion also has 3 imaginary parts, can a virtual quaternion correspond to a space point? In fact, we are doing this.

You can use **unit quaternion** to represent any rotation in 3D space, but this expression is subtly different from the plural. In the plural, multiplying by i means rotating 90° . Does this mean that the quaternion, multiplied by i is rotated around the i axis by 90° ? So, $ij = k$ does that mean, first around the i transfer 90° , then around J transfer 90° , equivalent to around k turn 90° ? Readers can find a cell phone to plan - then you will find that this is not the case. The correct situation should be that multiplying i corresponds to rotating 180° , in order to guarantee the nature of $ij = k$. And $i^2 = -1$ means that after rotating 360° around the i axis, I get an opposite thing. This thing has to be rotated for two weeks to be equal to its original appearance.

This seems a bit mysterious, the complete explanation needs to introduce too much extra things, we still calm down and come back to the eyes. At least, we know that a unit quaternion can express the rotation of a three-dimensional space. So what are the nature of the quaternions themselves, and what can they do with each other? Let us first examine the algorithm between quaternions.

1.4.2 Quad operation

A quaternion is the same as a normal complex, and a series of operations can be performed. Commonly there are four arithmetic operations, multiplication, inversion, conjugate, and so on. The following are introduced separately.

There are two quaternions $\boldsymbol{q}_a, \boldsymbol{q}_b$, whose vectors are represented as $[s_a, \boldsymbol{v}_a]^{\mathrm{T}}, [s_b,\ Bmv_b]^{\mathrm{T}}$, or the original quaternion is expressed as:

$$bmq_a = s_a + x_a i + y_a + z_a k,\ quad\ bmq_b = execution + x_b i + y_b j + z_b k.$$

Then, its operation can be expressed as follows.

1. *addition and subtraction*

   The addition and subtraction of the quaternion $\boldsymbol{q}_a, \boldsymbol{q}_b$ is:

   $$\boldsymbol{q}_a \pm \boldsymbol{q}_b = \left[s_a \pm s_b, \boldsymbol{v}_a \pm \boldsymbol{v}_b\right]^{\mathrm{T}}. \tag{1.22}$$

2. *multiplication*

   Multiplication is the multiplication of each item of $\boldsymbol{q}_a$ with each item of $\boldsymbol{q}_b$, and finally, the imaginary part is done according to the formula (**??**). Finishing is available:

   $$\begin{aligned} \boldsymbol{q}_a\boldsymbol{q}_b = & s_a s_b - x_a x_b - y_a y_b - z_a z_b \\ & + \left(s_a x_b + x_a s_b + y_a z_b - z_a y_b\right) i \\ & + \left(s_a y_b - x_a z_b + y_a s_b + z_a x_b\right) j \\ & + \left(s_a z_b + x_a y_b - y_a x_b + z_a s_b\right) k. \end{aligned} \tag{1.23}$$

Although a little complicated, the form is neat and orderly. If written in vector form and using internal and external product operations, the expression will be more concise:

$$\boldsymbol{q}_a\boldsymbol{q}_b = \left[s_a s_b - \boldsymbol{v}_a^{\mathrm{T}}\boldsymbol{v}_b, s_a\boldsymbol{v}_b + s_b\boldsymbol{v}_a + \boldsymbol{v}_a \times \boldsymbol{v}_b\right]^{\mathrm{T}}. \tag{1.24}$$

Under this multiplication definition, the two real quaternion products are still real, which is also consistent with the complex number. However, note that due to the existence of the last outer product, quaternion multiplication is usually not commutative unless $\boldsymbol{v}_a$ and $\boldsymbol{v}_b$ at $\mathbb{R}^3$ collinear line, at which point the outer product term is zero.

3. *module*

   The modulus of a quaternion is defined as

$$\|\boldsymbol{q}_a\| = \sqrt{s_a^2 + x_a^2 + y_a^2 + z_a^2}. \tag{1.25}$$

   It can be verified that the modulus of the product of two quaternions is the product of the modulo. This makes the unit quaternion still multiplied by the unit quaternion.

$$\|\boldsymbol{q}_a\boldsymbol{q}_b\| = \|\boldsymbol{q}_a\|\|\boldsymbol{q}_b\|. \tag{1.26}$$

4. *conjugate*

   The conjugate of a quaternion is to take the imaginary part as the opposite:

$$rmq_a^* = s_a - x_a i - y_a j - z_a k = [s_a, -\mathrm{the_a}]^{\frac{\mathrm{T}}{\cdot}} \tag{1.27}$$

   The quaternion conjugate is multiplied by itself, and a real quaternion is obtained. The actual part is the square of the modulo length:

$$\boldsymbol{q}^*\boldsymbol{q} = \boldsymbol{q}\boldsymbol{q}^* = [s_a^2 + \boldsymbol{v}^{\mathrm{T}}\boldsymbol{v}, \boldsymbol{0}]^{\mathrm{T}}. \tag{1.28}$$

5. *reverse*

   The inverse of a quaternion is

$$bmq^{-1} = \boldsymbol{q}^* / \ |\boldsymbol{q}\ |^2. \tag{1.29}$$

   According to this definition, the product of the quaternion and its own inverse is the real quaternion $\mathbf{1}$:

$$\boldsymbol{q}\boldsymbol{q}^{-1} = \boldsymbol{q}^{-1}\boldsymbol{q} = \mathbf{1}. \tag{1.30}$$

   If $\boldsymbol{q}$ is a unit quaternion, its inverse and conjugate are the same amount. At the same time, the inverse of the product has properties similar to matrices:

$$(\boldsymbol{q}_a\boldsymbol{q}_b)^{-1} = \boldsymbol{q}_b^{-1}\boldsymbol{q}_a^{-1}. \tag{1.31}$$

6. *multiplication*

   Similar to vectors, quaternions can be multiplied by numbers:

$$k\boldsymbol{q} = [ks, k\boldsymbol{v}]^{\mathrm{T}}. \tag{1.32}$$

1.4.3 Use quaternion to represent rotation

We can use a quaternion to express the rotation of a point. Suppose a spatial 3D point $\boldsymbol{p} = [x, y, z] \in \mathbb{R}^3$, and a rotation specified by the unit quaternion $\boldsymbol{q}$. The 3D point $\boldsymbol{p}$ is rotated to become $\boldsymbol{p}'$. If you use a matrix description, then there is $\boldsymbol{p}' = \boldsymbol{R}\boldsymbol{p}$. And if you use quaternions to describe rotation, how do they relate to their relationship?

First, describe the 3D space point with a virtual quaternion:

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

This is equivalent to matching the three imaginary parts of the quaternion with the three axes in the space. Then, the rotated point $\boldsymbol{p}'$ can be expressed as such a product:

$$\boldsymbol{p}' = \boldsymbol{q}\boldsymbol{p}\boldsymbol{q}^{-1}. \tag{1.33}$$

The multiplication here is quaternion multiplication, and the result is also a quaternion. Finally, take the imaginary part of $\boldsymbol{p}'$ and get the coordinates of the point after the rotation. Moreover, it can be verified (reserved as an exercise), and the real part of the calculation result is 0, so it is a pure virtual quaternion.

1.4.4 conversion of quaternions to other rotation representations

An arbitrary unit quaternion describes a rotation, which can also be described by a rotation matrix or a rotation vector. Now let's examine the conversion relationship between quaternions and rotation vectors and rotation matrices. Before that, we have to say that quaternion multiplication can also be written as a matrix multiplication. Let $\boldsymbol{q} = [s, \boldsymbol{v}]^{\mathrm{T}}$, then define the following symbols $^{+}$ and $^{\oplus}$ for $^{\text{auf cite}}$ Barfoot2011:

$$\boldsymbol{q}^{+} = \begin{bmatrix} s & -\boldsymbol{v}^{\mathrm{T}} \\ \boldsymbol{v} & s\boldsymbol{I} + \boldsymbol{v}^{\wedge} \end{bmatrix}, \quad \boldsymbol{q}^{\oplus} = \begin{bmatrix} s & -\boldsymbol{v}^{\mathrm{T}} \\ \boldsymbol{v} & s\boldsymbol{I} - \boldsymbol{v}^{\wedge} \end{bmatrix}, \tag{1.34}$$

These two symbols map the quaternion to a matrix of 4×4 . Then quaternion multiplication can be written in the form of a matrix:

$$\boldsymbol{q}_1^{+}\boldsymbol{q}_2 = \begin{bmatrix} s_1 & -\boldsymbol{v}_1^{T} \\ \boldsymbol{v}_1 & s_1\boldsymbol{I} + \boldsymbol{v}_1^{\wedge} \end{bmatrix} \begin{bmatrix} s_2 \\ \boldsymbol{v}_2 \end{bmatrix} = \begin{bmatrix} -\boldsymbol{v}_1^{T}\boldsymbol{v}_2 + s_1 s_2 \\ s_1\boldsymbol{v}_2 + s_2\boldsymbol{v}_1 + \boldsymbol{v}_1^{\wedge}\boldsymbol{v}_2 \end{bmatrix} = \boldsymbol{q}_1\boldsymbol{q}_2 \tag{1.35}$$

The same can also be proved:

$$\boldsymbol{q}_1\boldsymbol{q}_2 = \boldsymbol{q}_1^{+}\boldsymbol{q}_2 = \boldsymbol{q}_2^{\oplus}\boldsymbol{q}_1. \tag{1.36}$$

Then, consider the problem of using a quaternion to rotate a spatial point. According to the previous statement, there are:

$$\begin{aligned} \boldsymbol{p}' &= \boldsymbol{q}\boldsymbol{p}\boldsymbol{q}^{-1} = \boldsymbol{q}^{+}\boldsymbol{p}^{+}\boldsymbol{q}^{-1} \\ &= \boldsymbol{q}^{+}{\boldsymbol{q}^{-1}}^{\oplus}\boldsymbol{p}. \end{aligned} \tag{1.37}$$

Substituting the matrix corresponding to two symbols, you get:

$$\boldsymbol{q}^{+}\left(\boldsymbol{q}^{-1}\right)^{\oplus} = \begin{bmatrix} s & -\boldsymbol{v}^{T} \\ \boldsymbol{v} & s\boldsymbol{I} + \boldsymbol{v}^{\wedge} \end{bmatrix} \begin{bmatrix} s & \boldsymbol{v}^{T} \\ -\boldsymbol{v} & s\boldsymbol{I} + \boldsymbol{v}^{\wedge} \end{bmatrix} = \begin{bmatrix} 1 & \boldsymbol{0} \\ \boldsymbol{0}^{\mathrm{T}} & \boldsymbol{v}\boldsymbol{v}^{\mathrm{T}} + s^{2}\boldsymbol{I} + 2s\boldsymbol{v}^{\wedge} + \left(\boldsymbol{v}^{\wedge}\right)^{2} \end{bmatrix}. \tag{1.38}$$

Since $\boldsymbol{p}'$ and $\boldsymbol{p}$ are both virtual quaternions, the fact that the bottom right corner of the matrix gives the transformation of **from quaternion to rotation matrix**:

$$\boldsymbol{R} = \boldsymbol{v}\boldsymbol{v}^{\mathrm{T}} + s^{2}\boldsymbol{I} + 2s\boldsymbol{v}^{\wedge} + \left(\boldsymbol{v}^{\wedge}\right)^{2}. \tag{1.39}$$

In order to obtain the conversion formula of the quaternion to the rotation vector, the two sides of the above formula are traced to obtain:

$$\begin{aligned} \mathrm{tr}(\boldsymbol{R}) &= \mathrm{tr}(\boldsymbol{v}\boldsymbol{v}^{\mathrm{T}} + 3s^{2} + 2s \cdot 0 + \mathrm{tr}((\boldsymbol{v}^{\wedge})^{2}) \\ &= v_1^2 + v_2^2 + v_3 s^2 + 3s^2 - 2(v_1^2 + v_2^2 + v_3^2) \\ &= (1 - s^2) + 3s^2 - 2(1 - s^2) \\ &= 4s^2 - 1. \end{aligned} \tag{1.40}$$

Also obtained by the formula (**??**):

$$\begin{aligned} \theta &= \arccos(\frac{\mathrm{tr}(\boldsymbol{R} - 1)}{2}) \\ &= \arccos(2s^2 - 1). \end{aligned} \tag{1.41}$$

which is

$$\cos\theta = 2s^2 - 1 = 2\cos^2\frac{\theta}{2} - 1, \tag{1.42}$$

and so:

$$\theta = 2\arccos s. \tag{1.43}$$

As the rotary shaft, if the formula (**??**) by $\boldsymbol{Q}$ imaginary portion instead of $\boldsymbol{P}$, easy to know $\boldsymbol{Q}Thevectoroftheimaginarypartof$ is not moving when it is rotated, that is, it constitutes the rotation axis. So just remove it from its modulus, you get it. In summary, the conversion formula for quaternion to rotation vector can be written as follows:

$$\begin{cases} \theta = 2\arccos q_0 \\ [n_x, n_y, n_z]^{\mathrm{T}} = [q_1, q_2, q_3]^{\mathrm{T}} / \sin\frac{\theta}{2} \end{cases}. \tag{1.44}$$

As for how to switch from other methods to quaternions, you only need to reverse the above steps. In actual programming, the library usually prepares for the conversion between various forms for us. Whether it's a quaternion, a rotation matrix, or a shaft angle, they can all be used to describe the same rotation. We should choose the most convenient form in practice without having to stick to a particular form. In the subsequent practices and exercises, we will demonstrate the transition between various expressions to deepen the reader's impression.

The conversion of What is the relationship between The TODO(Hussein)

In addition to the Euclidean transformation, there are several other transformations in the 3D space, but the Euclidean transformation is the simplest. Some of them are related to the measurement geometry, as they may be mentioned in the following explanations, so list them first. The Euclidean transformation maintains the length and angle of the vector, which is equivalent to moving or rotating a rigid body intact without changing its appearance. Several other transformations will change its shape. They all have similar matrix representations.

1. *similar transformation*

   The similarity transformation has one more degree of freedom than the Euclidean transformation, which allows the object to be uniformly scaled, and its matrix is expressed as

   $$\boldsymbol{T}_S = \begin{bmatrix} s\boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{bmatrix}. \tag{1.45}$$

   Notice that the rotation part has a scaling factor of s , which means that we can evenly scale the three coordinates of x, y, z after the vector is rotated . Due to the inclusion of scaling, the similar transformation no longer keeps the area of the graphic unchanged. You can imagine a cube with a side length of 1 transforming into a side with a length of 10 (but still a cube). The set of three-dimensional similar transforms is also called **similar transform group**, which is denoted as Sim(3) .

2. *affine transformation*

   The matrix form of the affine transformation is as follows:

   $$\boldsymbol{T}_A = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{t} \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{bmatrix}. \tag{1.46}$$

   Unlike the Euclidean transformation, the affine transformation only requires $\boldsymbol{A}$ to be an invertible matrix, not necessarily an orthogonal matrix. An affine transformation is also called an orthogonal projection. After the affine transformation, the cube is no longer square, but the faces are still parallelograms.

3. *projection transformation*

   Projective transformation is the most general transformation, its matrix form is

$$\boldsymbol{T}_P = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{t} \\ \boldsymbol{a}^{\mathrm{T}} & v \end{bmatrix}. \tag{1.47}$$

   Its upper left corner is the reversible matrix $\boldsymbol{A}$, the upper right corner is the translation $\boldsymbol{t}$, and the lower left corner is the scale $\boldsymbol{a}^{\mathrm{T}}$. Since the homogeneous coordinates are used, when $v \neq 0$, we can divide the entire matrix by v to get a matrix with a bottom right corner of 1; otherwise we get a matrix with a lower right corner of 0 . Therefore, the 2D projective transformation has a total of 8 degrees of freedom, and 3D has a total of 15 degrees of freedom. Projective transformation is the most common form of transformation that has been said so far. The transformation from the real world to the camera photo can be seen as a projective transformation. The reader can imagine what a square tile would look like in a photo: first, it is no longer square. Due to the near-large and small relationship, it is not even a parallelogram, but an irregular quadrilateral.

?? summarizes the nature of several transformations currently covered. Note that in the "invariant nature", there is an inclusion relationship from top to bottom. For example, in addition to maintaining volume, the Euclidean transformation also has the properties of parallelism, intersection, and the like.

TODO(Hussein)

We will later say that the transformation from the real world to the camera photo is a projective transformation. If the focal length of the camera is infinity, then this transformation is an affine transformation. However, before we go into the details of the camera model, we just have a rough impression of them.

1.5 Practice: Eigen Geometry Module

1.5.1 Data demonstration of the Eigen geometry module

Now, let's actually practice the various rotation expressions mentioned earlier. We will use quaternions, Euler angles, and rotation matrices in Eigen to demonstrate how they are transformed. We will also give a visualization program to help the reader understand the relationship of these transformations.

Listing 1.6: slambook2/ch3/useGeometry/useGeometry.cpp

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

#include <Eigen/Core>
#include <Eigen/Geometry>

using namespace Eigen;
// This program demonstrates how to use the Eigen geometry module

int main(int argc, char **argv) {
    // The Eigen/Geometry module provides a variety of rotation and translation
    // 3D rotation matrix directly using Matrix3d or Matrix3f
    Matrix3d rotation_matrix = Matrix3d::Identity();
    // The rotation vector uses AngleAxis, the underlying layer is not directly
    AngleAxisd rotation_vector(M_PI / 4, Vector3d(0, 0, 1)); //Rotate 45 degrees
    cout.precision(3);
    cout << "rotation matrix =\n" << rotation_vector.matrix() << endl; //conve
    // can also be assigned directly
    rotation_matrix = rotation_vector.toRotationMatrix();
    // coordinate transformation with AngleAxis
    Vector3d v(1, 0, 0);
    Vector3d v_rotated = rotation_vector * v;
    cout << "(1,0,0) after rotation (by angle axis) = " << v_rotated.transpose()
    // Or use a rotation matrix
    v_rotated = rotation_matrix * v;
    cout << "(1,0,0) after rotation (by matrix) = " << v_rotated.transpose() <<

    // Euler angle: You can convert the rotation matrix directly into Euler angl
    Vector3d euler_angles = rotation_matrix.eulerAngles(2, 1, 0); // ZYX order,
    cout << "yaw pitch roll = " << euler_angles.transpose() << endl;

    // Euclidean transformation matrix using Eigen::Isometry
    Isometry3d T = Isometry3d::Identity(); // Although called 3d, it is essentia
    T.rotate(rotation_vector); // Rotate according to rotation_vector
    T.pretranslate(Vector3d(1, 3, 4)); // Set the translation vector to (1,3,4)
```

```
    cout << "Transform matrix = \n" << T.matrix() << endl;

    // Use the transformation matrix for coordinate transformation
    Vector3d v_transformed = T * v; // Equivalent to R*v+t
    cout << "v tranformed = " << v_transformed.transpose() << endl;

    // For affine and projective transformations, use Eigen::Affine3d and Eigen:

    // Quaternion
    // You can assign AngleAxis directly to quaternions, and vice versa
    Quaterniond q = Quaterniond(rotation_vector);
    cout << "quaternion from rotation vector = " << q.coeffs().transpose()
    << endl; // Note that the order of coeffs is (x, y, z, w), w is the real par
    // can also assign a rotation matrix to it
    q = Quaterniond(rotation_matrix);
    cout << "quaternion from rotation matrix = " << q.coeffs().transpose() << en
    // Rotate a vector with a quaternion and use overloaded multiplication
    V_rotated = q * v; // Note that the math is qvq^{-1}
    cout << "(1,0,0) after rotation = " << v_rotated.transpose() << endl;
    // expressed by regular vector multiplication, it should be calculated as fo
    cout << "should be equal to " << (q * Quaterniond(0, 1, 0, 0) * q.inverse())

    return 0;
}
```

The various forms of expression in Eigen are summarized below. Note that each type has both single and double data types and, as before, cannot be automatically converted by the compiler. Taking double precision as an example, you can change the last d to f, which is a single-precision data structure.

- rotation matrix (3×3): Eigen::Matrix3d.
- rotation vector (3×1): Eigen::AngleAxisd.
- Euler angle (3×1): Eigen::Vector3d.
- quaternion (4×1): Eigen::Quaterniond.
- Euclidean transformation matrix (4×4): Eigen::Isometry3d.
- affine transform (4×4): Eigen::Affine3d.
- projective transformation (4×4): Eigen::Projective3d.

This program can be compiled by referring to the corresponding CMakeLists in the code. In this program, I demonstrate how to use the rotation matrix, rotation vectors (AngleAxis), Euler angles, and quaternions in Eigen. We use these rotations to rotate a vector $\boldsymbol{v}$ and find that the result is the same (different is really a hell of it). At the same time, it also demonstrates how to

convert these expressions in the program. Readers who want to learn more about Eigen's geometry modules can refer to it (http://eigen.tuxfamily.org/dox/group__TutorialGeometry.html).

The reader is cautioned that the **program code has some subtle differences from the mathematical representation**. For example, by operator overloading, quaternions and three-dimensional vectors can directly calculate multiplication, but mathematically, the vector needs to be converted into a virtual quaternion, and then quaternion multiplication is used for calculation. The same applies to the transformation matrix. Multiply the case of a three-dimensional vector. In general, the usage in the program is more flexible than the mathematical formula.

1.5.2 actual coordinate transformation example

Let's take a small example to demonstrate the coordinate transformation.

example *The radish No. 1 and the radish No. 2 are located in the world coordinate system. Hutchison world coordinate system for the W , radishes's coordinate system R_1 and R_2 . The position of the radish No. 1 is $\boldsymbol{q}_1 = [0.35, 0.2, 0.3, 0.1]^{\mathrm{T}}, \boldsymbol{t}_1 = [0.3, 0.1, 0.1]^{\mathrm{T}}$. The position of the radish No. 2 is $\boldsymbol{q}_2 = [-0.5, 0.4, -0.1, 0.2]^{\mathrm{T}}, \boldsymbol{t}_2 = [-0.1, 0.5, 0.3]^{\mathrm{T}}$. Here $\boldsymbol{q}$ and $\boldsymbol{t}$ express $\boldsymbol{T}_{R_k,W}, k = 1, 2$, which is the world coordinate system to the camera coordinate system. Transform the relationship. Now, Little Radish No. 1 sees a point in its own coordinate system with coordinates of $\boldsymbol{p}_{R_1} = [0.5.0, 0.2]^{\mathrm{T}}$, find the coordinates of the vector in the radish No. 2 coordinate system.*

This is a very simple but representative example. In actual scenarios you often need to convert coordinates between different parts of the same robot or between different robots. Below we write a program to demonstrate this calculation.

Listing 1.7: slambook2/ch3/examples/coordinateTransform.cpp

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<Eigen/Core>
#include<Eigen/Geometry>

using namespace std;
using namespace Eigen;

int main(int argc, char** argv) {
    Quaterniond q1(0.35, 0.2, 0.3, 0.1), q2(-0.5, 0.4, -0.1, 0.2);
    q1.normalize();
    q2.normalize();
    Vector3d t1(0.3, 0.1, 0.1), t2(-0.1, 0.5, 0.3);
    Vector3d p1(0.5, 0, 0.2);
```

```
    Isometry3d T1w(q1), T2w(q2);
    T1w.pretranslate (t1);
    T2w.pretranslate (t2);

    Vector3d p2 = T2w * T1w.inverse() * p1;
    cout << endl << p2.transpose() << endl;
    return 0;
}
```

The answer to the program output is $[-0.0309731, 0.73499, 0.296108]^{\mathrm{T}}$, and the calculation process is very simple, just calculate

$$\boldsymbol{p}_{R_2} = \boldsymbol{T}_{R_2,W}\boldsymbol{T}_{W,R_1}\boldsymbol{p}_{R_1}$$

. Note that the quaternion needs to be normalized before use.

1.6 Visualization demo

1.6.1 Show motion track

If you are new to the concepts of rotation and translation, you may find that their form looks complicated, because after all, each expression can be converted to other ways, and the conversion formula is sometimes longer. However, although the values of the rotation matrix and transformation matrix may not be intuitive enough, we can easily draw them in the window.

In this section we demonstrate two visual examples. First, let's say that we recorded the trajectory of a robot in some way, and now I want to draw it into a window. Suppose the track file is stored in trajectory.txt, and each line is stored in the following format:

$$\text{time}, t_x, t_y, t_z, q_x, q_y, q_z, q_w,$$

where time refers The recording time of this pose, $\boldsymbol{t}$ is translation, $\boldsymbol{q}$ is the rotation quaternion, all recorded in the world coordinate system to the robot coordinate system. Below we read these tracks from the file and display them in a window. In principle, if you just talk about "robot pose", then you can use $\boldsymbol{T}_{WR}$ or $\boldsymbol{T}_{RW}$, in fact they are only one inverse. It means that knowing one of them makes it easy to get another. If you want to store **robot's track**, then you can store $\boldsymbol{T}_{WR}$ at all timesOr $\boldsymbol{T}_{RW}$, which doesn't make much difference.

When drawing the trajectory, we can draw the "trajectory" into a sequence of points, which is similar to the "trajectory" we imagined. Strictly speaking, this is actually the **the coordinates of the origin of the robot (camera) coordinate system in the world coordinate system**. Consider the origin of the robot coordinate system, ie $\boldsymbol{O}_R$, then the $\boldsymbol{O}_W$ at this time is the coordinates of the origin in the world coordinate system:

$$bmO_W = \ bmT_{WR}\ bmO_R = \ bmt_{WR}. \tag{1.48}$$

This is the translation part of $\boldsymbol{T}_{WR}$. So, you can see **where the camera is** directly from $\boldsymbol{T}_{WR}$, which is why we say $\boldsymbol{T}_{WR}$ is more intuitive. Therefore, in the visualization program, the track file stores $\boldsymbol{T}_{WR}$ instead of $\boldsymbol{T}_{RW}$.

Finally, we need a library that supports 3D drawing. There are many libraries that support 3D drawing, such as the familiar matlab, python matplotlib, OpenGL and so on. In linux, a common library is OpenGL-based Pangolin library [14], which provides some GUI based on the support of OpenGL drawing operations. Features. In the second edition of the book, we used git's submodule feature to manage the third-party libraries that this book relies on. Readers can go directly to the 3rdparty folder to install the required libraries, and git guarantees that I am consistent with the version you are using.

Listing 1.8: slambook2/ch3/examples/plotTrajectory.cpp

```
#include <pangolin/pangolin.h>
```

[14] https://github.com/stevenlovegrove/Pangolin

```
#include <Eigen/Core>
#include <unistd.h>

using namespace std;
using namespace Eigen;

// path to trajectory file
string trajectory_file = "./examples/trajectory.txt";

void DrawTrajectory(vector<Isometry3d, Eigen::aligned_allocator<Isometry3d>>);

int main(int argc, char **argv) {
    vector<Isometry3d, Eigen::aligned_allocator<Isometry3d>> poses;
    ifstream fin(trajectory_file);
    if (!fin) {
        cout << "cannot find trajectory file at " << trajectory_file << endl;
        return 1;
    }

    while (!fin.eof()) {
        double time, tx, ty, tz, qx, qy, qz, qw;
        fin >> time >> tx >> ty >> tz >> qx >> qy >> qz >> qw;
        Isometry3d Twr(Quaterniond(qw, qx, qy, qz));
        Twr.pretranslate(Vector3d(tx, ty, tz));
        poses.push_back(Twr);
    }
    cout << "read total " << poses.size() << " pose entries" << endl;

    // draw trajectory in pangolin
    DrawTrajectory(poses);
    return 0;
}

void DrawTrajectory(vector<Isometry3d, Eigen::aligned_allocator<Isometry3d>> pos
    // create pangolin window and plot the trajectory
    pangolin::CreateWindowAndBind("Trajectory Viewer", 1024, 768);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    pangolin::OpenGlRenderState s_cam(
    pangolin::ProjectionMatrix(1024, 768, 500, 500, 512, 389, 0.1, 1000),
    pangolin::ModelViewLookAt(0, -0.1, -1.8, 0, 0, 0, 0.0, -1.0, 0.0)
    );

    pangolin::View &d_cam = pangolin::CreateDisplay()
```

```
        .SetBounds(0.0, 1.0, 0.0, 1.0, -1024.0f / 768.0f)
        .SetHandler(new pangolin::Handler3D(s_cam));

    while (pangolin::ShouldQuit() == false) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        d_cam.Activate(s_cam);
        glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
        glLineWidth(2);
        for (size_t i = 0; i < poses.size(); i++) {
            // draw three axes of each pose
            Vector3d Ow = poses[i].translation();
            Vector3d Xw = poses[i] * (0.1 * Vector3d(1, 0, 0));
            Vector3d Yw = poses[i] * (0.1 * Vector3d(0, 1, 0));
            Vector3d Zw = poses[i] * (0.1 * Vector3d(0, 0, 1));
            glBegin(GL_LINES);
            glColor3f(1.0, 0.0, 0.0);
            glVertex3d (Ow [0], Ow [1], Ow [2]);
            glVertex3d (Xw [0], Xw [1], Xw [2]);
            glColor3f(0.0, 1.0, 0.0);
            glVertex3d (Ow [0], Ow [1], Ow [2]);
            glVertex3d (Is [0], Is [1], Is [2]);
            glColor3f(0.0, 0.0, 1.0);
            glVertex3d (Ow [0], Ow [1], Ow [2]);
            glVertex3d (Zw [0], Zw [1], Zw [2]);
            glEnd();
        }
        // draw a connection
        for (size_t i = 0; i < poses.size(); i++) {
            glColor3f(0.0, 0.0, 0.0);
            glBegin(GL_LINES);
            auto p1 = poses[i], p2 = poses[i + 1];
            glVertex3d(p1.translation()[0], p1.translation()[1], p1.translation(
            glVertex3d(p2.translation()[0], p2.translation()[1], p2.translation(
            glEnd();
        }
        pangolin::FinishFrame();
        usleep(5000);   // sleep 5 ms
    }
}
```

This program demonstrates how to draw a 3D pose in Panglin. We draw the three axes of each pose in red, green, and blue (actually we calculate the world coordinates of each axis), and then connect the traces with black lines. The program runs as shown in **??**.

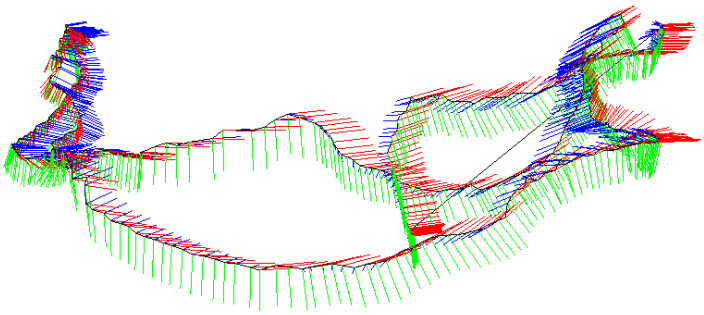
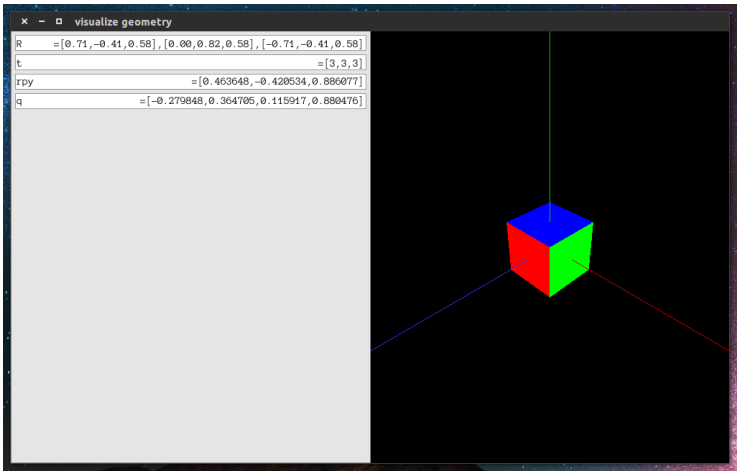


Figure 1.2: Results of pose visualization

1.6.2 Show camera pose



caption

Visualization program for rotation matrix, Euler angle, quaternion.

In addition to displaying the trajectory, we can also display the pose of the camera in the 3D window. In slambook2/ch3/visualizeGeometry, we visualize various expressions of camera poses (see **??**). When the reader uses the mouse to operate the camera, the box on the left side will display the rotation matrix, translation, Euler angle and quaternion of the camera pose in real time. You can see how the data changes. According to our experience, you should not see their intuitive meaning except for the Euler angle. However, although the rotation matrix or transformation matrix is not intuitive, it is not difficult to visually display them. This program uses the Pangolin library as a 3D display library. Please refer to Readme.txt to compile the program.

Exercises

1. verifies that the rotation matrix is an orthogonal matrix.

2. (optional) find the derivation process of Rodrigues formula and understand it.

3. verifies that after the quaternion rotates a point, the result is a virtual quaternion (the real part is zero), so it still corresponds to a three-dimensional space point, see (**??**). The

4. drawing table summarizes the conversion relationship of the rotation matrix, the axis angle, the Euler angle, and the quaternion.

5. Suppose there is a large Eigen matrix, its top left corner wants 3×3 blocks taken out, and then assigned to $\boldsymbol{I}_{3\times 3}$. Please programmatically implement.

6. (optional) General linear equation $\boldsymbol{Ax} = \boldsymbol{b}$ What are the practices? Can you implement it in Eigen?