14 Lectures on Visual SLAM
From Theory to Practice

Xiang Gao, Tao Zhang, Qinrui Yan and Yi Liu

April 4, 2020

Contents

Preface for English Version

A lot of friends at github asked me about this English version. I'm really sorry it takes so long to do the translation, and I'm glad to make it public available to help the readers. I encountered some issues on math equation in the web pages. Since the book is originally written in LaTeX, I'm going to release the LaTeX source along with the compiled pdf. You can directly access the pdf version for English book, and probably the publishing house is going to help me do the paper version.

As I'm not a native English speaker, the translation work is basically based on Google translation and some afterwards modification. If you think the quality of translation can be improved and you are willing to do this, please contact me or send an issue on github. Any help will be welcome!

Xiang

Chapter 1

Preface

1.1 What is this book about?

This is a book introducing visual SLAM, and it is probably the first Chinese book solely focused on this specific topic.

So, what is SLAM?

SLAM stands for **S**imultaneous **L**ocalization **a**nd **M**apping. It usually refers to a robot or a moving rigid body, equipped with a specific **sensor**, estimates its own **motion** and builds a **model** (certain kinds of description) of the surrounding environment, without a *priori* information[1]. If the sensor referred here is mainly a camera, it is called "**Visual SLAM**".

Visual SLAM is the subject of this book. We deliberately put a long definition into one single sentence, so that the readers can have a clear concept. First of all, SLAM aims at solving the "positioning" and "map building" issues at the same time. In other words, it is a problem of how to estimate the location of a sensor itself, while estimating the model of the environment. So how to achieve it? This requires a good understanding of sensor information. A sensor can observe the external world in a certain form, but the specific approaches for utilizing such observations are usually different. And, why is this problem worth spending an entire book to discuss? Simply because it is difficult, especially if we want to do SLAM in **real time** and **without any a priory knowledge**. When we talk about visual SLAM, we need to estimate the trajectory and map based on a set of continuous images (which form a video).

This seems to be quite intuitive. When we human beings enter an unfamiliar environment, aren't we doing exactly the same thing? So, the question is whether we can write programs and make computers do so.

At the birth of computer vision, people imagined that one day computers could act like human, watching and observing the world, and understanding the surrounding environment. The ability of exploring unknown areas is a wonderful and romantic dream, attracting numerous researchers striving on this problem day and night [**?**]. We thought that this would not be that difficult, but the progress turned out to be not as smooth as expected. Flowers, trees, insects, birds and animals, are recorded so differently in computers: they are simply matrices consisted of numbers. To make computers understand the contents of images, is as difficult as making us human understand those blocks of numbers. We didn't even know how we understand images, nor do we know how to make computers do so. However, after decades of

struggling, we finally started to see signs of success - through Artificial Intelligence (AI) and Machine Learning (ML) technologies, which gradually enable computers to identify objects, faces, voices, texts, although in a way (probabilistic modeling) that is still so different from us. On the other hand, after nearly three decades of development in SLAM, our cameras begin to capture their movements and know their positions, although there is still a huge gap between the capability of computers and human. Researchers have successfully built a variety of real-time SLAM systems. Some of them can efficiently track their own locations, and others can even do three-dimensional reconstruction in real-time.

This is really difficult, but we have made remarkable progress. What's more exciting is that, in recent years, we have seen emergence of a large number of SLAM-related applications. The sensor location could be very useful in many areas: indoor sweeping machines and mobile robots, automatic driving cars, Unmanned Aerial Vehicles (UAVs) in the air, Virtual Reality (VR) and Augmented Reality (AR). SLAM is so important. Without it, the sweeping machine cannot maneuver in a room autonomously, but wandering blindly instead; domestic robots can not follow instructions to reach a certain room accurately; Virtual Reality will always be limited within a prepared space. If none of these innovations could be seen in real life, what a pity it would be.

Today's researchers and developers are increasingly aware of the importance of the SLAM technology. SLAM has over 30 years of research history, and it has been a hot topic in both robotics and computer vision communities. Since the 21st century, visual SLAM technology has undergone a significant change and breakthrough in both theory and practice, and is gradually moving from laboratories into real-world. At the same time, we regretfully find that, at least in the Chinese language, SLAM-related papers and books are still very scarce, making many beginners of this area unable to get started smoothly. Although the theoretical framework of SLAM has basically become mature, to implement a complete SLAM system is still very challenging and requires high level of technical expertise. Researchers new to the area have to spend a long time learning a significant amount of scattered knowledge, and often have to go through a number of detours to get close to the real core.

This book systematically explains the visual SLAM technology. We hope that it will (at least in part) fill the current gap. We will detail SLAM's theoretical background, system architecture, and the various mainstream modules. At the same time, we place great emphasis on practice: all the important algorithms introduced in this book will be provided with runnable code that can be tested by yourself, so that readers can reach a deeper understanding. Visual SLAM, after all, is a technology for application. Although the mathematical theory can be beautiful, if you are not able to convert it into lines of code, it will be like a castle in the air, which brings little practical impact. We believe that practice verifies true knowledge, and practice tests true passion. Only after getting your hands dirty with the algorithms, you can truly understand SLAM, and claim that you have fallen in love with SLAM research.

Since its inception in 1986 [2], SLAM has been a hot research topic in robotics. It is very difficult to give a complete introduction to all the algorithms and their variants in the SLAM history, and we consider it as unnecessary as well. This book will be firstly introducing the background knowledge, such as projective geometry, computer vision, state estimation theory, Lie Group and Lie algebra, etc. On top of that, we will be showing the trunk of the SLAM tree, and omitting those complicated and oddly-shaped leaves. We think this is effective. If the reader can master the

essence of the trunk, they have already gained the ability to explore the details of the research frontier. So our aim is to help SLAM beginners quickly grow into qualified researchers and developers. On the other hand, even if you are already an experienced SLAM researcher, this book may still reveal areas that you are unfamiliar with, and may provide you with new insights.

There have already been a few SLAM-related books around, such as "Probabilistic Robotics" [3], "Multiple View Geometry in Computer Vision" [4], "State Estimation for Robotics: A Matrix-Lie-Group Approach"[**?**], etc. They provide rich contents, comprehensive discussions and rigorous derivations, and therefore are the most popular textbooks among SLAM researchers. However, there are two important issues: Firstly, the purpose of these books is often to introduce the fundamental mathematical theory, with SLAM being only one of its applications. Therefore, they cannot be considered as specifically visual SLAM focused. Secondly, they place great emphasis on mathematical theory, but are relatively weak in programming. This makes readers still fumbling when trying to apply the knowledge they learn from the books. Our belief is: only after coding, debugging and tweaking algorithms and parameters with his own hands, one can claim real understanding of a problem.

In this book, we will be introducing the history, theory, algorithms and research status in SLAM, and explaining a complete SLAM system by decomposing it into several modules: visual odometry, back-end optimization, map building, and loop closure detection. We will be accompanying the readers step by step to implement the core algorithms of each module, explore why they are effective, under what situations they are ill-conditioned, and guide them through running the code on their own machines. You will be exposed to the critical mathematical theory and programming knowledge, and will use various libraries including Eigen, OpenCV, PCL, g2o, and Ceres, and master their use in the Linux operating system.

Well, enough talking, wish you a pleasant journey!

1.2 How to use this book?

This book is entitled "14 Lectures on Visual SLAM". As the name suggests, we will organize the contents into "lectures" like we are learning in a classroom. Each lecture focuses on one specific topic,organized in a logical order. Each chapter will include both a theoretical part and a practical part, with the theoretical usually coming first. We will introduce the mathematics essential to understand the algorithms, and most of the time in a narrative way, rather than in a "definition, theorem, inference" approach adopted by most mathematical textbooks. We think this will be much easier to understand, but of course with a price of being less rigorous sometimes. In practical parts, we will provide code and discuss the meaning of the various parts, and demonstrate some experimental results. So, when you see chapters with the word "practice" in the title, you should turn on your computer and start to program with us, joyfully.

The book can be divided into two parts: The first part will be mainly focused on the fundamental math knowledge, which contains:

1. Lecture 1: preface (the one you are reading now), introducing the contents and structure of the book.

2. Lecture 2: an overview of a SLAM system. It describes each module of a SLAM system and explains what they do and how they do it. The practice

section introduces basic C++ programming in Linux environment and the use of an IDE.

3. Lecture 3: rigid body motion in 3D space. You will learn knowledge about rotation matrices, quaternions, Euler angles, and practice them with the Eigen library.

4. Lecture 4: Lie group and Lie algebra. It doesn't matter if you have never heard of them. You will learn the basics of Lie group, and manipulate them with Sophus.

5. Lecture 5: pinhole camera model and image expression in computer. You will use OpenCV to retrieve camera's intrinsic and extrinsic parameters, and then generate a point cloud using the depth information through PCL (Point Cloud Library).

6. Lecture 6: nonlinear optimization, including state estimation, least squares and gradient descent methods, e.g. Gauss-Newton and Levenburg-Marquardt. You will solve a curve fitting problem using the Ceres and g2o library.

   From lecture 7, we will be discussing SLAM algorithms, starting with Visual Odometry (VO) and followed by the map building problems:

7. Lecture 7: feature based visual odometry, which is currently the mainstream in VO. Contents include feature extraction and matching, epipolar geometry calculation, Perspective-n-Point (PnP) algorithm, Iterative Closest Point (ICP) algorithm, and Bundle Adjustment (BA), etc. You will run these algorithms either by calling OpenCV functions or by constructing you own optimization problem in Ceres and g2o.

8. Lecture 8: direct (or intensity-based) method for VO. You will learn the principle of optical flow and direct method, and then use g2o to achieve a simple RGB-D direct method based VO (the optimization in most direct VO algorithms will be more complicated).

9. Lecture 9: back-end optimization. We will discuss Bundle Adjustment in detail, and show the relationship between its sparse structure and the corresponding graph model. You will use Ceres and g2o separately to solve a same BA problem.

10. Lecture 10: pose graph in the back-end optimization. Pose graph is a more compact representation for BA which marginalizes all map points into constraints between keyframes. You will use g2o and gtsam to optimize a pose graph.

11. Lecture 11: loop closure detection, mainly Bag-of-Word (BoW) based method. You will use dbow3 to train a dictionary from images and detect loops in videos.

12. Lecture 12: map building. We will discuss how to estimate the depth of feature points in monocular SLAM (and show why they are unreliable). Compared with monocular depth estimation, building a dense map with RGB-D cameras is much easier. You will write programs for epipolar line search and patch matching to estimate depth from monocular images, and then build a point cloud map and octagonal tree map from RGB-D data.

13. Lecture 13: a practice chapter for VO. You will build a visual odometer framework by yourself by integrating the previously learned knowledge, and solve problems such as frame and map point management, key frame selection and optimization control.

14. Lecture 14: current open source SLAM projects and future development direction. We believe that after reading the previous chapters, you will be able to understand other people's approaches easily, and be capable to achieve new ideas of your own.

Finally, if you don't understand what we are talking about at all, congratulations! This book is right for you!

1.3 Source code

All source code in this book is hosted on github:

`https://github.com/gaoxiang12/slambook2`

Note the slambook2 refers to the second version which I added a lot of extra experiments.

It is strongly recommended that readers download them for viewing at any time. The code is divided by chapters, for example, the contents of the 7th lecture will be placed in folder "ch7". In addition, some of the small libraries used in the book can be found in the "3rd party" folder as compressed packages. For large and medium-sized libraries like OpenCV, we will introduce their installation methods when they first appear. If you have any questions regarding the code, click the "Issues" button on GitHub to submit. If there is indeed a problem with the code, we will make changes in a timely manner. Even if your understanding is biased, we will still reply as much as possible. If you are not accustomed to using Git, you can also click the button on the right which contains the word "download" to download a zipped file to your local drive.

1.4 Oriented readers

This book is for students and researchers interested in SLAM. Reading this book needs certain prerequisites, we assume that you have the following knowledge:

- Calculus, Linear Algebra, Probability Theory. These are the fundamental mathematical knowledge that most readers should have learned during undergraduate study. You should at least understand what a matrix and a vector are, and what it means by doing differentiation and integration. For more advanced mathematical knowledge required, we will introduce in this book as we proceed.

- Basic C++ Programming. As we will be using C++ as our major programming language, it is recommended that the readers are at least familiar with its basic concepts and syntax. For example, you should know what a class is, how to use the C++ standard library, how to use template classes, etc. We will try our best to avoid using tricks, but in certain situations we really can not avert. In addition, we will adopt some of C++ 11 standard, but don't worry, they will be explained as they appear.

- Linux Basics. Our development environment is Linux instead of Windows, and we will only provide source code for Linux. **We believe that mastering Linux is an essential skill for SLAM researchers, and please take it to begin. After going through the contents of this book, we believe you will agree with us** *. In Linux, the configuration of related libraries is so convenient, and you will gradually appreciate the benefit of mastering it. If you have never used a Linux system, it will be beneficial if you can find some Linux learning materials and spend some time reading them (to master Linux basics, the first few chapters of an introductory book should be sufficient). We do not ask readers to have superb Linux operating skills, but we do hope readers at least know how to fire an terminal, and enter a code directory. There are some self-test questions on Linux at the end of this chapter. If you have answers to them, you shouldn't have much problem in understanding the code in this book.

Readers interested in SLAM but do not have the above mentioned knowledge may find it difficult to proceed with this book. If you do not understand the basics of C++, you can read some introductory books such as *C ++ Primer Plus*. If you do not have the relevant math knowledge, we also suggest that you read some relevant math textbooks first. Nevertheless, we think that most readers who have completed undergraduate study should already have the necessary mathematical arsenal. Regarding the code, we recommend that you spend time typing them by yourself, and tweaking the parameters to see how they affect outputs. This will be very helpful.

This book can be used as a textbook for SLAM-related courses, but also suitable as extra-curricular self-study materials.

1.5 Style

This book covers both mathematical theory and programming implementation. Therefore, for the convenience of reading, we will be using different layouts to distinguish different contents.

1. Mathematical formulas will be listed separately, and important formulas will be assigned with an equation number on the right end of the line, for example:

$$\mathbf{y} = \mathbf{A}\mathbf{x}. \tag{1.1}$$

   Italics are used for scalars, e.g., a. Bold symbols are used for vectors and matrices, e.g.$\mathbf{a}$, $\mathbf{A}$. Hollow bold represents special sets, e.g. real number $\mathbb{R}$ and integer set $\mathbb{Z}$. Gothic is used for Lie Algebra, e.g. $\mathfrak{se}(3)$.

2. Source code will be framed into boxes, using a smaller font size, with line numbers on the left. If a code block is long, the box may continue to the next page:

```
1  #include <iostream>
2  using namespace std;
3
```

* Linux is not that popular in China as our computer science education starts very lately around 1990s.

```
4  int main (int argc, char** argv) {
5    cout << "Hello" << endl;
6    return 0;
7  }
```

3. When the code block is too long or contains repeated parts with previously listed code, it is not appropriate to be listed entirely. We will only give **important snippets** and mark it with "Snippet". Therefore, we strongly recommend that readers download all the source code on GitHub and complete the exercises to better understand the book.

4. Due to typographical reasons, the code shown in the book may be slightly different from the code hosted on GitHub. In that case please use the code on GitHub.

5. For each of the libraries we use, it will be explained in details when first appearing, but not repeated in the follow-up. Therefore, it is recommended that readers read this book in order.

6. An abstract will be presented at the beginning of each lecture. A summary and some exercises will be given at the end. The cited references are listed at the end of the book.

7. The chapters with an asterisk mark in front are optional readings, and readers can read them according to their interest. Skipping them will not hinder the understanding of subsequent chapters.

8. Important contents will be marked in **bold** or *italic*, as we are already accustomed to.

9. Most of the experiments we designed are demonstrative. Understanding them does not mean that you are already familiar with the entire library. So we recommend that you spend time on yourselves in further exploring the important libraries frequently used in the book.

10. The book's exercises and optional readings may require you to search for additional materials, so you need to learn to use search engines.

1.6 Acknowledgments

The online English version of this book is currently public available and open source. The Chinese version

1.7 Exercises (self-test questions)

1. There is a linear equation $\mathbf{Ax} = \mathbf{b}$, if $\mathbf{A}$ and $\mathbf{b}$ are known, how to solve for $\mathbf{x}$? What are the requirements for $\mathbf{A}$ and $\mathbf{b}$ if we want an unique $\mathbf{x}$? (Hint: check the rank of $\mathbf{A}$ and $\mathbf{b}$).

2. What is a Gaussian distribution? What does it look like in one-dimensional case? How about in high-dimensional case?

3. Do you know what a **class** is in C++? Do you know STL? Have you ever used them?

4. How do you write a C++ program? (It's completely fine if your answer is "using Visual C++ 6.0" *. As long as you have C++ or C programming experience, you are in good hand).

5. Do you know the C++11 standard? Which new features have you heard of or used? Are you familiar with any other standard?

6. Do you know Linux? Have you used at least one flavor (not including Android), such as Ubuntu?

7. What is the directory structure of Linux? What basic commands do you know? (e.g. ls, cat, etc.)

8. How to install a software in Ubuntu (without using the Software Center)? What directories are software usually installed under? If you only know the fuzzy name of a software (for example, you want to install a library with a word "eigen" in its name), how would you do it?

9. *Spend an hour learning Vim, you will be using it sooner or later. You can type "vimtutor" into an terminal and read through its contents. We do not require you to operate it very skillfully, as long as you can use it to edit the code in the process of learning this book. Do not waste time on its plugins, do not try to turn Vim into an IDE for now, we will only use it for text editing in this book.

* As I know many of our undergraduate students are still using this VC++ 6.0 in the university.

Chapter 2

First Glance of Visual SLAM

Goal of Study

1. Understand which modules a visual SLAM framework consists of, and what task each module carries out.
2. Set up the programming environment, and prepare for experiments.
3. Understand how to compile and run a program under Linux. If there is a problem, how to debug it.
4. Learn the basic usage of cmake.

2.1 Introduction

This lecture summarizes the structure of a visual SLAM system as an outline of subsequent chapters. Practice part introduces the fundamentals of environment setup and program development. We will make a small "Hello SLAM" program at the end.

2.2 Meet "Little Carrot"

Suppose we assembled a robot called *Little Carrot*, as shown in the following figure:

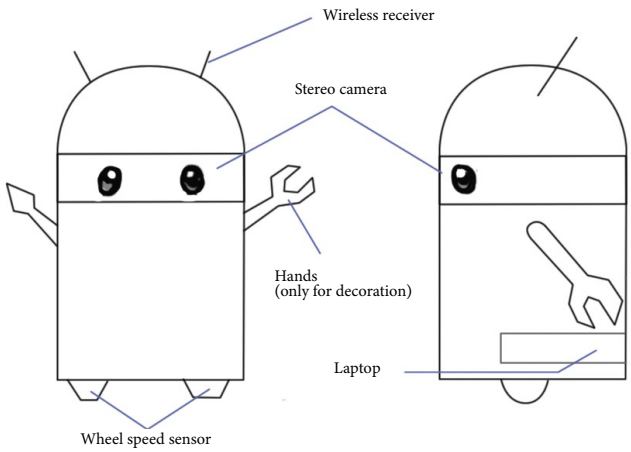

Figure 2-1: The sketch of robot *Little Carrot*

Although it looks a bit like the Android robot, it has nothing to do with the Android system. We put a laptop into its trunk (so that we can debug programs at any time). So, what is our robot capable to do?

We hope Little Carrot has the ability of *autonomous moving*. Although there are many *robots* placed statically on desktops, capable of chatting with people or playing music, but a tablet computer nowadays can also deliver the same tasks. As a robot, we hope Little Carrot can move freely in a room. Wherever we say hello to it, it can come to us right away.

First of all, such a robot needs wheels and motors to move, so we installed wheels under Little Carrot (gait control for humanoid robots is very complicated, which we will not be considering here). Now with the wheels, the robot is able to move, but without an effective navigation system, Little Carrot does not know where a target of action is, and it can do nothing but wander around blindly. Even worse, it may hit a wall and cause damage. In order to avoid this, we installed cameras on its head, with the intuition that such a robot *should look similar to human*. Certainly, with eyes, brains and limbs, human can walk freely and explore any environment, so we (somehow naively) think that our robot should be able to achieve it too. Well, in order to make Little Carrot able to explore a room, we find it at least needs to know two things:

1. Where am I? - It's about *localization*.

2. What is the surrounding environment like? -It's about *map building*.

Localization and *map building*, can be seen as the perception in both inward and outward directions. As a completely autonomous robot, Little Carrot need not only to understand its own *state* (i.e. the location), but also the external *environment* (i.e. the map). Of course, there are many different approaches to solve these two problems. For example, we can lay guiding rails on the floor of the room, or paster a lot of artificial markers such as QR code pictures on the wall, or mount radio positioning devices on the table. If you are outdoor, you can also install a GNSS receiver (like the one in a cell phone or a car) on the head of Little Carrot. With these devices, can we claim that the positioning problem has been resolved? Let's categorize these sensors (see Fig. 2-2) into two classes.

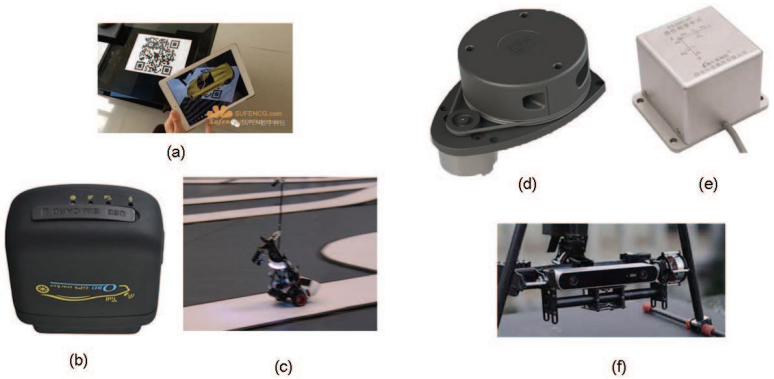

Figure 2-2: Different kinds of sensors: (a) QR code (b) GNSS receiver (c) guiding rails (d) Laser range finder (e) Inertial measurement unit (f) stereo camera

The first class are *non-intrusive* sensors which are completely self-contained inside a robot, such as wheel encoders, cameras, laser scanners, etc. They do not assume an cooperative environment around the robot. The other class are *intrusive* sensors depending on a prepared environment, such as the above mentioned guiding rails, QR codes, etc. Intrusive sensors can usually locate a robot directly, solving the positioning problem in a simple and effective manner. However, since they require changes on the environment, the scope of usage is often limited within a certain degree. For example, if there is no GPS signal, or guiding rails cannot be laid, what should we do in those cases?

We can see that the intrusive sensors place certain *constraints* to the external environment. A localization system based on them can only function properly when those constraints are met in the real world. Otherwise, the localization approach cannot be carried out anymore, like GPS positioning system normally doesn't work well in indoor environments. Therefore, although this type of sensor is simple and reliable, they do not work as a general solution. In contrast, non-intrusive sensors, such as laser scanners, cameras, wheel encoders, Inertial Measurement Units (IMUs), etc., can only observe indirect physical quantities rather than the direct locations. For example, a wheel encoder measures the wheel rotation angle, an IMU measures the angular velocity and the acceleration, a camera or a laser scanner observe the external environment in a certain form like point-clouds and images. We have to apply algorithms to infer positions from these indirect observations. While this sounds like a roundabout tactic, the more obvious benefit is that it does not make any

demands on the environment, making it possible for this localization framework to be applied to an unknown environment. Therefore, they are called as self-localization in many research area.

Looking back at the SLAM definitions discussed earlier, we emphasized an *unknown environment* in SLAM problems. In theory, we should not presume which environment the Little Carrot will be used (but in reality we will have a rough range, such as indoor or outdoor), which means that we can not assume that the external sensors like GPS can work smoothly. Therefore, the use of portable non-intrusive sensors to achieve SLAM is our main focus. In particular, when talking about visual SLAM, we generally refer to the using of *cameras* to solve the localization and map building problems.

Visual SLAM is the main subject of this book, so we are particularly interested in what the Little Carrot's eyes can do. The cameras used in SLAM are different from the commonly seen SLR cameras. It is often much simpler and does not carry expensive lens. It shoots at the surrounding environment at a certain rate, forming a continuous video stream. An ordinary camera can capture images at 30 frames per second, while high-speed cameras can do faster. The camera can be roughly divided into three categories: Monocular, Stereo and RGB-D, as shown by the following figure 2-3. Intuitively, a monocular camera has only one camera, a stereo camera has two. The principle of a RGB-D camera is more complex, in addition to being able to collect color images, it can also measure the distance of the scene from the camera for each pixel. RGB-D cameras usually carry multiple cameras, and may adopt a variety of different working principles. In the fifth lecture, we will detail their working principles, and readers just need an intuitive impression for now. In addition, there are also specialty and emerging camera types can be applied to SLAM, such as panorama camera [5], event camera [6]. Although they are occasionally seen in SLAM applications, so far they have not become the mainstream. From the appearance we can infer that Little Carrot seems to carry a stereo camera.

Figure 2-3: Different kinds of cameras: monocular, RGB-D and stereo.

Now, let's take a look at the pros and cons of using different type of camera for SLAM.

Monocular Camera

The SLAM system that uses only one camera is called Monocular SLAM. This sensor structure is particularly simple, and the cost is particularly low, therefore the monocular SLAM has been very attractive to researchers. You must have seen the output data of a monocular camera: photo. Yes, as a photo, what are its characteristics?

A photo is essentially a *projection* of a scene onto a camera's imaging plane. It reflects a three-dimensional world in a two-dimensional form. Obviously, there is one dimension lost during this projection process, which is the so-called depth (or distance). In a monocular case, we can not obtain the *distance* between objects in the scene and the camera by using a single image. Later we will see that this distance is actually critical for SLAM. Because we human have seen a large number of images, we formed a natural sense of distances for most scenes, and this can help us determine the distance relationship among the objects in the image. For example, we can recognize objects in the image and correlate them with their approximate size obtained from daily experience. The close objects will occlude the distant objects; the sun, the moon and other celestial objects are infinitely far away; an object will have shadow if it is under sunlight. This common sense can help us determine the distance of objects, but there are also certain cases that confuse us, and we can no longer determine the distance and true size of an object. The following figure 2-4 is shown as an example. In this image, we can not determine whether the figures are real person or small toys purely based on the image itself. Unless we change our view angle, explore the three-dimensional structure of the scene. In other words, from a single image, we can not determine the true size of an object. It may be a big but far away object, but it may also be a close but small object. They may appear to be the same size in an image due to the perspective projection effect.

Figure 2-4: We cannot tell if the people are real humans or just small toys from a single image

Since the image taken by an monocular camera is just a 2D projection of the 3D space, if we want to recover the 3D structure, we have to change the camera's view angle. Monocular SLAM adopts the same principle. We move the camera and

estimate its own *motion*, as well as the distances and sizes of the objects in the scene, namely the *structure* of the scene. So how should we estimate these movements and structures? From the everyday experience we know that if a camera moves to the right, the objects in the image will move to the left which gives us an inspiration of inferring motion. On the other hand, we also know that closer objects move faster, while distant objects move slower. Thus, when the camera moves, the movement of these objects on the image forms pixel disparity. Through calculating the disparity, we can quantitatively determine which objects are far away and which objects are close.

However, even if we know which objects are near and which are far, they are still only relative values. For example, when we are watching a movie, we can tell which objects in the movie scene are bigger than the others, but we can not determine the *real size* of those objects – are the buildings real high-rise buildings or just models on a table? Is it a real monster that destructs a building, or just an actor wearing special clothing? Intuitively, if the camera's movement and the scene size are doubled at the same time, monocular cameras see the same. Likewise, multiplying this size by any factor, we will still get the same picture. This demonstrates that the trajectory and map obtained from monocular SLAM estimation will differ from the actual trajectory and map with a factor, which is just the so-called *scale* [*]. Since monocular SLAM can not determine this real scale purely based on images, this is also called the *scale ambiguity*.

In monocular SLAM, depth can only be calculated with translational movement, and the real scale cannot be determined. These two things could cause significant trouble when applying monocular SLAM into real-world applications. The fundamental cause is that depth can not be determined from a single image. So, in order to obtain real-scaled depth, we start to use stereo and RGB-D cameras.

Stereo Camera and RGB-D Camera

The purpose of using stereo and RGB-D cameras is to measure the distance between objects and the camera, to overcome the shortcomings of monocular cameras that distances are unknown. Once distances are known, the 3D structure of a scene can be recovered from a single frame, and also eliminates the scale ambiguity. Although both stereo and RGB-D cameras are able to measure the distance, their principles are not the same. A stereo camera consists of two synchronized monocular cameras, displaced with a known distance, namely the *baseline*. Because the physical distance of the baseline is know, we are able to calculate the 3D position of each pixel, in a way that is very similar to our human eyes. We can estimate the distances of the objects based on the differences between the images from left and right eye, and we can try to do the same on computers (see Fig. 2-5). We can also extend stereo camera to multi-camera systems if needed, but basically there is no much difference.

Stereo cameras usually require significant amount of computational power to (unreliably) estimate depth for each pixel. This is really clumsy compared to human beings. The depth range measured by a stereo camera is related to the baseline length. The longer a baseline is, the farther it can measure. So stereo cameras mounted on autonomous vehicles are usually quite big. Depth estimation for stereo cameras is achieved by comparing images from the left and right cameras, and does not rely on other sensing equipment. Thus stereo cameras can be applied both indoor and outdoor. The disadvantage of stereo cameras or multi-camera systems is

[*] Mathematical reason will be explained in the visual odometry chapter.

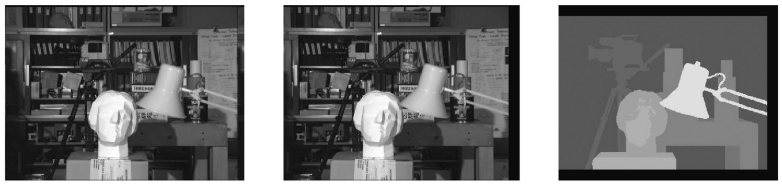

Figure 2-5: Distance is calculated from the disparity of two stereo image pair.

that the configuration and calibration process is complicated, and their depth range and accuracy are limited by baseline length and camera resolution. Moreover, stereo matching and disparity calculation also consumes much computational resource, and usually requires GPU or FPGA to accelerate in order to generate real-time depth maps. Therefore, in most of the state-of-the-art algorithms, computational cost is still one of the major problems of stereo cameras.

Depth camera (also known as RGB-D camera, RGB-D will be used in this book) is a type of new cameras rising since 2010. Similar to laser scanners, RGB-D cameras adopt infrared structure of light or Time-of-Flight (ToF) principles, and measure the distance between objects and the camera by actively emitting light to the object and receive the returned light. This part is not solved by software as a stereo camera, but by physical sensors, so it can save much computational resource compared to stereo cameras (see Fig. 2-6). Common RGB-D cameras include Kinect / Kinect V2, Xtion Pro Live, RealSense, etc. However, most of the RGB-D cameras still suffer from issues including narrow measurement range, noisy data, small field of view, susceptible to sunlight interference, and unable to measure transparent material. For SLAM purpose, RGB-D cameras are mainly used in indoor environments, and are not suitable for outdoor applications.

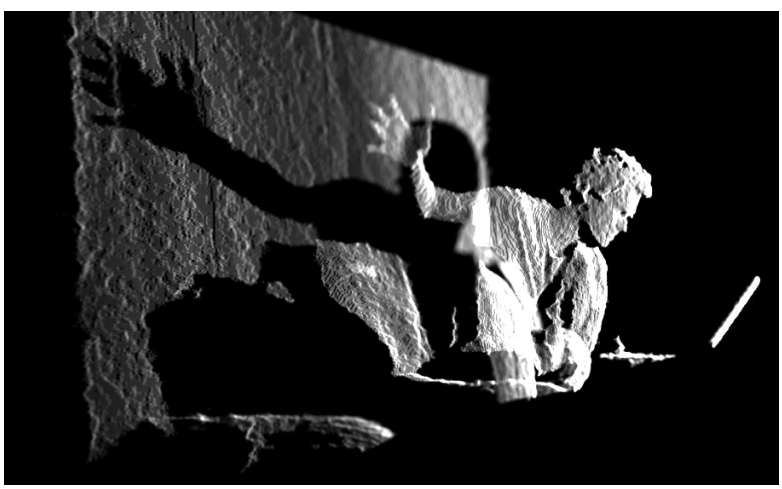

Figure 2-6: RGBD cameras measure the distance and can build a point cloud with a single image frame.

We have discussed the common types of cameras, and we believe you should have gained an intuitive understanding of them. Now, imagine a camera is moving in a

scene, we will get a series of continuously changing images *. The goal of visual SLAM is to localize and build a map using these images. This is not as simple task as you would think. It is not a single algorithm that continuously output positions and map information as long as we feed it with input data. SLAM requires a good algorithm framework, and after decades of hard work by researchers, the framework has been matured in recent years.

2.3 The Classic Visual SLAM Framework

Let's take a look at the classic visual SLAM framework, shown in the following figure 2-7:

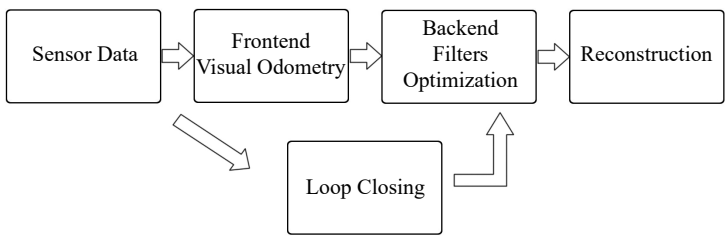

Figure 2-7: The classic visual SLAM framework.

A typical visual SLAM work-flow includes the following steps:

1. Sensor data acquisition. In visual SLAM, this mainly refers to for acquisition and preprocessing for camera images. For a mobile robot, this will also include the acquisition and synchronization with motor encoders, IMU sensors, etc.

2. Visual Odometry (VO). The task of VO is to estimate the camera movement between adjacent frames (ego-motion), as well as to generate a rough local map. VO is also known as the *Front End*.

3. Backend filtering/optimization. The back end receives camera poses at different time stamps from VO, as well as results from loop closing, and apply optimization to generate a fully optimized trajectory and map. Because it is connected after the VO, it is also known as the *Back End*.

4. Loop Closing. Loop closing determines whether the robot has returned to its previous position in order to reduce the accumulated drift. If a loop is detected, it will provide information to the back end for further optimization.

5. Reconstruction. It constructs a task specific map based on the estimated camera trajectory.

The classic visual SLAM framework is the result of more than a decade's research endeavor. The framework itself and the algorithms have been basically finalized and have been provided as basic functions in several public vision and robotics libraries. Relying on these algorithms, we are able to build visual SLAM systems performing real-time localization and mapping in static environments. Therefore, a

* You can try to use your phone to record a video clip.

rough conclusion can be reached that if the working environment is limited to static and rigid with stable lighting conditions and no human interference, visual SLAM problem is basically solved [7].

The readers may have not fully understood the concepts of the above mentioned modules yet, so we will detail the functionality of each module in the following sections. However, an deeper understanding of their working principles requires certain mathematical knowledge which will be expanded in the second part of this book. For now, an intuitive and qualitative understanding of each module is good enough.

Visual Odometry

The visual odometry is concerned with the movement of a camera between *adjacent image frames*, and the simplest case is of course the motion between two successive images. For example, when we see the images in Fig. 2-8, we will naturally tell that the right image should be the result of the left image after a rotation to the left with a certain angle (it will be easier if we have a video input). Let's consider this question: how do we know the motion is "turning left"? Humans have long been accustomed to using our eyes to explore the world, and estimating our own positions, but this intuition is often difficult to explain, especially in natural language. When we see these images, we will naturally think that, ok, the bar is close to us, the walls and the blackboard are farther away. When the camera turns to left, the closer part of the bar started to appear, and the cabinet on the right side started to move out of our sight. With this information, we conclude that the camera should be be rotating to the left.

Figure 2-8: Camera motion can be inferred from two consecutive image frames. Images are from NYUD dataset.

But if we go a step further: can we determine how much the camera has rotated or translated, in units of degrees or centimeters? It is still difficult for us to give an quantitative answer. Because our intuition is not good at calculating numbers. But for a computer, movements have to be described with such numbers. So we will ask: how should a computer determine a camera's motion only based on images?

As mentioned earlier, in the field of computer vision, a task that seems natural to a human can be very challenging for a computer. Images are nothing but numerical matrices in computers. A computer has no idea what these matrices mean (this is the problem that machine learning is also trying to solve). In visual SLAM, we can only see blocks of pixels, knowing that they are the results of projections by spatial points onto the camera's imaging plane. In order to quantify a camera's movement, we must first *understand the geometric relationship between a camera and the spatial points*.

Some background knowledge is needed to clarify this geometric relationship and the realization of VO methods. Here we only want to convey an intuitive concept. For now, you just need to take away that VO is able to estimate camera motions from images of adjacent frames and restore the 3D structures of the scene. It is named as an "odometry", because similar to an actual wheel odometry which only calculates the ego-motion at neighboring moments, and does not estimate a global map or a absolute pose. In this regard, VO is like a species with only a short memory.

Now, assuming that we have a visual odometry, we are able to estimate camera movements between every two successive frames. If we connect the adjacent movements, this constitutes the movement of the robot trajectory, and therefore addresses the positioning problem. On the other hand, we can calculate the 3D position for each pixel according to the camera position at each time step, and they will form an map. Up to here, it seems with an VO, the SLAM problem is already solved. Or, is it?

Visual odometry is indeed an key technology to solving visual SLAM problem. We will be spending a great part to explain it in details. However, using only a VO to estimate trajectories will inevitably cause *accumulative drift*. This is due to the fact that the visual odometry (in the simplest case) only estimates the movement between two frames. We know that each estimate is accompanied by a certain error, and because the way odometry works, errors from previous moments will be carried forward to the following moments, resulting in inaccurate estimation after a period of time (see Fig. 2-9). For example, the robot first turns left 90° and then turns right 90°. Due to error, we estimate the first 90° as 89°, which is possible to happen in real-world applications. Then we will be embarrassed to find that after the right turn, the estimated position of the robot will not return to the origin. What's worse, even the following estimates are perfectly estimated, they will always be carrying this 1° error compared to the true trajectory.

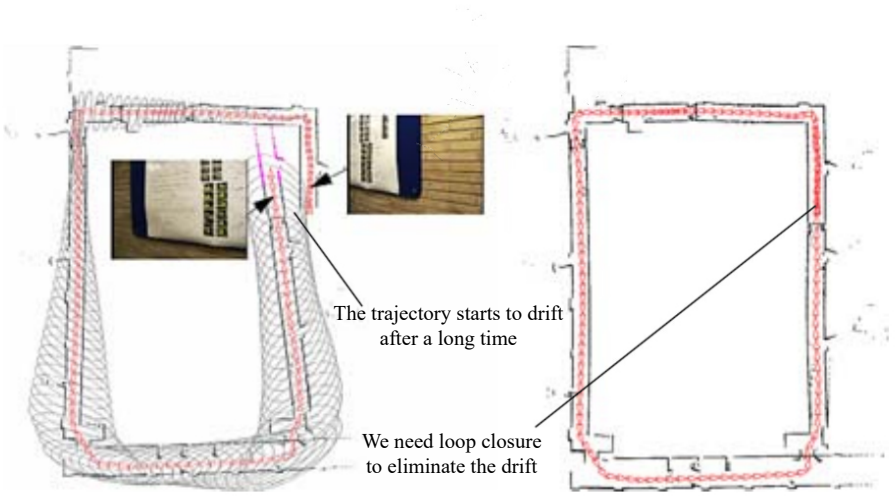

Figure 2-9: Drift will be accumulated if we only have a relative motion estimation.

The accumulated drift will make us unable to build a consistent map. A straight corridor may oblique, and a 90° angle may be crooked - this is really an unbearable matter! In order to solve the drifting problem, we also need other two components:

the *back-end optimization* * and *loop closing*. Loop closing is responsible for detecting whether the robot returns to its previous position, while the back-end optimization corrects the shape of the entire trajectory based on this information.

Back-end Optimization

Generally speaking, the back-end optimization mainly refers to the process of dealing with the *noise* in SLAM systems. We wish that all the sensor data is accurate, but in reality, even the most expensive sensors still have certain amount of noise. Cheap sensors usually have larger measurement errors, while that of expensive ones may be small. Moreover, performance of many sensors are affected by changes in magnetic field, temperature, etc. Therefore, in addition to solving the problem of estimating camera movements from images, we also care about how much noise this estimation contains, how these noise is carried forward from the last time step to the next, and how confident we have on the current estimation. So the problem that back-end optimization solves can be summarized as: to estimate the state of the entire system from noisy input data and calculate how uncertain these estimations are. The state here includes both the robot's own trajectory and the environment map.

In contrast, the visual odometry part is usually referred to as the *front end*. In a SLAM framework, the front end provides data to be optimized by the back end, as well as the initial values. Because the back end is responsible for the overall optimization, we only care about the data itself instead of where it comes from. In other words, we only have numbers and matricies in backend without those beatiful images. In visual SLAM, the front end is more relevant to *computer vision* topics, such as image feature extraction and matching, while the backend is relevant to *state estimation* research area.

Historically, the back-end optimization part has been equivalent to "SLAM research" for a long time. In the early days, SLAM problem was described as a state estimation problem, which is exactly what the back-end optimization tries to solve. In the earliest papers on SLAM, researchers at that time called it "estimation of spatial uncertainty" [2, 8]. Although sounds a little obscure, it does reflect the nature of the SLAM problem: *the estimation of the uncertainty of the self-movement and the surrounding environment*. In order to solve the SLAM problem, we need state estimation theory to express the uncertainty of localization and map construction, and then use filters or nonlinear optimization to estimate the mean and uncertainty (covariance) of the states. The details of state estimation and non-linear optimization will be explained in chapter 6, 10 and 11.

Loop Closing

Loop Closing, also known as *Loop Closure Detection*, is mainly to address the drifting problem of position estimation in SLAM. So how to solve it? Assuming that a robot has returned to its origin after a period of movement, but the estimated position does not return to the origin due to drift. How to correct it? Imagine that if there is some way to let the robot know that it has returned to the origin, then we can then "pull" the estimated locations to the origin to eliminate drifts, which is, exactly, called loop closing.

---

* It is usually known as the back end. Since it is often implemented by optimization so we use the term back-end optimization.

Loop closing has close relationship with both localization and map building. In fact, the main purpose of building a map is to enable a robot to know the places it has been to. In order to achieve loop closing, we need to let the robot has the ability to identify the scenes it has visited before. There are different alternatives to achieve this goal. For example, as we mentioned earlier, we can set a marker at where the robot starts, such as a QR code. If the sign was seen again, we know that the robot has returned to the origin. However, the marker is essentially an intrusive sensor which sets additional constraints to the application environment. We prefer the robot can use its non-intrusive sensors, e.g. the image itself, to complete this task. A possible approach would be to detect similarities between images. This is inspired by us humans. When we see two similar images, it is easy to identify that they are taken from the same place. If the loop closing is successful, accumulative error can be significantly reduced. Therefore, visual loop detection is essentially an algorithm for calculating similarities of images. Note that the loop closing problem also exists in laser based SLAM, but here the rich information contained in images can remarkably reduce the difficulty of making a correct loop detection.

After a loop is detected, we will tell the back-end optimization algorithm that, OK, "A and B are the same point". Then, based on this new information, the trajectory and the map will be adjusted to match the loop detection result. In this way, if we have sufficient and reliable loop detection, we can eliminate cumulative errors, and get globally consistent trajectories and maps.

Mapping

Mapping means the process of building a map, whatever kind it is. A map (see Fig. 2-10) is a description of the environment, but the way of description is not fixed and depends on the actual application.

Let's take the domestic cleaning robots as an example. Since they basically move on the ground, a two-dimensional map with marks for open areas and obstacles, built by a single line laser scanner, would be sufficient for navigation for them. And for a camera, we need at least a three-dimensional map for its 6 degrees of freedom movement. Sometimes, we want a smooth and beautiful reconstruction result, not just a set of points, but also with texture of triangular faces. And at other times, we do not care about the map, just need to know things like "point A and point B are connected, while point B and point C are not", which is a topological way to understand the environement. Sometimes maps may not even be needed, for instance, a level-3 autonomous driving car can make a lane-following driving only knowing its relative motion with the lanes.

For maps, we have various ideas and demands. So compared to the previously mentioned VO, loop closure detection and back-end optimization, map building does not have a certain algorithm. A collection of spatial points can be called a map, a beautiful 3D model is also a map, so is a picture of a city, a village, railways, and rivers. The form of the map depends on the application of SLAM. In general, they can be divided into to categories: *metrical map* and *topological map*.

Metric Map Metrical maps emphasize the exact metrical locations of the objects in maps. They are usually classified as either sparse or dense. Sparse metric maps store the scene into a compact form, and do not express all the objects. For example, we can construct a sparse map by selecting representative landmarks such as the lanes and traffic signs, and ignore other parts. In contrast, dense metrical maps

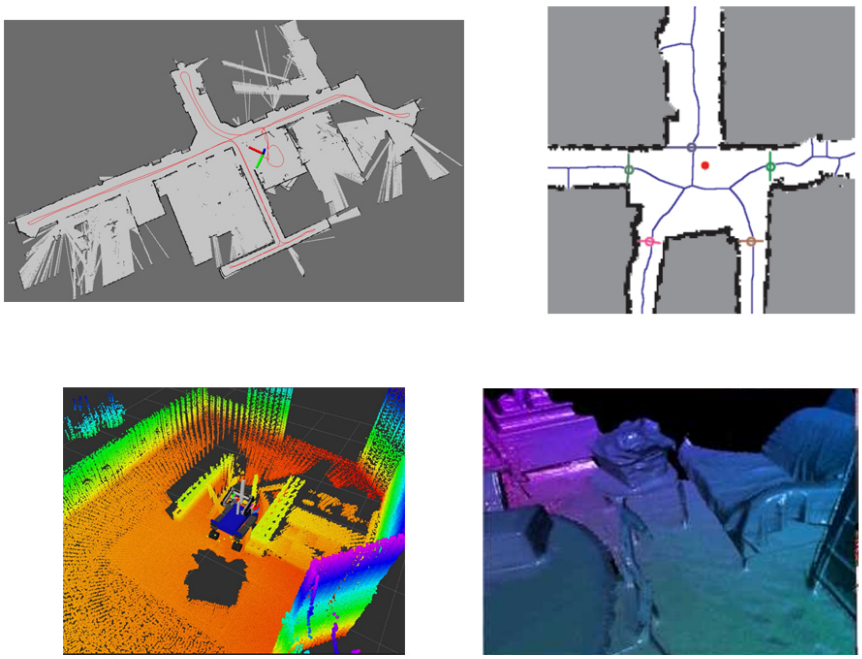

Figure 2-10: Different kinds of maps: 2D grid map, 2D topological map, 3D point clouds and 3D meshes.

focus on modeling all the things that are seen. For localization, sparse map would be enough, while for navigation, a dense map is usually needed (otherwise we may hit a wall between two landmarks). A dense map usually consists of a number of small pieces at a certain resolution. It can be small grids for 2D metric maps, or small voxels for 3D maps. For example, in a grid map, an grid may have three states: occupied, idle, and unknown, to express whether there is an object. When a spatial location is queried, the map can give the information about whether the location can be passed through. This type of maps can be used for a variety of navigation algorithms, such as A*, D**, etc., and thus attracts the attention of robotics researchers. But we can also see that all the grid status are store in the map, and thus being storage expensive. There are also some open issues in building a metrical map, for example, in large-scale metrical maps, a little bit of steering error may cause the walls of two rooms to overlap with each other, and thus making the map ineffective.

Topological Map Compared to the accurate metrical maps, topological maps emphasize the relationships among map elements. A topological map is a graph composed of nodes and edges, only considering the connectivity between nodes. For instance, we only care about that point A and point B are connected, regardless how we could travel from point A to point B. It relaxes the requirements on precise locations of a map by removing map details, and is therefore a more compact expres-

* See `https://en.wikipedia.org/wiki/A*_search_algorithm`.

sion. However, topological maps are not good at representing maps with complex structures. Questions such as how to split a map to form nodes and edges, and how to use a topological map for navigation and path planning, are still open problems to be studied.

2.4 Mathematical Formulation of SLAM Problems

Through the previous introduction, readers should have gained an intuitive understanding of the modules in a SLAM system and the main functionality of each module. However, we cannot write runable programs only based on intuitive impressions. We want to rise it to a rational and rigorous level, that is, using mathematical symbols to formulate a SLAM process. We will be using variables and formulas, but please rest assured that we will try our best to keep it clear enough.

Assuming that our Little Carrot is moving in an unknown environment, carrying some sensors. How can this be described in mathematical language? First, since sensors usually collect data at different some time points, we are only concerned with the locations and map at these moments. This turns a continuous process into discrete time steps, say $1, \cdots, k$, at which data sampling happens. We use $\mathbf{x}$ to indicate positions of Little Carrot. So the positions at different time steps can be written as $\mathbf{x}_1, \cdots, \mathbf{x}_k$, which constitute the trajectory of Little Carrot. In terms of the map, we assume that the map is made up of a number of *landmarks*, and at each time step, the sensors can see a part of the landmarks and record their observations. Assume there are total N landmarks in the map, and we will use $\mathbf{y}_1, \cdots, \mathbf{y}_N$ to denote them.

With such a setting, the process that "Little Carrot move in the environment with sensors" basically has two parts:

1. What is its *motion*? We want to describe how $\mathbf{x}$ is changed from time step $k-1$ to k.
2. What are the sensor *observations*? Assuming that the Little Carrot detects a certain landmark, say $\mathbf{y}_j$ at position $\mathbf{x}_k$, we need to describe this event in mathematical language.

Let's first take a look at motion. Typically, we may send some motion message to the robots like "turn 15 degree to left". These messages or orders will be finally carried out by the controller, but probably in may different ways. Sometimes we control the position of robots, but acceleration or angular velocity would always be reasonable alternates. However, no matter what the controller is, we can use a universal and abstract mathematical model to describe it:

$$\mathbf{x}_k = f\left(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{w}_k\right), \tag{2.1}$$

where $\mathbf{u}_k$ is the input orders, and $\mathbf{w}_k$ is noise. Note that we use a general $f(\cdot)$ to describe the process, instead of specifying the exact form of f. This allows the function to represent any motion input, rather than being limited to a particular one, and thus becoming a general equation. We call it the *motion equation*.

The presence of noise turns this model into a stochastic model. In other words, even if we give the order like "move forward one metes", it does not mean that our robot really advances one meter. If all the instructions are accurate, there is no need to *estimate* anything. In fact, the robot may only advance by, say, 0.9 meters, and

at another moment, it moves by 1.1 meters. Thus, the noise during each movement is random. If we ignore this noise, the position determined only by the command may be a hundred miles away from the actual position after several minutes.

Corresponding to the motion equation, there is also a *observation equation*. The observation equation describes the process that the Little Carrot sees a landmark point $\mathbf{y}_j$ at $\mathbf{x}_k$ and generates an observation data $\mathbf{z}_{k,j}$. Likewise, we will describe this relationship with an abstract function $h(\cdot)$:

$$\mathbf{z}_{k,j} = h\left(\mathbf{y}_j, \mathbf{x}_k, \mathbf{v}_{k,j}\right), \tag{2.2}$$

where $\mathbf{v}_{k,j}$ is the noise in this observation. Since there are various forms of observation sensors, the observed data $\mathbf{z}$ and the observation equation h may also have many different forms.

Readers may say that the function f, h here does not seem to specify what is going on in the motion and observation. Also, what does $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ mean here? In fact, depending on the actual motion and the type of sensor, there are several kinds of **parameterization** methods. What is parameterization? For example, suppose our robot moves in a plane, then its pose * is described by two $x-y$ coordinates and an angle, ie $\mathbf{x}_k = [x_1, x_2, \theta]_k^{\mathrm{T}}$, where x_1, x_2 are positions on two axes and θ is the angle. At the same time, the input command is the position and angle change between the time interval: $\mathbf{u}_k = [\Delta x_1, \Delta x_2, \Delta\theta]_k^{\mathrm{T}}$, so the motion equation can be parameterized as:

$$\begin{bmatrix} x_1 \\ x_2 \\ \theta \end{bmatrix}_k = \begin{bmatrix} x_1 \\ x_2 \\ \theta \end{bmatrix}_{k-1} + \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \Delta\theta \end{bmatrix}_k + \mathbf{w}_k, \tag{2.3}$$

where $\mathbf{w}_k$ is the noise again. This is a simple linear relationship. However, not all input commands are position and angular changes. For example, the input of "throttle" or "joystick" is the speed or acceleration, so there are other forms of more complex equations of motion. At that time, we would say the kinetic analysis is required.

Regarding the observation equation, for example, the robot carries a two-dimensional laser sensor. We know that when a laser sensor observes a 2D landmark by measuring two quantities: the distance between the landmark point and the robot r, and the angle ϕ. Let's say the landmark is at $\mathbf{y}_j = [y_1, y_2]_j^{\mathrm{T}}$, the pose is $\mathbf{x}_k = [x_1, x_2]_k^{\mathrm{T}}$ and the observed data is $\mathbf{z}_{k,j} = [r_{k,j}, \phi_{k,j}]^{\mathrm{T}}$, then the observation equation is written as:

$$\begin{bmatrix} r_{k,j} \\ \phi_{k,j} \end{bmatrix} = \begin{bmatrix} \sqrt{\left(y_{1,j} - x_{1,k}\right)^2 + \left(y_{2,j} - x_{2,k}\right)^2} \\ \arctan\left(\frac{y_{2,j} - x_{2,k}}{y_{1,j} - x_{1,k}}\right) \end{bmatrix} + \mathbf{v}. \tag{2.4}$$

When considering about visual SLAM, the sensor is a camera, then the observation equation is a process like "getting the pixels in the image of the landmarks." This process involves a description of the camera model, which will be covered in detail in Chapter 5, which is skipped here.

Obviously, it can be seen that the two equations have different parameterized forms for different sensors. If we maintain versatility and take them into a common abstract form, then the SLAM process can be summarized into two basic equations:

$$\begin{cases} \mathbf{x}_k = f\left(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{w}_k\right), & k = 1, \cdots, K \\ \mathbf{z}_{k,j} = h\left(\mathbf{y}_j, \mathbf{x}_k, \mathbf{v}_{k,j}\right), & (k,j) \in \mathcal{O} \end{cases}, \tag{2.5}$$

* In this book, we use the word "pose" to mean "position" plus "rotation".

where $\mathcal{O}$ is a set that contains the information at which pose the landmark was observed (usually not all the landmarks can be seen at every moment – we are likely to see only a small part of the landmarks at a single moment). These two equations together describe a most basic SLAM problem: how to solve the estimate $\mathbf{x}$ (localization) and $\mathbf{y}$ (mapping) problem with the noisy control input $\mathbf{u}$ and the sensor reading $\mathbf{z}$ data? Now, as we see, we have modeled the SLAM problem as a **state estimation problem**: How to estimate the internal, hidden state variables through the noisy measurement data?

The solution to the state estimation problem is related to the specific form of the two equations and which distribution the noise obeys. According to whether the motion and observation equations are **linear**, and whether the noise is assumed to be **Gaussian**, it is divided into **linear/nonlinear** and **Gaussian/non-Gaussian** systems. The Linear Gaussian (LG system) is the simplest, and its unbiased optimal estimation can be given by the Kalman Filter (KF). In the complex nonlinear non-Gaussian (NLNG system), we basically rely on two methods: Extended Kalman Filter (EKF) and nonlinear optimization. Until the early 21st century, the EKF-based filter method still dominated SLAM. We will linearize the system at the working point and solve it in two steps: predict-update (see Lecture 10). The earliest real-time visual SLAM system was developed based on EKF [1]. Subsequently, in order to overcome the shortcomings of EKF (such as the linearization error and noise Gaussian distribution assumptions), people began to use other filters such as particle filters, and even use nonlinear optimization methods. Today, the mainstream of visual SLAM uses state-of-the-art optimization techniques represented by Graph Optimization [9]. We believe that optimization methods are clearly superior to filters, and as long as computing resources allow, optimization methods are often preferred (see lectures 10 and 11).

Now we believe the reader has a general understanding of the mathematical model of SLAM, but we still need to clarify some issues. First, we should explain what the **pose** $\boldsymbol{x}$ is. The word **pose** we just said is somewhat vague. Perhaps the reader can understand that a robot moving in the plane can be parameterized by two coordinates plus an angle. However, more robots are more like things in the three-dimensional space. We know that the motion of the three-dimensional space consists of three axes, so the movement of the robot is described by the translation on the three axes and the rotation around the three axes, totally having six Degrees of Freedom (DoF). But wait, does that mean that we can describe it with a vector in $\mathbb{R}^6$? We will find that things are not that simple. For a 6 DoF **pose**[*], how to express it? How to optimize it? How to describe its mathematical properties? This will be the main content of the third and fourth chapters. Next, we will explain how the **observation equation** is parameterized in the visual SLAM. In other words, how the landmark points in space are projected onto a photo? This requires an explanation of the camera's projection model and distortion, which we will cover in chapter 5. Finally, when you know this information, **how to solve the above state estimation problem**? This requires knowledge of nonlinear optimization and is the content of Lecture 6.

These contents form part of the mathematical knowledge of this book. After laying the groundwork for them, we can discuss more detailed knowledge of visual odometry, back-end optimization, and more. It can be seen that the content of this lecture constitutes a summary of the book. If you don't understand the above

[*] We will call it **pose** in the future to distinguish it from the position. The pose we are talking about includes **Rotation** and **Translation**.

concepts well, you may want to go back and read them again. Let's start the introduction of programming!

2.5 Practice: Basics

2.5.1 Installing Linux

Finally we come to the exciting practice session! Are you ready? In order to complete the practice of this book, we need to prepare a computer. You can use a laptop or desktop, preferably your personal computer, because we need to install an operating system on it for experiments.

Our program is based on C++ programs on Linux. During the experiment, we will use a number of open source libraries. Most libraries are only supported in Linux, while configuration on Windows is relatively (or quite) cumbersome. Therefore, we have to assume that you already have a basic knowledge of Linux (see the exercises in the previous lecture), including using basic commands to understand how the software is installed. Of course, you don't have to know how to develop C++ programs under Linux, which is exactly what we want to talk about below.

Let's start from installing the experimental environment required for this book. As a book for beginners, we use Ubuntu as a development environment. Ubuntu and its variances have enjoyed a good reputation as a novice user in all major Linux distributions. Ubuntu is an open source operating system. Its system and software can be downloaded freely on the official website (`http://ubuntu.com`), which provides detailed instructions on how to install it. At the same time, Tsinghua University, China Science and Technology University and other major universities in China have also provided Ubuntu software mirrors, making the software installation very convenient (probably there are also mirror websites in your country).

The first version of this book uses Ubuntu 14.04 as the default development environment. In the second edition, we updated the default version to the newer **Ubuntu 18.04** (Figure 2-11) for later research. If you want to change the styles, then Ubuntu Kylin, Debian, Deepin and Linux Mint are also good choices. I promise that all the code in the book has been well tested under Ubuntu 18.04, but if you choose a different distribution, I am not sure if you will encounter some minor problems. You may need to spend some time solving small issues (but you can also take them as opportunities to exercise yourself). In general, Ubuntu's support for various libraries is relatively complete, and the software is also very rich. Although we don't limit which Linux distribution you use, in the explanation, **we will use Ubuntu 18.04 as an example**, and mainly use Ubuntu commands (such as apt-get), so in other versions of Ubuntu there will be no obvious differences below. In general, the migration of programs between Linux is not very difficult. But if you want to use the programs in this book under Windows or OS X, you need to have some porting experience.

Now, I assume there's an Ubuntu 18.04 installed on your PC. Regarding the installation of Ubuntu, you can find a lot of tutorials on the Internet. Just do it, I'll skip here. The easiest way is to use a virtual machine (see Figure 2-11), but it takes a lot of memory (our experience is more than 4GB) and CPU to be smooth. You can also install dual systems, which will be faster, but a blank USB flash drive is required as the boot disk. In addition, virtual machine software support for external hardware is often not good enough. If you want to use real sensors

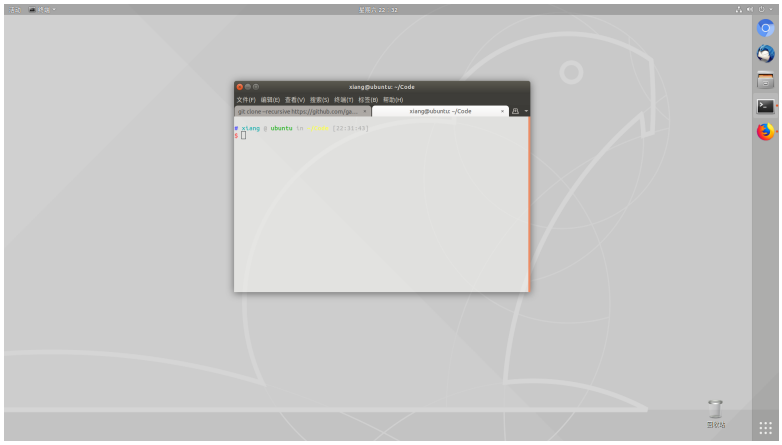

Figure 2-11: Ubuntu 1804 in virtual machine.

(binocular cameras, Kinects, etc.), it is recommended that you use dual systems to install Linux.

Now, let's say you have successfully installed Ubuntu, either it's a virtual machine or a dual system. If you are not familiar with Ubuntu, try its various software and experience its interface and interaction mode*. But I have to **remind** you, especially to the novice friends: don't spend too much time on Ubuntu's user interface! Linux has a lot of chances to waste your time, you may find some niche software, some games, and even spend a lot of time looking for a wallpaper. But remember, you are working with Linux. Especially in this book, you are using Linux to learn SLAM, so try to spend your time on learning SLAM.

Ok, let's choose a directory and put the code for the SLAM program in this book. For example, you can put the code under "slambook2" in the home directory (/home). We will refer to this directory as "**code root**" in the future. At the same time, you can choose another directory to copy the Git code of this book, which is convenient for comparison when doing experiments. The code for this book is divided into chapters. For example, the code for this chapter will be under slambook2/ch2, and the next one will be under slambook2/ch3. So, now please go into the slambook2/ch2 directory (you should create a new folder and enter the folder name).

2.5.2 Hello SLAM

Like many computer books, let's write a HelloSLAM program. But before we do this, let's talk about what a program is.

In Linux, a program is a file with execute permissions. It can be a script or a binary file, but we don't limit its suffix name (unlike Windows, you need to specify it as an .exe file). The commonly used binaries such as **cd** and **ls** are executable files located in the /bin directory. For an executable program elsewhere, as long as it has executable permissions, it will run when we enter the program name in the terminal. When programming in C++, we first write a text file:

Listing 2.1: slambook2/ch2/helloSLAM.cpp

* Most people think Ubuntu is cool for the first time.

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

int main(int argc, char **argv) {
    cout << "Hello SLAM!" << endl;
    return 0;
}
```

Then use a program called **compiler** to compile this text file into an executable program. Obviously this is a very simple program which just prints a hello slam text. You should be able to understand it effortlessly, so there is no explanation here - if this is not the case, please take a look at the basics of C++. This program just outputs a string to the screen. You can use the text editor like **gedit** (or **Vim**, if you have learned Vim in the previous tutorial) and enter the code and save it in the path listed above. Now, we compile it into an executable using the compiler g++ (g++ is a C++ compiler). Enter:

Listing 2.2: Terminal input:

```
g++ helloSLAM.cpp
```

If it goes well, this command should have no output. If the "command not found" error message appears on the screen, you may not have g++ installed yet. Please use the following command to install it:

Listing 2.3: terminal input:

```
sudo apt-get install g++
```

If there are other errors, please check again if the program you just entered is correct.

Just now this compile command compiles the text file helloSLAM.cpp into an executable program. We check the current directory and find that there is an additional a.out file, and it has execute permissions (the colors in the terminal are different, should be green in default settings). We can enter ./a.out to run the program *:

Listing 2.4: terminal input:

```
% ./a.out
Hello SLAM!
```

As we thought, this program outputs "Hello SLAM!", telling us that it is running correctly.

Please review what we did before. In this example, we used the editor to enter the source code for helloSLAM.cpp, then called the g++ compiler to compile it and get the executable. By default, g++ compiles the source file into a program of the name a.out (it is a bit weird, but acceptable). If we like, we can also specify the file name of this output. This is an extremely simple example, we actually **use a lot of hidden default parameters, almost omitting all intermediate steps**, in order to give the reader a simple impression (although you may not have realized it). Below we will use cmake to compile this program.

* Don't type the first %.

2.5.3 Use cmake

Theoretically, any C++ program can be compiled with g++. But when the program size is getting bigger and bigger, a project may have many folders and source files, and the compiled commands will be longer and longer. Usually a small C++ project may contain more than a dozen classes, and there are complex dependencies between these classes. Some of them are compiled into executables, and some are compiled into libraries. If we only rely on the g++ command, we need to enter a lot of commands, and the whole compilation process will become very cumbersome. Therefore, for C++ projects, using some engineering management tools is more efficient. In history, engineers used **makefile** to compile automatically, but the cmake to be discussed below is more convenient than it. And cmake is widely used in engineering, we will see that most of the libraries mentioned later use cmake to manage the source code.

In a cmake project, we will use the cmake command to generate a makefile, and then use the make command to compile the entire project based on the contents of the makefile. The reader may not know what a makefile is, but it doesn't matter, we will learn by example. Still taking the above helloSLAM.cpp as an example, this time we are not using g++ directly, but using cmake to build a project and then compiling it. Create a new CMakeLists.txt file in slambook2/ch2/ with the following contents:

Listing 2.5: slambook2/ch2/CMakeLists.txt

```
cmake_minimum_required( VERSION 2.8 )
project( HelloSLAM )
add_executable( helloSLAM helloSLAM.cpp )
```

The CMakeLists.txt file is used to tell cmake what we want to do with the files in this directory. The contents of the CMakeLists.txt file need to follow the cmake syntax. In this example, we demonstrate the most basic project: specifying a project name and an executable program. According to the comments, the reader should understand what each sentence does.

Now, in the current directory (slambook2/ch2/), call cmake to compile the project: *:

Listing 2.6: Terminal input

```
cmake .
```

cmake will output some compilation information, and then generate some intermediate files in the current directory, the most important of which is the makefile†. Since MakeFile is automatically generated, we don't have to modify it. Now, compile the project with the make command.

Listing 2.7: Terminal input

```
% make
Scanning dependencies of target helloSLAM
[100%] Building CXX object CMakeFiles/helloSLAM.dir/helloSLAM.cpp.o
Linking CXX executable helloSLAM
[100%] Built target helloSLAM
```

* Note that there's a dot at the end of the command, please don't forget it, which means using cmake in the current directory.

† Makefile is an automated compilation script, the reader can now understand it as a system automatically generated compiler instructions, without taking care of its content.

The compiler will show a process percent during compilation. We then get the declared executable **helloSLAM** in our CMakeLists.txt if the compilation is successful. Just type:

Listing 2.8: Terminal Input

```
% ./helloSLAM
Hello SLAM!
```

to run it. Because we didn't modify the source code, we got the same result as before. Please think about the difference between this practice and the previous use of g++ compiler. This time we used the cmake-make process. The cmake process handles the relationship between the project files, and the make process actually calls g++ to compile the program. By calling this cmake-make process, we have a good management for the project: **from inputting a string of g++ commands to maintaining several relatively intuitive CMakeLists.txt files**, which will obviously reduce the difficulty of maintaining the entire project. For example, if you want to add another executable file, just add a line "add_executable" in CMakeLists.txt, and the subsequent steps are unchanged. Cmake will help us resolve code dependencies without having to type in a bunch of g++ commands.

The only thing that is dissatisfied with this process is that the intermediate files generated by cmake are still in our code files. When we want to release the code, we don't want to publish these intermediate files together. At this time, we still need to delete them one by one, which is very inconvenient. A better approach is to have these intermediate files in an intermediate directory. After the compilation is successful, we will delete the intermediate directory. Therefore, the more common practice of compiling cmake projects is as follows:

Listing 2.9: Terminal input

```
mkdir build
cd build
cmake ..
make
```

We created a new intermediate folder "build", and then entered the build folder, using the cmake .. command to compile the previous folder, which is the folder where the code is located. In this way, the intermediate files generated by cmake will be in the "build" folder, separate from the source code. When publishing the source code, we just delete the build folder. Please try to compile the code in ch2 in this way, and then call the generated executable (please remember to delete the intermediate file generated in the last section).

2.5.4 Use Libraries

In a C++ project, not all code is compiled into executables. Only executable files with the main function will generate executable programs. For other code, we just want to package them into a packet for other programs to call. This packet is called **library**.

A library is often just a collection of many algorithms and programs, and we will be exposed to many libraries in later exercises. For example, the OpenCV library provides many computer vision related algorithms, while the Eigen library provides calculations of matrix algebra. Therefore, we need to learn how to use cmake to

generate libraries and use the functions in the library. Now let's demonstrate how to write a library yourself. Write the following libHelloSLAM.cpp file:

Listing 2.10: slambook2/ch2/libHelloSLAM.cpp

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

// Just a funtion printing hello message
void printHello() {
    cout << "Hello SLAM" << endl;
}
```

This library provides a printHello function that will output a message. But it doesn't have a main function, which means there are no executables in this library. We add the following to CMakeLists.txt:

Listing 2.11: slambook2/ch2/CMakeLists.txt

```
add_library( hello libHelloSLAM.cpp )
```

This line tells cmake that we want to compile this file into a library called "hello". Then, as above, compile the entire project using cmake:

Listing 2.12: Terminal input

```
cd build
cmake ..
make
```

At this point, a libhello.a file is generated in the build folder, which is the library we declared.

In Linux, the library files are divided into **static library** and **shared library**. Static libraries have a .a extension and shared libraries end with .so. All libraries are collections of functions that are packaged. The difference is that **a static library will generate a copy each time it is called, and the shared library has only one copy**, which saves space. If you want to generate a shared library instead of a static library, just use the following statement:

Listing 2.13: slambook2/ch2/CMakeLists.txt

```
add_library( hello_shared SHARED libHelloSLAM.cpp )
```

Then we will get a libhello_shared.so.

The library file is a compressed package with compiled binary functions. However, if there is only a .a or .so library file, then we don't know what the function is and how to call it. In order for others (or ourselves) to use this library, we need to provide a **header file** to indicate what is in the library. Therefore, for the user of the library, **you can call this library as long as you get the header and library files**. Write the header file for libhello below.

Listing 2.14: slambook2/ch2/libHelloSLAM.h

```
#ifndef LIBHELLOSLAM_H_
#define LIBHELLOSLAM_H_

// Declares a funtion in header file
void printHello();

#endif
```

In this way, according to this file and the library file we just compiled, you can use the printHello function. Finally, we write an executable program to call this simple function:

Listing 2.15: slambook2/ch2/useHello.cpp

```
#include "libHelloSLAM.h"

// Call printHello() in libHelloSLAM.h
int main(int argc, char **argv) {
    printHello();
    return 0;
}
```

Then, declare an executable in CMakeLists.txt and **link** it to the library:

Listing 2.16: slambook2/ch2/CMakeLists.txt

```
add_executable( useHello useHello.cpp )
target_link_libraries( useHello hello_shared )
```

Through these two lines of statements, the useHello program can successfully use the code in the hello_shared library. This small example demonstrates how to generate and call a library. Please note that for libraries provided by others, we can also call them in the same way and integrate them into our own programs.

In addition to the features already demonstrated, cmake has many more syntax and options. Of course we can not list all of them here. In fact, cmake is very similar to a normal programming language, with variables and conditional control statements, so you can learn cmake just like learning programming. The exercises contain some reading materials for cmake, which can be read by interested readers. Now, a brief review of what we did before:

1. First, the program code consists of a header file and a source file.
2. The source file with the main function is compiled into an executable program, and the other is compiled into a library file.
3. If the executable wants to call a function in the library file, it needs to refer to the header file provided by the library to understand the format of the call. Also, link the executable to the library file.

These steps should be simple and clear, but you may encounter some problems in the actual operation. For example, what happens if the executable references a library function but we forget to link the library? Try removing the link command in CMakeLists.txt and see what happens. Can you understand the error message reported by cmake?

2.5.5 Use IDE

Finally, let's talk about how to use the Integrated Development Environments (IDEs). The previous programming can be done with a simple text editor. However, you may need to jump between files to query the declaration and implementation of a function. This can be a little annoying when there are too many files. The IDE provides developers with a lot of convenient functions such as jump, completion, breakpoint debugging, etc. Therefore, we recommend that the reader choose an IDE for development.

There are many kinds of IDEs under Linux. Although there are still some gaps with the best IDE (I mean Visual Studio in Windows), there are several supported C++ developments, such as Eclipse, Qt Creator, Code::Blocks, Clion, Visual Studio Code, and so on. Again, we don't force readers to use a particular IDE, but only give our advice. We are using KDevelop and Clion (see Figure 2-12 and Figure 2-15)*. KDevelop is a free software located in Ubuntu's software repository, meaning you can install it with apt-get; Clion is a paid software, but you can use the student mailbox for free for one year. Both are good C++ development environments, the advantages are listed below:

1. Support cmake projects.
2. Support C++ better (including the 11 and later standards). There are highlighting, jumping, and finishing functions. Can automatically format the code.
3. Makes it easy to see individual files and directory trees.
4. Has one-click compilation, breakpoint debugging and other functions.

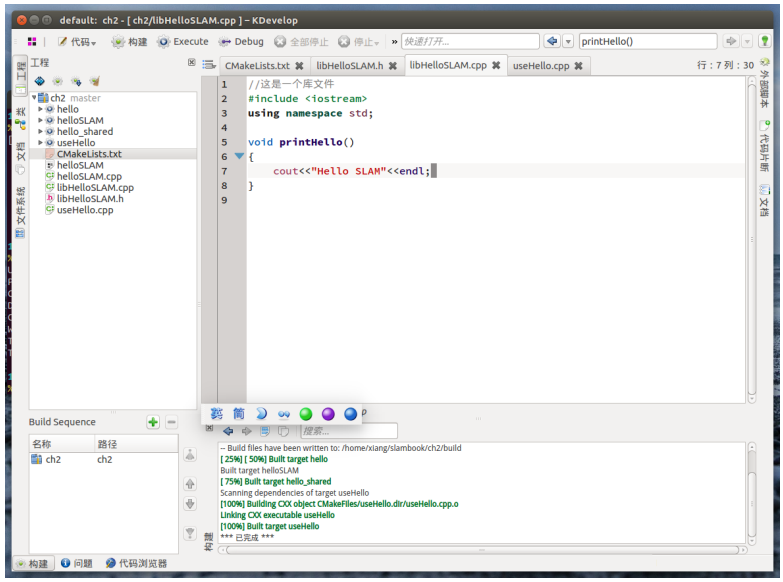

Figure 2-12: Kdevelop in Ubuntu.

Below we take a little bit of space to introduce KDevelop and Clion.

Use KDE

Kdevelop natively supports the cmake project. To do this, after creating CMakeLists.txt in the terminal, open CMakeLists.txt with "Project →Open/Import Project" in KDevelop. The software will ask you a few questions, and by default create a build folder to help you call the cmake and make commands. These can be done

* However, the recent Visual Studio Code is getting better and better. It's free. It's very popular among developers. You may have a try.

automatically by pressing the shortcut key F8. The following section of Figure 2-12 shows the compilation information.

We hand over the task of adapting to the IDE to the reader. If you are transferring from Windows, you will find its interface similar to Visual C++ or Visual Studio. Please use KDevelop to open the previous project and compile it to see what information it outputs. I believe you will feel more convenient than opening the terminal.

Next, let's show to debug in the IDE. Most of the students who program under Windows will have experience of breakpoint debugging under Visual Studio. However, in Linux, the default debugging tool gdb only provides a text interface, which is not convenient for novices. Some IDEs provide breakpoint debugging (the bottom layer is still gdb), and KDevelop is one of them. To use KDevelop's breakpoint debugging feature, you need to do the following:

1. Set the project to Debug compilation mode in CMakeLists.txt, and don't use optimization options (not used by default).

2. Tell KDevelop which program you want to run. If there are parameters, also configure its parameters and working directory.

3. Enter the breakpoint debugging interface, you can single step, see the value of the intermediate variable.

The first step is to set the compilation mode by adding the following command to CMakeLists.txt:

Listing 2.17: slambook2/ch2/CMakeLists.txt

```
1 Set( CMAKE_BUILD_TYPE "Debug" )
```

Cmake has some compilation-related built-in variables that give you more detailed control over the compilation process. For the compilation type, there is usually a Debug mode for debugging and a Release mode for publishing. In Debug mode, the program runs slower, but breakpoint debugging is possible, and you can see the values of the variables; while Release mode is faster, but there is probably no debugging information. We set the program to Debug mode and place the breakpoint. Next, tell KDevelop which program you want to launch.

In the second step, open "Run →Configure Launcher" and click on "Add New → Application" on the left. In this step, our task is to tell KDevelop which program to launch. As shown in Figure 2-13, you can either select a cmake project target (that is, the executable we built with the add_executable directive) or point to a binary file. The second approach is recommended, and in our experience, this is less of a problem.

In the second column, you can set the program's parameters and working directory. Sometimes programs have runtime parameters that are passed in as arguments to the main function. If not, leave it blank, as is the working directory. After configuring these two items, click the "OK" button to save the configuration results.

In just these steps we have configured an application startup item. For each startup item, we can click the "Execute" button to start the program directly, or click the "Debug" button to debug it. Readers can try to click the "Execute" button to see the results of the output. Now, to debug this program, click on the left side of the printHello line and add a breakpoint. Then, click on the "Debug" button and the program will wait at the breakpoint, as shown by Figure 2-14.

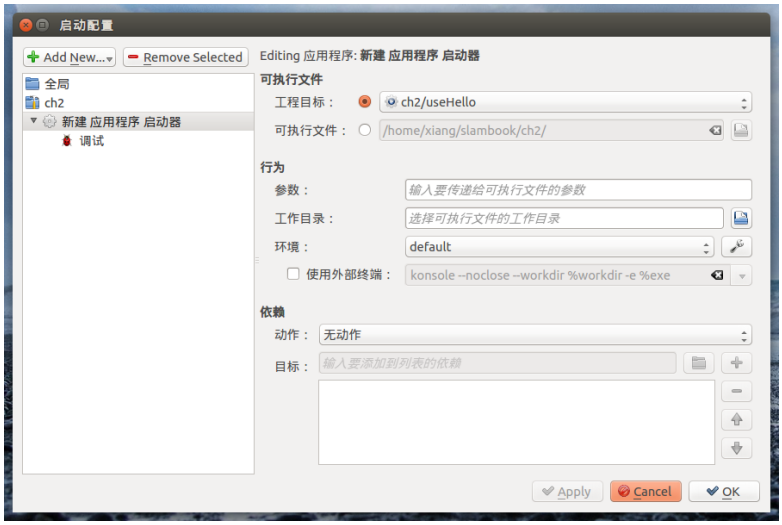

Figure 2-13: Config launches. We can choose a launch target and set parameters here.

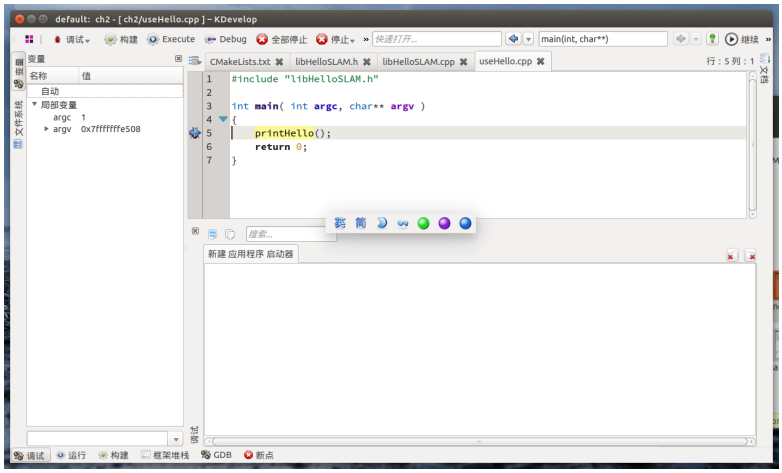

Figure 2-14: Debug interface.

When debugging, KDevelop will switch to debug mode and the interface will change a bit. At the breakpoint, you can control the operation of the program with single step operation (F10 key), single step follow up (F11 key), and single step jump (F12 key) function. At the same time, you can click the interface on the left to view the value of the local variable. Or select the "Stop" button to end debugging. After debugging, KDevelop will return to the normal development interface.

Now you should be familiar with the entire process of breakpoint debugging. In the future, if an error occurs during the running phase of the program, causing the program to crash, you can use breakpoint debugging to determine the location of the error, and then modify *.

* instead of directly sending us an email asking how to deal with the problem.

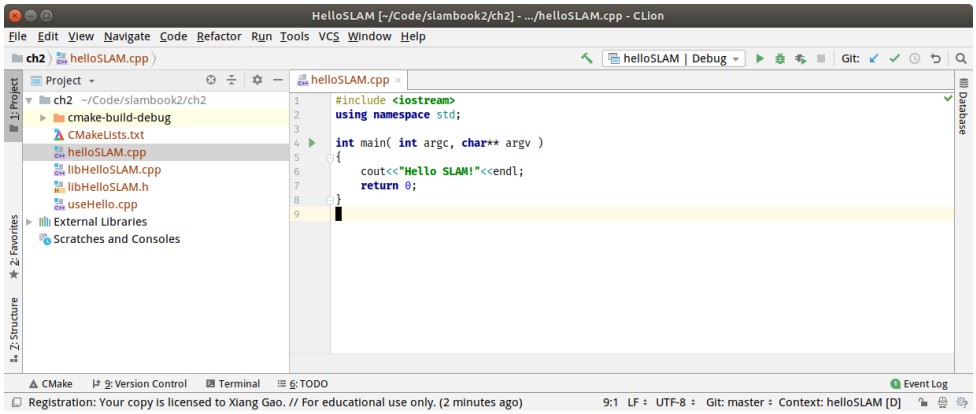

Figure 2-15: Clion interface

Use Clion

Clion is more complete than KDevelop, but it requires an user account, and the memory/CPU requirements for the host will be higher. *. In Clion, you can also open a CMakeLists.txt or specify a directory. Clion will complete the cmake-make process for you. Its running interface is shown in Figure 2-15.

Similarly, after opening Clion, you can select the programs you want to run or debug in the upper right corner of the interface, and adjust their startup parameters and working directory. Click the small beetle button in this column to start the breakpoint debugging mode. Clion also has a number of convenient features, such as automatically creating classes, changing functions, and automatically adjusting the coding style. Please try it.

Ok, if you are already familiar with the use of the IDE, then the second chapter will stop here. You may already feel that I have talked too much, so in the following practice section, we will not introduce things like how to create a new build folder, call the cmake and make commands to compile the program. I believe that readers should master these simple steps. Similarly, since most of the third-party libraries used in this book are cmake projects, you will continue to be familiar with the compilation process. Next we will start the formal chapter and introduce some related mathematics.

Exercises

1. Read the survey SLAM literature [10] and [11], can you read the contents? (They are Chinese papers for beginners. For English reader, see the next exercise.)

2.* Read SLAM's review literature, such as [7, 12–15] and so on. What are the similarities and differences between these papers on SLAM and this book?

3. What are the parameters of the

* CLion is abnormally slow in the version after 2018. It is recommended that you use the release version around 2017.

4. g++ command? If I want to change the generated program file name, how should I call g++?

5. Use the build folder to compile your cmake project, then try it in KDevelop.

6. Deliberately add some syntax errors to the code to see what information the build will generate. Can you read the error message of g++?

7. If you forgot to link the library to the executable, will the compiler report an error? What kind of mistakes are reported?

8.* Read "cmake practice" (or other cmake materials) to learn about the grammars of cmake.

9.* Improve the hello SLAM problem, make it a small library, and install it on your local hard drive. Then, create a new project, use find_package to find the library and call it.

10.* Read other cmake instructional materials, such as `https://github.com/TheErk/CMake-tutorial`.

11. Find the official website of KDevelop and see what other features it has. Are you using it?

12. If you learned Vim in the last lecture, please try KDevelop's/Clion's Vim editing function.

Chapter 3

3D Rigid Body Motion

Goal of Study

1. Understand the description of rigid body motion in three-dimensional space: rotation matrix, transformation matrix, quaternion and Euler angle.
2. Understand 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.

3.1 Rotation Matrix

3.1.1 Point, Vector and 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: **points** and **vectors**. Points are the basic elements in space, no length, no volume. Connecting the two points forms a vector.

A vector can be thought of as an arrow pointing from one point to another. We need to remind the reader that, please do not confuse the vector with its **coordinates**. A vector is one of the things in space, such as $\mathbf{a}$. Here $\mathbf{a}$ 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** * $(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$, then, the arbitrary vector $\mathbf{a}$ has a **coordinate** under this set of bases:

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

Here $(a_1, a_2, a_3)^{\mathrm{T}}$ is called $\mathbf{a}$'s coordinates †. The specific values of the coordinates are related to the vector itself, and also the selection of the bases. In $\mathbb{R}^3$, 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 $\mathbf{x}$ and $\mathbf{y}$ axis, the $\mathbf{z}$ axis can be found using the right-hand (or left-hand) rule by $\mathbf{x} \times \mathbf{y}$. 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 the operations between vectors/vectors, and vectors/numbers, such as scalar multiplication, vector addition, subtraction, inner product, outer product, and so on. Multiplication, addition and subtraction are 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 $\mathbf{a}, \mathbf{b} \in \mathbb{R}^3$, in the usual sense ‡ , the inner product of $\mathbf{a}, \mathbf{b}$ can be written as:

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

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

$$\mathbf{a} \times \mathbf{b} = \begin{Vmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{Vmatrix} = \begin{bmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{bmatrix} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \mathbf{b} \triangleq \mathbf{a}^{\wedge}\mathbf{b}. \tag{3.3}$$

The result of the outer product is a vector whose direction is perpendicular to the two vectors, and the length is $|\mathbf{a}| \, |\mathbf{b}| \, \langle \mathbf{a}, \mathbf{b} \rangle$, which is also the area of the quadrilateral of the two vectors. For the outer product operations, we introduce the $^{\wedge}$ operator

---

* Just a reminder here, the base is a set of linearly independent vectors in the space, normally being orthognal and has unit-length.

† We use column vectors in this book which is same as most of the mathematics books.

‡ the inner product also has formal rules, but this book only discusses the usual inner product.

here, which means writing $\mathbf{a}$ as a matrix. In fact, it is a **skew-symmetric matrix***. You can take $^\wedge$ as an skew-symmetric symbol. It turns the outer product $\mathbf{a} \times \mathbf{b}$ into the multiplication of the matrix and the vector $\mathbf{a}^\wedge\mathbf{b}$, which turns it into a linear operator. This symbol will be used frequently in the following sections, and this symbol is a one-to-one mapping, meaning that for any vector, it corresponds to a unique anti-symmetric matrix, and vice versa:

$$\mathbf{a}^\wedge = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}. \tag{3.4}$$

At the same time, note that the vector operations such as addition, subtraction, internal and external products, can be calculated even when we do not have their coordinates. For example, although the inner product can be expressed by the sum of the product products of the two vectors when we know the coordinates, it can also be calculated by the length and the angle even if their coordinates are unknown. Therefore, the inner product result of the two vectors is independent of the selection of the coordinate system.

3.1.2 Euclidean Transforms

We often define a variety of coordinate systems in the real scene. In robotics, you define one coordinate system for each link and joint; in 3D mapping, we also define the coordinate system for each cuboid and cylinder. If we 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 the x_W, y_W, z_W defined in Fig. 3-1. 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 $\mathbf{p}$ in the camera's field of view, has coordinates $\mathbf{p}_c$ in the camera coordinate system; and in the world coordinate system, its coordinates are $\mathbf{p}_w$, 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 way to describe this transformation. As we will see later, we can describe it with a matrix $\mathbf{T}$.

Intuitively, the motion between two coordinate systems consists of a rotation plus a translation, which is called **rigid body motion**. Obviously, the camera movement is a rigid body one. During the rigid body motion, the length and angle of the a vector will not change. Imagine you throw your phone into the air and †, 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 or be stretched during this motion. At this point, we say that the phone's motion is a **Euclidean Transform**.

The Euclidean transform consists of rotation and translation. Let's first consider about the rotation. We have an unit-length orthogonal base $(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$. After a rotation it becomes $(\mathbf{e}'_1, \mathbf{e}'_2, \mathbf{e}'_3)$. Then, for the same vector $\mathbf{a}$ (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:

* Skew-symmetric matrix means $\mathbf{A}$ satisfies $\mathbf{A}^\mathrm{T} = -\mathbf{A}$.

† Please don't put it into practice because it will fall on the ground and crash.

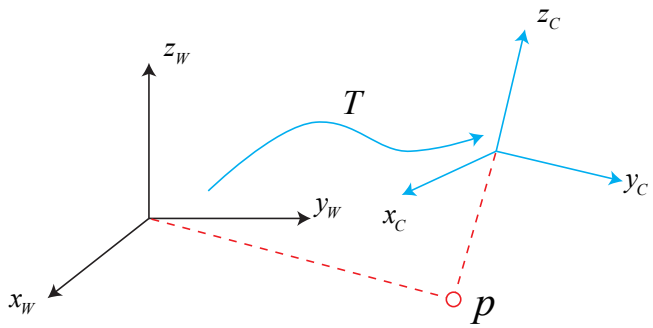

Figure 3-1: Coordinate transformation. For the same vector $\mathbf{p}$, it coordinates in the world $\mathbf{p}_W$ and coordinates in the camera system $\mathbf{p}_C$ is different. This transformation relationship is described by the transform matrix $\mathbf{T}$.

$$[\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3] \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = [\mathbf{e}'_1, \mathbf{e}'_2, \mathbf{e}'_3] \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \end{bmatrix}. \tag{3.5}$$

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

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1^{\mathrm{T}}\mathbf{e}'_1 & \mathbf{e}_1^{\mathrm{T}}\mathbf{e}'_2 & \mathbf{e}_1^{\mathrm{T}}\mathbf{e}'_3 \\ \mathbf{e}_2^{\mathrm{T}}\mathbf{e}'_1 & \mathbf{e}_2^{\mathrm{T}}\mathbf{e}'_2 & \mathbf{e}_2^{\mathrm{T}}\mathbf{e}'_3 \\ \mathbf{e}_3^{\mathrm{T}}\mathbf{e}'_1 & \mathbf{e}_3^{\mathrm{T}}\mathbf{e}'_2 & \mathbf{e}_3^{\mathrm{T}}\mathbf{e}'_3 \end{bmatrix} \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} \triangleq \mathbf{R}\mathbf{a}'. \tag{3.6}$$

We take the intermediate matrix out and define it as a matrix $\mathbf{R}$. This matrix consists of the inner product between the two sets of bases, describing the coordinate transformation relationship of the same vector before and after the rotation. As long as the rotation is the same, this matrix is the same. It can be said that the matrix $\mathbf{R}$ describes the rotation itself. So we call it the **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 the rotation matrix in the following.

The rotation matrix has some special properties. In fact, it is an orthogonal matrix with a determinant of 1 [*] [†]. 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:

$$\mathrm{SO}(n) = \{\mathbf{R} \in \mathbb{R}^{n\times n} | \mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{I}, \det(\mathbf{R}) = 1\}. \tag{3.7}$$

$\mathrm{SO}(n)$ is the meaning of **Special Orthogonal Group**. We leave the contents of the "group" to the next lecture. This collection consists of a rotation matrix of n

[*] Orthogonal matrix is a matrix whose inverse is its transpose. The orthogonality of the rotation matrix can be derived directly from the definition.

[†] The determinant is 1 is artificially defined. In fact, its determinant is ± 1, but the rotation with determinant -1 is called improper rotation, that is, one rotation plus one reflection.

dimensional space, in particular, SO(3) refers to the rotation of the three-dimensional space. By this way, we can talk directly about the rotation transformation between the two coordinate systems without having to start from the bases.

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

$$\mathbf{a}' = \mathbf{R}^{-1}\mathbf{a} = \mathbf{R}^{\mathrm{T}}\mathbf{a}. \tag{3.8}$$

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

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

$$\mathbf{a}' = \mathbf{R}\mathbf{a} + \mathbf{t}. \tag{3.9}$$

Where $\mathbf{t}$ 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 $\mathbf{R}$ and a translation vector $\mathbf{t}$. In practice, we will define the coordinate system 1, coordinate system 2, then the vector $\mathbf{a}$ under the two coordinates is $\mathbf{a}_1, \mathbf{a}_2$, they are The relationship between the two, in accordance with the complete writing, should be:

$$\mathbf{a}_1 = \mathbf{R}_{12}\mathbf{a}_2 + \mathbf{t}_{12}. \tag{3.10}$$

Here $\mathbf{R}_{12}$ means "rotation the vector from coordinate system 2 to coordinate system 1". Since the vector is multiplied to the right of this matrix, its subscript is **read from right to left**. This is just a 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 $\mathbf{R}_{21}$. The reader must be clear about the notation here, because different books have different notations, some will be recorded as the top left/subscript, and the text will be written on the right side.

About $\mathbf{t}_{12}$, it actually corresponds a vector from the coordinate system 1 origin pointing to the coordinate system 2 origin, whose **coordinates are taken under coordinate system 1**, so I suggest readers to put it as "a vector from 1 to 2". But the reverse $\mathbf{t}_{21}$, which is a vector from 2's origin to 1's origin, whose **coordinates are taken in coordinate system 2**, is not equal to $-\mathbf{t}_{12}$, but is also related to the rotation of the two systems*. Therefore, when beginners ask the question "Where is my coordinates?", we need to clearly explain the meaning of this sentence. Here "my coordinates" normally refers to the vector from the world system pointing to the origin of the robot system, and then take the coordinates in the world's base. Corresponding to the mathematical symbol, it should be the value of $\mathbf{t}_{WC}$. For the same reason, it is not $-\mathbf{t}_{CW}$, but actually $-\mathbf{R}_{CW}^{T}\mathbf{t}_{CW}$.

3.1.3 Transform Matrix and Homogeneous Coordinates

The formula (3.9) fully expresses the rotation and translation of Euclidean space, but there is still a small problem: the transformation relationship here is not a linear relationship. Suppose we made two transformations: $\mathbf{R}_1, \mathbf{t}_1$ and $\mathbf{R}_2, \mathbf{t}_2$:

* Although from the vector level, they are indeed inverse relations, but the coordinates of the two vectors are not opposite. Can you find out why it looks like this?

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

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

$$\mathbf{c} = \mathbf{R}_2 (\mathbf{R}_1\mathbf{a} + \mathbf{t}_1) + \mathbf{t}_2.$$

This form is not elegant after multiple transformations. Therefore, we introduce homogeneous coordinates and transformation matrices, rewriting the form (3.9):

$$\begin{bmatrix} \mathbf{a}' \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ 1 \end{bmatrix} \triangleq \mathbf{T} \begin{bmatrix} \mathbf{a} \\ 1 \end{bmatrix}. \tag{3.11}$$

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 one matrix, making the whole relationship a linear relationship. In this formula, the matrix $\mathbf{T}$ is called **Transform Matrix**.

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

$$\tilde{\mathbf{b}} = \mathbf{T}_1\tilde{\mathbf{a}}, \ \tilde{\mathbf{c}} = \mathbf{T}_2\tilde{\mathbf{b}} \quad \Rightarrow \tilde{\mathbf{c}} = \mathbf{T}_2\mathbf{T}_1\tilde{\mathbf{a}}. \tag{3.12}$$

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 make it a normal one*. So, without ambiguity, we will write it directly as $\mathbf{b} = \mathbf{T}\mathbf{a}$, and by default we just assume a homogeneous coordinate conversion is made if needed†.

Regarding the transformation matrix $\mathbf{T}$, 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 $\mathbf{0}$ vector, and the lower right corner is 1. This matrix is also known as the Special Euclidean Group:

$$\mathrm{SE}(3) = \left\{ \mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \in \mathbb{R}^{4\times 4} | \mathbf{R} \in \mathrm{SO}(3), \mathbf{t} \in \mathbb{R}^3 \right\}. \tag{3.13}$$

Like $\mathrm{SO}(3)$, solving the inverse of the matrix represents an inverse transformation:

$$\mathbf{T}^{-1} = \begin{bmatrix} \mathbf{R}^{\mathrm{T}} & -\mathbf{R}^{\mathrm{T}}\mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix}. \tag{3.14}$$

Again, we use the notation of $\mathbf{T}_{12}$ to represent a transformation from 2 to 1. Moreover, in order to keep the symbol concise, in the case of no ambiguity, the symbols of the homogeneous coordinates and the ordinary coordinates are not deliberately distinguished in the later sections. For example, when we write $\mathbf{T}\mathbf{a}$, we use homogeneous coordinates (otherwise we can't calculate). When you write $\mathbf{R}\mathbf{a}$, you use non-homogeneous coordinates. If they are written in the same equation, it is assumed that the conversion from homogeneous coordinates to normal coordinates is already done - because the conversion between homogeneous and non-homogeneous

* But the purpose of the homogeneous coordinates is not limited to this, we will come back to it in Chapter 7.

† Note that if homogeneous coordinate transformation is not performed, the matrix multiplication here does not make sense.

coordinates is actually very easy. In C++ programs. You can do this with **operator overloading** to ensure that the operations you see in the program are correct.

Let's take a review now. 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 .

3.2 Practice: Using 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* 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 it:

Listing 3.1: Terminal input:

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

Most of the commonly used libraries in our book 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 should be in "/usr/include/eigen3/". If you are not sure, you can find it by entering the following command:

Listing 3.2: 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 3.3: slambook2/ch3/useEigen/eigenMatrix.cpp

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

#include <ctime>
// Eigen core
```

* Official home page: `http://eigen.tuxfamily.org/index.php?title=Main_Page`.

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

#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;
```

```
61      cout << "[1,2,3;4,5,6]*[3,2,1]=" << result.transpose() << endl;
62
63      Matrix<float, 2, 1> result2 = matrix_23 * vd_3d;
64      cout << "[1,2,3;4,5,6]*[4,5,6]: " << result2.transpose() << endl;
65
66      // Also you can't misjudge the dimensions of the matrix
67      // Try canceling the comments below to see what Eigen will report.
68      // Eigen::Matrix<double, 2, 3> result_wrong_dimension =
69      // matrix_23.cast<double>() * v_3d;
70
71      // some matrix operations
72      // Four operations are not demonstrated, just use +-*/.
73      Matrix_33 = Matrix3d::Random(); // Random Number Matrix
74      cout << "random matrix: \n" << matrix_33 << endl;
75      cout << "transpose: \n" << matrix_33.transpose() << endl;
76      cout << "sum: " << matrix_33.sum() << endl;
77      cout << "trace: " << matrix_33.trace() << endl;
78      cout << "times 10: \n" << 10 * matrix_33 << endl;
79      cout << "inverse: \n" << matrix_33.inverse() << endl;
80      cout << "det: " << matrix_33.determinant() << endl;
81
82      // Eigenvalues
83      // Real symmetric matrix can guarantee successful diagonalization
84      SelfAdjointEigenSolver<Matrix3d> eigen_solver(matrix_33.transpose() *
85      matrix_33);
86      cout << "Eigen values = \n" << eigen_solver.eigenvalues() << endl;
87      cout << "Eigen vectors = \n" << eigen_solver.eigenvectors() << endl;
88
89      // Solving equations
90      // We solve the equation of matrix_NN * x = v_Nd
91      // The size of N is defined in the previous macro, which is generated
            by a
92      // random number Direct inversion is the most direct, but the amount of
93      // inverse operations is large.
94
95      Matrix<double, MATRIX_SIZE, MATRIX_SIZE> matrix_NN =
96      MatrixXd::Random(MATRIX_SIZE, MATRIX_SIZE);
97      matrix_NN =
98      matrix_NN * matrix_NN.transpose(); // Guarantee semi-positive definite
99      Matrix<double, MATRIX_SIZE, 1> v_Nd = MatrixXd::Random(MATRIX_SIZE, 1);
100
101     Clock_t time_stt = clock(); // timing
102     // Direct inversion
103     Matrix<double, MATRIX_SIZE, 1> x = matrix_NN.inverse() * v_Nd;
104     cout << "time of normal inverse is "
105     << 1000 * (clock() - time_stt) / (double)CLOCKS_PER_SEC << "ms" << endl
            ;
106     cout << "x = " << x.transpose() << endl;
107
108     // Usually solved by matrix decomposition, such as QR decomposition,
            the speed
109     // will be much faster
110     time_stt = clock();
111     x = matrix_NN.colPivHouseholderQr().solve(v_Nd);
112     cout << "time of Qr decomposition is "
113     << 1000 * (clock() - time_stt) / (double)CLOCKS_PER_SEC << "ms" << endl
            ;
114     cout << "x = " << x.transpose() << endl;
115
116     // For positive definite matrices, you can also use cholesky
            decomposition to
117     // solve equations.
118     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 3.4: 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 3.5: 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. Please enter the above code by yourself if you are a beginner in C++ (not including comments). At least compile and run the above program for once.

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.

3. The matrix provided by 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.

4. The matrix implementation inside 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.

5. The 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 variable and double variable, and **the compiler will automatically cast 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 very long "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 some compilation error about Eigen, 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?

3.3 Rotation Vector and Euler Angle

3.3.1 Rotation Vector

Now let's return to the theoretical part. With a rotation matrix to describe the rotation, is it enough to use a 4×4 transformation matrix to describe a 6-degree-of-freedom 3D rigid body motion? Obviously the matrix representation has at least the following disadvantages:

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

2. The 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, a rotation can be described by **a rotation axis and a rotation angle**. Thus, we can use a vector whose direction is parallel with the axis of rotation and the length is equal to the angle of rotation, which is called the **rotation vector** (or 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 $\mathbf{R}$. If described by a rotation vector, assuming that the rotation axis is a unit length vector $\mathbf{n}$ and the angle is θ, then the vector $\theta\mathbf{n}$ 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 a little complicated, it is not described here. Only the result of the conversion is given*:

$$\mathbf{R} = \cos\theta\mathbf{I} + (1 - \cos\theta)\,\mathbf{n}\mathbf{n}^{\mathrm{T}} + \sin\theta\mathbf{n}^{\wedge}. \tag{3.15}$$

The symbol $^{\wedge}$ is a vector to skew-symmetric conversion, see the formula (3.3). Conversely, we can also calculate the conversion from a rotation matrix to a rotation vector. For the corner θ, take the **trace** †, we have:

$$\begin{aligned} \mathrm{tr}\,(\mathbf{R}) &= \cos\theta\,\mathrm{tr}\,(\mathbf{I}) + (1 - \cos\theta)\,\mathrm{tr}\left(\mathbf{n}\mathbf{n}^{\mathrm{T}}\right) + \sin\theta\,\mathrm{tr}(\mathbf{n}^{\wedge}) \\ &= 3\cos\theta + (1 - \cos\theta) \\ &= 1 + 2\cos\theta. \end{aligned} \tag{3.16}$$

therefore:

$$\theta = \arccos(\frac{\mathrm{tr}(\mathbf{R}) - 1}{2}). \tag{3.17}$$

* 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 view.

† see **trace** on both sides to find the sum of the diagonal elements of the matrix.

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

$$\mathbf{Rn} = \mathbf{n}. \tag{3.18}$$

Therefore, the axis $\mathbf{n}$ is the eigen vector corresponding to the matrix $\mathbf{R}$'s eigenvalue 1. Solving this equation and normalizing it gives the axis of 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) .

3.3.2 Euler Angle

Let's talk about the Euler angle.

Whether it is a rotation matrix or a rotation vector, although they can describe the rotation, they are very unintuitive to us humans. When we see a rotation matrix or a rotation vector, it is hard to imagine what this rotation is like. When they change, we don't know which direction the object is turning. The Euler angle provides a very intuitive way to describe rotation—it uses **3 primal axes** to decompose a rotation into three rotations around different axes. Humans can easily understand the process of rotating around a single axis. However, due to the variety of decomposition methods, there are many different and confusing definition methods for Euler angles. For example, we can first rotate around the X axis, then around the Y axis, and finally around the Z axis, and by this way we get a rotation like XYZ order. Similarly, you can define rotation orders such as ZYZ and ZYX. You also need to distinguish whether it is rotated around the **fixed axis** or around the **axis after rotation**, which will also give a different definition.

This uncertainty in the axis orders brings many practical difficulties. Fortunately, in certain research areas, Euler angles usually have a uniform definition. You may have heard the words "pitch angle" and "yaw angle" of an aircraft. One of the most commonly used Euler angles is the yaw-pitch-roll angles. Since it is equivalent to the rotation of the ZYX axis, the ZYX is taken as an example. Suppose the front of a rigid body (toward our direction) is the X axis, the right side is the Y axis, and the top is the Z axis, as shown by Figure 3-2. Then, the ZYX angle is equivalent to decompose any rotation into the following three axes:

1. Rotate around the Z axis of the object to get the yaw angle $\theta_{\text{yaw}} = y$;
2. Rotate around the Y axis of **after rotation** to get the pitch angle $\theta_{\text{pitch}} = p$;
3. Rotate around the X axis of **after rotation** to get the roll angle $\theta_{\text{roll}} = r$.

By this way, you can use a three-dimensional vector such as $[r, p, y]^{\mathrm{T}}$ to describe any rotation. This vector is very intuitive, we can imagine the rotation process from this vector. The other Euler angles are also decomposed into three axes to obtain a three-dimensional vector, but the axes and order maybe different. The *rpy* angle introduced here is a widely used one, and only a few Euler angles have such a popular name as *rpy*. Different Euler angles are referred to in the order of the axes of rotation. For example, the rotation order of the rpy angle is ZYX. Similarly, there are Euler angles like XYZ, ZYZ - but they don't have a specific name. It is worth mentioning that most areas have their own coordinate directions and order habits when using Euler angles, not necessarily the same as we said here.

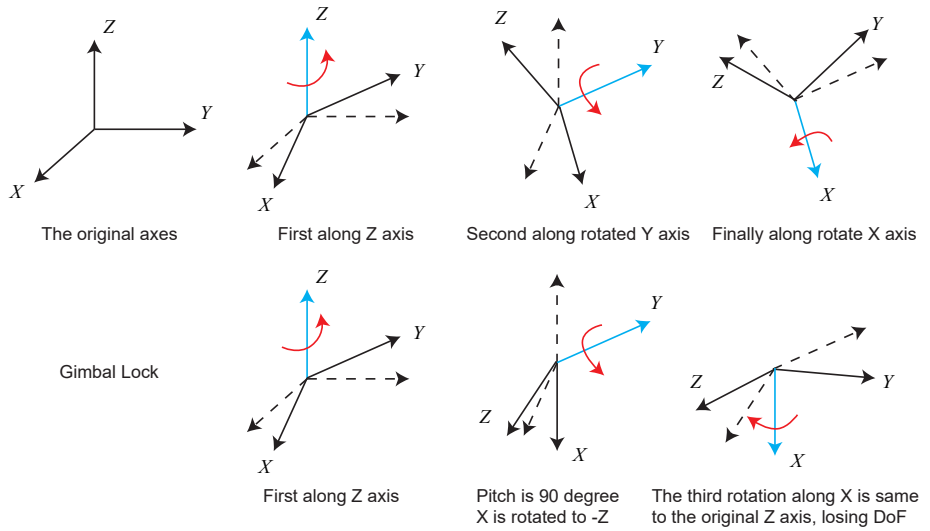

Figure 3-2: Euler angles. The top is defined for the ZYX order (rpy order). The bottoms shows when pitch=90°, the third rotation is using the same axis as the first one, causing the system to lose a degree of freedom. If you don't understand the universal lock, please take a look at the related videos and it will be more convenient to understand.

A major drawback of Euler Angle is that it encounters the famous **Gimbal Lock** *): in *rpy*'s case, when the pitch angle is ±90°, the first rotation and the third rotation will use the same axis, causing the system to lose a degree of freedom (from 3 rotations to 2 rotations). This is called the singularity problem and also exists in other forms of Euler Angles. In theory, it can be proved that as long as you want to use three real numbers to express the three-dimensional rotation, you will inevitably encounter the singularity problem†. Due to this principle, Euler angles are not suitable for interpolation and iteration, and are often only used in human-computer interaction. We also rarely use Euler angles to express poses directly in the SLAM program, nor do we use Euler angles to express rotation in filtering or optimization (because it has singularity). However, if you want to verify that your algorithm is correct or not, converting to Euler angles can help you quickly determine if the results are correct. In some cases where the main body is mainly 2D motion (such as sweepers, self-driving vehicles), we can also decompose the rotation into three Euler angles, and then take one of them (such as the yaw angle) as the positioning information output.

3.4 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 has

* `https://en.wikipedia.org/wiki/Gimbal_lock`.

† The rotation vector also has singularity, which occurs when the angle θ exceeds 2π. Obviously rotating 2π is same with no rotation.

singularity. In fact, we **cannot find a three-dimensional vector description without singularity**[16]. This is somewhat similar to using two coordinates to represent the Earth's surface (such as longitude and latitude), and there will be singularity (longitude is meaningless when latitude is $\pm 90^\circ$).

Recall the complex number that we have studied before. We use the complex set $\mathbb{C}$ to represent the vector on the 2D complex plane, and the complex multiplication with an unit complex number can represent the rotation on the 2D 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: **quaternions**. The quaternion is an extended complex number found by Hamilton. It **is both compact and not singular**. If we must find some 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 $\mathrm{e}^{i\theta}$, which is a complex number represented by polar coordinates. It can also be written in the usual form like the famous Euler equation:

$$\mathrm{e}^{i\theta} = \cos\theta + i\sin\theta. \tag{3.19}$$

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

A quaternion $\mathbf{q}$ has a real part and three imaginary parts. We write the real part in the front (and there are also some books where the real part is written in the last), like this:

$$\mathbf{q} = q_0 + q_1 i + q_2 j + q_3 k, \tag{3.20}$$

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

$$\begin{cases} i^2 = j^2 = k^2 = -1 \\ ij = k, ji = -k \\ jk = i, kj = -i \\ ki = j, i = -j \end{cases}. \tag{3.21}$$

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:

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

Here, s is the real part of the quaternion, and $\mathbf{v}$ is 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**.

We can use a **unit quaternion** to represent any rotation in 3D space, but this expression is subtly different from the complex numbers. In the complex, multiplying by i means rotating 90° . Does this mean that in the quaternion, multiplied by i is rotated around the i axis by 90° ? So, does $ij = k$ mean, first rotating around the i by 90° , then around j by 90° , is equivalent to rotating around k by 90° ? Readers can use a cell phone to simulate that, 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, we get an opposite thing. This thing has to be rotated for 720 ° to be equal to its original appearance.

This seems a bit mysterious, the complete explanation needs too much extra things, let's calm down and come back to the quaternions. At least, we know that a unit quaternion can express the rotation of a three-dimensional space. So what are the properties of the quaternions? And how can they operate with each other?

3.4.1 Quaternion Operations

Quaternions are very similar to complex numbers, and a series of operations can be performed. We can easily plus, minus, multiplies to quaternions just like doing with two complex numbers. Assume there are two quaternions $\mathbf{q}_a, \mathbf{q}_b$, whose vectors are represented as $[s_a, \mathbf{v}_a]^{\mathrm{T}}, [s_b, \mathbf{v}_b]^{\mathrm{T}}$, or the original quaternion is expressed as:

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

Then, their operations can be expressed as follows.

1. *Addition and Subtraction.* The addition and subtraction of the quaternion $\mathbf{q}_a, \mathbf{q}_b$ is:
$$\mathbf{q}_a \pm \mathbf{q}_b = [s_a \pm s_b, \mathbf{v}_a \pm \mathbf{v}_b]^{\mathrm{T}}. \tag{3.22}$$

2. *Multiplication.* Multiplication is the multiplication of each item of $\mathbf{q}_a$ with each item of $\mathbf{q}_b$, and finally, the imaginary part is done according to the formula (3.21):
$$\begin{aligned} \mathbf{q}_a \mathbf{q}_b = & s_a s_b - x_a x_b - y_a y_b - z_a z_b \\ & + (s_a x_b + x_a s_b + y_a z_b - z_a y_b) i \\ & + (s_a y_b - x_a z_b + y_a s_b + z_a x_b) j \\ & + (s_a z_b + x_a y_b - y_a x_b + z_a s_b) k. \end{aligned} \tag{3.23}$$
Although a little complicated, the form is neat and orderly. If written in vector form and using inner and outer product operations, the expression will be more concise:
$$\mathbf{q}_a \mathbf{q}_b = \left[s_a s_b - \mathbf{v}_a^{\mathrm{T}} \mathbf{v}_b, s_a \mathbf{v}_b + s_b \mathbf{v}_a + \mathbf{v}_a \times \mathbf{v}_b \right]^{\mathrm{T}}. \tag{3.24}$$
Under this multiplication definition, the product of two real quaternion is still real, which is also consistent with the real number multiplication. However, note that due to the existence of the last outer product, quaternion multiplication is usually not commutative unless $\mathbf{v}_a$ and $\mathbf{v}_b$ at $\mathbb{R}^3$ are parallel which means the outer product term is zero.

3. *Length.* The length of a quaternion is defined as:
$$\|\mathbf{q}_a\| = \sqrt{s_a^2 + x_a^2 + y_a^2 + z_a^2}. \tag{3.25}$$
It can be verified that the length of the product is the product of the length. This makes the unit quaternion keep unit length when multiplied by another unit quaternion:
$$\|\mathbf{q}_a \mathbf{q}_b\| = \|\mathbf{q}_a\| \|\mathbf{q}_b\|. \tag{3.26}$$

4. *Conjugate.* The conjugate of a quaternion is to take the imaginary part as the opposite:

$$\mathbf{q}_a^* = s_a - x_a i - y_a j - z_a k = [s_a, -\mathbf{v}_a]^{\mathrm{T}}. \tag{3.27}$$

We get a real quaternion if the quaternion is multiplied by its conjugate. The real part is the square of its length:

$$\mathbf{q}^*\mathbf{q} = \mathbf{q}\mathbf{q}^* = [s_a^2 + \mathbf{v}^{\mathrm{T}}\mathbf{v}, \mathbf{0}]^{\mathrm{T}}. \tag{3.28}$$

5. *Inverse.* The inverse of a quaternion is:

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

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

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

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

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

6. *Scalar Multiplication.* Similar to vectors, quaternions can be multiplied by numbers:

$$k\mathbf{q} = [ks, k\mathbf{v}]^{\mathrm{T}}. \tag{3.32}$$

3.4.2 Use Quaternion to Represent Rotation

We can use a quaternion to express the rotation of a point. Suppose a spatial 3D point $\mathbf{p} = [x, y, z]^{\mathrm{T}} \in \mathbb{R}^3$, and a rotation is specified by a unit quaternion $\mathbf{q}$. The 3D point $\mathbf{p}$ is rotated to become $\mathbf{p}'$. If we use matrix, then there is $\mathbf{p}' = \mathbf{R}\mathbf{p}$. And if we use quaternion to describe rotation, how do we operate a 3D vector with a quaternion?

First, we use extends the 3D point to a imaginary quaternion:

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

We just put the three coordinates into the imaginary part and leave the real part to zero. Then, the rotated point $\mathbf{p}'$ can be expressed as such a product:

$$\mathbf{p}' = \mathbf{q}\mathbf{p}\mathbf{q}^{-1}. \tag{3.33}$$

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

3.4.3 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/matrices. Before that, we have to say that quaternion multiplication can also be written as a matrix multiplication. Let $\mathbf{q} = [s, \mathbf{v}]^{\mathrm{T}}$, then define the following symbols $^+$ and $^\oplus$ as [17]:

$$\mathbf{q}^+ = \begin{bmatrix} s & -\mathbf{v}^{\mathrm{T}} \\ \mathbf{v} & s\mathbf{I} + \mathbf{v}^\wedge \end{bmatrix}, \quad \mathbf{q}^\oplus = \begin{bmatrix} s & -\mathbf{v}^{\mathrm{T}} \\ \mathbf{v} & s\mathbf{I} - \mathbf{v}^\wedge \end{bmatrix}, \tag{3.34}$$

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

$$\mathbf{q}_1^+ \mathbf{q}_2 = \begin{bmatrix} S_1 & -\mathbf{v}_1^T \\ \mathbf{v}_1 & s_1\mathbf{I} + \mathbf{v}_1^\wedge \end{bmatrix} \begin{bmatrix} s_2 \\ \mathbf{v}_2 \end{bmatrix} = \begin{bmatrix} -\mathbf{v}_1^T \mathbf{v}_2 + s_1 s_2 \\ s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1^\wedge \mathbf{v}_2 \end{bmatrix} = \mathbf{q}_1 \mathbf{q}_2 \tag{3.35}$$

We simply get:

$$\mathbf{q}_1 \mathbf{q}_2 = \mathbf{q}_1^+ \mathbf{q}_2 = \mathbf{q}_2^\oplus \mathbf{q}_1. \tag{3.36}$$

Then, consider the problem of using a quaternion to rotate a spatial point. According to the previous section, we have:

$$\begin{aligned} \mathbf{p}' &= \mathbf{q}\mathbf{p}\mathbf{q}^{-1} = \mathbf{q}^+ \mathbf{p}^+ \mathbf{q}^{-1} \\ &= \mathbf{q}^+ \mathbf{q}^{-1^\oplus} \mathbf{p}. \end{aligned} \tag{3.37}$$

Substituting the matrix corresponding to two symbols, we get:

$$\mathbf{q}^+ \left(\mathbf{q}^{-1}\right)^\oplus = \begin{bmatrix} s & -\mathbf{v}^T \\ \mathbf{v} & s\mathbf{I} + \mathbf{v}^\wedge \end{bmatrix} \begin{bmatrix} s & \mathbf{v}^T \\ -\mathbf{v} & s\mathbf{I} + \mathbf{v}^\wedge \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0}^{\mathrm{T}} & \mathbf{v}\mathbf{v}^{\mathrm{T}} + s^2\mathbf{I} + 2s\mathbf{v}^\wedge + \left(\mathbf{v}^\wedge\right)^2 \end{bmatrix}. \tag{3.38}$$

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

$$\mathbf{R} = \mathbf{v}\mathbf{v}^{\mathrm{T}} + s^2\mathbf{I} + 2s\mathbf{v}^\wedge + \left(\mathbf{v}^\wedge\right)^2. \tag{3.39}$$

In order to obtain the conversion formula of the quaternion to the rotation vector, we take the trace on both of two sides of the above formula:

$$\begin{aligned} \mathrm{tr}(\mathbf{R}) &= \mathrm{tr}(\mathbf{v}\mathbf{v}^{\mathrm{T}} + 3s^2 + 2s \cdot 0 + \mathrm{tr}((\mathbf{v}^\wedge)^2) \\ &= v_1^2 + v_2^2 + v_3^2 + 3s^2 - 2(v_1^2 + v_2^2 + v_3^2) \\ &= (1 - s^2) + 3s^2 - 2(1 - s^2) \\ &= 4s^2 - 1. \end{aligned} \tag{3.40}$$

Also obtained by the formula (3.17):

$$\begin{aligned} \theta &= \arccos(\frac{\mathrm{tr}(\mathbf{R}) - 1}{2}) \\ &= \arccos(2s^2 - 1). \end{aligned} \tag{3.41}$$

which is

$$\cos\theta = 2s^2 - 1 = 2\cos^2\frac{\theta}{2} - 1, \tag{3.42}$$

and so we have:

$$\theta = 2 \arccos s. \tag{3.43}$$

For the rotation axis, if we replace $\mathbf{p}$ with the imaginary part of $\mathbf{q}$ in the formula (3.38), it is easy to know the imaginary part of $\mathbf{q}$ is not moving when it is rotated, that is, it constitutes exactly the rotation axis. So we get the rotation axis just by normalizing $\mathbf{q}$'s imaginary part. In summary, the conversion formula for quaternion to rotation vector can be written as follows:

$$\begin{cases} \theta = 2 \arccos q_0 \\ [n_x, n_y, n_z]^{\mathrm{T}} = [q_1, q_2, q_3]^{\mathrm{T}} / \sin \frac{\theta}{2} \end{cases}. \tag{3.44}$$

As for how to convert from other representations to quaternions, we only need to reversly follow 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 an angle-axis, 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.

3.5 Affine and Projective Transformation

In addition to the Euclidean transformation, there are several other transformations in the 3D space, in which the Euclidean is the simplest. Some of them are related to the measurement geometry. We will introducte them in the following chapters, so here we only list their basic properties. The Euclidean transformation keeps the length and angle of the vector, which is equivalent to moving or rotating a rigid body without changing its appearance. The other transformations will change its shape, and they all have similar matrix representations.

1. *Similarity 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:

   $$\mathbf{T}_S = \begin{bmatrix} s\mathbf{R} & \mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix}. \tag{3.45}$$

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

2. *Affine transformation.*

   The matrix form of the affine transformation is as follows:

   $$\mathbf{T}_A = \begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix}. \tag{3.46}$$

Unlike the Euclidean transformation, the affine transformation requires only $\mathbf{A}$ 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. *Perspective transformation.*

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

$$\mathbf{T}_P = \begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{a}^{\mathrm{T}} & v \end{bmatrix}. \tag{3.47}$$

   Its upper left corner is the invertible matrix $\mathbf{A}$, the upper right corner is the translation $\mathbf{t}$, and the lower left corner is the scale $\mathbf{a}^{\mathrm{T}}$. 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 perspective transformation has a total of 8 degrees of freedom, and 3D has a total of 15 degrees of freedom. Perspective transformation is the most general transformation that has been said so far. The transformation from the real world to a camera photo can be seen as a perspective transformation. The reader can imagine what a square tile would look like in a photo: first, it is no longer square. Due to the close part is larger then the far away part, it is not even a parallelogram, but an irregular quadrilateral.

Table 3-1 summarizes the properties of several transformations currently covered. Note that in the "invariance", 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.

Table 3-1: comparison of common transformation properties

Transform Name	Matrix Form	Degrees of Freedom	Invariance
Euclidean	$\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix}$	6	Length, Angle, Volume
Similarity	$\begin{bmatrix} s\mathbf{R} & \mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix}$	7	Volume ratio
Affine	$\begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix}$	12	Parallelism, Volume ratio
Perspective	$\begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{a}^{\mathrm{T}} & v \end{bmatrix}$	15	Plane intersection and tangency

We will introduce later that the transformation from the real world to the camera photo is a perspective 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, let's do some experiments to have a rough impression of these transformations.

3.6 Practice: Eigen Geometry Module

3.6.1 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 3.6: 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 representations
  // 3D rotation matrix directly using Matrix3d or Matrix3f
  Matrix3d rotation_matrix = Matrix3d::Identity();
  // The rotation vector uses AngleAxis, the underlying layer is not
      directly Matrix, but the operation can be treated as a matrix (
      because the operator is overloaded)
  AngleAxisd rotation_vector(M_PI / 4, Vector3d(0, 0, 1)); //Rotate 45
      degrees along the Z axis
  cout.precision(3);
  cout << "rotation matrix = \n " << rotation_vector.matrix() << endl; //
      convert to matrix with matrix()
  // 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() << endl;
  // Or use a rotation matrix
  v_rotated = rotation_matrix * v;
  cout << "(1,0,0) after rotation (by matrix) = " << v_rotated.transpose()
      << endl;

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

  // Euclidean transformation matrix using Eigen::Isometry
  Isometry3d T = Isometry3d::Identity(); // Although called 3d, it is
      essentially a 4*4 matrix
  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
```

```
41   cout << "v tranformed = " << v_transformed.transpose() << endl;
42
43   // For affine and projective transformations, use Eigen::Affine3d and
         Eigen::Projective3d.
44
45   // Quaternion
46   // You can assign AngleAxis directly to quaternions, and vice versa
47   Quaterniond q = Quaterniond(rotation_vector);
48   cout << "quaternion from rotation vector = " << q.coeffs().transpose() <<
         endl;
49   // Note that the order of coeffs is (x, y, z, w), w is the real part, the
         first three are the imaginary part
50   // can also assign a rotation matrix to it
51   q = Quaterniond(rotation_matrix);
52   cout << "quaternion from rotation matrix = " << q.coeffs().transpose() <<
         endl;
53   // Rotate a vector with a quaternion and use overloaded multiplication
54   V_rotated = q * v; // Note that the math is qvq^{-1}
55   cout << "(1,0,0) after rotation = " << v_rotated.transpose() << endl;
56   // expressed by regular vector multiplication, it should be calculated as
         follows
57   cout << "should be equal to " << (q * Quaterniond(0, 1, 0, 0) * q.inverse
         ()).coeffs().transpose() << endl;
58
59   return 0;
60 }
```

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 the double precision as an example, you can simply 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.
- Perspective 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 $\mathbf{v}$ and find that the result is the same. 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 `http://eigen.tuxfamily.org/dox/group__TutorialGeometry.html`.

Note that the **program code has some subtle differences from the mathematical representation**. For example, by operator overloading in C++, quaternions and three-dimensional vectors can directly be multiplied, but mathematically, the vector needs to be converted into a imaginary quaternion like we talked in the last section, and then quaternion multiplication is used for calculation. The same applies to the transformation matrix multiplying with a three-dimensional vector. In general, the usage in the program is more flexible than the mathematical formula.

3.6.2 Coordinate Transformation Example

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

Example 1. The robot No. 1 and the robot No. 2 are located in the world coordinate system. We use the world coordinate system as W , robot coordinate system as R_1 and R_2 . The pose of the robot No. 1 is $\mathbf{q}_1 = [0.35, 0.2, 0.3, 0.1]^{\mathrm{T}}, \mathbf{t}_1 = [0.3, 0.1, 0.1]^{\mathrm{T}}$. The pose of the robot No. 2 is $\mathbf{q}_2 = [-0.5, 0.4, -0.1, 0.2]^{\mathrm{T}}, \mathbf{t}_2 = [-0.1, 0.5, 0.3]^{\mathrm{T}}$. Here $\mathbf{q}$ and $\mathbf{t}$ express $\mathbf{T}_{R_k,W}, k = 1, 2$, which is the world coordinate system to the robot coordinate system. Now, assume that robot No. 1 sees a point in its own coordinate system with coordinates of $\mathbf{p}_{R_1} = [0.5.0, 0.2]^{\mathrm{T}}$, 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 3.7: 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 is $[-0.0309731, 0.73499, 0.296108]^{\mathrm{T}}$, and the calculation process is very simple, just by calculating

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

Note that the quaternion needs to be normalized before use.

3.7 Visualization Demo

3.7.1 Plotting Trajectory

If you are new to the concepts of rotation and translation, you may find that their form looks a little complicated. There are so many representation methods and we need to convert to a preferred one if necessary. Fortunately, although the values of

the rotation and transformation matrix may not be intuitive enough, we can easily draw them in a 3D 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 in a figure. Suppose the trajectory file is stored in a text file called "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, $\mathbf{t}$ is translation, $\mathbf{q}$ is the 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 we just talk about "robot pose", then we can use any one of $\mathbf{T}_{WR}$ or $\mathbf{T}_{RW}$ because they are just the inverse of each other. It means that knowing one of them makes it easy to get the other. If we want to store **robot's trajectory**, then saving $\mathbf{T}_{WR}$ or $\mathbf{T}_{RW}$ doesn't make much difference.

When drawing the trajectory, we should draw the "trajectory" as 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 in the world coordinate system**. Consider the origin of the robot coordinate system, i.e., $\mathbf{O}_R$, then the $\mathbf{O}_W$ at this time is the coordinates of the origin in the world coordinate system:

$$\mathbf{O}_W = \mathbf{T}_{WR}\mathbf{O}_R = \mathbf{t}_{WR}. \tag{3.48}$$

This is exactly the translation part of $\mathbf{T}_{WR}$. So, you can see **where the camera is** directly from $\mathbf{T}_{WR}$. Therefore, in most of the public datasets, the trajectory file stores $\mathbf{T}_{WR}$ instead of $\mathbf{T}_{RW}$.

Finally, we need a library that supports 3D drawing. There are many libraries that support 3D drawing, such as the famous matlab, python matplotlib, OpenGL and so on. In linux, a common library is OpenGL-based Pangolin library *, which provides some drawing operations based on OpenGL. 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 3.8: slambook2/ch3/examples/plotTrajectory.cpp

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

* See https://github.com/stevenlovegrove/Pangolin.

```
17      cout << "cannot find trajectory file at " << trajectory_file << endl;
18      return 1;
19    }
20
21    while (!fin.eof()) {
22      double time, tx, ty, tz, qx, qy, qz, qw;
23      fin >> time >> tx >> ty >> tz >> qx >> qy >> qz >> qw;
24      Isometry3d Twr(Quaterniond(qw, qx, qy, qz));
25      Twr.pretranslate(Vector3d(tx, ty, tz));
26      poses.push_back(Twr);
27    }
28    cout << "read total " << poses.size() << " pose entries" << endl;
29
30    // draw trajectory in pangolin
31    DrawTrajectory(poses);
32    return 0;
33  }
34
35  void DrawTrajectory(vector<Isometry3d, Eigen::aligned_allocator<Isometry3d
        >> poses) {
36    // create pangolin window and plot the trajectory
37    pangolin::CreateWindowAndBind("Trajectory Viewer", 1024, 768);
38    glEnable(GL_DEPTH_TEST);
39    glEnable(GL_BLEND);
40    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
41
42    pangolin::OpenGlRenderState s_cam(
43    pangolin::ProjectionMatrix(1024, 768, 500, 500, 512, 389, 0.1, 1000),
44    pangolin::ModelViewLookAt(0, -0.1, -1.8, 0, 0, 0, 0.0, -1.0, 0.0)
45    );
46
47    pangolin::View &d_cam = pangolin::CreateDisplay()
48    .SetBounds(0.0, 1.0, 0.0, 1.0, -1024.0f / 768.0f)
49    .SetHandler(new pangolin::Handler3D(s_cam));
50
51    while (pangolin::ShouldQuit() == false) {
52      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
53      d_cam.Activate(s_cam);
54      glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
55      glLineWidth(2);
56      for (size_t i = 0; i < poses.size(); i++) {
57        // draw three axes of each pose
58        Vector3d Ow = poses[i].translation();
59        Vector3d Xw = poses[i] * (0.1 * Vector3d(1, 0, 0));
60        Vector3d Yw = poses[i] * (0.1 * Vector3d(0, 1, 0));
61        Vector3d Zw = poses[i] * (0.1 * Vector3d(0, 0, 1));
62        glBegin(GL_LINES);
63        glColor3f(1.0, 0.0, 0.0);
64        glVertex3d (Ow [0], Ow [1], Ow [2]);
65        glVertex3d (Xw [0], Xw [1], Xw [2]);
66        glColor3f(0.0, 1.0, 0.0);
67        glVertex3d (Ow [0], Ow [1], Ow [2]);
68        glVertex3d (Is [0], Is [1], Is [2]);
69        glColor3f(0.0, 0.0, 1.0);
70        glVertex3d (Ow [0], Ow [1], Ow [2]);
71        glVertex3d (Zw [0], Zw [1], Zw [2]);
72        glEnd();
73      }
74      // draw a connection
75      for (size_t i = 0; i < poses.size(); i++) {
76        glColor3f(0.0, 0.0, 0.0);
77        glBegin(GL_LINES);
78        auto p1 = poses[i], p2 = poses[i + 1];
```

```
79          glVertex3d(p1.translation()[0], p1.translation()[1], p1.translation()
                [2]);
80          glVertex3d(p2.translation()[0], p2.translation()[1], p2.translation()
                [2]);
81          glEnd();
82        }
83        pangolin::FinishFrame();
84        usleep(5000);   // sleep 5 ms
85      }
86    }
```

This program demonstrates how to draw a 3D pose in Pangolin. 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 poses with black lines. The result is shown in Figure 3-3.

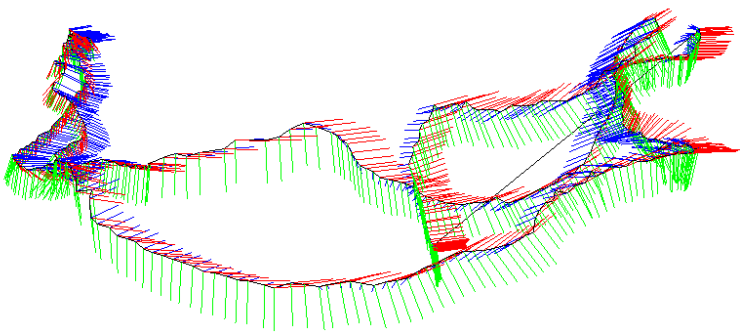

Figure 3-3: Results of pose visualization

3.7.2 Displaying Camera Pose

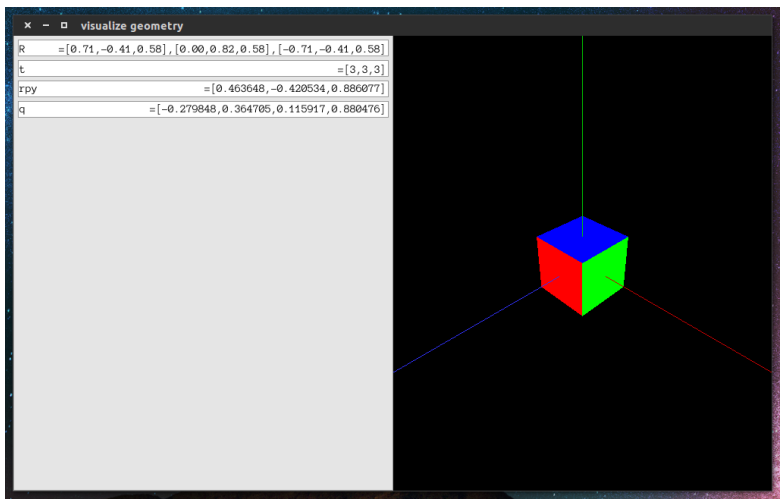

Figure 3-4: Visualization program for rotation matrix, Euler angle, quaternion.

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 Figure 3-4). When the reader uses the mouse to move 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, it is hard to infer the exact rotation from quaternions or matrices. 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. Verify that the rotation matrix is an orthogonal matrix.
2. Prove the Rodrigues formula.
3. Verify that after the quaternion rotates a point, the result is a imaginary quaternion (the real part is zero), so it still corresponds to a three-dimensional space point, see (3.33).
4. Draw a table that summarizes the conversion relationship of the rotation matrix, rotation angle, Euler angle and quaternion.
5. Suppose there is a large Eigen matrix, we want to know the value in the top left 3×3 blocks, and then assign it to $\mathbf{I}_{3\times 3}$. Please implement it in C++.
6. When does a general linear equation $\mathbf{Ax} = \mathbf{b}$ has an unique solution of $\mathbf{x}$? How to solve it numerically? Can you implement it in Eigen?

Chapter 4

Lie Group and Lie Algebra

Main Goal

1. Understand the concept of Lie group, Lie algebra, and their applications of $\mathrm{SO}(3), \mathrm{SE}(3)$ and the corresponding Lie algebras.
2. Understands the meaning of BCH formula.
3. Learn the perturbation model on Lie algebra.
4. Use Sophus to perform operations on Lie algebras.

In the last lecture, we introduced the description of rigid body motion in the three-dimensional world, including the rotation matrix, rotation vector, Euler angle, quaternion and so on. We focus on the representation of rotation, but in SLAM we have to estimate and optimize them in addition to the representation. Because the pose is unknown in SLAM, we need to solve the problem of "**which camera pose best matches the current observation**". A typical way is to build it into an optimization problem, solving the optimal $\mathbf{R}, \mathbf{t}$, and minimizing the error.

As mentioned before, the rotation matrix itself is constrained (orthogonal and the determinant is 1). When used as optimization variables, it introduces additional constraints on matrices that makes optimization difficult. Through the transformation relationship between Lie group and Lie algebra, we are able to turn the pose estimation into an unconstrained optimization problem and simplify the solution. Considering that the reader may not have the basic knowledge of Lie Group and Lie algebra, we will start with the most basic knowledge.

4.1 Basics of Lie Group and Lie Algebra

In the last lecture, we introduced the definition of the rotation matrix and the transformation matrix. At the time, we said that the three-dimensional rotation matrix constitutes **special orthogonal group** SO(3), and the transformation matrix constitutes **special Euclidean group** SE(3). They are written like this:

$$\mathrm{SO}(3) = \{\mathbf{R} \in \mathbb{R}^{3\times 3} | \mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{I}, \det(\mathbf{R}) = 1\}. \tag{4.1}$$

$$\mathrm{SE}(3) = \left\{ \mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \in \mathbb{R}^{4\times 4} | \mathbf{R} \in \mathrm{SO}(3), \mathbf{t} \in \mathbb{R}^3 \right\}. \tag{4.2}$$

However, at the time we did not explain the meaning of **group** in detail. Readers should note that both of the rotation matrix and the transformation matrix **are not closed to addition**. In other words, for any two rotation matrices $\mathbf{R}_1, \mathbf{R}_2$, according to the definition, their addition is no longer a rotation matrix:

$$\mathbf{R}_1 + \mathbf{R}_2 \notin \mathrm{SO}(3), \quad \mathbf{T}_1 + \mathbf{T}_2 \notin \mathrm{SE}(3). \tag{4.3}$$

You can also say that the two matrices do not have a well-defined addition operator, or the matrix addition is not closed in these two sets. We find they have only one closed operation: the multiplication:

$$\mathbf{R}_1\mathbf{R}_2 \in \mathrm{SO}(3), \quad \mathbf{T}_1\mathbf{T}_2 \in \mathrm{SE}(3). \tag{4.4}$$

We know that the matrix multiplication corresponds to the composition of two rotations or transformations. For a set that only has one "well-defined" operation, we call it a **group**.

4.1.1 Group

For the next contents we need to talk about a little bit abstract algebra. I think this is a necessary condition for discussing Lie Group and Lie Algebra, but in fact, except for the students of mathematics and physics, most of the students will not have this knowledge in undergraduate classes. So let's look at some basic concepts first.

A group is an algebraic structure of **a set** plus **an operation**. We denote the set as A and the operation as $\cdot$, then the group can be denoted as $G = (A, \cdot)$. We say G is a **group** if the operation satisfies the following conditions:

1. Closure: $\forall a_1, a_2 \in A,\ a_1 \cdot a_2 \in A$.
2. Combination: $\forall a_1, a_2, a_3 \in A,\ (a_1 \cdot a_2) \cdot a_3 = a_1 \cdot (a_2 \cdot a_3)$.
3. Unit element: $\exists a_0 \in A$, s.t. $\forall a \in A,\ a_0 \cdot a = a \cdot a_0 = a$.
4. Inverse element: $\forall a \in A,\ \exists a^{-1} \in A,\ st\ a \cdot a^{-1} = a_0$.

It is easy to verify that the rotation matrix set with the normal matrix multiplication form a group, and the same for transformation matrix with matrix multiplication, so they can be called as rotation matrix group and transformation matrix group. Other common groups include the addition of integers $(\mathbb{Z}, +)$, the rational numbers with multiplication after removing 0 $(\mathbb{Q}\backslash 0, \cdot)$, etc. Common groups in the matrix are:

- General Linear group GL(n). The invertible matrix of $n \times n$ with matrix multiplication.
- Special Orthogonal Group SO(n). Or the rotation matrix group, where SO(2) and SO(3) is the most common.
- Special Euclidean group SE(n). Or the n dimensional transformation described earlier, such as SE(2) and SE(3).

The group structure guarantees that the operations on the group have very good properties, and the **group theory** is the theory that studies the various structures and properties of the groups. Readers interested in group theory can refer to any of the modern algebra books. **Lie Group** refers to a group with continuous (smooth) properties. Discrete groups like the integer group $\mathbb{Z}$ have no continuous properties, so they are not Lie groups. And obviously, SO(n) and SE(n) are continuous in real space because we can intuitively imagine that a rigid body moving continuously in space, so they are all Lie Groups. Since SO(3) and SE(3) are especially important for camera pose estimation, we mainly discuss these two Lie groups. However, strictly discussing the concepts of "continuous" and "smooth" requires knowledge of analysis and topology, but we are not mathematics books, so only some important conclusions directly related to SLAM are introduced. If the reader is interested in the theoretical nature of Lie Groups, please refer to the special books like [18].

Normally we have two ways to introduce the Lie Groups or Lie Algebras. The first is to directly introduce Lie group and Lie algebra, and then tell the reader that each Lie group corresponds to a Lie algebra, but in this case, the reader will think that Lie algebra seems to be a symbol that jumps out with no reason, and does not know its physical meaning. So, I am going to take a little time to draw the Lie algebra from the rotation matrix, similar to the practice of [19]. Let's start with the simpler SO(3), leading to the Lie algebra $\mathfrak{so}(3)$ above SO(3).

4.1.2 Introduction of the Lie Algebra

Consider an arbitrary rotation matrix $\mathbf{R}$, we know that it satisfies:

$$\mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{I}. \tag{4.5}$$

Now, we say that $\mathbf{R}$ is the rotation of a camera that changes continuously over time, which is a function of time: $\mathbf{R}(t)$. Since it is still a rotation matrix, we have

$$\mathbf{R}(t)\mathbf{R}(t)^{T} = \mathbf{I}.$$

Deriving time on both sides of the equation yields (we use $\dot{\mathbf{R}}$ to represent the derivative of $\mathbf{R}$ on time t, just like many control books):

$$\dot{\mathbf{R}}(t)\mathbf{R}(t)^{\mathrm{T}} + \mathbf{R}(t)\dot{\mathbf{R}}(t)^{\mathrm{T}} = 0.$$

Put the second item to right:

$$\dot{\mathbf{R}}(t)\mathbf{R}(t)^{\mathrm{T}} = -\left(\dot{\mathbf{R}}(t)\mathbf{R}(t)^{\mathrm{T}}\right)^{\mathrm{T}}. \tag{4.6}$$

It can be seen that $\dot{\mathbf{R}}(t)\mathbf{R}(t)^{\mathrm{T}}$ is a **skew-symmetric** matrix. Recall that we introduced the $\wedge$ symbol when we mention about the cross product in (3.3), which

turns a vector into an skew-symmetric matrix. Similarly, for any skew-symmetric matrix, we can also find a unique vector corresponding to it. Let this operation be represented by the symbol $^\vee$:

$$\mathbf{a}^\wedge = \mathbf{A} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}, \quad \mathbf{A}^\vee = \mathbf{a}. \tag{4.7}$$

So, since $\dot{\mathbf{R}}(t)\mathbf{R}(t)^{\mathrm{T}}$ is an skew-symmetric matrix, we can find a three-dimensional vector $\boldsymbol{\phi}(t) \in \mathbb{R}^3$ corresponds to it:

$$\dot{\mathbf{R}}(t)\mathbf{R}(t)^{\mathrm{T}} = \boldsymbol{\phi}(t)^\wedge.$$

Right multiply with $\mathbf{R}(t)$ on both sides, since $\mathbf{R}$ is an orthogonal matrix, we have:

$$\dot{\mathbf{R}}(t) = \boldsymbol{\phi}(t)^\wedge \mathbf{R}(t) = \begin{bmatrix} 0 & -\phi_3 & \phi_2 \\ \phi_3 & 0 & -\phi_1 \\ -\phi_2 & \phi_1 & 0 \end{bmatrix} \mathbf{R}(t). \tag{4.8}$$

It can be seen that we can take the time derivative of a ration matrix just by multiplying a $\boldsymbol{\phi}^\wedge(t)$ matrix on the left. Consider at time $t_0 = 0$, and the rotation matrix is $\mathbf{R}(0) = \mathbf{I}$. According to the derivative definition, $\mathbf{R}(t)$ can be used to perform a first-order Taylor expansion around $t = 0$:

$$\begin{aligned} \mathbf{R}(t) &\approx \mathbf{R}(t_0) + \dot{\mathbf{R}}(t_0)(t - t_0) \\ &= \mathbf{I} + \boldsymbol{\phi}(t_0)^\wedge (t). \end{aligned} \tag{4.9}$$

We see that $\boldsymbol{\phi}$ reflects the derivative of $\mathbf{R}$, so it is called the **Tangent Space** near the origin of SO(3). Also, if time t is close to t_0, we assume $\boldsymbol{\phi}(t)$ to close to be a constant $\boldsymbol{\phi}(t_0) = \boldsymbol{\phi}_0$. Then according to the formula (4.8), we have:

$$\dot{\mathbf{R}}(t) = \boldsymbol{\phi}(t_0)^\wedge \mathbf{R}(t) \approx \boldsymbol{\phi}_0^\wedge \mathbf{R}(t).$$

The above formula is a differential equation for $\mathbf{R}$, and with the initial value $\mathbf{R}(0) = \mathbf{I}$, we have solution like:

$$\mathbf{R}(t) = \exp(\boldsymbol{\phi}_0^\wedge t). \tag{4.10}$$

The reader can verify that the above equation holds for both the differential equation and the initial value. This means that around $t = 0$, the rotation matrix can be calculated from $\exp(\boldsymbol{\phi}_0^\wedge t)$*. We see that the rotation matrix $\mathbf{R}$ is associated with another skew-symmetric matrix $\boldsymbol{\phi}_0^\wedge t$ through an exponential relationship. But what is the exponential of a matrix? Here we have two questions that need to be clarified:

1. Given $\mathbf{R}$ at a certain moment, we can find a $\boldsymbol{\phi}$ that describes the local derivative relationship of $\mathbf{R}$. How are they correlated with each other? We will say that $\boldsymbol{\phi}$ corresponds to the Lie algebra $\mathfrak{so}(3)$ on SO(3);
2. Second, when a vector $\boldsymbol{\phi}$ is given, how is $\exp(\boldsymbol{\phi}^\wedge)$ calculated? Conversely, given $\mathbf{R}$, is there an opposite operation to calculate $\boldsymbol{\phi}$? In fact, this is the exponential/logarithmic mapping between Lie group and Lie algebra.

Let's solve these two problems below.

* At this point we have not explained what this exp means and how it works. We will talk about its definition and calculation process right after this section.

4.1.3 The Definition of Lie Algebra

Now let's give the strict definition of Lie Algebra. Each Lie group has a Lie algebra corresponding to it. Lie algebra describes the local structure of the Lie group around it origin point, or in other words, is the tangent space. The general definition of Lie algebra is listed as follows:

A Lie algebra consists of a set $\mathbb{V}$, a scalar field $\mathbb{F}$, and a binary operation $[,]$. If they satisfy the following properties, then $(\mathbb{V}, \mathbb{F}, [,])$ is a Lie algebra, denoted as $\mathfrak{g}$.

1. Closure. $\forall \mathbf{X}, \mathbf{Y} \in \mathbb{V}, [\mathbf{X}, \mathbf{Y}] \in \mathbb{V}$.
2. Bilinear. $\forall \mathbf{X}, \mathbf{Y}, \mathbf{Z} \in \mathbb{V}, a, b \in \mathbb{F}$, we have:

$$[a\mathbf{X} + b\mathbf{Y}, \mathbf{Z}] = a[\mathbf{X}, \mathbf{Z}] + b[\mathbf{Y}, \mathbf{Z}], \quad [\mathbf{Z}, a\mathbf{X} + b\mathbf{Y}] = a[\mathbf{Z}, \mathbf{X}] + b[\mathbf{Z}, \mathbf{Y}].$$

3. Reflexive * $\forall \mathbf{X} \in \mathbb{V}, [\mathbf{X}, \mathbf{X}] = \mathbf{0}$.
4. Jacobi equation. $\forall \mathbf{X}, \mathbf{Y}, \mathbf{Z} \in \mathbb{V}, [\mathbf{X}, [\mathbf{Y}, \mathbf{Z}]] + [\mathbf{Z}, [\mathbf{X}, \mathbf{Y}]] + [\mathbf{Y}, [\mathbf{Z}, \mathbf{X}]] = \mathbf{0}$.

The binary operation $[,]$ is called **Lie brackets**. On the first glance, we require a lot of properties about the operator of Lie bracket. Compared to the simpler binary operations in the group, the Lie bracket expresses the difference between the two elements. It does not require a combination law, but requires the element and itself to be zero after the brackets. As an example, the cross product $\times$ defined on the 3D vector $\mathbb{R}^3$ is a kind of Lie bracket, so $\mathfrak{g} = (\mathbb{R}^3, \mathbb{R}, \times)$ constitutes a Lie algebra. The reader can try to substitute the cross product into the above four properties to verify the above conclusion.

4.1.4 Lie Algebra $\mathfrak{so}(3)$

The previously mentioned $\boldsymbol{\phi}$ is actually a kind of Lie algebra. The Lie algebra corresponding to SO(3) is a vector defined on $\mathbb{R}^3$, which we will denote as $\boldsymbol{\phi}$. According to the previous derivation, each $\boldsymbol{\phi}$ can generate an skew-symmetric matrix:

$$\boldsymbol{\Phi} = \boldsymbol{\phi}^\wedge = \begin{bmatrix} 0 & -\phi_3 & \phi_2 \\ \phi_3 & 0 & -\phi_1 \\ -\phi_2 & \phi_1 & 0 \end{bmatrix} \in \mathbb{R}^{3\times 3}. \tag{4.11}$$

Under this definition, the two vectors $\boldsymbol{\phi}_1, \boldsymbol{\phi}_2$'s Lie brackets are:

$$[\boldsymbol{\phi}_1, \boldsymbol{\phi}_2] = (\Phi_1\Phi_2 - \Phi_2\Phi_1)^\vee. \tag{4.12}$$

The reader can verify that the Lie brackets under this definition satisfy the above properties. Since the vector $\boldsymbol{\phi}$ is one-to-one with the skew-symmetric matrix, we say the elements of $\mathfrak{so}(3)$ are three-dimensional vectors or three-dimensional skew-symmetric matrices, without any ambiguity:

$$\mathfrak{so}(3) = \left\{ \boldsymbol{\phi} \in \mathbb{R}^3 \text{ or } \boldsymbol{\Phi} = \boldsymbol{\phi}^\wedge \in \mathbb{R}^{3\times 3} \right\}. \tag{4.13}$$

Some books also use the symbol $\widehat{\boldsymbol{\phi}}$ to represent $\boldsymbol{\phi}^\wedge$, but the meaning is the same. At this point, we have made it clear about the contents of $\mathfrak{so}(3)$. They are just a set

* Reflexive means that an element operates with itself results in zero.

of **3D vectors** which can be used to express the derivative of the rotation matrix. Its relationship to SO(3) is given by the exponential map:

$$\mathbf{R} = \exp(\boldsymbol{\phi}^\wedge). \tag{4.14}$$

The exponential map will be introduced later. Since we have introduced $\mathfrak{so}(3)$, we will first look at the corresponding Lie algebra on SE(3).

4.1.5 Lie Algebra $\mathfrak{se}(3)$

For SE(3), it also has a corresponding Lie algebra $\mathfrak{se}(3)$. To save space, I won't start by taking time derivatives. Similar to $\mathfrak{so}(3)$, $\mathfrak{se}(3)$ is located in the $\mathbb{R}^6$ space:

$$\mathfrak{se}(3) = \left\{ \boldsymbol{\xi} = \begin{bmatrix} \boldsymbol{\rho} \\ \boldsymbol{\phi} \end{bmatrix} \in \mathbb{R}^6, \boldsymbol{\rho} \in \mathbb{R}^3, \boldsymbol{\phi} \in \mathfrak{so}(3), \boldsymbol{\xi}^\wedge = \begin{bmatrix} \boldsymbol{\phi}^\wedge & \boldsymbol{\rho} \\ \mathbf{0}^{\mathrm{T}} & 0 \end{bmatrix} \in \mathbb{R}^{4\times 4} \right\}. \tag{4.15}$$

We write each $\mathfrak{se}(3)$ element as $\boldsymbol{\xi}$, which is a six-dimensional vector. The first three dimensions are "translation part" (but keep in mind that the meaning is **different** from the translation in the transformation matrix, we will see later), which is denoted as $\boldsymbol{\rho}$; after the three-dimensional rotation, there is a $\boldsymbol{\phi}$, which is essentially a $\mathfrak{so}(3)$ element*. At the same time, we extended the meaning of the $^\wedge$ symbol. In $\mathfrak{se}(3)$, a six-dimensional vector is also converted to a four-dimensional matrix using the $^\wedge$ symbol, but no longer a skew-symmetric one:

$$\boldsymbol{\xi}^\wedge = \begin{bmatrix} \boldsymbol{\phi}^\wedge & \boldsymbol{\rho} \\ \mathbf{0}^{\mathrm{T}} & 0 \end{bmatrix} \in \mathbb{R}^{4\times 4}. \tag{4.16}$$

We still use the $^\wedge$ and $^\vee$ symbols to refer to the relationship from "vector to matrix" and "matrix to vector" to maintain consistency with $\mathfrak{so}(3)$. They are still one-to-one correspondence. The readers can simply take $\mathfrak{se}(3)$ as a "vector consisting of a translation plus a $\mathfrak{so}(3)$ element" (although $\boldsymbol{\rho}$ is not the direct translation). Similarly, the Lie algebra $\mathfrak{se}(3)$ also has a Lie bracket similar to $\mathfrak{so}(3)$:

$$[\boldsymbol{\xi}_1, \boldsymbol{\xi}_2] = \left(\boldsymbol{\xi}_1^\wedge \boldsymbol{\xi}_2^\wedge - \boldsymbol{\xi}_2^\wedge \boldsymbol{\xi}_1^\wedge\right)^\vee. \tag{4.17}$$

The reader can verify that it satisfies the definition of Lie algebra (I'll leave it as an exercise). So far we have seen two important Lie algebras $\mathfrak{so}(3)$ and $\mathfrak{se}(3)$.

4.2 Exponential and Logarithmic Mapping

4.2.1 Exponential Map of SO(3)

Now consider the second question: How to calculate $\exp(\boldsymbol{\phi}^\wedge)$? Obviously it is an exponential map of a matrix. Again, we will first discuss the exponential mapping of $\mathfrak{so}(3)$ and then the case of $\mathfrak{se}(3)$.

The exponential of an arbitrary matrix can be written as a Taylor expansion if it is converged, whose result is still a matrix:

$$\exp(\mathbf{A}) = \sum_{n=0}^{\infty} \frac{1}{n!} \mathbf{A}^n. \tag{4.18}$$

* Please note that in some books the authors may put the rotation in front and the translation in the back, which has no significant difference.

Similarly, for any element in $\boldsymbol{\phi} \in \mathfrak{so}(3)$„ we can also define its exponential map in this way:

$$\exp(\boldsymbol{\phi}^\wedge) = \sum_{n=0}^{\infty} \frac{1}{n!} (\boldsymbol{\phi}^\wedge)^n. \tag{4.19}$$

But this definition cannot be calculated directly because we don't want to calculate the infinite power of a matrix. Below we derive a convenient way to calculate the exponential mapping. Since $\boldsymbol{\phi}$ is a three-dimensional vector, we can define its length and direction, denoted as θ and $\mathbf{a}$, respectively. So we have $\boldsymbol{\phi} = \theta\mathbf{a}$, where $\mathbf{a}$ is a unit-length direction vector, i.e., $\|\mathbf{a}\| = 1$. First, for such a unit-length vector $\mathbf{a}$, there are two properties:

$$\mathbf{a}^\wedge \mathbf{a}^\wedge = \begin{bmatrix} -a_2^2 - a_3^2 & a_1 a_2 & a_1 a_3 \\ a_1 a_2 & -a_1^2 - a_3^2 & a_2 a_3 \\ a_1 a_3 & a_2 a_3 & -a_1^2 - a_2^2 \end{bmatrix} = \mathbf{a}\mathbf{a}^{\mathrm{T}} - \mathbf{I}, \tag{4.20}$$

as well as

$$\mathbf{a}^\wedge \mathbf{a}^\wedge \mathbf{a}^\wedge = \mathbf{a}^\wedge (\mathbf{a}\mathbf{a}^{\mathrm{T}} - \mathbf{I}) = -\mathbf{a}^\wedge. \tag{4.21}$$

These two formulas provide a way to handle the high-order $\mathbf{a}^\wedge$ items. Now we can write the exponential map as:

$$\begin{aligned}
\exp(\boldsymbol{\phi}^\wedge) = \exp(\theta\mathbf{a}^\wedge) &= \sum_{n=0}^{\infty} \frac{1}{n!} (\theta\mathbf{a}^\wedge)^n \\
&= \mathbf{I} + \theta\mathbf{a}^\wedge + \frac{1}{2!}\theta^2 \mathbf{a}^\wedge \ Bma^\wedge + \frac{1}{3!}\theta^3 \mathbf{a}^\wedge \mathbf{a}^\wedge \mathbf{a}^\wedge + \frac{1}{4!}\theta^4 (\mathbf{a}^\wedge)^4 + ... \\
&= \mathbf{a}\mathbf{a}^{\mathrm{T}} - \mathbf{a}^\wedge \mathbf{a}^\wedge + \theta\mathbf{a}^\wedge + \frac{1}{2!}\theta^2 \mathbf{a}^\wedge \mathbf{a}^\wedge - \frac{1}{3!}\theta^3 \mathbf{a}^\wedge - \frac{1}{4!}\theta^4 (\mathbf{a}^\wedge)^2 + ... \\
&= \mathbf{a}\mathbf{a}^{\mathrm{T}} + \underbrace{\left(\theta - \frac{1}{3!}\theta^3 + \ Frac15!\theta^5 - ...\right)}_{\sin\theta} \mathbf{a}^\wedge - \underbrace{\left(1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - ...\right)}_{\cos\theta} \mathbf{a}^\wedge \mathbf{a}^\wedge \\
&= \mathbf{a}^\wedge \mathbf{a}^\wedge + \mathbf{I} + \sin\theta \mathbf{a}^\wedge - \cos\theta \ Bma^\wedge \mathbf{a}^\wedge \\
&= (1 - \cos\theta)\mathbf{a}^\wedge \mathbf{a}^\wedge + \mathbf{I} + \sin\theta \mathbf{a}^\wedge \\
&= \cos\theta \mathbf{I} + (1 - \cos\theta)\mathbf{a}\mathbf{a}^{\mathrm{T}} + \sin\theta \mathbf{a}^\wedge.
\end{aligned}$$

Finally, we got a very familiar equation:

$$\exp(\theta\mathbf{a}^\wedge) = \cos\theta \mathbf{I} + (1 - \cos\theta)\mathbf{a}\mathbf{a}^{\mathrm{T}} + \sin\theta \mathbf{a}^\wedge. \tag{4.22}$$

Recall the previous lesson, this equation is exactly the same as the Rodriguez formula, i.e. equation (3.15). This shows that $\mathfrak{so}(3)$ is actually the **rotation vector**, and the exponential map is the just the Rodriguez formula. Through them, we map any vector in $\mathfrak{so}(3)$ to a rotation matrix in SO(3). Conversely, if we define a logarithmic map, we can also map the elements in SO(3) to $\mathfrak{so}(3)$:

$$\boldsymbol{\phi} = \ln(\mathbf{R})^\vee = \left(\sum_{n=0}^{\infty} \frac{(-1)^n}{n+1} (\mathbf{R} - \mathbf{I})^{n+1} \right)^\vee. \tag{4.23}$$

Just like the exponential mapping, we don't have to use Taylor to expand the logarithmic mapping. In Lecture 3, we have already introduced how to calculate the

corresponding Lie algebra according to the rotation matrix, that is, using the formula (3.17), and use the properties of the trace to solve the rotation angle and the rotation axis separately, which is more convenient.

Now, we've introduced the calculation method of exponential mapping. Readers may ask, what is the property of the exponential mapping? Can I find a unique $\boldsymbol{\phi}$ for any $\mathbf{R}$? Unfortunately, the exponential map is just a surjective map, not injective. This means that for each element in SO(3) we can find a $\mathfrak{so}(3)$ element corresponding to it; however, there may be multiple $\mathfrak{so}(3)$ elements corresponding to the same SO(3) element. At least for the rotation angle θ, we know that rotating multiple 360° will give the same rotation - it has periodicity. However, if we fix the rotation angle between $\pm\pi$, then the Lie group and the Lie algebra elements are one-to-one correspondence.

The conclusion of SO(3) and $\mathfrak{so}(3)$ seems to be in our expectation. It is very similar to the rotation vector we talked about earlier, and the exponential mapping is the Rodrigues formula. The derivative of the rotation matrix can be specified by the rotation vector, which guides how to perform calculus operations in the rotation matrix.

4.2.2 Exponential Map of SE(3)

The exponential map on $\mathfrak{se}(3)$ is described below. In order to save space, we no longer deduct the exponential mapping in detail like $\mathfrak{so}(3)$. The exponential mapping on $\mathfrak{se}(3)$ is as follows:

$$\exp\left(\boldsymbol{\xi}^{\wedge}\right) = \begin{bmatrix} \sum_{n=0}^{\infty} \frac{1}{n!}(\boldsymbol{\phi}^{\wedge})^n & \sum_{n=0}^{\infty} \frac{1}{(n+1)!}(\boldsymbol{\phi}^{\wedge})^n \boldsymbol{\rho} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \tag{4.24}$$

$$\triangleq \begin{bmatrix} \mathbf{R} & \mathbf{J}\rho \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} = \mathbf{T}. \tag{4.25}$$

With a little patience, you can derive the Taylor expansion from the practice of $\mathfrak{so}(3)$. Let $\boldsymbol{\phi} = \theta\mathbf{a}$, where $\mathbf{a}$ is the unit vector, then:

$$\begin{aligned} \sum_{n=0}^{\infty} \frac{1}{(n+1)!}(\boldsymbol{\phi}^{\wedge})^n &= \mathbf{I} + \frac{1}{2!}\theta\mathbf{a}^{\wedge} + \frac{1}{3!}\theta^2(\mathbf{a}^{\wedge})^2 + \frac{1}{4!}\theta^3(\mathbf{a}^{\wedge})^3 + \frac{1}{5!}\theta^4(\mathbf{a}^{\wedge})^4 \cdots \\ &= \frac{1}{\theta}\left(\frac{1}{2!}\theta^2 - \frac{1}{4!}\theta^4 + \cdots\right)(\mathbf{a}^{\wedge}) + \frac{1}{\theta}\left(\frac{1}{3!}\theta^3 - \frac{1}{5}\theta^5 + \cdots\right)(\mathbf{a}^{\wedge})^2 + \mathbf{I} \\ &= \frac{1}{\theta}(1-\cos\theta)(\mathbf{a}^{\wedge}) + \frac{\theta - \sin\theta}{\theta}\left(\mathbf{a}\mathbf{a}^{T} - \mathbf{I}\right) + \mathbf{I} \\ &= \frac{\sin\theta}{\theta}\mathbf{I} + \left(1 - \frac{\sin\theta}{\theta}\right)\mathbf{a}\mathbf{a}^{T} + \frac{1-\cos\theta}{\theta}\mathbf{a}^{\wedge} \triangleq \mathbf{J}. \end{aligned} \tag{4.26}$$

From the results, we can see the $\mathbf{R}$ in the upper left corner of the exponential map of $\boldsymbol{\xi}$ is just the well-known SO(3), which means the rotation part of $\boldsymbol{\xi}$ is just the rotation part in $\mathfrak{so}(3)$. The $\mathbf{J}$ in the upper right corner is given by the above derivation:

$$\mathbf{J} = \frac{\sin\theta}{\theta}\mathbf{I} + \left(1 - \frac{\sin\theta}{\theta}\right)\mathbf{a}\mathbf{a}^{\mathrm{T}} + \frac{1-\cos\theta}{\theta}\mathbf{a}^{\wedge}. \tag{4.27}$$

This formula is somewhat similar to the Rodrigues formula, but not exactly the same. We see that after passing the exponential map, the translation part is

multiplied by a linear jacobian matrix $\mathbf{J}$. Please pay attention to the $\mathbf{J}$ here, as it will be used later.

Similarly, although we can also derive the logarithmic mapping analytically, there is a more trouble-free way to find the corresponding vector on $\mathfrak{so}(3)$ according to the transformation matrix $\mathbf{T}$: from the upper left corner $\mathbf{R}$ we can calculate the rotation vector, while $\mathbf{t}$ the upper right corner satisfies:

$$\mathbf{t} = \mathbf{J}\boldsymbol{\rho}. \tag{4.28}$$

Since $\mathbf{J}$ can be obtained from $\boldsymbol{\phi}$, $\boldsymbol{\rho}$ can also be solved by this linear equation. Now, we have clarified the definition of Lie Group and Lie algebra and their mutual conversion relationship, as summarized in Figure 4-1 . If the reader doesn't understand everything, please go back to a few pages to look the formula derivation again.

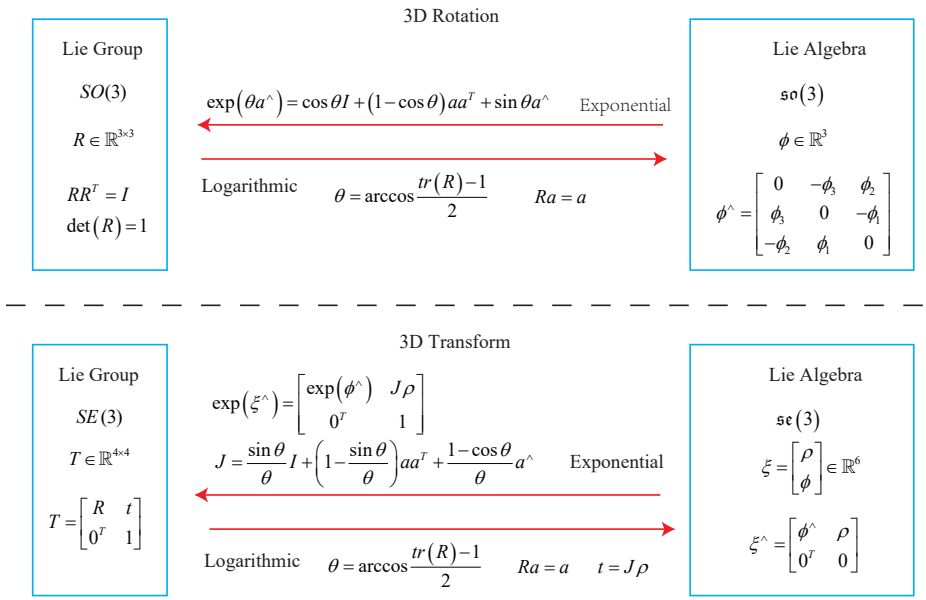

Figure 4-1: The correspondence between $\mathrm{SO}(3), \mathrm{SE}(3), \mathfrak{so}(3), \mathfrak{se}(3)$.

4.3 Lie Algebra Derivation and Perturbation Model

4.3.1 BCH Formula and its Approximation

A major motivation for using Lie algebra is to do optimization, and the derivative is a very necessary information in the optimization process (we will talk about it in detail in Lecture 6). Let's consider a problem below. Although we have already understood the relationship between Lie group and Lie algebra on SO(3) and SE(3), but what happens in $\mathfrak{so}(3)$ when two matrix are multiplied in SO(3)? Conversely, when we add two vectors in $\mathfrak{so}(3)$, does SO(3) correspond to the product of the two matrices? If we write it out, it should be:

$$\exp\left(\boldsymbol{\phi}_1^\wedge\right)\exp\left(\boldsymbol{\phi}_2^\wedge\right) = \exp\left(\left(\boldsymbol{\phi}_1 + \boldsymbol{\phi}_2\right)^\wedge\right)?$$

If $\boldsymbol{\phi}_1, \boldsymbol{\phi}_2$ are scalars, then obviously this is true; but here we calculate the exponential function of **matrix** instead of a scalar. In other words, we are studying whether the following formula holds:

$$\ln\left(\exp\left(\mathbf{A}\right)\exp\left(\mathbf{B}\right)\right) = \mathbf{A}+\mathbf{B}\ ?$$

for matrices. Unfortunately, this formula is not true in the matrix. The complete form of the product is given by the Baker-Campbell-Hausdorff formula (BCH formula)*. Due to the complexity of its complete form, we only give the first few items of its expansion:

$$\ln\left(\exp\left(\mathbf{A}\right)\exp\left(\mathbf{B}\right)\right) = \mathbf{A}+\mathbf{B}+\frac{1}{2}\left[\mathbf{A},\mathbf{B}\right]+\frac{1}{12}\left[\mathbf{A},\left[\mathbf{A},\mathbf{B}\right]\right]-\frac{1}{12}\left[\mathbf{B},\left[\mathbf{A},\mathbf{B}\right]\right]+\cdots \tag{4.29}$$

Where [] is the Lie brackets. The BCH formula tells us that how to deal with the product of two matrices: they produce some extra Lie brackets compared with the scalar form. In particular, consider the case of SO(3), the $\ln\left(\exp\left(\boldsymbol{\phi}_1^\wedge\right)\exp\left(\boldsymbol{\phi}_2^\wedge\right)\right)^\vee$, when $\boldsymbol{\phi}_\mathbf{1}$ or $\boldsymbol{\phi}_\mathbf{2}$ is small, small items with more than quadratic can be simply ignored. At this time, BCH has a linear approximation†:

$$\ln\left(\exp\left(\boldsymbol{\phi}_1^\wedge\right)\exp\left(\boldsymbol{\phi}_2^\wedge\right)\right)^\vee \approx \begin{cases} \mathbf{J}_l(\boldsymbol{\phi}_2)^{-1}\boldsymbol{\phi}_1+\boldsymbol{\phi}_2 & \text{when } \boldsymbol{\phi}_1 \text{ is a small amount,} \\ \mathbf{J}_r(\boldsymbol{\phi}_1)^{-1}\boldsymbol{\phi}_2+\boldsymbol{\phi}_1 & \text{when } \boldsymbol{\phi}_2 \text{ is a small amount.} \end{cases} \tag{4.30}$$

Take the first approximation as an example. This formula tells us to left multiply a tiny rotation matrix $\mathbf{R}_1$ on a rotation matrix $\mathbf{R}_2$ (whose Lie algebra is $\boldsymbol{\phi}_1$ and $\boldsymbol{\phi}_2$, respectively), in $\mathfrak{so}(3)$ it can be approximated by adding a $\mathbf{J}_l\left(\boldsymbol{\phi}_2\right)^{-1}\boldsymbol{\phi}_1$ to the original Lie algebra $\boldsymbol{\phi}_2$. Similarly, the second approximation describes the case where $\mathbf{R}_1$ is right multiplied by a small rotation. Therefore, under the BCH approximation, the Lie algebra is divided into a left-multiplying approximation and a right-multiplying approximation. In daily usage, we must pay attention to whether the left model or the right model is used. This book takes the left multiplication as an example. The jacobian in our left model $\mathbf{J}_l$ is actually the content of the form (4.27) :

$$\mathbf{J}_l = \mathbf{J} = \frac{\sin\theta}{\theta}\mathbf{I}+\left(1-\frac{\sin\theta}{\theta}\right)\mathbf{a}\mathbf{a}^{\mathrm{T}}+\frac{1-\cos\theta}{\theta}\mathbf{a}^\wedge. \tag{4.31}$$

Its inverse is:

$$\mathbf{J}_l^{-1} = \frac{\theta}{2}\cot\frac{\theta}{2}\mathbf{I}+\left(1-\frac{\theta}{2}\cot\frac{\theta}{2}\right)\mathbf{a}\mathbf{a}^{\mathrm{T}}-\frac{\theta}{2}\mathbf{a}^\wedge. \tag{4.32}$$

if θ is not zero (in that case we take both $\mathbf{J}_l$ and its inverse as identity). To get the right jacobian we only need to take a negative sign for the argument:

$$\mathbf{J}_r(\boldsymbol{\phi}) = \mathbf{J}_l(-\boldsymbol{\phi}). \tag{4.33}$$

By this way, we've made it clear about the relationship between Lie group multiplication and Lie algebra addition.

For the convenience of the reader, we restate the meaning of the BCH approximation. Suppose we have a rotation $\mathbf{R}$, the corresponding Lie algebra is $\boldsymbol{\phi}$. We give

* See `https://en.wikipedia.org/wiki/Baker-Campbell-Hausdorff_formula`.

† We are not going to do the detailed derivation of BCH approximation, see [20] if you are interested.

it a small perturbation to the left, denoted as $\Delta\mathbf{R}$, and so that the corresponding Lie algebra is $\Delta\boldsymbol{\phi}$. Then, on Lie group, the result is $\Delta\mathbf{R}\cdot\mathbf{R}$, and on the Lie algebra, according to the BCH approximation, it is $\mathbf{J}_l^{-1}(\boldsymbol{\phi})\Delta\boldsymbol{\phi}+\boldsymbol{\phi}$. Put them together, we can simply write:

$$\exp\left(\Delta\boldsymbol{\phi}^{\wedge}\right)\exp\left(\boldsymbol{\phi}^{\wedge}\right)=\exp\left(\left(\boldsymbol{\phi}+\mathbf{J}_l^{-1}\left(\boldsymbol{\phi}\right)\Delta\boldsymbol{\phi}\right)^{\wedge}\right). \tag{4.34}$$

Conversely, if we do addition on Lie algebra by adding $\boldsymbol{\phi}$ with $\Delta\boldsymbol{\phi}$, we can approximate the multiplication on the Lie group as:

$$\exp\left(\left(\boldsymbol{\phi}+\Delta\boldsymbol{\phi}\right)^{\wedge}\right)=\exp\left(\left(\mathbf{J}_l\Delta\boldsymbol{\phi}\right)^{\wedge}\right)\exp\left(\boldsymbol{\phi}^{\wedge}\right)=\exp\left(\boldsymbol{\phi}^{\wedge}\right)\exp\left(\left(\mathbf{J}_r\Delta\boldsymbol{\phi}\right)^{\wedge}\right). \tag{4.35}$$

This provides a theoretical basis for calculus on Lie algebra. Similarly, for SE(3), there is a similar BCH approximation:

$$\exp\left(\Delta\boldsymbol{\xi}^{\wedge}\right)\exp\left(\boldsymbol{\xi}^{\wedge}\right)\approx\exp\left(\left(\boldsymbol{\mathcal{J}}_l^{-1}\Delta\boldsymbol{\xi}+\boldsymbol{\xi}\right)^{\wedge}\right), \tag{4.36}$$

$$\exp\left(\boldsymbol{\xi}^{\wedge}\right)\exp\left(\Delta\boldsymbol{\xi}^{\wedge}\right)\approx\exp\left(\left(\boldsymbol{\mathcal{J}}_r^{-1}\Delta\boldsymbol{\xi}+\boldsymbol{\xi}\right)^{\wedge}\right). \tag{4.37}$$

Here the $\boldsymbol{\mathcal{J}}_l$ and $\boldsymbol{\mathcal{J}}_r$ are a more complicated 6×6 matrices. Readers can find its detailed contents in [20]. Since we did not use these two jacobians matrices in the calculation (we will see in the next subsection), the exact form is omitted here.

4.3.2 Derivative on SO(3)

Now let's talk about how to compute the derivation if our target function is related to a rotation or a transform, which has a very strong practical meaning since we usually have these functions to optimize in solving SLAM problem. Assume we want to estimate a pose described by SO(3) or SE(3) elements. Our robot observes a point with world coordinate $\mathbf{p}$ and generates an observation data $\mathbf{z}$, which can be written as:

$$\mathbf{z}=\mathbf{T}\mathbf{p}+\mathbf{w}, \tag{4.38}$$

where $\mathbf{w}$ is the noise (and is unknown). Because of the noise, the real observed data is not absolutely same with the one we computed from the observation model, so we can calculate the error of predicted observation with the real one:

$$\mathbf{e}=\mathbf{z}-\mathbf{T}\mathbf{p}. \tag{4.39}$$

Suppose we have N points in total, then we find a best $\mathbf{T}$ to make the error minimized:

$$\min_{\mathbf{T}}J(\mathbf{T})=\sum_{i=1}^{N}\left\|\mathbf{z}_i-\mathbf{T}\mathbf{p}_i\right\|_2^2. \tag{4.40}$$

To solve such an optimized problem (which is a least square), we need to calculate the derivative of J by $\mathbf{T}$. We leave the least square problem to the next section, here we just want to make it cleat that we normally have some functions that have rotations or transforms as their variables. We have to adjust those rotations or transforms to find a better/best estimation. But, as we mentioned before, since SO(3) and SE(3) does not have a well-defined addition (they are just groups), so the derivatives cannot be defined in their common form. If we treat the $\mathbf{R}$ or $\mathbf{T}$ as common matrices, then we have to introduce the constraints into our optimization.

However, from the perspective of Lie algebra, since it consists of vectors, it has a good addition operation. Therefore, there are two ways to solve the problem of derivation using Lie algebra:

1. Assume we add a infinitesimal amount on Lie algebra, then compute the change of the object function.
2. Assume we multiply a infinitesimal perturbation on the Lie group **left multiplication** or **right multiplication**, and use Lie algebra to describe the perturbation, then compute the derivative on this perturbation. This is called as left perturbation or right perturbation model.

The first method corresponds to the normal derivation model of the Lie algebra, and the second corresponds to the perturbation model. Let's discuss the similarities and differences between these two approaches.

4.3.3 Derivative Model

First consider the situation on SO(3). Suppose we rotate a space point $\mathbf{p}$ and get $\mathbf{Rp}$. Now, to calculate derivative of the point coordinates by the rotation, we informally write it as*:

$$\frac{\partial (\mathbf{Rp})}{\partial \mathbf{R}}.$$

Since SO(3) has no addition, it cannot be calculated by the common derivative definition. Let the Lie algebra corresponding to $\mathbf{R}$ be $\boldsymbol{\phi}$, and we will calculate instead of the common derivative:†:

$$\frac{\partial \left(\exp\left(\boldsymbol{\phi}^{\wedge}\right)\mathbf{p}\right)}{\partial \boldsymbol{\phi}}.$$

According to the definition of the derivative, we have:

$$\begin{aligned}
\frac{\partial \left(\exp\left(\boldsymbol{\phi}^{\wedge}\right)\mathbf{p}\right)}{\partial \boldsymbol{\phi}} &= \lim_{\delta\boldsymbol{\phi}\to\mathbf{0}} \frac{\exp\left(\left(\boldsymbol{\phi}+\delta\boldsymbol{\phi}\right)^{\wedge}\right)\mathbf{p} - \exp\left(\boldsymbol{\phi}^{\wedge}\right)\mathbf{p}}{\delta\boldsymbol{\phi}} \\
&= \lim_{\delta\boldsymbol{\phi}\to\mathbf{0}} \frac{\exp\left(\left(\mathbf{J}_l\delta\boldsymbol{\phi}\right)^{\wedge}\right)\exp\left(\boldsymbol{\phi}^{\wedge}\right)\mathbf{p} - \exp\left(\boldsymbol{\phi}^{\wedge}\right)\mathbf{p}}{\delta\boldsymbol{\phi}} \\
&= \lim_{\delta\boldsymbol{\phi}\to\mathbf{0}} \frac{\left(\mathbf{I}+\left(\mathbf{J}_l\delta\boldsymbol{\phi}\right)^{\wedge}\right)\exp\left(\boldsymbol{\phi}^{\wedge}\right)\mathbf{p} - \exp\left(\boldsymbol{\phi}^{\wedge}\right)\mathbf{p}}{\delta\boldsymbol{\phi}} \\
&= \lim_{\delta\boldsymbol{\phi}\to\mathbf{0}} \frac{\left(\mathbf{J}_l\delta\boldsymbol{\phi}\right)^{\wedge}\exp\left(\boldsymbol{\phi}^{\wedge}\right)\mathbf{p}}{\delta\boldsymbol{\phi}} \\
&= \lim_{\delta\boldsymbol{\phi}\to\mathbf{0}} \frac{-\left(\exp\left(\boldsymbol{\phi}^{\wedge}\right)\mathbf{p}\right)^{\wedge}\mathbf{J}_l\delta\boldsymbol{\phi}}{\delta\boldsymbol{\phi}} = -\left(\mathbf{Rp}\right)^{\wedge}\mathbf{J}_l.
\end{aligned}$$

---

* Please note that the derivative cannot be defined by matrix differentiation, here we just write it for convenience.

† Strictly speaking, in matrix differentiation, we can only compute the derivative of a row vector to a column vector, whose result is a matrix. However, in this book we write the derivative of the column vector to the column vector for convenience. The reader can think that the numerator is transposed first, and after the computation, the final result is also transposed. This makes the formula look simple, otherwise we have to add a transpose to each line of the equations. In this sense, we can use equations like $\mathrm{d}(\mathbf{Ax})/\mathrm{d}\mathbf{x} = \mathbf{A}$.

The second line is BCH approximation, the third line is Taylor's approximation after throwing the high-order terms (but because the limit is taken, we still write the equal here), and the fourth line to the fifth row treat the skew-symmetric symbol as a cross product so that $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$. Thus, we compute the derivative of the rotated point relative to the addition in Lie algebra:

$$\frac{\partial (\mathbf{Rp})}{\partial \boldsymbol{\phi}} = (-\mathbf{Rp})^{\wedge} \mathbf{J}_l. \tag{4.41}$$

However, since there is still a very complicated form of $\mathbf{J}_l$, we don't want to calculate it. The perturbation model described below provides a simpler way to calculate derivatives.

4.3.4 Perturbation Model

Another way to do this is to perturb $\mathbf{R}$ for $\Delta\mathbf{R}$ and see the change of the result relative to the disturbance. This disturbance can be multiplied on the left or on the right. The final result will be slightly different. Let's take the left disturbance as an example. Let the left perturbation $\Delta\mathbf{R}$ correspond to the Lie algebra as $\boldsymbol{\varphi}$. Then, for $\boldsymbol{\varphi}$, that is:

$$\frac{\partial (\mathbf{Rp})}{\partial \boldsymbol{\varphi}} = \lim_{\boldsymbol{\varphi} \to \mathbf{0}} \frac{\exp(\boldsymbol{\varphi}^{\wedge}) \exp(\boldsymbol{\phi}^{\wedge}) \mathbf{p} - \exp(\boldsymbol{\phi}^{\wedge}) \mathbf{p}}{\boldsymbol{\varphi}}. \tag{4.42}$$

The derivation of this formula is simpler than the above:

$$\begin{aligned} \frac{\partial (\mathbf{Rp})}{\partial \boldsymbol{\varphi}} &= \lim_{\boldsymbol{\varphi} \to \mathbf{0}} \frac{\exp(\boldsymbol{\varphi}^{\wedge}) \exp(\boldsymbol{\phi}^{\wedge}) \mathbf{p} - \exp(\boldsymbol{\phi}^{\wedge}) \mathbf{p}}{\boldsymbol{\varphi}} \\ &= \lim_{\boldsymbol{\varphi} \to \mathbf{0}} \frac{(\mathbf{I} + \boldsymbol{\varphi}^{\wedge}) \exp(\boldsymbol{\phi}^{\wedge}) \mathbf{p} - \exp(\boldsymbol{\phi}^{\wedge}) \mathbf{p}}{\boldsymbol{\varphi}} \\ &= \lim_{\boldsymbol{\varphi} \to \mathbf{0}} \frac{\boldsymbol{\varphi}^{\wedge} \mathbf{Rp}}{\boldsymbol{\varphi}} = \lim_{\boldsymbol{\varphi} \to \mathbf{0}} \frac{-(\mathbf{Rp})^{\wedge} \boldsymbol{\varphi}}{\boldsymbol{\varphi}} = -(\mathbf{Rp})^{\wedge}. \end{aligned}$$

It can be seen that the calculation of a Jacobian $\mathbf{J}_l$ is omitted compared to the direct derivation of Lie algebra. This makes the perturbation model more practical. Please keep in mind the derivative here since we are going to use it in the pose estimation sections.

4.3.5 Derivative on SE(3)

Finally, we give the perturbation model on SE(3), and skip the derivative model. Suppose a point $\mathbf{p}$ is transformed by $\mathbf{T}$ (corresponding to Lie algebra $\boldsymbol{\xi}$), and the result is $\mathbf{Tp}$*. Now, give $\mathbf{T}$ a left perturbation $\Delta\mathbf{T} = \exp(\delta\boldsymbol{\xi}^{\wedge})$, whose Lie algebra is $\delta\boldsymbol{\xi} = [\delta\boldsymbol{\rho}, \delta\boldsymbol{\phi}]^{\mathrm{T}}$, then:

* Please note that to make multiplication make sense, $\mathbf{p}$ must use homogeneous coordinates.

$$
\begin{aligned}
\frac{\partial (\mathbf{Tp})}{\partial \delta \boldsymbol{\xi}} &= \lim_{\delta \boldsymbol{\xi} \to \mathbf{0}} \frac{\exp\left(\delta \boldsymbol{\xi}^{\wedge}\right) \exp\left(\boldsymbol{\xi}^{\wedge}\right) \mathbf{p} - \exp\left(\boldsymbol{\xi}^{\wedge}\right) \mathbf{p}}{\delta \boldsymbol{\xi}} \\
&= \lim_{\delta \boldsymbol{\xi} \to \mathbf{0}} \frac{\left(\mathbf{I} + \delta \boldsymbol{\xi}^{\wedge}\right) \exp\left(\boldsymbol{\xi}^{\wedge}\right) \mathbf{p} - \exp\left(\boldsymbol{\xi}^{\wedge}\right) \mathbf{p}}{\delta \boldsymbol{\xi}} \\
&= \lim_{\delta \boldsymbol{\xi} \to \mathbf{0}} \frac{\delta \boldsymbol{\xi}^{\wedge} \exp\left(\boldsymbol{\xi}^{\wedge}\right) \mathbf{p}}{\delta \boldsymbol{\xi}} \\
&= \lim_{\delta \boldsymbol{\xi} \to \mathbf{0}} \frac{\begin{bmatrix} \delta \boldsymbol{\phi}^{\wedge} & \delta \boldsymbol{\rho} \\ \mathbf{0}^{\mathrm{T}} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{Rp} + \mathbf{t} \\ 1 \end{bmatrix}}{\delta \boldsymbol{\xi}} \\
&= \lim_{\delta \boldsymbol{\xi} \to \mathbf{0}} \frac{\begin{bmatrix} \delta \boldsymbol{\phi}^{\wedge} \left(\mathbf{Rp} + \mathbf{t}\right) + \delta \boldsymbol{\rho} \\ \mathbf{0}^{\mathrm{T}} \end{bmatrix}}{[\delta \boldsymbol{\rho}, \delta \boldsymbol{\phi}]^{\mathrm{T}}} = \begin{bmatrix} \mathbf{I} & -(\mathbf{Rp} + \mathbf{t})^{\wedge} \\ \mathbf{0}^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} \end{bmatrix} \triangleq (\mathbf{Tp})^{\odot}.
\end{aligned}
$$

We define the final result as an operator $^{\odot *}$, which transforms a spatial point of homogeneous coordinates into a matrix of 4×6. This equation requires a little explanation about matrix differentiation. Assuming that $\mathbf{a}, \mathbf{b}, \mathbf{x}, \mathbf{y}$ are column vectors, then in our book, there are following rules:

$$
\frac{\mathrm{d} \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix}}{\mathrm{d} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}} = \left(\frac{\mathrm{d}[\mathbf{a}, \mathbf{b}]^{\mathrm{T}}}{\mathrm{d} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}} \right)^{\mathrm{T}} = \begin{bmatrix} \frac{\mathrm{d}\mathbf{a}}{\mathrm{d}\mathbf{x}} & \frac{\mathrm{d}\mathbf{b}}{\mathrm{d}\mathbf{x}} \\ \frac{\mathrm{d}\mathbf{a}}{\mathrm{d}\mathbf{y}} & \frac{\mathrm{d}\mathbf{b}}{\mathrm{d}\mathbf{y}} \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} \frac{\mathrm{d}\mathbf{a}}{\mathrm{d}\mathbf{x}} & \frac{\mathrm{d}\mathbf{a}}{\mathrm{d}\mathbf{y}} \\ \frac{\mathrm{d}\mathbf{b}}{\mathrm{d}\mathbf{x}} & \frac{\mathrm{d}\mathbf{b}}{\mathrm{d}\mathbf{y}} \end{bmatrix} \tag{4.43}
$$

Substituting this into the last line, you can get the final result. So far, we have introduced the differential operation on Lie group Lie algebra. In the following chapters, we will apply this knowledge to solve practical problems.

4.4 Practice: Sophus

4.4.1 Basic Usage of Sophus

We have introduced the basic knowledge of Lie algebra, and now it is time to consolidate what we have learned through practical exercises. Let's discuss how to manipulate Lie algebra in a program. In Lecture 3, we saw that Eigen provided geometry modules, but did not provide support for Lie algebra. A better Lie algebra library is the Sophus library maintained by Strasdat (`https://github.com/strasdat/Sophus`)[†]. The Sophus library supports SO(3) and SE(3), which are mainly discussed in this chapter. In addition, it also contains two-dimensional motion SO(2), SE(2) and the similar transformation of Sim(3). It is developed directly on top of Eigen and we don't need to install additional dependencies. Readers can get Sophus directly from GitHub, or the Sophus source code is also available in our book's code directory slambook2/3rdparty. For historical reasons, earlier versions of Sophus only provided double-precision Lie group/Lie algebra classes. Subsequent versions have been rewritten as template classes, so that different precision of Lie group/Lie algebra can be used in the Sophus from the template class, but at the

* I will read it as "Duang", like a stone falling into a well.

† Sophus Lie first proposed the Lie algebra. The library is named after him.

same time it increases the difficulty of use. In the second edition of this book, we use the Sophus library of **with templates**. The Sophus provided in the 3rdparty of this book is the **template** version, which should have been copied to your computer when you downloaded the code for this book. Sophus itself is also a cmake project. Presumably you already know how to compile the cmake project, so I won't go into details here. The Sophus library only needs to be compiled, no need to install it.

Let's demonstrate the SO(3) and SE(3) operations in the Sophus library:

Listing 4.1: slambook/ch4/useSophus.cpp

```
#include <iostream>
#include <cmath>
#include <Eigen/Core>
#include <Eigen/Geometry>
#include "sophus/se3.hpp"

using namespace std;
using namespace Eigen;

/// This program demonstrates the basic usage of Sophus
int main(int argc, char **argv) {
    // Rotation matrix with 90 degrees along Z axis
    Matrix3d R = AngleAxisd(M_PI / 2, Vector3d(0, 0, 1)).toRotationMatrix()
        ;
    // or quaternion
    Quaterniond q(R);
    Sophus::SO3d SO3_R(R);              // Sophus::SO3d can be constructed
        from rotation matrix
    Sophus::SO3d SO3_q(q);              // or quaternion
    // they are equivalent of course
    cout << "SO(3) from matrix:\n" << SO3_R.matrix() << endl;
    cout << "SO(3) from quaternion:\n" << SO3_q.matrix() << endl;
    cout << "they are equal" << endl;

    // Use logarithmic map to get the Lie algebra
    Vector3d so3 = SO3_R.log();
    cout << "so3 = " << so3.transpose() << endl;
    // hat is from vector to skew-symmetric matrix
    cout << "so3 hat=\n" << Sophus::SO3d::hat(so3) << endl;
    // inversely from matrix to vector
    cout << "so3 hat vee= " << Sophus::SO3d::vee(Sophus::SO3d::hat(so3)).
        transpose() << endl;

    // update by perturbation model
    Vector3d update_so3(1e-4, 0, 0); // this is a small update
    Sophus::SO3d SO3_updated = Sophus::SO3d::exp(update_so3) * SO3_R;
    cout << "SO3 updated = \n" << SO3_updated.matrix() << endl;

    cout << "********************************" << endl;
    // Similar for SE(3)
    Vector3d t(1, 0, 0);                 // translation 1 along X
    Sophus::SE3d SE3_Rt(R, t);           // construction SE3 from R,t
    Sophus::SE3d SE3_qt(q, t);           // or q,t
    cout << "SE3 from R,t= \n" << SE3_Rt.matrix() << endl;
    cout << "SE3 from q,t= \n" << SE3_qt.matrix() << endl;
    // Lie Algebra is 6d vector, we give a typedef
    typedef Eigen::Matrix<double, 6, 1> Vector6d;
    Vector6d se3 = SE3_Rt.log();
    cout << "se3 = " << se3.transpose() << endl;
    // The output shows Sophus puts the translation at first in se(3), then
         rotation.
    // Save as SO(3) wehave hat and vee
```

```
    cout << "se3 hat = \n" << Sophus::SE3d::hat(se3) << endl;
    cout << "se3 hat vee = " << Sophus::SE3d::vee(Sophus::SE3d::hat(se3)).
        transpose() << endl;

    // Finally the update
    Vector6d update_se3;
    update_se3.setZero();
    update_se3(0, 0) = 1e-4d;
    Sophus::SE3d SE3_updated = Sophus::SE3d::exp(update_se3) * SE3_Rt;
    cout << "SE3 updated = " << endl << SE3_updated.matrix() << endl;

    return 0;
}
```

The demo is divided into two parts. The first half introduces the operation on SO(3), and the second half is SE(3). We demonstrate how to construct $\mathrm{SO}(3), \mathrm{SE}(3)$ objects, exponentially, logarithmically map them, and update the lie group elements when we know the update amount. If the reader has a good understanding of the content of this lecture, then this program should not be difficult for you. In order to compile it, add the following lines to CMakeLists.txt:

Listing 4.2: slambook/ch4/useSophus/CMakeLists.txt

```
# we use find_package to make cmake find sophus
find_package( Sophus REQUIRED )
include_directories( ${Sophus_INCLUDE_DIRS} ) # sohpus is header only

add_executable( useSophus useSophus.cpp )
```

The find_package is a command provided by cmake to find the header and library files of a library. If cmake can find it, it will provide the variables for the directory where the header and library files are located. In the example of Sophus, it is Sophus_INCLUDE_DIRS. The template-based Sophus library, like Eigen, contains only header files and no source files. Based on them, we can introduce the Sophus library into our own cmake project. Readers are asked to see the output of this program on their own, which is consistent with our previous derivation.

4.4.2 Example: Evaluating the trajectory

In practical engineering, we often need to evaluate the difference between the estimated trajectory of an algorithm and the real trajectory to evaluate the accuracy of the algorithm. The real (or ground-truth) trajectory is often obtained by some higher precision systems, and the estimated one is calculated by the algorithm to be evaluated. In the last lecture we demonstrated how to display a trajectory stored in a file. In this section we will consider how to calculate the error of two trajectories. Consider an estimated trajectory $\mathbf{T}_{\mathrm{esti},i}$ and the real trajectory $\mathbf{T}_{\mathrm{gt},i}$, where $i = 1, \cdots, N$, then we can define some error indicators to describe the difference between them.

There are many kinds of error indicators. The common used one is **Absolute Trajectory Error** (ATE), which is like:

$$\mathrm{ATE}_{\mathrm{all}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \| \log(\mathbf{T}_{\mathrm{gt},i}^{-1}\mathbf{T}_{\mathrm{esti},i})^{\vee} \|_2^2}, \tag{4.44}$$

This is actually the Root-Mean-Squared Error (RMSE) for each pose in Lie algebra. This error can describe both of the rotation and translation error. At the same time,

some literatures only consider the translation error [21], so we can define **Average Translational Error**:

$$\mathrm{ATE}_{\mathrm{trans}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \|\mathrm{trans}(\mathbf{T}_{\mathrm{gt},i}^{-1}\mathbf{T}_{\mathrm{esti},i})\|_2^2}, \tag{4.45}$$

Where the suffix "trans" represents the translation of the internal variables of the parentheses. Because from the perspective of the entire trajectory, after the error occurs in the rotation, the subsequent trajectory will also have an error in the translation, so both indicators are applicable in practice.

In addition to this, relative error indicators can also be defined. For example, consider the movement from i to the time of $i + \Delta t$, then the Relative Pose Error (RPE) can be defined as:

$$\mathrm{RPE}_{\mathrm{all}} = \sqrt{\frac{1}{N-\Delta t}\sum_{i=1}^{N-\Delta t} \| \log\left(\left(\mathbf{T}_{\mathrm{gt},i}^{-1}\mathbf{T}_{\mathrm{gt},i+\Delta t}\right)\right)^{-1}\left(\mathbf{T}_{\mathrm{esti},i}^{-1}\mathbf{T}_{\mathrm{esti},i+\Delta t}\right)\Big)^{\vee} \|_2^2}, \tag{4.46}$$

Similarly, you can only take the translation part:

$$\mathrm{RPE}_{\mathrm{trans}} = \sqrt{\frac{1}{N-\Delta t}\sum_{i=1}^{N-\Delta t} \|\mathrm{trans}\left(\left(\mathbf{T}_{\mathrm{gt},i}^{-1}\mathbf{T}_{\mathrm{gt},i+\Delta t}\right)\right)^{-1}\left(\mathbf{T}_{\mathrm{esti},i}^{-1}\mathbf{T}_{\mathrm{esti},i+\Delta t}\right)\Big) \|_2^2}. \tag{4.47}$$

This part of the calculation is easy to implement with the Sophus library. Below we demonstrate the calculation of the absolute trajectory error. In this example, we have two trajectories: groundtruth.txt and estimated.txt. The following code will read the two trajectories, calculate the error, and display it in a 3D window. For the sake of brevity, the code for the trajectory plotting has been omitted, as we have done similar work in the previous section.

Listing 4.3: slambook/ch4/example/trajectoryError.cpp

```
#include <iostream>
#include <fstream>
#include <unistd.h>
#include <pangolin/pangolin.h>
#include <sophus/se3.hpp>

using namespace Sophus;
using namespace std;

string groundtruth_file = "./example/groundtruth.txt";
string estimated_file = "./example/estimated.txt";

typedef vector<Sophus::SE3d, Eigen::aligned_allocator<Sophus::SE3d>>
    TrajectoryType;

void DrawTrajectory(const TrajectoryType &gt, const TrajectoryType &esti);

TrajectoryType ReadTrajectory(const string &path);

int main(int argc, char **argv) {
    TrajectoryType groundtruth = ReadTrajectory(groundtruth_file);
    TrajectoryType estimated = ReadTrajectory(estimated_file);
    assert(!groundtruth.empty() && !estimated.empty());
```

```
    assert(groundtruth.size() == estimated.size());

    // compute rmse
    double rmse = 0;
    for (size_t i = 0; i < estimated.size(); i++) {
        Sophus::SE3d p1 = estimated[i], p2 = groundtruth[i];
        double error = (p2.inverse() * p1).log().norm();
        rmse += error * error;
    }
    rmse = rmse / double(estimated.size());
    rmse = sqrt(rmse);
    cout << "RMSE = " << rmse << endl;

    DrawTrajectory(groundtruth, estimated);
    return 0;
}

TrajectoryType ReadTrajectory(const string &path) {
    ifstream fin(path);
    TrajectoryType trajectory;
    if (!fin) {
        cerr << "trajectory " << path << " not found." << endl;
        return trajectory;
    }

    while (!fin.eof()) {
        double time, tx, ty, tz, qx, qy, qz, qw;
        fin >> time >> tx >> ty >> tz >> qx >> qy >> qz >> qw;
        Sophus::SE3d p1(Eigen::Quaterniond(qx, qy, qz, qw), Eigen::Vector3d
            (tx, ty, tz));
        trajectory.push_back(p1);
    }
    return trajectory;
}
```

The result of this program is 2.207, and the image is shown as Figure 4-2. You can also try to remove the rotating part and only calculates the error of the translation part. In fact, in this example, we have helped the reader to do some pre-processing tasks, including time alignment of the trajectory and external parameter estimation. These contents have not been mentioned yet, and we will talk about it in future.

4.5 Similar Transforma Group and its Lie Algebra

Finally, we would like to mention the similar transform group Sim(3) used in monocular vision, and the corresponding Lie algebra $\mathfrak{sim}(3)$. If you are only interested in stereo or RGB-D SLAM, you can skip this section.

We have already introduced the concept of scale ambiguity. If SE(3) is used in the monocular SLAM to represent the pose, then the scale in the entire SLAM process will change due to scale uncertainty and scale drift, which is what SE(3) not reflects. Therefore, in the case of monocular we generally express the scale factor explicitly. In mathematical terms, for the point $\mathbf{p}$ in space, a **similar transformation** is passed in the camera coordinate system instead of the Euclidean transformation:

$$\mathbf{p}' = \begin{bmatrix} s\mathbf{R} & \mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \mathbf{p} = s\mathbf{R}\mathbf{p} + \mathbf{t}. \tag{4.48}$$

In the similarity transformation, we express the scale as s. It also acts on top of the three coordinates of $\mathbf{p}$ and scales $\mathbf{p}$ once. Similar to SO(3), SE(3), the simi-

Figure 4-2: Calculates the error between the estimated trajectory and the real trajectory.

larity transform also forms a group on matrix multiplication, called the similarity transform group Sim(3):

$$\mathrm{Sim}(3) = \left\{ \mathbf{S} = \begin{bmatrix} s\mathbf{R} & \mathbf{t} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \in \mathbb{R}^{4\times 4} \right\}. \tag{4.49}$$

Similarly, Sim(3) also has corresponding Lie algebra, exponential mapping, logarithmic mapping, and so on. The Lie algebra $\mathfrak{sim}(3)$ element is a 7-dimensional vector $\boldsymbol{\zeta}$. Its first 6 dimensions are the same as $\mathfrak{se}(3)$, and followed by a σ to denote the scale.

$$\mathfrak{sim}\left(3\right) = \left\{ \boldsymbol{\zeta} | \boldsymbol{\zeta} = \begin{bmatrix} \boldsymbol{\rho} \\ \boldsymbol{\phi} \\ \sigma \end{bmatrix} \in \mathbb{R}^7, \boldsymbol{\zeta}^{\wedge} = \begin{bmatrix} \sigma\mathbf{I} + \boldsymbol{\phi}^{\wedge} & \boldsymbol{\rho} \\ \mathbf{0}^{\mathrm{T}} & 0 \end{bmatrix} \in \mathbb{R}^{4\times 4} \right\}. \tag{4.50}$$

It has an additional σ compared with $\mathfrak{se}(3)$. The Sim(3) and $\mathfrak{sim}(3)$ are still associated with exponential maps and logarithm maps. The exponential mapping is:

$$\exp\left(\boldsymbol{\zeta}^{\wedge}\right) = \begin{bmatrix} \mathrm{e}^{\sigma}\exp\left(\boldsymbol{\phi}^{\wedge}\right) & \mathbf{J}_s\boldsymbol{\rho} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix}. \tag{4.51}$$

Where $\mathbf{J}_s$ is:

$$\begin{aligned} \mathbf{J}_s =& \frac{\mathrm{e}^{\sigma}-1}{\sigma}\mathbf{I} + \frac{\sigma\mathrm{e}^{\sigma}\sin\theta + \left(1-\mathrm{e}^{\sigma}\cos\theta\right)\theta}{\sigma^2+\theta^2}\mathbf{a}^{\wedge} \\ &+ \left(\frac{\mathrm{e}^{\sigma}-1}{\sigma} - \frac{\left(\mathrm{e}^{\sigma}\cos\ Theta - 1\right)\sigma + \left(\mathrm{e}^{\sigma}\sin\theta\right)\theta}{\sigma^2+\theta^2}\right)\mathbf{a}^{\wedge}\mathbf{a}^{\wedge}. \end{aligned}$$

Through exponential mapping, we can find the relationship between Lie algebra and Lie group. For the Lie algebra $\boldsymbol{\zeta}$, its correspondence with Lie group is:

$$s = \mathrm{e}^{\sigma},\ \mathbf{R} = \exp(\boldsymbol{\phi}^{\wedge}),\ \mathbf{t} = \mathbf{J}_s\boldsymbol{\rho}. \tag{4.52}$$

The rotation is consistent with SO(3). In the translation part, we need to multiply a Jacobian $\mathcal{J}$ in $\mathfrak{se}(3)$, and the similarly transformed Jacobi is more complicated. For the scale factor, you can see that s in the Lie group is the exponential function of σ in the Lie algebra.

The BCH approximation of Sim(3) is similar to SE(3). We can discuss a derivative of $\mathbf{S}$ after a similar transformation of $\mathbf{Sp}$ relative to $\mathbf{S}$. Similarly, there are two ways of differential model and perturbation model, and the perturbation model is the simpler one. We omit the derivation process and directly give the results of the perturbation model. Let $\mathbf{Sp}$ a small perturbation $\exp(\boldsymbol{\zeta}^\wedge)$ on the left and ask for $\mathbf{Sp}$ The derivative of the disturbance. Since $\mathbf{Sp}$ is a 4-dimensional homogeneous coordinate, $\boldsymbol{\zeta}$ is a 7-dimensional vector, which should have 4×7 Jacobian. For convenience, remember the first 3 dimensional composition vector $\mathbf{q}$ of $\mathbf{Sp}$, then:

$$\frac{\partial \mathbf{Sp}}{\partial \boldsymbol{\zeta}} = \begin{bmatrix} \mathbf{I} & -\mathbf{q}^\wedge & \mathbf{q} \\ \mathbf{0}^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} & 0 \end{bmatrix}. \tag{4.53}$$

We will end here about the contents of Sim(3). For more detailed information on Sim(3), please refer to the literature [22].

4.6 Summary

This lecture introduces Lie group SO(3) and SE(3), and their corresponding Lie algebras $\mathfrak{so}(3)$ and $\mathfrak{se}(3)$. We introduce the expression and transformation of poses on them, and then through the linear approximation of BCH, we can perturb and predict the pose. This lays the theoretical foundation for the optimization of the posture afterwards, because we need to adjust the estimate of a certain pose frequently so that the corresponding error is reduced. Only after we have figured out how to adjust and update the pose can we continue to the next step.

The content of this lecture may be more theoretical. After all, it is not as good as computer vision. Compared to the mathematics textbooks that explain Lie group Lie algebra, since we only care about practical content, the process is very streamlined and the speed is relatively fast. The reader must understand the content of this lecture, which is the basis for solving many subsequent problems, especially the pose estimation part.

It should be mentioned that in addition to the Lie algebra, the rotation can also be expressed by means of quaternion, Euler angle, etc., but the subsequent processing is troublesome. In practice, you can also use SO(3) plus panning instead of SE(3) to avoid some Jacobian calculations.

Exercises

1. Verify SO(3), SE(3), and Sim(3) are groups on matrix multiplication.
2. Verify that $(\mathbb{R}^3, \mathbb{R}, \times)$ constitutes a Lie algebra.
3. Verify that $\mathfrak{so}(3)$ and $\mathfrak{se}(3)$ satisfy the requirements of Lie algebra.
4. Verify the properties (4.20) and (4.21).
5. Show that:
$$\mathbf{Rp}^\wedge \mathbf{R}^{\mathrm{T}} = (\mathbf{Rp})^\wedge.$$

6. Show that:

$$\mathbf{R}\exp(\mathbf{p}^\wedge)\mathbf{R}^\mathrm{T} = \exp((\mathbf{R}\mathbf{p})^\wedge).$$

This is called The **adjoint** property on SO(3). Similarly, there is an adjoint property on SE(3):

$$\mathbf{T}\exp(\boldsymbol{\xi}^\wedge)\mathbf{T}^{-1} = \exp\left((\mathrm{Ad}(\mathbf{T})\boldsymbol{\xi})^\wedge\right), \tag{4.54}$$

where

$$\mathrm{Ad}(\mathbf{T}) = \begin{bmatrix} \mathbf{R} & \mathbf{t}^\wedge\mathbf{R} \\ \mathbf{0} & \mathbf{R} \end{bmatrix}. \tag{4.55}$$

7. Follow the derivation of the left perturbation and derives the derivatives of SO(3) and SE(3) under the right perturbation.

8. Search how cmake's find_package works. What optional parameters does it have? What are the prerequisites for cmake to find a library?

Chapter 5

Cameras and Images

Goal of This Chapter

1. Understand the pin-hole camera model, intrinsics, extrinsics and distortion.
2. Understand how to project a spatial point into image planes.
3. Understand how to cope with the OpenCV images.
4. Understand the basic calibration methods.

In the previous two lectures, we introduced the problem that how to express and optimize the robot's 6 DoF pose, and partially explained the meaning of the variables and the equations of motion and observation in SLAM. In this chapter we will discuss "How robots observe the outside world", which is part of the observation equation. In the camera-based visual SLAM, the observation mainly refers to the process of **image projection**.

We can see a lot of photos in real life. In a computer, a photo consists of millions of pixels, each of which records the information about color or brightness. We will see a bundle of light reflected or emitted by an object in the three-dimensional world pass through the camera's optical center and is projected onto the imaging plane of the camera. After the camera's light sensor receives the light, it produces a measurement and we get the pixels, which form the photo we see. Can this process be described by mathematical equations? This lecture will first discuss the camera model, explain how the projection relationship is described, and what is the internal parameters in this projection process. At the same time, we are also going to give a brief introduction to the stereo and RGB-D cameras. Then, we introduce the basic operations of 2D images in OpenCV. Finally, an experiment of point cloud stitching is demonstrated to show the meaning of intrinsics and extrinsics parameters.

5.1 Pin-hole Camera Models

The process of projecting a 3D point (in meters) to a 2D image plane (in pixels) can be described by a geometric model. Actually there are several models to describe this, the simplest of which is called the **pinhole model**. We will start from this pin-hole projection. At the same time, due to the presence of the lens on the camera lens, **distortion** is generated during the projection. Therefore, we are going to use the pin-hold model plus with a distortion model to describe the entire projection process.

5.1.1 Pinhole Camera Geometry

Most of us have seen the candle projection experiment in the physics class of high school: a lit candle is placed in front of a dark box, and the light of the candle is projected through a small hole in the dark box on the rear plane of the black box. Then an inverted candle image is formed on this plane. In this process, the small hole is able to project a candle in a three-dimensional world onto a two-dimensional imaging plane. For the same reason, we can use this simple model to explain the imaging process of the camera, as shown in Figure 5-1.

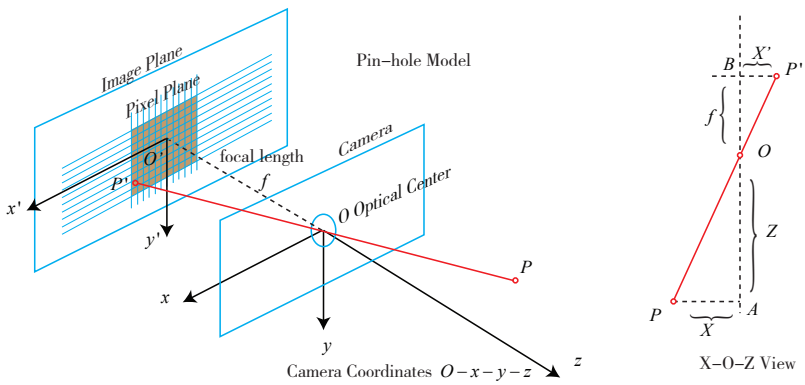

Figure 5-1: Pinhole camera model.

Let's take a look at the simple geometry in this model. Let $O-x-y-z$ be the camera coordinate system. Commonly we put the z axis to the front of the camera, x to the right, and y to the down (so in this figure we should stand on the left side to see the right side). O is the camera's **Optical Center**, which is also the "hole" in the pinhole model. The 3D point P, after being projected through the hole O, falls on the physical imaging plane $O'-x'-y'$, and produce the image point P'. Let the coordinates of P be $[X, Y, Z]^{\mathrm{T}}$, P' is $[X', Y', Z']^{\mathrm{T}}$, and set the physical distance from the imaging plane to camera plane is f (focal length). Then, according to the similarity of the triangles, there are:

$$\frac{Z}{f} = -\frac{X}{X'} = -\frac{Y}{Y'}. \tag{5.1}$$

The negative sign indicates that the image is inverted. However, the image obtained by the actual camera is not an inverted image (otherwise the usage of the camera would be very inconvenient). In order to make the model more realistic, we can

equivalently place the imaging plane symmetrically in front of the camera, along with the 3D space points on the same side of the camera coordinate system, as shown by Figure 5-2. This can remove the negative sign in the formula to make the formula more compact:

$$\frac{Z}{f} = \frac{X}{X'} = \frac{Y}{Y'}. \tag{5.2}$$

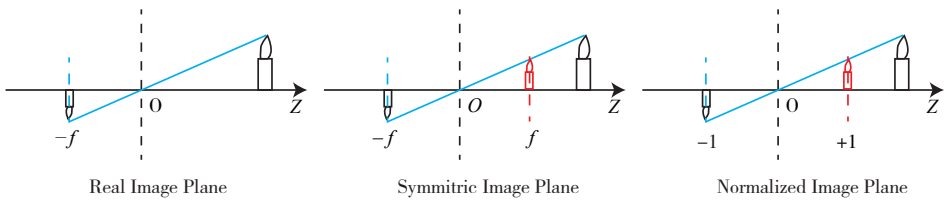

Figure 5-2: The real, symmetric and normalized image plane.

Put X', Y' to the left side:

$$\begin{matrix} X' = f\frac{X}{Z} \\ Y' = f\frac{Y}{Z} \end{matrix}. \tag{5.3}$$

Readers may ask why can we seem to arbitrarily move the imaging plane to the front? In fact this is just a mathematical approach to handle the camera projection, and most of the images captured by the camera are not upside-down - the camera's software will flip the image for you, so what we actually get is the image on the symmetric plane. So, although from the physical principle, the pin-hold image should be inverted, but since we have pre-processed the image, it is not bad to take the symmetric one. Therefore, without causing ambiguity, we often emit the minus symbol in the pinhole model.

The formula (5.3) describes the spatial relationship between the point P and its image, where the units of all points are meters, for example, a focal length may be 0.2 meters and X' be 0.14 meters. However, in the camera, we end up with pixels, where we need to sample and quantize the pixels on the imaging plane. In order to describe the process by which the sensor converts the perceived light into image pixels, we set a pixel plane $o - u - v$ fixed on the physical imaging plane. Finally we get **pixel coordinates** of P' in the pixel plane: $[u, v]^{\mathrm{T}}$.

The usual definition ot the **pixel coordinate system*** is : the origin o' is in the upper left corner of the image, the u axis is parallel to the x axis, and the v axis is parallel to the y axis. Between the pixel coordinate system and the imaging plane, there is an obvious **zoom** and a **translation of the origin**. We set the pixel coordinates to scale α times on the u axis and β times on v. At the same time, the origin is translated by $[c_x, c_y]^{\mathrm{T}}$. Then, the relationship between the coordinates of P' and the pixel coordinate $[u, v]^{\mathrm{T}}$ is:

$$\begin{cases} u = \alpha X' + c_x \\ v = \beta Y' + c_y \end{cases}. \tag{5.4}$$

Put it into (5.3) and set αf as f_x, βf as f_y:

$$\begin{cases} u = f_x \frac{X}{Z} + c_x \\ v = f_y \frac{Y}{Z} + c_y \end{cases}, \tag{5.5}$$

* Or image coordinate system, see section 2 of this lecture.

where f is the focal length in meters, $\alpha, and \beta$ is in pixels/meter, so f_x, f_y and c_x, c_y are in pixels. It would be more compact to write this form as a matrix, but we need to use homogeneous coordinates on the left and non-homogeneous coordinates on the right:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{Z} \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \triangleq \frac{1}{Z}\mathbf{KP}. \tag{5.6}$$

Let put Z to the left side as in most books:

$$Z \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \triangleq \mathbf{KP}. \tag{5.7}$$

In this equation, we refer to the matrix composed of the middle quantities as the camera's **inner parameter matrix** (o Intrinsics)$\mathbf{K}$. It is generally believed that the internal parameters of the camera are fixed after manufacturing and will not change during usage. Some camera manufacturers will tell you the internal parameters of the camera, and sometimes you need to estimate the internal parameters by yourself, which is called **calibration**. In view of the maturity of the calibration algorithm (such as the famous Zhang Zhengyou's calibration [23]), it will not be introduced here *.

There are internal parameters, and naturally there must be something like "external parameters". In the equation (5.6) , we use the coordinates of P in the camera coordinate system, but in fact the coordinates of P should be its world coordinates because the camera is moving (we use symbol $\mathbf{P}_w$). It should be converted to the camera coordinate system based on the current pose of the camera. The pose of the camera is described by its rotation matrix $\mathbf{R}$ and the translation vector $\mathbf{t}$. Then there are:

$$Z\mathbf{P}_{uv} = Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K}\left(\mathbf{RP}_w + \mathbf{t}\right) = \mathbf{KTP}_w. \tag{5.8}$$

Note that the latter formula implies a conversion from homogeneous to non-homogeneous coordinates (can you see it?)†. It describes the projection relationship of world coordinates to pixel coordinates of P. Among them, the camera's pose $\mathbf{R}, \mathbf{t}$ is also called the camera's **extrinsics** ‡. Compared with the intrinsics, the extrinsics may change with the camera installation, and is also the target to be estimated in the SLAM if we only have a camera.

The projection process can also be viewed from another perspective. The formula (5.8) shows that we can convert a world coordinate point to the camera coordinate system first, and then remove the value of its last dimension (that is, the depth of the point from the imaging plane of the camera), which is equivalent to the **normalization** on the last dimension. By this way we get the projection of the

---

* I'm sure professor Zhang has a copy of this book now.

† We use homogeneous coordinates in $\mathbf{TP}$, then convert to non-homogeneous coordinates, and then multiply it by $\mathbf{K}$.

‡ In robots or autonomous vehicles, the extrinsics is sometimes explained the transform between the camera coordinate system and the robot body coordinate system, describing "where the camera is installed".

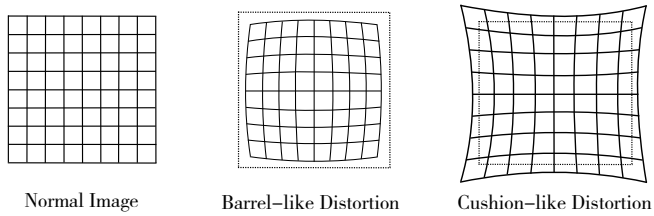

Figure 5-3: The radical distortion.

point P on the camera **normalized plane**:

$$(\mathbf{R}\mathbf{P}_w + \mathbf{t}) = \underbrace{[X, Y, Z]^{\mathrm{T}}}_{\text{Camera Coordinates}} \rightarrow \underbrace{[X/Z, Y/Z, 1]^{\mathrm{T}}}_{\text{Normalized Coordinates}} . \tag{5.9}$$

The **normalized coordinates** can be seen as a point in the $z = 1$ plane in front of the camera*.This $z = 1$ plane is also called **normalized plane**. We normalize the coordinates and then multiply it with the intrinsic matrix, yielding the pixel coordinates, so we can also consider the pixel coordinates $[u, v]^{\mathrm{T}}$ as the result of quantitative measurements on points on the normalized plane. It can also be seen from this model that if the camera coordinates are multiplied by any non-zero constant at the same time, the normalized coordinates are the same, which means that the **depth is lost during the projection process**, so in monocular vision the depth value of the pixel cannot be obtained by a single image.

5.1.2 Distortion

In order to get a larger FoV (Field-of-View), we normally add a lens in front of the camera. The addition of the lens has an influence on the propagation of light during imaging: (1) the shape of lens may affect the propagation way of light, (2) during the mechanical assembly, the lens and the imaging plane are not completely parallel, which also makes the projected position change.

There are some mathematical models to describe the **distortion** caused by the shape of the lens. In the pinhole model, a straight line keeps straight when projected onto the pixel plane. However, in real photos, the lens of the camera tends to make a straight line in the real environment become a curve †. The closer to the edge of the image, the more obvious this phenomenon is. Since the lenses actually produced are often center-symmetrical, this makes the irregular distortion generally radially symmetrical. They fall into two main categories: **barrel-like distortion** and **cushion-like distortion**, as shown by Figure 5-3.

In barrel distortion the radius of pixels decreases as the distance from the optical axis increases, while the cushion distortion is just the opposite. In both distortions, the line that intersects the intersection of the center of the image and the optical axis remains the same.

In addition to the shape of the lens, which introduces radial distortion, **tangential distortion** is introduced in during assembly of the camera because the lens and the imaging surface cannot be strictly parallel, as shown by Figure 5-4.

To better understand radial and tangential distortion, we describe them in more rigorous mathematical form. Consider any point on the **normalized plane**, **p**,

* Note that in the actual calculation, it is necessary to check whether Z is positive, because the negative Z can also get a point on the normalized plane by this method. However, the camera

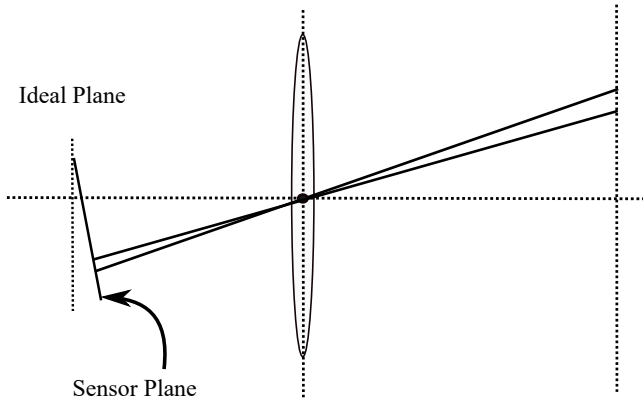

Figure 5-4: Tangential distortion.

whose coordinates are $[x, y]^{\mathrm{T}}$, or $[r, \theta]^{\mathrm{T}}$ in the form of polar coordinates, where r represents the distance between the point $\mathbf{p}$ and the origin of the coordinate system, and θ represents the angle to the horizontal axis. Radial distortion can be seen as a change in the coordinate point along the length, that is, its radius from the origin. Tangential distortion can be seen as a change in the coordinate point along the tangential direction, that is, the horizontal angle has changed. It is generally assumed that these distortions are polynomial, namely:

$$\begin{aligned} x_{\text{distorted}} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{\text{distorted}} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6), \end{aligned} \tag{5.10}$$

where $[x_{\text{distorted}}, y_{\text{distorted}}]^{\mathrm{T}}$ is the **normalized coordinates** of the point after distortion. On the other hand, for **tangential distortion**, we can use the other two parameters p_1, p_2 to describe it:

$$\begin{aligned} x_{\text{distorted}} &= x + 2p_1 xy + p_2(r^2 + 2x^2) \\ y_{\text{distorted}} &= y + p_1(r^2 + 2y^2) + 2p_2 xy. \end{aligned} \tag{5.11}$$

Put (5.10) and (5.11) together we get a joint model with 5 distortion coefficients. The complete form is:

$$\begin{cases} x_{\text{distorted}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 xy + p_2(r^2 + 2x^2) \\ y_{\text{distorted}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y^2) + 2p_2 xy \end{cases}. \tag{5.12}$$

In the above process of correcting distortion, we used 5 distortion coefficients. In practical applications, you can flexibly choose to number of parameters, for example, only selecting k_1, p_1, p_2, or use k_1, k_2, p_1, p_2, etc.

In this section, we modeled the camera's imaging process using a pinhole model and described the radial and tangential distortions caused by the lens. In the actual image system, researchers have proposed many other models, such as the affine model and perspective model, and there are many other types of distortion. In most of the visual SLAM systems, pinhole models and rad-tan distortion models are sufficient, so we will not describe other one.

---

does not capture the scene behind the imaging plane.

† Yes, it is no longer straight, but becomes curved. If it makes an inside curve, it is called barrel-like distortion; otherwise if the curve looks outward, it is cushion-like distortion.

It is worth mentioning that there are two ways of undistortion (or correction). We can choose to undistort the entire image first, get the corrected image, and then discuss the spatial position of the points on the image. Alternatively, we can also discuss some feature points in the distorted image, and find its real position through the distortion equation. Both are feasible, but the former seems to be more common in visual SLAM. Therefore, when an image is undistorted, we can directly establish a projection relationship with the pinhole model without considering distortion. Therefore, in the discussion that follows, we can directly assume that the image has been undistorted.

Finally, let's summarize the imaging process of a monocular camera:

1. First, there is a point P in the world coordinate system, and its world coordinates are $\mathbf{P}_w$.
2. Since the camera is moving, its motion is described by $\mathbf{R}, \mathbf{t}$ or transform matrix $\mathbf{T} \in \mathrm{SE}(3)$. The camera coordinates for P are $\tilde{\mathbf{P}}_c = \mathbf{R}\mathbf{P}_w + \mathbf{t}$.
3. The $\tilde{\mathbf{P}}_c$ component is X, Y, Z, and they are projected onto the normalized plane $Z = 1$ to get the normalized coordinates: $\mathbf{P}_c = [X/Z, Y/Z, 1]^{\mathrm{T}}$*.
4. If the image is distorted, the coordinates of $\mathbf{P}_c$ after distortion are calculated according to the distortion parameters.
5. Finally, the distorted coordinates of P pass through the intrinsics and we find its pixel coordinates: $\mathbf{P}_{uv} = \mathbf{K}\mathbf{P}_c$.

In summary, we have talked about four coordinates: the world coordinates, the camera coordinates, the normalized coordinates, and the pixel coordinates. Readers should clarify their relationship, which reflects the entire imaging process and will be used in future.

5.1.3 Stereo Cameras

The pinhole camera model describes the imaging model of a single camera. However, we cannot determine the specific location of a spatial point on by a single pixel. This is because all points on the line from the camera's optical center to the normalized plane can be projected onto that pixel. Only when the depth of P is determined (such as through a binocular or RGB-D camera) can we know exactly its spatial location, as shown in Figure 5-5 .

There are many ways to measure the pixel distance (or depth). For example, the human eye can judge the distance of the object according to the difference (or parallax) of the scene seen by the left and right eyes. The principle of the binocular camera is also the same: by simultaneously acquiring the images of the left and right cameras, and calculating the parallax/disparity between the images, the depth of each pixel is estimated. In the following paragraph we briefly describe the imaging principle of the stereo camera (as shown in Figure 5-6).

A binocular camera is generally composed of a left-eye camera and a right-eye camera. Of course, it can also be made up and down, but the mainstream binoculars we've seen are all left and right. In left and right cameras are often regarded as simple pin-hole cameras. They are synchrozied and placed horizontally, meaning that the

* Note that Z may be less than 1, indicating that the point is behind the normalization plane and it should not be projected on the camera plane.

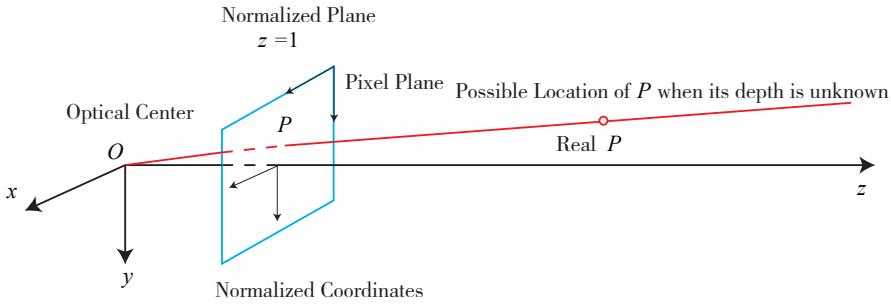

Figure 5-5: The possible location of a single pixel.

centers of both cameras are on the same x axis. The distance between the two centers is called **baseline** (denoted as b), which is an important parameter.

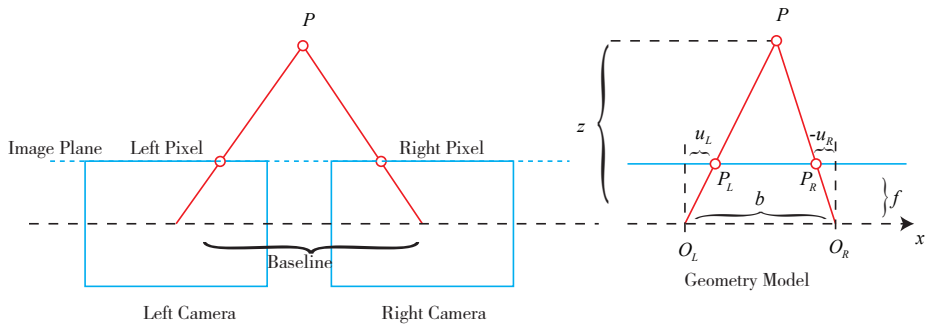

Figure 5-6: Geometry model of stereo cameras from upside down. The O_L, O_R are left and right optical centers. f is the focal length, u_L and u_R are pixel corrdinates of a same point along x axis. Note that u_R should be a negative value in this figure, so the pysical distance should be $-u_R$.

Now consider a 3D point P, which is projected into the left-eye and the right-eye, written as P_L, P_R. Due to the presence of the camera baseline, these two imaging positions are different. Ideally, since the left and right cameras are only shifted on the x axis, the image of P also differs only on the x axis (corresponding to the u axis of the image). Take the left pixel coordinate as u_L and the right coordinate as u_R. The geometric relationship is shown on the right of Figure 5-6. According to the similarity relationship between $\triangle PP_LP_R$ and $\triangle PO_LO_R$, there are:

$$\frac{z-f}{z} = \frac{b-u_L+u_R}{b}. \tag{5.13}$$

Rearrangle it and we have:

$$z = \frac{fb}{d}, \quad d \triangleq u_L - u_R, \tag{5.14}$$

where d is defined as the difference between the horizontal coordinates of the left and right figures, and is called **disparity** or **parallax**. Based on parallax, we can

estimate the distance between a pixel and the camera. Parallax is inversely proportional to distance: the larger the parallax is, the closer the distance is *. At the same time, since the parallax is at least one pixel, there is a theoretical maximum value for the binocular depth, which is determined by fb. We see that the in order to see the far away things, we need a larger stereo camera; conversely, small binocular devices can only measure very close distances. By analogy, when the human eye looks at a very distant object (such as a very distant airplane), it is usually impossible to accurately determine its distance.

Although the formula for calculating the depth from parallax is simple, the calculation of parallax d itself is more difficult. We need to know exactly where a pixel of the left-eye image appears in the right-eye image (that is, the corresponding relationship). This also belongs to the kind of task that is "easy for humans but difficult for computers". When we want to calculate the depth of each pixel in an image, the calculation amount and accuracy will become a problem, and the parallax can be calculated only in the place where the image texture is rich. Due to the amount of calculation, binocular depth estimation still needs to use GPU or FPGA to make the distance calculation run in real time. This will be mentioned in lecture 13.

5.1.4 RGB-D Cameras

Compared to the binocular camera's way of calculating depth, the RGB-D camera's approach is more "active": it can actively measure the depth of each pixel. The current RGB-D cameras can be divided into two categories according to their principle (see Figure 5-7):

1. The first kind of RGB-D sensor uses **Infrared Structured Light** (Structured Light) to measure pixel distance. Many of the old RGB-D sensors are belong to this kind, for example, the Kinect 1st generation, Project Tango 1st generation, Intel RealSense, etc.

2. The second kind measures pixel distance using the **Time-of-flight (ToF)**. Examples are Kinect 2 and some existing ToF sensors.

Regardless of the type, the RGB-D camera needs to emit a beam of light (usually infrared light) to the target object. In the principle of structured light, the camera calculates the distance between the object and itself based on the returned structured light pattern. In the ToF principle, the camera emits light pulse to the target, and then determines the distance according to the time of flight of the beam. The ToF principle is very similar to the laser sensor, except that the laser obtains the distance by scanning point by point (or line by line), and the ToF camera can obtain the pixel depth of the entire image, which is also the characteristics of the RGB-D camera. So, if you take apart an RGB-D camera, you will usually find that there will be at least one transmitter and one receiver in addition to the ordinary camera.

After measuring the depth, the RGB-D camera usually completes the pairing between the depth and color map pixels according to the position of each camera at the time of production, and outputs a pixel-to-pixel corresponding color image and depth image. We can read the color information and distance information at the same image position, calculate the 3D camera coordinates of the pixels, and generate a point cloud. RGB-D data can be processed either at the image level or the point

* Readers can simulate it with your own eyes.

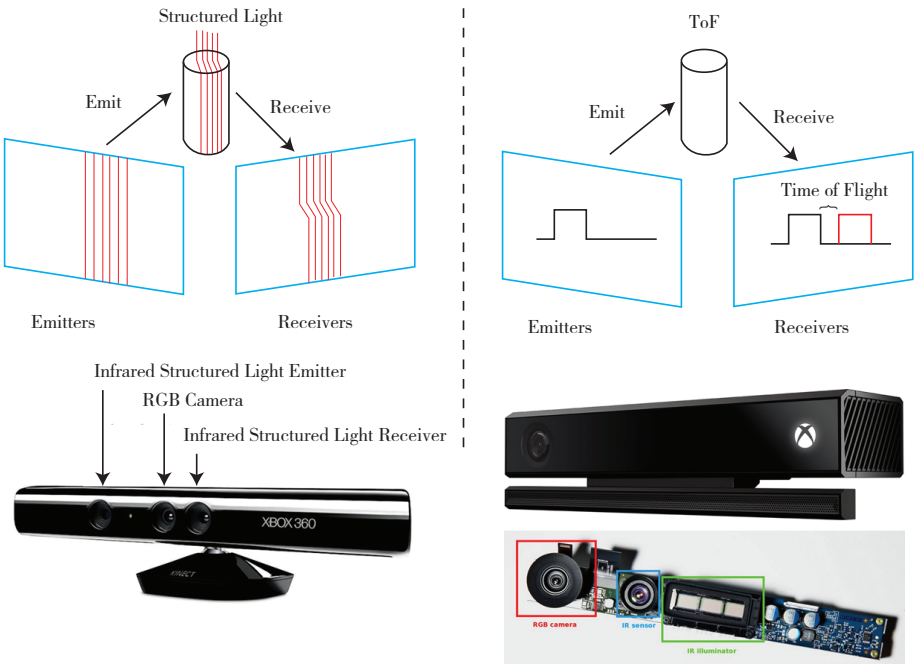

Figure 5-7: RGB-D Cameras

cloud level. The second experiment of this lecture will demonstrate the point cloud construction of RGB-D cameras.

The RGB-D camera can measure the distance of each pixel in real time. However, due to this measurement of transmitting and receiving, its range of use is limited. RGB-D cameras that use infrared light for depth measurement are susceptible to interference from infrared light emitted by daylight or other sensors, so they cannot be used outdoors. Without modulation, multiple RGB-D cameras can interfere with each other. For transmissive objects, the position of these points cannot be measured because they cannot receive reflected light. In addition, RGB-D cameras have some disadvantages in terms of cost and power consumption.

5.2 Images

Cameras and lens convert the information in the three-dimensional world into a photo composed of pixels, which is then stored in the computer as a data source for subsequent processing. In mathematics, images can be described by a matrix; in computers, they occupy a continuous disk or memory space, which can be represented by a two-dimensional array. In this way, the program does not have to distinguish whether they are dealing with a numerical matrix or a meaningful image.

In this section, we will introduce some basic operations of computer image processing. In particular, we are going to introduce the basic steps of processing images with OpenCV, and lay the foundation for subsequent chapters. Let's start with the simplest case, the grayscale image. In a grayscale image, each pixel position (x, y)

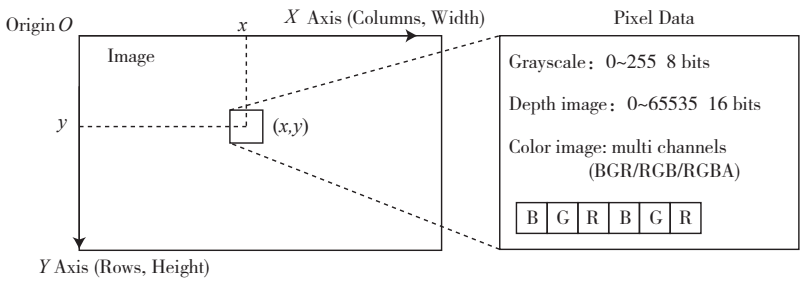

Figure 5-8: Pixels in an image.

corresponds to a grayscale value of I. Therefore, an image with a width of w and a height of h can be mathematically written as a function:

$$(I)(x, y) : \mathbb{R}^2 \mapsto \mathbb{R},$$

where (x, y) is the coordinate of the pixel. However, computers cannot express real space, so we need to quantify the subscripts and image readings within a certain range. For example, x, y are usually integers starting with 0 to $w - 1, h - 1$. In common grayscale images, an integer of 0~255 (that is, an unsigned char in C++, 1 byte) is used to express the grayscale reading of the image. Then, a grayscale image with a width of 640 pixels and a height of 480 pixels can be expressed as:

Listing 5.1: Use 2D array to express an image

```
unsigned char image[480][640];
```

Why does the two-dimensional array here has the size of 480 × 640? Because in the program, the first index of 2D array is the row, and the second index is the column. In an image, the number of rows (or the y axis) in the array corresponds to the height of the image, and the number of columns (or the x axis) corresponds to the width of the image.

Let's examine the content of this image. Images are naturally made up of pixels. When accessing a certain pixel, you need to specify its coordinates, as shown in Figure 5-8 . The left side of the figure shows how the traditional pixel coordinate system is defined. The origin is in the upper left corner of the image, X axis is from left to right, and Y is top-down. If it has a third axis, the Z axis, then according to the right-hand rule, the Z axis should be from outside to inside (or front in 3D space). This definition is consistent with the camera coordinate system. The width or number of columns of an image corresponds to the X axis; the number of rows, or the height of an image corresponds to its Y axis.

According to this definition, if we discuss a pixel located at x, y, then the code of accessing its memory in computer should be:

Listing 5.2: Accessing image pixels

```
unsigned char pixel = image[y][x];
```

It corresponds to the reading of the gray value $I(x, y)$. Please note the order of x and y here. Although we tirelessly discuss the problem of coordinate systems, errors like this index sequence will still be one of the most common errors encountered by

novices during debugging. If you accidentally change the coordinates of x, y when writing a program, the compiler cannot provide any useful information, and all you can see is an segment fault in runtime.

The gray scale of a pixel can be recorded as an 8-bit unsigned integer, which is a value of 0~255. If we have more information to record, one byte is probably not enough. For example, in the depth map of an RGB-D camera, the distance between each pixel is recorded. This distance is usually measured in millimeters, and the range of RGB-D cameras is usually around a dozen meters, exceeding 255. At this time, people will use 16-bit integers (unsigned short in C++) to record the depth map information, that is, the value at 0~65535. When converted to meters, it can represent up to 65 meters, which is enough for RGB-D cameras.

The representation of a color image requires the concept of a channel. In computers, we use a combination of three colors: red, green, and blue to express any color. Therefore, for each pixel, three values of R, G, and B are recorded, and each value is called a channel. For example, the most common color image has three channels, each of which is represented by an 8-bit integer. Under this rule, one pixel occupies 24-bit space.

The number and order of channels can be freely defined. In OpenCV color images, the default order of channels is B, G, R. That is, when we get a 24-bit pixel, the first 8 bits represent the blue value, the middle 8 bits represent the green value, and the last 8 bits represent the red value. In the same way, the order of R, G, and B can also be used to represent a color image. If you want to express the transparency of the image, use R, G, B, A four channels.

5.3 Practice: Images in Computer Vision

5.3.1 Basic Usage of OpenCV

The following is a demo program to understand how to access the image in OpenCV, and how to visit its pixels.

Install OpenCV

OpenCV * provides a large number of open source image algorithms, and is a very widely used image processing algorithm library in computer vision. This book also uses OpenCV for basic image processing. Before using, readers must install it from library or from source code. Under Ubuntu, there are two options: **install from source code** or **only install binary library files**:

1. Install from source means to download all OpenCV source code from the OpenCV website, compile and install on the machine for usage. The advantage is that you can freely choose which version to install, and the source code is accessible, but it takes some compilation time.

2. Or, we can only installs the binary library file, which means it was pre-compiled by the Ubuntu community, so that there is no need to recompile it.

Since we use a newer version of OpenCV, we must install it from the source code. First, you can adjust some compilation options to match the programming

* Official homepage: `http://opencv.org`.

environment (for example, whether you need GPU acceleration, etc.); furthermore, from the source code installation we can use some additional functions. OpenCV currently maintains two major versions, divided into OpenCV 2.4 series and OpenCV 3 series *. This book uses the OpenCV **3** series.

Because the OpenCV project is relatively large, it will not be placed under 3rd-party in this book. Readers can download it from `http://opencv.org/downloads.html` and select the OpenCV for Linux version. You will get a compressed package like opencv-3.1.0.zip. Unzip it to any directory, we found that OpenCV is also a cmake project.

Before compiling, first install the dependencies of OpenCV:

Listing 5.3: Terminal input:

```
sudo apt-get install build-essential libgtk2.0-dev libvtk5-dev libjpeg-dev
    libtiff4-dev libjasper-dev libopenexr-dev libtbb-dev
```

In fact, OpenCV has many dependencies, and the lack of certain compilation items will affect some of its functions (but we will not use all the functions). OpenCV will check whether the dependencies will be installed during cmake and adjust its own functions. If you have a GPU on your computer and the relevant dependencies are installed, OpenCV will also enable GPU acceleration. But for this book, the above dependencies are sufficient.

Subsequent compilation and installation are the same as ordinary cmake projects. After make, please call “sudo make install” to install OpenCV on your machine (instead of just compiling it). Depending on the machine configuration, this compilation process may take from 20 minutes to an hour. If your CPU is strong, you can use commands like ”make -j4” to call multiple threads to compile (the parameter after -j is the number of threads used). After installation, OpenCV is stored in the /usr/local directory by default. You can look for the installation location of OpenCV header files and library files to see where they are. In addition, if you have installed the OpenCV 2 series before, it is recommended that you install OpenCV 3 elsewhere (think about how this should be done).

* In 2020 we can also use version 4.0 or higher.

Bibliography

[1] A. Davison, I. Reid, N. Molton, and O. Stasse, "Monoslam: Real-time single camera SLAM," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 1052–1067, 2007.

[2] R. C. Smith and P. Cheeseman, "On the representation and estimation of spatial uncertainty," *International Journal of Robotics Research*, vol. 5, no. 4, pp. 56–68, 1986.

[3] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT Press, 2005.

[4] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge university press, 2003.

[5] A. Pretto, E. Menegatti, and E. Pagello, "Omnidirectional dense large-scale mapping and navigation based on meaningful triangulation," *2011 IEEE International Conference on Robotics and Automation (ICRA 2011)*, pp. 3289–96, 2011.

[6] B. Rueckauer and T. Delbruck, "Evaluation of event-based algorithms for optical flow with ground-truth from inertial measurement sensor," *Frontiers in neuroscience*, vol. 10, 2016.

[7] C. Cesar, L. Carlone, H. C., Y. Latif, D. Scaramuzza, J. Neira, I. D. Reid, and L. John J., "Past, present, and future of simultaneous localization and mapping: Towards the robust-perception age," *arXiv preprint arXiv:1606.05830*, 2016.

[8] R. Smith, M. Self, and P. Cheeseman, "Estimating uncertain spatial relationships in robotics," in *Autonomous robot vehicles*, pp. 167–193, Springer, 1990.

[9] H. Strasdat, J. M. Montiel, and A. J. Davison, "Visual slam: Why filter?," *Image and Vision Computing*, vol. 30, no. 2, pp. 65–77, 2012.

[10] L. Haomin, Z. Guofeng, and B. Hujun, "A survey of monocular simultaneous localization and mapping," *Journal of Computer-Aided Design and Compute Graphics*, vol. 28, no. 6, pp. 855–868, 2016. in Chinese.

[11] M. Liang, H. Min, and R. Luo, "Graph-based slam: A survey," *ROBOT*, vol. 35, no. 4, pp. 500–512, 2013. in Chinese.

[12] J. Fuentes-Pacheco, J. Ruiz-Ascencio, and J. M. Rendón-Mancha, "Visual simultaneous localization and mapping: a survey," *Artificial Intelligence Review*, vol. 43, no. 1, pp. 55–81, 2015.

[13] J. Boal, Á. Sánchez-Miralles, and Á. Arranz, "Topological simultaneous localization and mapping: a survey," *Robotica*, vol. 32, pp. 803–821, 2014.

[14] S. Y. Chen, "Kalman filter for robot vision: A survey," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 11, pp. 4409–4420, 2012.

[15] Z. Chen, J. Samarabandu, and R. Rodrigo, "Recent advances in simultaneous localization and map-building using computer vision," *Advanced Robotics*, vol. 21, no. 3-4, pp. 233–265, 2007.

[16] J. Stuelpnagel, "On the parametrization of the three-dimensional rotation group," *SIAM Review*, vol. 6, no. 4, pp. 422–430, 1964.

[17] T. Barfoot, J. R. Forbes, and P. T. Furgale, "Pose estimation using linearized rotations and quaternion algebra," *Acta Astronautica*, vol. 68, no. 1-2, pp. 101–112, 2011.

[18] V. S. Varadarajan, *Lie groups, Lie algebras, and their representations*, vol. 102. Springer Science & Business Media, 2013.

[19] Y. Ma, S. Soatto, J. Kosecka, and S. S. Sastry, *An invitation to 3-d vision: from images to geometric models*, vol. 26. Springer Science & Business Media, 2012.

[20] T. Barfoot, "State estimation for robotics: A matrix lie group approach," 2016.

[21] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A benchmark for the evaluation of rgb-d SLAM systems," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 573–580, IEEE, 2012.

[22] H. Strasdat, *Local accuracy and global consistency for efficient visual slam*. PhD thesis, Citeseer, 2012.

[23] Z. Zhang, "Flexible camera calibration by viewing a plane from unknown orientations," in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 1, pp. 666–673, Ieee, 1999.