14 Lectures on Visual SLAM
From Theory to Practice

Xiang Gao, Tao Zhang, Qinrui Yan and Yi Liu

February 1, 2021

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 publicly available to help the readers. I encountered some issues with the math equations on 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 the 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 afterward modifications. 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 humans, watching and observing the world, and understanding the surrounding environment. The ability to explore unknown areas is a wonderful and romantic dream, attracting numerous researchers striving on this problem day and night [2]. 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 humans 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 humans. Researchers have successfully built a variety of real-time SLAM systems. Some of them can efficiently track their 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 the 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, self-driving cars, Unmanned Aerial Vehicles (UAVs), 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 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 the 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 a 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 several 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 real-applications. 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 [3], 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 first introduce 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 we aim 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" [4], "Multiple View Geometry in Computer Vision" [2], "State Estimation for Robotics: A Matrix-Lie-Group Approach"[5], etc. They provide rich content, 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 a 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 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 the 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 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 a 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 the 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 your 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 the 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 treemap 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, keyframe 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 in which we 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 the 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 a 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 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 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 the 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 detail 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 publicly available and open-source.

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 a 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 a one-dimensional case? How about in a 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 hands).

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 the 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 a 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, for now, do not try to turn Vim into an IDE, 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. The 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:

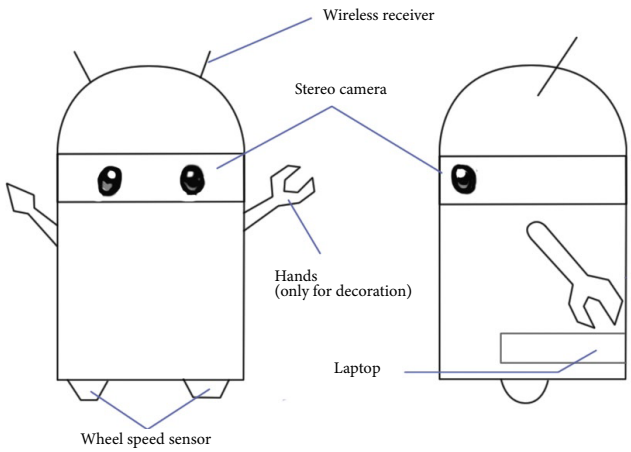


Figure 2-1: The sketch of robot *Little Carrot*

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 considered 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, humans 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 paste 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.

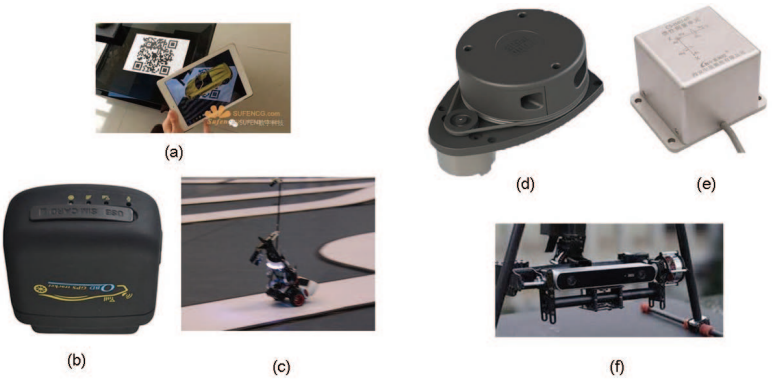


Figure 2-2: Different kinds of sensors: (a) QR code (b) GNSS receiver (c) guiding rails (d) Laser range finder (e) Inertial measurement unit (f) stereo camera

The first class is *non-intrusive* sensors which are completely self-contained inside a robot, such as wheel encoders, cameras, laser scanners, etc. They do not assume a cooperative environment around the robot. The other class is *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 in the environment, the scope of usage is often limited to 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, it does 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 location. 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 observes 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 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 Single Lens Reflex (SLR) cameras. It is often much simpler and does not carry an 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 an 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 devices 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 some special and emerging camera types that can be applied to SLAM, such as panorama camera [6], event camera [7]. Although they are occasionally seen in SLAM applications, so far they have not become 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 types of cameras 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, like 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. Evidently, 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 humans 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 people 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 a 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 known, 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 the 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 a 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.

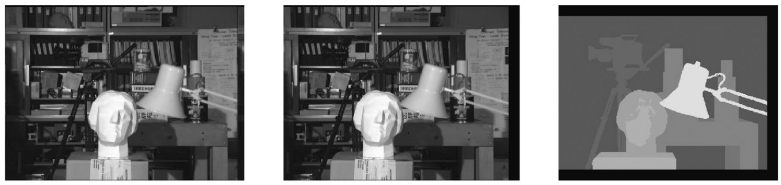


Figure 2-5: Distance is calculated from the disparity of two stereo image pair.

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 many computational resources and usually requires GPU or FPGA to accelerate in order to generate real-time depth maps. Therefore, in most state-of-the-art algorithms, the 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 new type of camera 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 many computational resources 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, susceptibility to sunlight interference, and unable to measure transparent material. For SLAM purposes, RGB-D cameras are mainly used in indoor environments and are not suitable for outdoor applications.

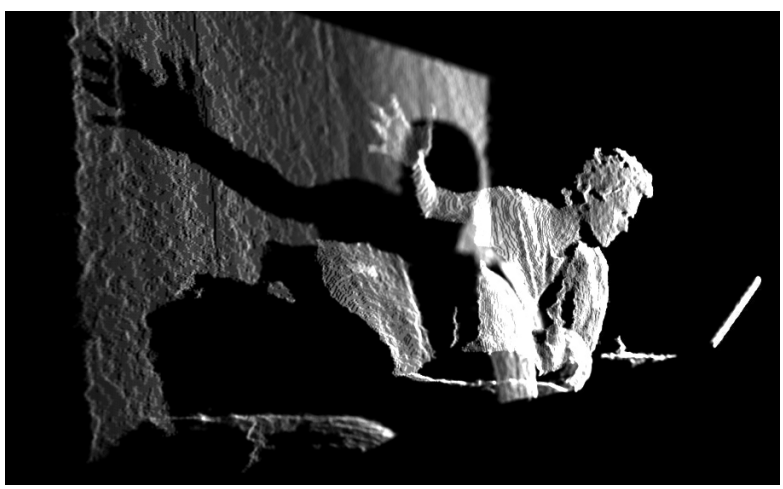


Figure 2-6: RGBD cameras measure the distance and can build a point cloud with a single image frame.

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 an as simple task as you would think. It is not a single algorithm that continuously outputs 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 Classic Visual SLAM Framework

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

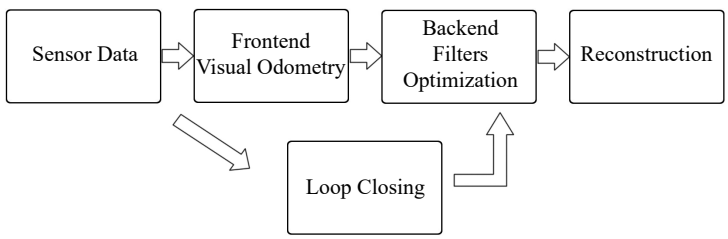


Figure 2-7: The classic visual SLAM framework.

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 of 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, the visual SLAM problem is basically solved [8].

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, a 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 the 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 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 a 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 an "odometry", because similar to actual wheel odometry which only calculates the ego-motion at neighboring moments, and does not estimate a global map or an absolute pose. In this regard, VO is like a species with only a short memory.

Now, assuming that we have 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 a map. Up to here, it seems with a VO, the SLAM problem is already solved. Or, is it?

Visual odometry is indeed a key technology for solving the 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 of 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.

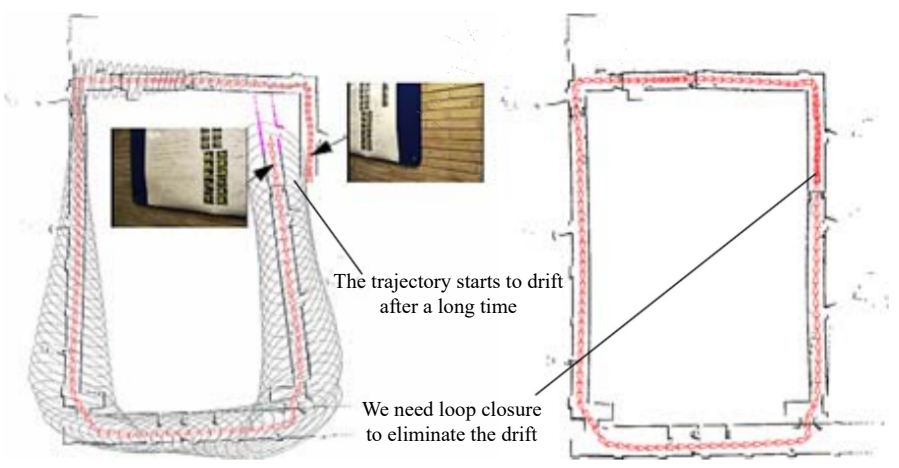


Figure 2-9: Drift will be accumulated if we only have a relative motion estimation.

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 a certain amount of noise. Cheap sensors usually have larger measurement errors, while that of expensive ones may be small. Moreover, the performance of many sensors is affected by changes in the 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 it is carried forward from the last time step to the next, and how confident we have in 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 to 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 matrices in the backend without those beautiful 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, the 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" [3, 9]. 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 a 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 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 that 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, the 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 a 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 environment. 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

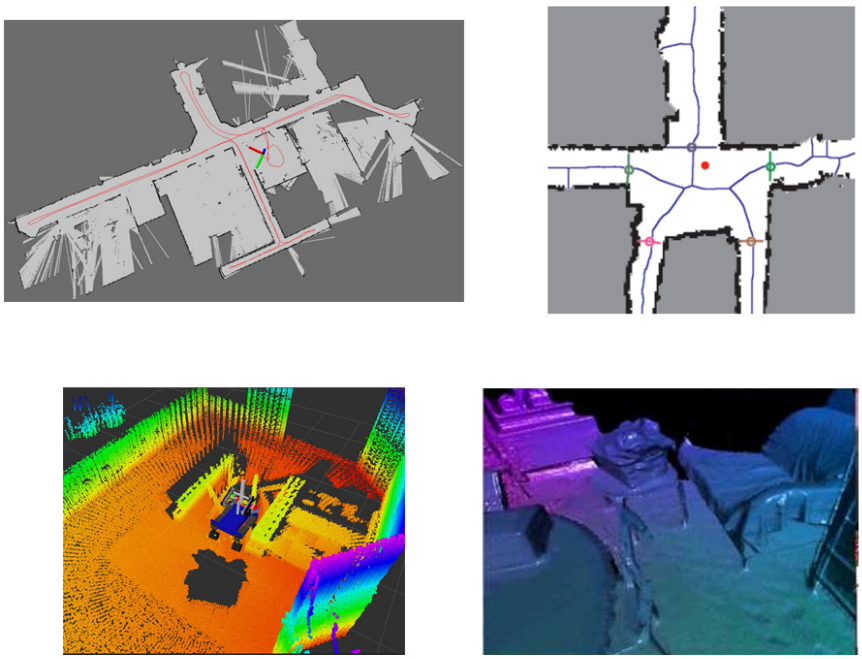


Figure 2-10: Different kinds of maps: 2D grid map, 2D topological map, 3D point clouds and 3D meshes.

focus on modeling all the things that are seen. For localization, a 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, a 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 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 attract 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 of 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 expression. However, topological maps are not good at representing maps with

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

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 runnable 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 time points, we are only concerned with the locations and the 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 is a total of 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 $\mathbf{x}$ has changed from time step $k-1$ to k.
2. What are the sensor *observations*? Assuming that the Little Carrot detects a certain landmark, let's 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 messages to the robots like "turn 15 degrees to left". These messages or orders will be finally carried out by the controller, but probably in many 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 an order as "move forward one meter", 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 maybe a hundred miles away from the actual position after several minutes.

Corresponding to the motion equation, there is also an *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 r between the landmark point and the robot, 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:

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

When considering 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 **x** (localization) and **y** (mapping) problem with the noisy control input **u** and the sensor reading **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 [10]. 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 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 several 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 by 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

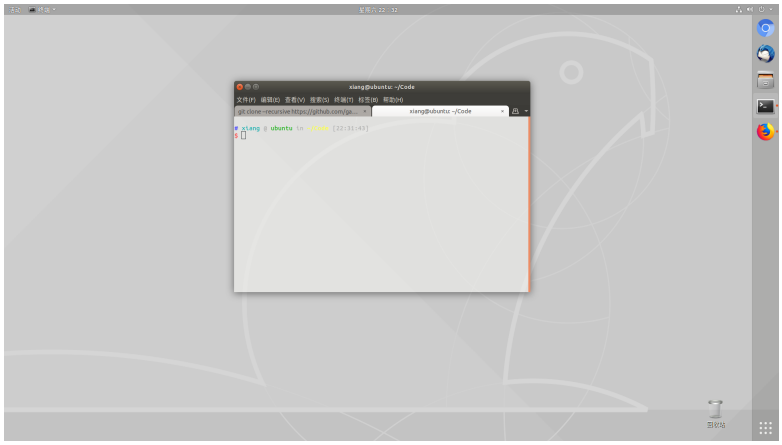


Figure 2-11: Ubuntu 1804 in virtual machine.

(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 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 a .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 that 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 executed 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.

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

* Don't type the first %.

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
```

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

* 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.

```
% ./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 the 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 good management for the project: **from inputting a string of g++ commands to maintaining several relatively intuitive "CMakeLists.txt" files**, which will drastically 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 codes, 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 function 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 the 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 function 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"
```

```
2
3 // Call printHello() in libHelloSLAM.h
4 int main(int argc, char **argv) {
5     printHello();
6     return 0;
7 }
```

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

Listing 2.16: slambook2/ch2/CMakeLists.txt

```
1 add_executable( useHello useHello.cpp )
2 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)*.

* 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.

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.

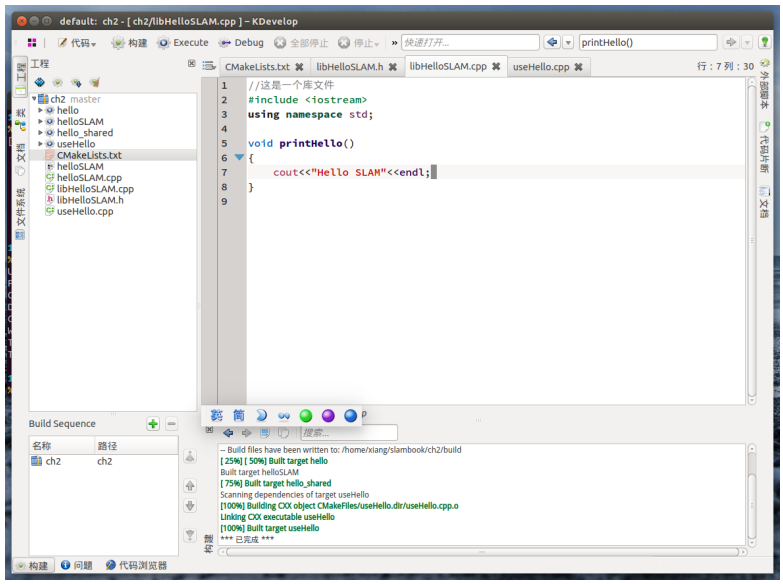


Figure 2-12: Kdevelop in Ubuntu.

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 the "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 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.

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 a 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.

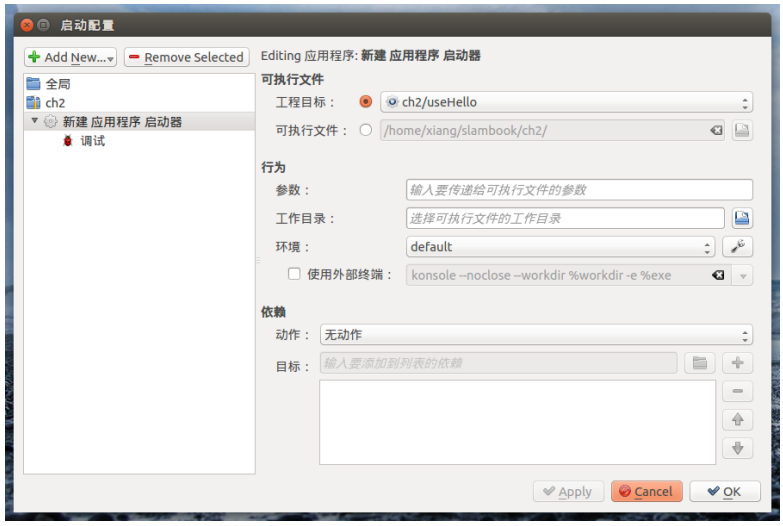


Figure 2-13: Config launches. We can choose a launch target and set parameters here.

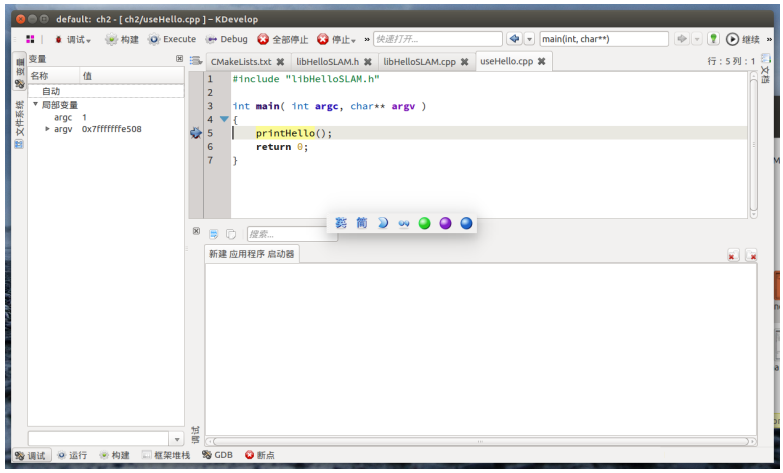


Figure 2-14: Debug interface.

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 *.

Use Clion

Clion is more complete than KDevelop, but it requires a user account, and the memory/CPU requirements for the host will be higher. †. In Clion, you can also

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

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

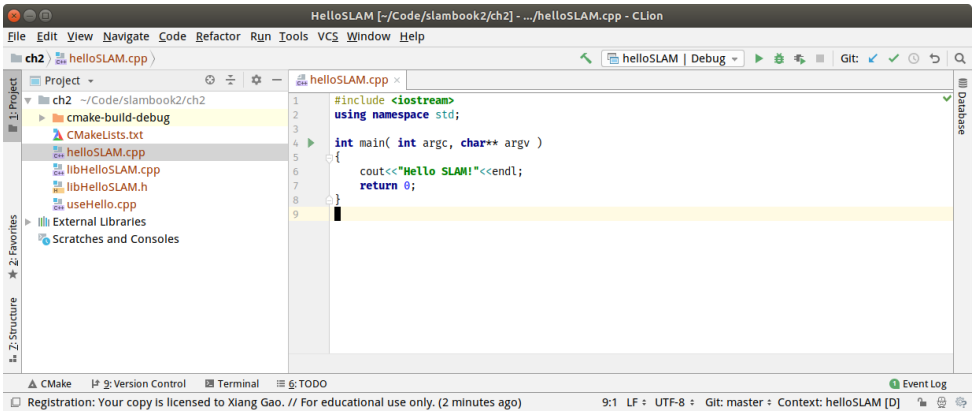


Figure 2-15: Clion interface

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 several 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 [11] and [12], 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 [8, 13–16] and so on. What are the similarities and differences between these papers on SLAM and this book?

3. What are the parameters of the g++ command? If I want to change the generated program file name, how should I call g++?

4. Use the build folder to compile your CMake project, then try it in KDevelop.

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

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

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

8.* 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.

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

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

11. 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 rule is opposite to the right-hand rule. Most 3D libraries use right-handed coordinates (such as OpenGL, 3D Max, etc.), and some libraries use left-handed coordinates (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 a 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 an 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 setting 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 $\mathbf{a}$ 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 an **Euclidean Transform**.

The Euclidean transform consists of rotation and translation. Let's first consider the rotation. We have a 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 these 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.

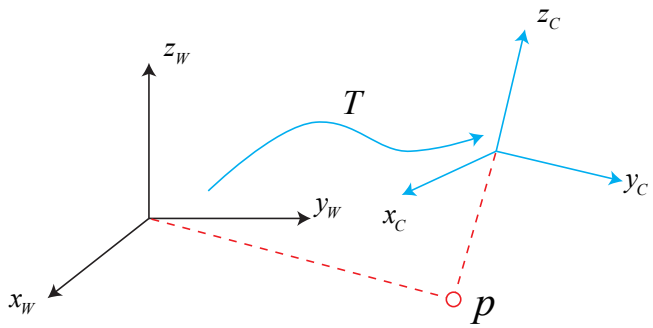


Figure 3-1: Coordinate transformation. For the same vector $\mathbf{p}$, its coordinates in the world $\mathbf{p}_W$ and the coordinates in the camera system $\mathbf{p}_C$ are different. This transformation relationship is described by the transform matrix $\mathbf{T}$.

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

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 a 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 of 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 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 later sections. For example, when we write $\mathbf{Ta}$, we use homogeneous coordinates (otherwise we can't calculate). When you write $\mathbf{Ra}$, 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
#include <Eigen/Core>
```

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

```cpp
// 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;
    cout << "[1,2,3;4,5,6]*[3,2,1]=" << result.transpose() << endl;

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

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

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

```
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();
119     x = matrix_NN.ldlt().solve(v_Nd);
120     cout << "time of ldlt decomposition is "
121     << 1000 * (clock() - time_stt) / (double)CLOCKS_PER_SEC << "ms" << endl;
122     cout << "x = " << x.transpose() << endl;
123
124     return 0;
125 }
```

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

```
1 # Add header file
2 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:

```
1 % build/eigenMatrix
2 matrix 2x3 from 1 to 6:
3 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' 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:

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

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

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

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

* 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.

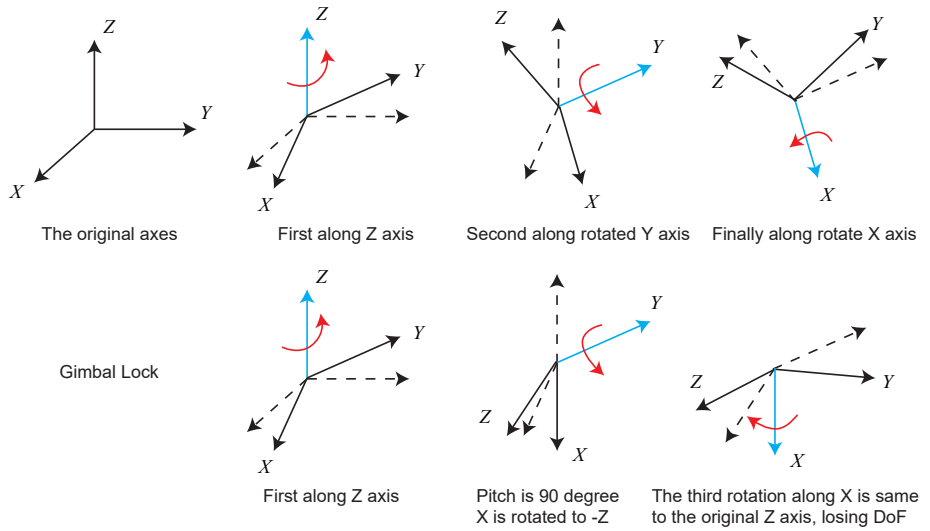


Figure 3-2: Euler angles. The top is defined for the ZYX order (rpy order). The bottoms shows when pitch=90°, the third rotation is using the same axis as the first one, causing the system to lose a degree of freedom. If you don't understand the universal lock, please take a look at the related videos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

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 $[r, p, y]^{\text{T}}$ 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.

A major drawback of Euler Angle is that it encounters the famous **Gimbal Lock** *): in *rpy*'s case, when the pitch angle is $\pm 90°$, 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

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

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 singularity. In fact, we **cannot find a three-dimensional vector description without singularity**[17]. 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 a 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

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

parts satisfy the following relationship:

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

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 by 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:

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

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:

$$\mathbf{q}_a \mathbf{q}_b = \begin{bmatrix} s_a s_b - \mathbf{v}_a^{\mathrm{T}} \mathbf{v}_b, s_a \mathbf{v}_b + s_b \mathbf{v}_a + \mathbf{v}_a \times \mathbf{v}_b \end{bmatrix}^{\mathrm{T}}. \tag{3.24}$$

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 lengths. 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:

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

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 extend the 3D point to an 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 [18]:

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

These two symbols map the quaternion to a matrix of 4×4. Then the 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 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, *and* z of a vector after it is rotated. Due to the scaling, the similar transformation no longer keeps the volume 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.

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,

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

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
```

```
26   v_rotated = rotation_matrix * v;
27   cout << "(1,0,0) after rotation (by matrix) = " << v_rotated.transpose() << endl;
28
29   // Euler angle: You can convert the rotation matrix directly into Euler angles
30   Vector3d euler_angles = rotation_matrix.eulerAngles(2, 1, 0); // ZYX order, ie roll
         pitch yaw order
31   cout << "yaw pitch roll = " << euler_angles.transpose() << endl;
32
33   // Euclidean transformation matrix using Eigen::Isometry
34   Isometry3d T = Isometry3d::Identity(); // Although called 3d, it is essentially a
         4*4 matrix
35   T.rotate(rotation_vector); // Rotate according to rotation_vector
36   T.pretranslate(Vector3d(1, 3, 4)); // Set the translation vector to (1,3,4)
37   cout << "Transform matrix = \n" << T.matrix() << endl;
38
39   // Use the transformation matrix for coordinate transformation
40   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.txt" 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 we 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 you are consistent with the version we 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";

```

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

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

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

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

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

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

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

  pangolin::View &d_cam = pangolin::CreateDisplay()
  .SetBounds(0.0, 1.0, 0.0, 1.0, -1024.0f / 768.0f)
  .SetHandler(new pangolin::Handler3D(s_cam));

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

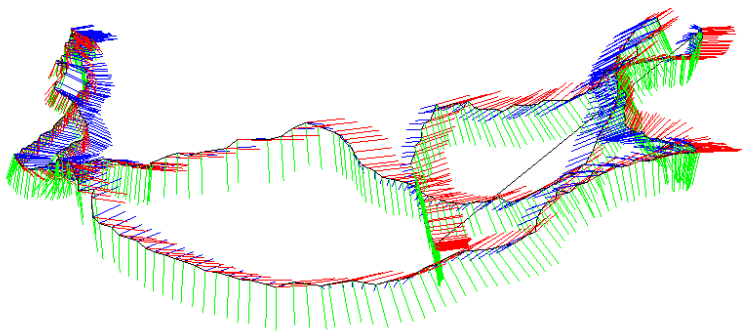


Figure 3-3: Results of pose visualization

3.7.2 Displaying Camera Pose

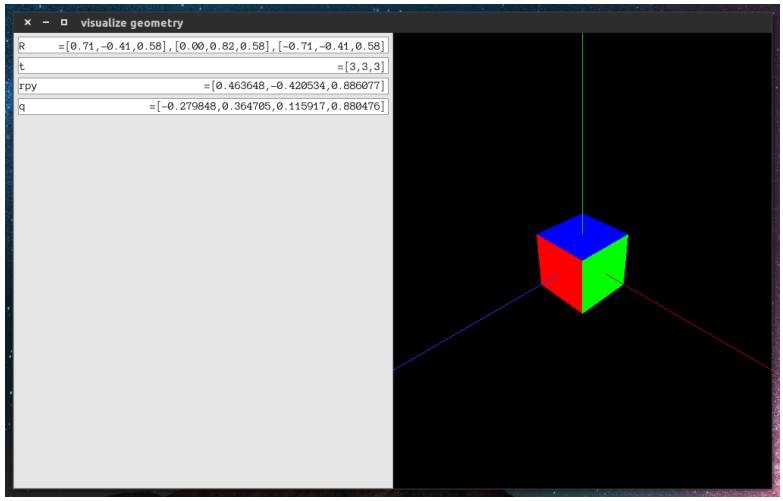


Figure 3-4: Visualization program for rotation matrix, Euler angle, quaternion.

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 $\mathbf{I}_{3\times3}$. Please implement it in C++.
6. When does a general linear equation $\mathbf{Ax} = \mathbf{b}$ has a 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. Understand the meaning of BCH (Baker-Campbell-Hausdorff) 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 focused 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 a little bit about 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 this text is not a mathematics book, 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 [19].

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 present to the reader that each Lie group corresponds to a Lie algebra, but in this case, the reader may 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 [20]. 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 other control books):

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

Move the second term to right and commute the matrices by using the transposed relation:

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

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 the cross product in the formula (3.3), which

turns a vector into a 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 a 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 $t_0 = 0$ that 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 Composition**: $\forall \mathbf{X}, \mathbf{Y}, \mathbf{Z} \in \mathbb{V}; a, b \in \mathbb{F}$, 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 Identity**: $\forall \mathbf{X}, \mathbf{Y}, \mathbf{Z} \in \mathbb{V}; [\mathbf{X}, [\mathbf{Y}, \mathbf{Z}]] + [\mathbf{Z}, [\mathbf{X}, \mathbf{Y}]] + [\mathbf{Y}, [\mathbf{Z}, \mathbf{X}]] = \mathbf{0}$.

The binary operations $[,]$ are called **Lie brackets**. At 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 a 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 bracket is:

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

The reader can verify that the Lie bracket 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, we 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 converted to a four-dimensional matrix also 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 $\exp(\boldsymbol{\phi}^\wedge)$? In other words, 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 has 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 $\mathbf{n}$, respectively. So we have $\boldsymbol{\phi} = \theta\mathbf{n}$, where $\mathbf{n}$ is a unit-length direction vector, i.e., $\|\mathbf{n}\| = 1$. First, for such a unit-length vector $\mathbf{n}$, there are two properties:

$$\mathbf{n}^\wedge \mathbf{n}^\wedge = \begin{bmatrix} -n_2^2 - n_3^2 & n_1 n_2 & n_1 n_3 \\ n_1 n_2 & -n_1^2 - n_3^2 & n_2 n_3 \\ n_1 n_3 & n_2 n_3 & -n_1^2 - n_2^2 \end{bmatrix} = \mathbf{n}\mathbf{n}^\mathrm{T} - \mathbf{I}, \tag{4.20}$$

as well as

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

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

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

Finally, we got a very familiar equation:

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

Recall the previous lesson, this equation is exactly the same as the Rodrigues' formula, i.e. equation (3.15). This shows that $\mathfrak{so}(3)$ is actually the **rotation vector**, and the exponential map is just the Rodrigues' 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 have to use Taylor series 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 **J**. Please pay attention to the **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 **T**: from the upper left corner **R** we can calculate the rotation vector, while **t** the upper right corner satisfies:

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

Since **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.

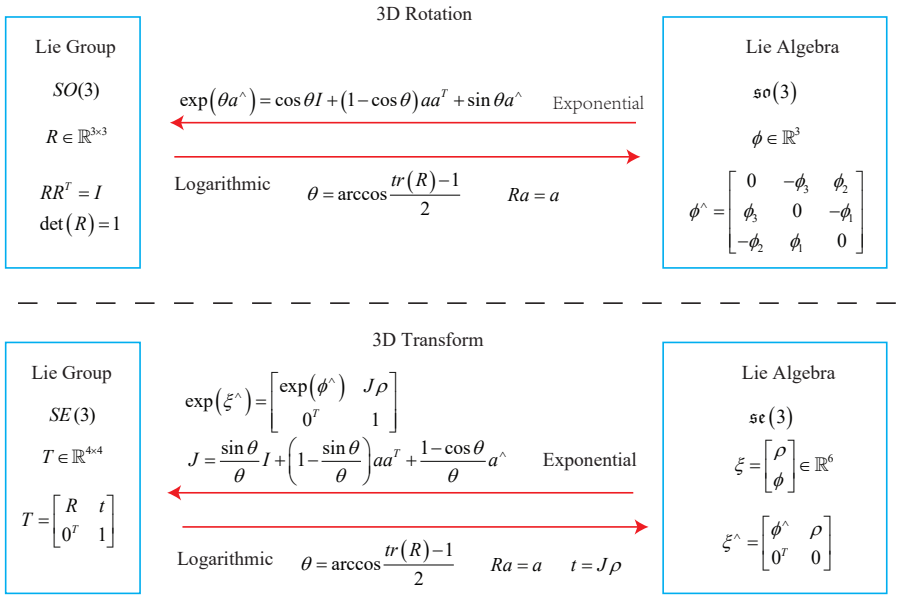


Figure 4-1: The correspondence between $\mathrm{SO}(3), \mathrm{SE}(3), \mathfrak{so}(3), \mathfrak{se}(3)$.

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 [5] 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 [5]. 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:

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

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;
```

```
60 }
```

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 SO(3), 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

```
1 # we use find_package to make CMake find sophus
2 find_package( Sophus REQUIRED )
3 include_directories( ${Sophus_INCLUDE_DIRS} ) # sohpus is header only
4
5 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 $\mathbf{T}_{\text{esti},i}$ and the real trajectory $\mathbf{T}_{\text{gt},i}$, 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:

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

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**:

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

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.

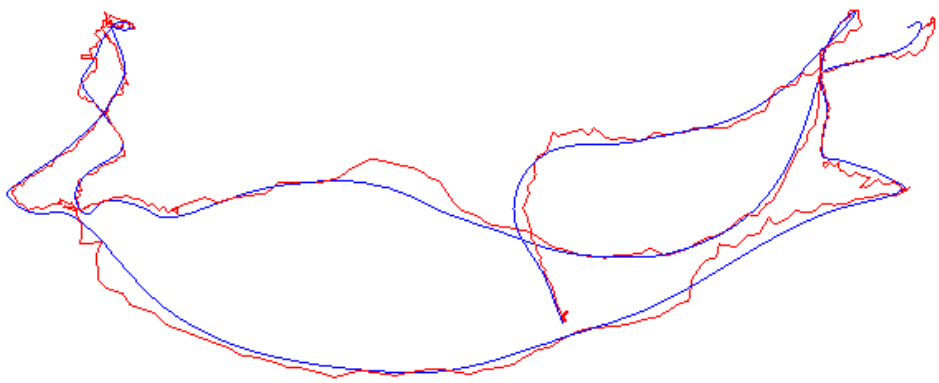


Figure 4-2: Calculates the error between the estimated trajectory and the real trajectory.

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 similarity 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.

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

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:

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

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 $\boldsymbol{\mathcal{J}}$ 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 **S** after a similar transformation of **Sp** relative to **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 **Sp** a small perturbation $\exp(\boldsymbol{\zeta}^\wedge)$ on the left and ask for **Sp** The derivative of the disturbance. Since **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 **q** of **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:
$$\mathbf{R}\mathbf{p}^{\wedge}\mathbf{R}^{\mathrm{T}} = (\mathbf{R}\mathbf{p})^{\wedge}.$$
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.

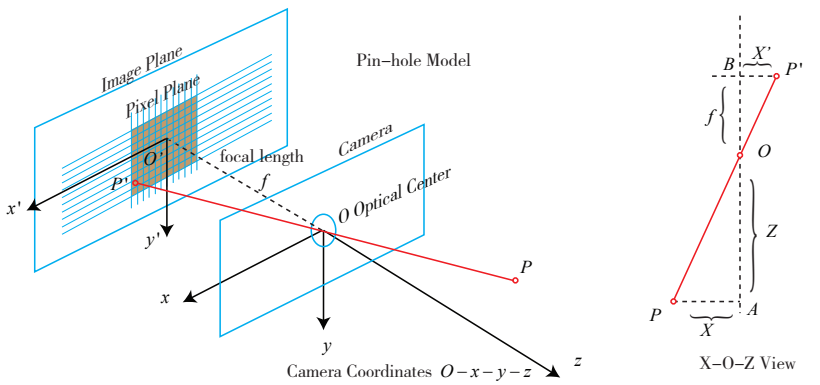


Figure 5-1: Pinhole camera model.

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:

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

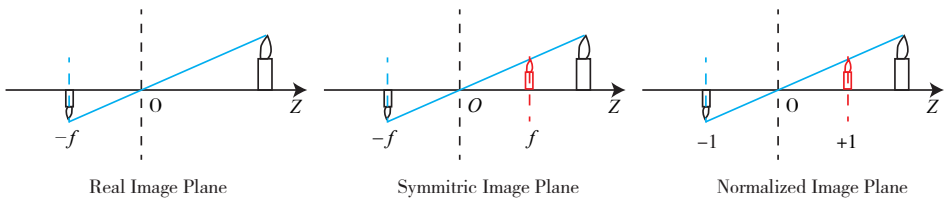


Figure 5-2: The real, symmetric and normalized image plane.

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 **TP**, then convert to non-homogeneous coordinates, and then multiply it by **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".

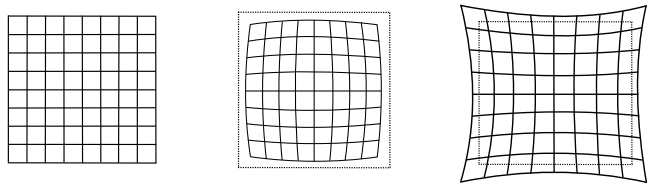


Figure 5-3: The radical distortion.

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

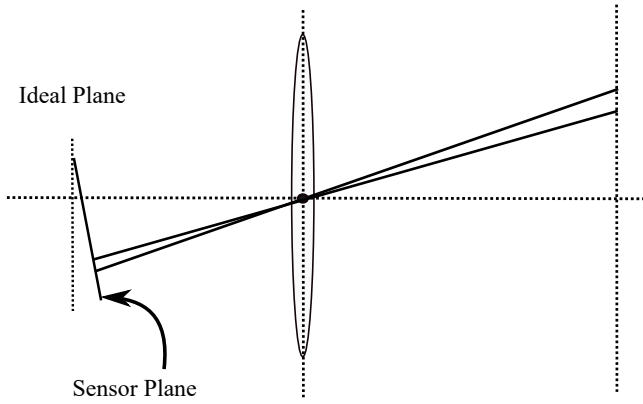


Figure 5-4: Tangential distortion.

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 the 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.

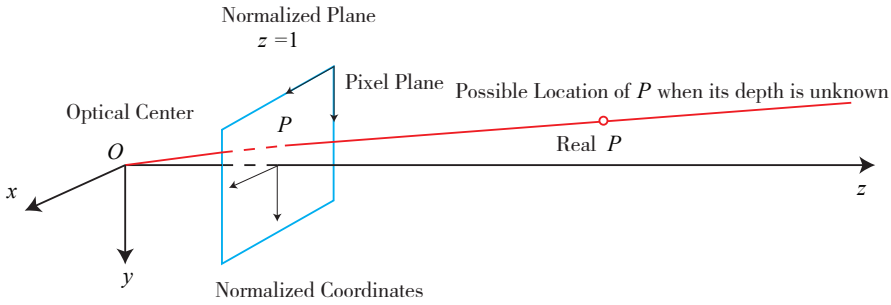


Figure 5-5: The possible location of a single pixel.

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.

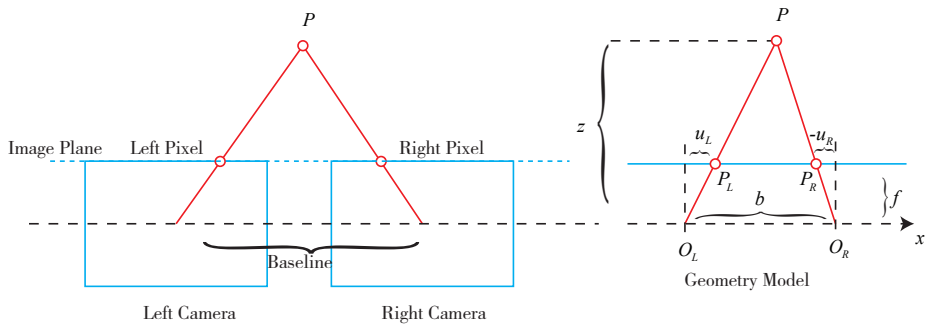


Figure 5-6: Geometry model of stereo cameras from upside down. The O_L, O_R are left and right optical centers. f is the focal length, u_L and u_R are pixel corrdinates of a same point along x axis. Note that u_R should be a negative value in this figure, so the pysical distance should be $-u_R$.

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 the 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.

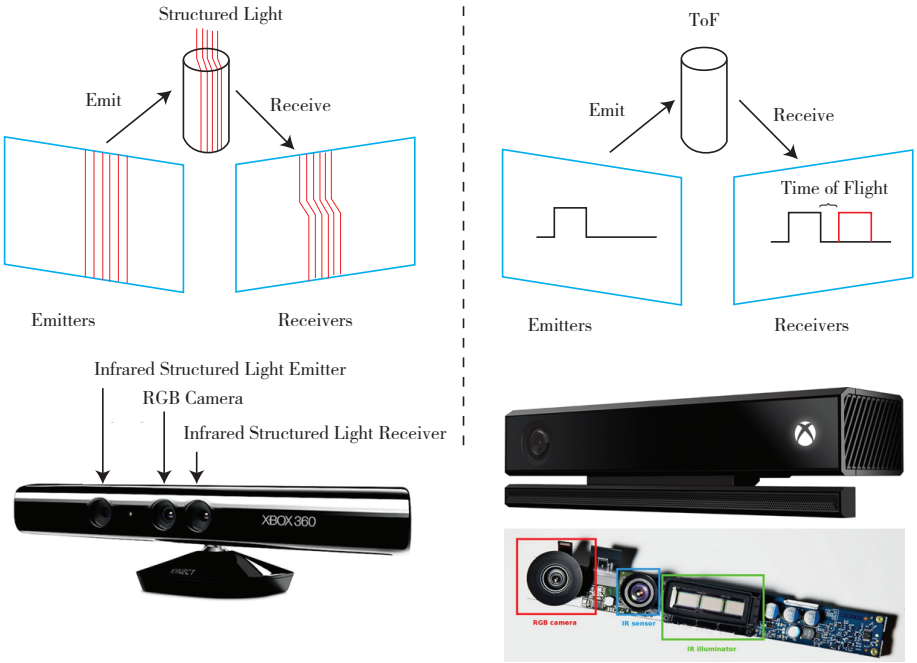


Figure 5-7: RGB-D Cameras

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)

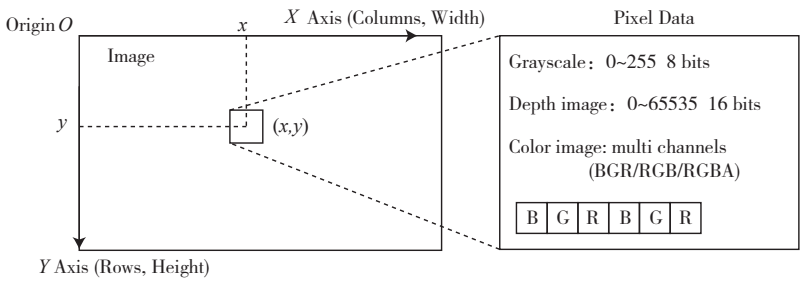


Figure 5-8: Pixels in an image.

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 a 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 environment (for example, whether you need GPU acceleration, etc.); furthermore,

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

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 powerful, 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).

5.3.2 Basic OpenCV Images Operations

Now let's go through the basic image operations in OpenCV from a simple example.

Listing 5.4: slambook/ch5/imageBasics/imageBasics.cpp

```
#include <iostream>
#include <chrono>

using namespace std;

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

int main(int argc, char **argv) {
    // Read the image in argv[1]
    cv::Mat image;
    image = cv::imread(argv[1]); // call cv::imread to read the image from file

    // check the data is correctly loaded
    if (image.data == nullptr) { // maybe the file does not exist
        cerr << "file" << argv[1] << " not exist." << endl;
        return 0;
    }

    // print some basic information
```

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

```
21      cout << "Image cols: " << image.cols << ", rows: " << image.rows
22      << ", channels: " << image.channels() << endl;
23      cv::imshow("image", image);      // use cv::imshow to show the image
24      cv::waitKey(0);                  // display and wait for a keyboard input
25
26      // check image type
27      if (image.type() != CV_8UC1 && image.type() != CV_8UC3) {
28          // we need grayscale image or RGB image
29          cout << "image type incorrect." << endl;
30          return 0;
31      }
32
33      // check hte pixels
34      chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
35      for (size_t y = 0; y < image.rows; y++) {
36          // use cv::Mat::ptr to get the pointer of each row
37          unsigned char *row_ptr = image.ptr<unsigned char>(y);  // row_ptr is the
                 pointer to y-th row
38          for (size_t x = 0; x < image.cols; x++) {
39              // read the pixel on (x,y), x=column, y=row
40              unsigned char *data_ptr = &row_ptr[x * image.channels()]; // data_ptr is
                     the pointer to (x,y)
41              // visit the pixel in each channel
42              for (int c = 0; c != image.channels(); c++) {
43                  unsigned char data = data_ptr[c]; // data should be pixel of I(x,y) in
                         c-th channel
44              }
45          }
46      }
47      chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
48      chrono::duration<double> time_used = chrono::duration_cast < chrono::duration <
             double >> (t2 - t1);
49      cout << "time used: " << time_used.count() << " seconds." << endl;
50
51      // copying cv::Mat
52      // operator = will not copy the image data, but only the reference
53      cv::Mat image_another = image;
54      // changing image_another will also change image
55      image_another(cv::Rect(0, 0, 100, 100)).setTo(0); // set top-left 100*100 block to
            zero
56      cv::imshow("image", image);
57      cv::waitKey(0);
58
59      // use cv::Mat::clone to actually clone the data
60      cv::Mat image_clone = image.clone();
61      image_clone(cv::Rect(0, 0, 100, 100)).setTo(255);
62      cv::imshow("image", image);
63      cv::imshow("image_clone", image_clone);
64      cv::waitKey(0);
65
66      // We are not going to copy the OpenCV's documentation here
67      // please take a look at it for other image operations like clipping, rotating and
            scaling.
68
69      cv::destroyAllWindows();
70      return 0;
71  }
```

In this example, we demonstrated the following operations: image reading, displaying, pixel vising, copying, assignment, etc. When compiling the program, you need to add the OpenCV header file in your "CMakeLists.txt", and then link the program to the OpenVC's library. At the same time, due to the use of C ++ 11 standards (such as nullptr and chrono), you also need to set up the c++ standard in compiler flag:

Listing 5.5: slambook/ch5/imageBasics/CMakeLists.txt

```
1  # use c++11 standard
2  set( CMAKE_CXX_FLAGS "-std=c++11" )
3
4  # find OpenCV
5  find_package( OpenCV REQUIRED )
```

```
6  # include its headers
7  include_directories( ${OpenCV_INCLUDE_DIRS} )
8
9  add_executable( imageBasics imageBasics.cpp )
10
11 # link the exe to opencv's libs
12 target_link_libraries( imageBasics ${OpenCV_LIBS} )
```

Let's give some notes for the code:

1. The program reads the image position from argv[1], the first parameter on the command line. We prepared an image (ubuntu.png, an Ubuntu wallpaper, I hope you like it) for readers to test. Therefore, after compilation, use the following command to call this program:

   Listing 5.6: Terminal input:

   ```
   1 ./build/imageBasics ubuntu.png
   ```

   If you call this program in the IDE, be sure to give it parameters at the same time. This can be configured in the launch configuration dialog if you are using Clion.

2. In line 10 ~to 18, use the cv::imread function to read the image, and display the image and basic information.

3. In line 35 ~46, we iterate over all pixels in the image and calculates the time spent in the entire loop. Please note that the pixel visiting method is not unique, and the method given by the example is not the most efficient way. OpenCV provides an iterator of cv::Mat, you can traverse the pixels of the image through the iterator. Or, cv::Mat::data provides a raw pointer to the beginning of the image data, you can directly calculate the offset through this pointer, and then get the actual memory location of the pixel. The method used in the example is to facilitate the reader to understand the structure of the image.

   On the author's machine (virtual machine), it takes about 12.74ms to traverse this image. You can compare the speed on your machine. However, we are using the default debug mode of CMake, which is much slower than the release mode.

4. OpenCV provides many functions for manipulating images, we will not list them one by one, otherwise this book will become an OpenCV operation manual. The example shows the most common things like image reading and displaying, as well as the deep copy function in cv::Mat. During the programming process, readers will also encounter operations such as image rotation and interpolation. At this time, you should refer to the corresponding documentation of the function to understand their principles and usage.

It should be noted that OpenCV is not the only image library, it is just one of the more widely used one. However, most image libraries have the similar image operations. We hope that readers can understand the representation of images in other libraries after understanding the representation of images in OpenCV, so that they can quickly adjust to any other libraries. In addition, since cv::Mat is also a matrix class, in addition to representing images, we can also use it to store matrix data such as rotation matrix. But it is generally believed that Eigen is more efficient for use with fixed-size matrices.

5.3.3 Image Undistortion

We've introduced the rad-tan distortion model in previous section, now let write an example to show the implementation. OpenCV has provided the cv::Undistort function for us, but we will also give a hand-written undistortion function to show the principles.

Listing 5.7: slambook/ch5/imageBasics/undistortImage.cpp

```
#include <opencv2/opencv.hpp>
#include <string>
using namespace std;
string image_file = "./distorted.png";   // the distorted image

int main(int argc, char **argv) {
    // In thie program we implement the undistortion by ourselves rather than using
         opencv
    // rad-tan model params
    double k1 = -0.28340811, k2 = 0.07395907, p1 = 0.00019359, p2 = 1.76187114e-05;
    // intrinsics
    double fx = 458.654, fy = 457.296, cx = 367.215, cy = 248.375;

    cv::Mat image = cv::imread(image_file, 0);   // the image type is CV_8UC1
    int rows = image.rows, cols = image.cols;
    cv::Mat image_undistort = cv::Mat(rows, cols, CV_8UC1);   // the undistorted image

    // computate the pixels in the undistorted one
    for (int v = 0; v < rows; v++) {
        for (int u = 0; u < cols; u++) {
            // note we are computing the pixel of (u,v) in the undistorted image
            // according to the rad-tan model, compute the coordinates in the
                 distorted image
            double x = (u - cx) / fx, y = (v - cy) / fy;
            double r = sqrt(x * x + y * y);
            double x_distorted = x * (1 + k1 * r * r + k2 * r * r * r * r * r) + 2 * p1 *
                 x * y + p2 * (r * r + 2 * x * x);
            double y_distorted = y * (1 + k1 * r * r + k2 * r * r * r * r * r) + p1 * (r *
                  r + 2 * y * y) + 2 * p2 * x * y;
            double u_distorted = fx * x_distorted + cx;
            double v_distorted = fy * y_distorted + cy;

            // check if the pixel is in the image boarder
            if (u_distorted >= 0 && v_distorted >= 0 && u_distorted < cols &&
                 v_distorted < rows) {
                image_undistort.at<uchar>(v, u) = image.at<uchar>((int) v_distorted, (
                     int) u_distorted);
            } else {
                image_undistort.at<uchar>(v, u) = 0;
            }
        }
    }

    // show the undistorted image
    cv::imshow("distorted", image);
    cv::imshow("undistorted", image_undistort);
    cv::waitKey();
    return 0;
}
```

Please check the difference between two images by yourself.

5.4 Practice: 3D Vision

5.4.1 Stereo Vision

We have introduced the imaging principle of stereo vision. Now we start from the left and right images, calculate the disparity map corresponding to the left eye, and then calculate the coordinates of each pixel in the camera coordinate system, which

will form a **point cloud**. We have prepared left and right images for the readers, as shown in Figure 5-9. The following code demonstrates the calculation of disparity map and point cloud:

Listing 5.8: slambook/ch5/stereoVision/stereoVision.cpp (Part)

```
int main(int argc, char **argv) {
    // intrinsics
    double fx = 718.856, fy = 718.856, cx = 607.1928, cy = 185.2157;
    // baseline
    double b = 0.573;

    cv::Mat left = cv::imread(left_file, 0);
    cv::Mat right = cv::imread(right_file, 0);
    cv::Ptr<cv::StereoSGBM> sgbm = cv::StereoSGBM::create(
        0, 96, 9, 8 * 9 * 9, 32 * 9 * 9, 1, 63, 10, 100, 32);    // SGBM is senstive to parameters
    cv::Mat disparity_sgbm, disparity;
    sgbm->compute(left, right, disparity_sgbm);
    disparity_sgbm.convertTo(disparity, CV_32F, 1.0 / 16.0f);

    // compute the point cloud
    vector<Vector4d, Eigen::aligned_allocator<Vector4d>> pointcloud;

    // change v++ and u++ to v+=2, u+=2 if your machine is slow to get a sparser cloud
    for (int v = 0; v < left.rows; v++)
    for (int u = 0; u < left.cols; u++) {
        if (disparity.at<float>(v, u) <= 10.0 || disparity.at<float>(v, u) >= 96.0) continue;

        Vector4d point(0, 0, 0, left.at<uchar>(v, u) / 255.0); // the first three dimensions are xyz, the 4-th is the color

        // compute the depth from disparity
        double x = (u - cx) / fx;
        double y = (v - cy) / fy;
        double depth = fx * b / (disparity.at<float>(v, u));
        point[0] = x * depth;
        point[1] = y * depth;
        point[2] = depth;

        pointcloud.push_back(point);
    }

    cv::imshow("disparity", disparity / 96.0);
    cv::waitKey(0);

    // show the point cloud in pangolin
    showPointCloud(pointcloud);
    return 0;
}
```

In this example, we call the SGBM (Semi-global Batch Matching) [24] algorithm implemented by OpenCV to calculate the disparity of the left and right images, and then transform it into the 3D space of the camera through the geometric model of the binocular camera. We use a classic parameter configuration from the internet, and we mainly adjust the maximum and minimum disparity. The disparity data combined with the camera's internal parameters and baseline can determine the position of each point in three-dimensional space. We omit the code related to displaying the point cloud to save some space.

This book is not going to introduce the disparity calculation algorithm of the binocular camera. Interested readers can read the relevant references [25, 26]. In addition to the binocular algorithm implemented by OpenCV, there are many other libraries focused on achieving efficient parallax calculations. It is still an active and complex subject today.

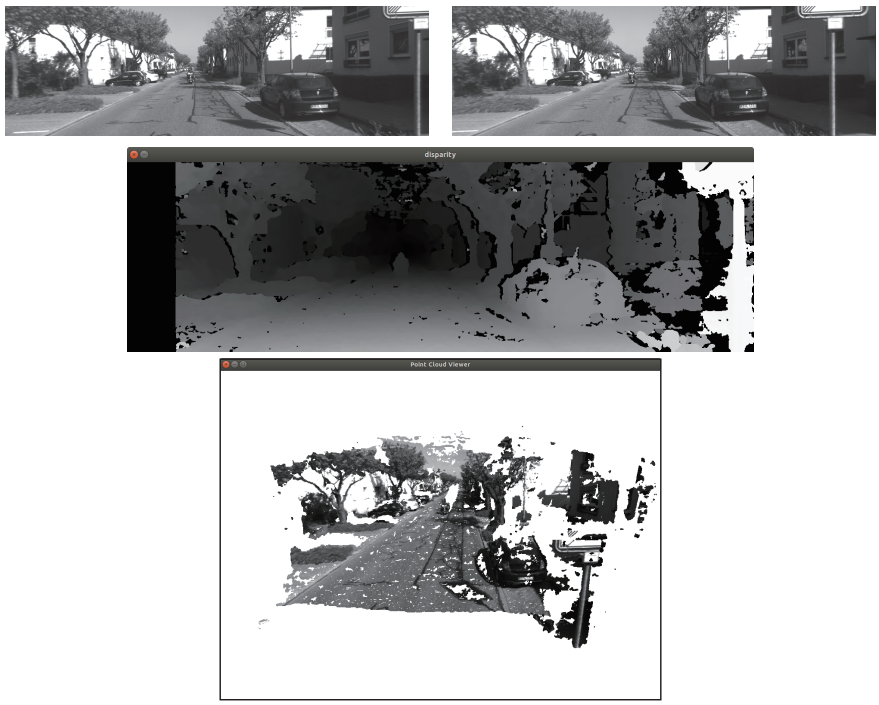


Figure 5-9: Stereo image example. Top-left: left image, top-right: right image, mid: SGBM disparity map, bottom: point cloud. Note that since some of the pixels in the left image is not seen in the right one, so the disparity map will have some empty values.

5.4.2 RGB-D Vision

Finally, we demonstrate an example of RGB-D vision. The convenience of RGB-D cameras is that they can obtain pixel depth information through physical methods. If the internal and external parameters of the camera are known, we can calculate the position of any pixel in the world coordinate system, thereby creating a point cloud map. Now let's demonstrate how to do it.

We have prepared 5 pairs of images, located in the slambook2/ch5/rgbd folder. There are 5 RGB images from 1.png to 5.png under the color/ directory, and 5 corresponding depth images under the depth/. At the same time, the "pose.txt" file gives the camera poses of the 5 images (in the form of $\mathbf{T}_{\mathrm{wc}}$). The form of the pose record is the same as before, with the translation vector plus a rotation quaternion:

$$[x, y, z, q_x, q_y, q_z, q_w],$$

where q_w is the real part of the quaternion. For example, the parameters of the first pair of image are:

$$[-0.228993, 0.00645704, 0.0287837, -0.0004327, -0.113131, -0.0326832, 0.993042].$$

Below we write a program to accomplish two things: (1). Calculate the point cloud corresponding to each pair of RGB-D images based on internal parameters;

(2). According to the camera pose of each image (that is, external parameters), put the points to a global cloud by the camera poses.

Listing 5.9: slambook/ch5/rgbd/jointMap.cpp (Part)

```
int main(int argc, char **argv) {
    vector<cv::Mat> colorImgs, depthImgs;
    TrajectoryType poses;         // camera poses

    ifstream fin("./pose.txt");
    if (!fin) {
        cerr << "Please run the program in the directory that has pose.txt" << endl;
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        boost::format fmt("./%s/%d.%s"); // the image format
        colorImgs.push_back(cv::imread((fmt % "color" % (i + 1) % "png").str()));
        depthImgs.push_back(cv::imread((fmt % "depth" % (i + 1) % "pgm").str(), -1));
             // use -1 flag to load the depth image

        double data[7] = {0};
        for (auto &d:data) fin >> d;
        Sophus::SE3d pose(Eigen::Quaterniond(data[6], data[3], data[4], data[5]),
        Eigen::Vector3d(data[0], data[1], data[2]));
        poses.push_back(pose);
    }

    // compute the point cloud using camera intrinsics
    double cx = 325.5;
    double cy = 253.5;
    double fx = 518.0;
    double fy = 519.0;
    double depthScale = 1000.0;
    vector<Vector6d, Eigen::aligned_allocator<Vector6d>> pointcloud;
    pointcloud.reserve(1000000);

    for (int i = 0; i < 5; i++) {
        cout << "Converting RGBD images " << i + 1 << endl;
        cv::Mat color = colorImgs[i];
        cv::Mat depth = depthImgs[i];
        Sophus::SE3d T = poses[i];
        for (int v = 0; v < color.rows; v++)
        for (int u = 0; u < color.cols; u++) {
            unsigned int d = depth.ptr<unsigned short>(v)[u]; // depth value is 16-bit
            if (d == 0) continue; // 0 means no valid value
            Eigen::Vector3d point;
            point[2] = double(d) / depthScale;
            point[0] = (u - cx) * point[2] / fx;
            point[1] = (v - cy) * point[2] / fy;
            Eigen::Vector3d pointWorld = T * point;

            Vector6d p;
            p.head<3>() = pointWorld;
            p[5] = color.data[v * color.step + u * color.channels()];   // blue
            p[4] = color.data[v * color.step + u * color.channels() + 1]; // green
            p[3] = color.data[v * color.step + u * color.channels() + 2]; // red
            pointcloud.push_back(p);
        }
    }

    cout << "global point cloud has " << pointcloud.size() << " points." << endl;
    showPointCloud(pointcloud);
    return 0;
}
```

We can see the point cloud in Pangolin after building it (see Figure 5-10).

Through these examples, we demonstrated some common monocular, binocular, and depth camera algorithms in computer vision. We hope readers can understand the meaning of the intrinsics, extrinsics and distortion model through these simple examples.

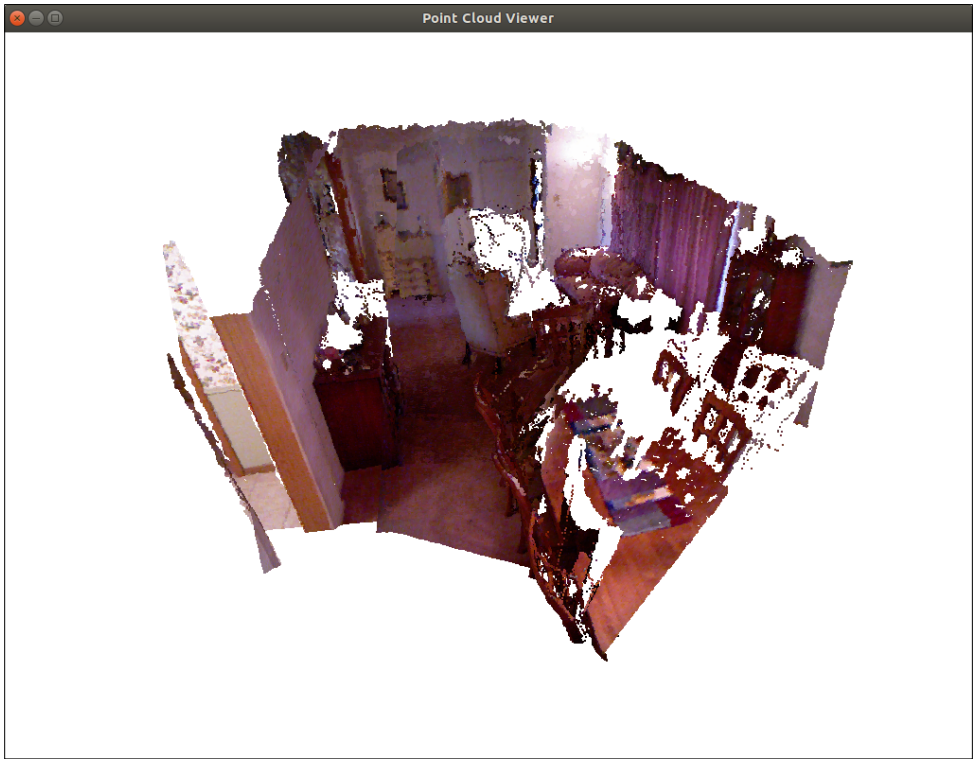


Figure 5-10: See the global point cloud by 5 RGBD image pairs.

Exercise

1.* Find a camera (use the camera of your mobile phone or laptop if you don't have one) and calibrate its internal parameters You may use a calibration board, or print out a checkerboard for calibration.

2. Describes the physical meaning of the camera's intrinsics. If the resolution of a camera is doubled and the rest is unchanged, how does its intrinsics change?

3. Search for the calibration method of special cameras (fisheye or panoramic cameras). Where are the differences between them and the pinhole models?

4. Investigate the similarities and differences between a global shutter camera and a rolling shutter camera. What are their advantages and disadvantages in SLAM?

5. How are RGB-D cameras calibrated? Taking Kinect as an example, what parameters need to be calibrated? (Refer to `https://github.com/code-iai/iai_kinect2`.)

6. In addition to the way of traversing the image demonstrated by the sample program, what other methods can you give to traverse the image?

7.* Read the official OpenCV tutorial to learn its basic usage.

Chapter 6

Nonlinear Optimization

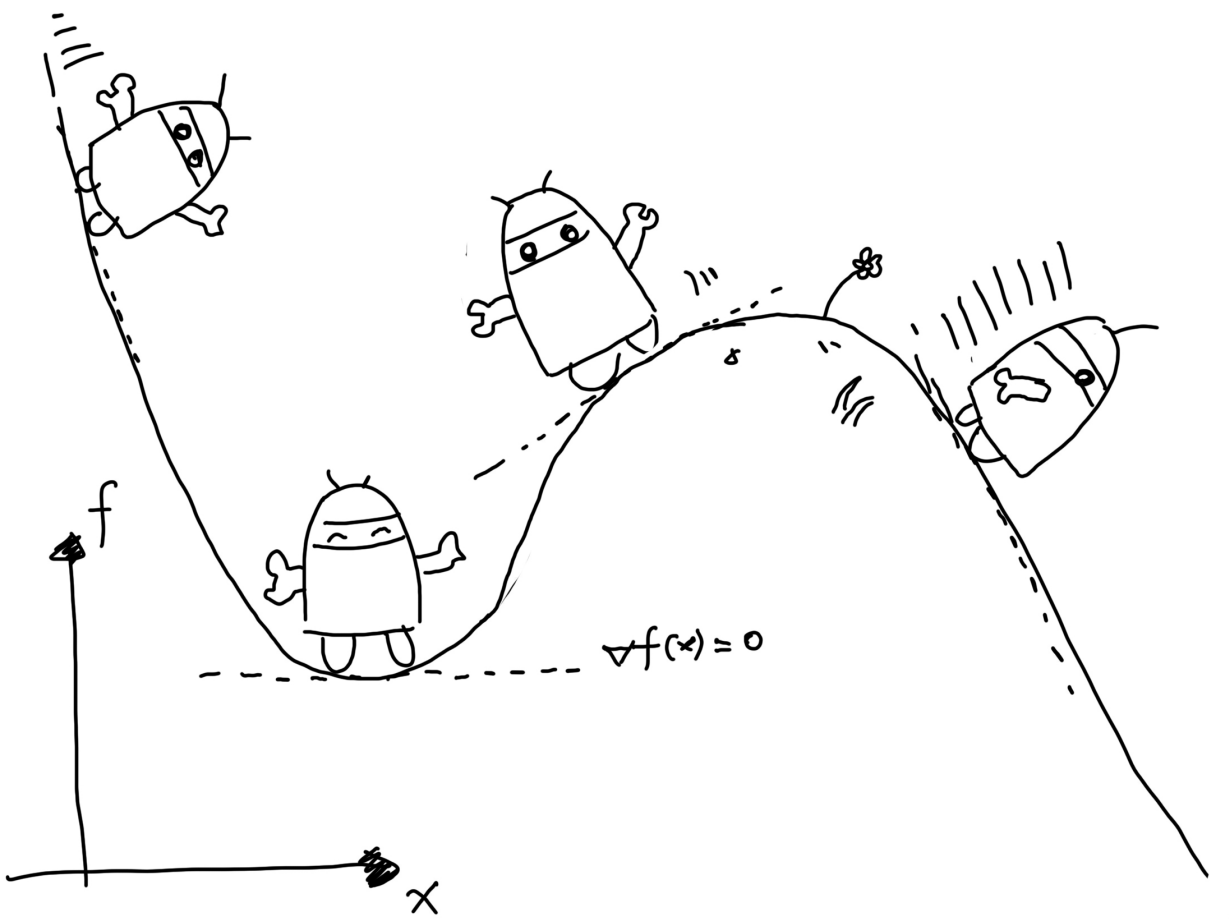
Goal of This Chapter

1. Understand how to form the batch state estimation problem into a least square and how to solve the least square problem.
2. Understand the Gauss-Newton and Levenburg-Marquadt method and implement them.
3. Learn how to use the Google Ceres and g2o library to solve a least square problem.

In the previous lectures, we introduced the motion and observation equations of the classic SLAM model. Now we know that the pose in the equation can be described by the transformation matrix and then optimized by Lie algebra. The observation equation is given by the camera imaging model, in which the internal parameter is fixed with the camera, and the external parameter is the pose of the camera. So, we have figured out the concrete expression of the classic visual SLAM model.

However, due to the presence of noise, the equations of motion and observation can not be exactly met. Although the camera can fit the pinhole model very well, unfortunately, the data we get is usually affected by various unknown noises. Even if we have a high-precision camera and controller, the motion and observations equations can only be approximated. Therefore, instead of assuming that the data must conform to the equation, it is better to estimate the state from the noisy data accurately.

Solving the state estimation problem requires a certain degree of optimization background knowledge. This section will introduce the basic unconstrained nonlinear optimization method and introduce optimization libraries g2o and Ceres.



$$f(x+\Delta x) \approx f(x) + \nabla f(x)\Delta x + \frac{1}{2}\Delta x^T H(x)\Delta x + \cdots$$

6.1 State Estimation

6.1.1 From Batch State Estimation to Least Square

According to the previous sections, the SLAM process can be described by a discrete time motion and observation equations like (2.5):

$$\begin{cases} \mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k \\ \mathbf{z}_{k,j} = \mathbf{h}(\mathbf{y}_j, \mathbf{x}_k) + \mathbf{v}_{k,j} \end{cases}. \tag{6.1}$$

Through the knowledge in Lecture 4, we learned that $\mathbf{x}_k$ is the pose of the camera, which can be described by SE(3). As for the observation equation, we have already explained in Lecture 5, the pinhole camera model. In order to give readers a deeper impression of them, we may wish to discuss their specific parameterized form. First, the pose variable $\mathbf{x}_k$ can be expressed by $\mathbf{T}_k \in \mathrm{SE}(3)$. Second, the motion is related to the specific form of the input, but there is no particularity in visual SLAM (should be same as the case of ordinary robots and vehicles), we will not talk about it for the time being. The observation equation is given by the pinhole model. Assuming an observation of the road sign $\mathbf{y}_j$ at $\mathbf{x}_k$, corresponding to the pixel position on the image $\mathbf{z}_{k,j}$, then, observe The equation can be expressed as:

$$s\mathbf{z}_{k,j} = \mathbf{K}(\mathbf{R}_k\mathbf{y}_j + \mathbf{t}_k), \tag{6.2}$$

where $\mathbf{K}$ is the intrinsic matrix of the camera, and s is the distance of pixels, which is also the third element of $(\mathbf{R}_k\mathbf{y}_j + \mathbf{t}_k)$. If we use transformation matrix $\mathbf{T}_k$ to describe the pose, then the points $\mathbf{y}_j$ must be described in homogeneous coordinates, and then converted to non-homogeneous coordinates afterwards. If you are not familiar with this process, please go back to the previous lectures.

Now, consider what happens when the data is affected by noise. In the motion and observation equations, we **usually** assume that the two noise terms $\mathbf{w}_k, \mathbf{v}_{k,j}$ satisfy a Gaussian distribution with zero mean, like this:

$$\mathbf{w}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_k), \mathbf{v}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_{k,j}), \tag{6.3}$$

where $\mathcal{N}$ means Gaussian distribution, $\mathbf{0}$ means zero mean, and $\mathbf{R}_k, \mathbf{Q}_{k,j}$ is the covariance matrix. Under the influence of these noises, we hope to infer the pose $\mathbf{x}$ and the map $\mathbf{y}$ from the noisy data $\mathbf{z}$ and $\mathbf{u}$ (and their probability density distribution), which constitutes a state estimation problem.

There are two ways to deal with this state estimation problem. Since these data come gradually over time in the SLAM process, we should intuitively hold an estimated state at the current moment and then update it with new data. This method is called the **incremental** method, or **filtering**. For a long time in history, researchers have used filters, especially the extended Kalman filter (EKF) and its derivatives, to solve it. The other way is to record the data into a file and looking for the best trajectory and map overall time. This method is called the **batch** estimation. In other words, we can put all the input and observation data from time 0 to k together and ask, with such input and observation, how to estimate the entire trajectory and map from time 0 to k?

These two different processing methods lead to many estimation methods. In general, the incremental method only cares about the state estimation of the **current moment** $\mathbf{x}_k$, but does not consider much about the previous state; relatively, the batch method can be used to get an optimized trajectory in larger time or space

scope, which is considered to be superior to the traditional filters [10], and has become the mainstream method of current visual SLAM. In extreme cases, we can let robots or drones collect data and then bring it back to the computing center for unified processing, which is also the mainstream practice of SfM (Structure from Motion). Of course, in these cases, the method is obviously not **real time**, which is not the most common application scenario of SLAM. So in SLAM, practical methods are usually some compromises. For example, we fix some historical trajectories and only optimize some trajectories close to the current moment, which leads to the **sliding window estimation** method to be described later.

In theory, the batch method is easier to introduce. At the same time, understanding the batch method also makes it easier to understand the incremental method. Therefore, in this section, we focus on the batch optimization method based on nonlinear optimization, and the Kalman filter and more in-depth knowledge will be discussed in the back-end chapter. Since the batch method is discussed, we will consider all the moments from time 1 to N and assume M map points. Define the robot pose and map coordinates at all times as:

$$\mathbf{x} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}, \quad \mathbf{y} = \{\mathbf{y}_1, \ldots, \mathbf{y}_M\}.$$

Similarly, $\mathbf{u}$ without subscript is used for input at all times, and $\mathbf{z}$ is used for observation data at all times. Then we say that the state estimation of the robot, from a probabilistic point of view, is to find the state $\mathbf{x}, \mathbf{y}$ under the condition that the input data $\mathbf{u}$ and the observation data $\mathbf{z}$. Or, the conditional probability distribution of:

$$P(\mathbf{x}, \mathbf{y}|\mathbf{z}, \mathbf{u}). \tag{6.4}$$

In particular, when we do not know the control input and only have one image, that is, only considering the data brought by the observation equation, it is equivalent to estimate the conditional probability distribution $P(\mathbf{x}, \mathbf{y}|\mathbf{z})$. Such a problem is also called Structure from Motion (SfM), that is, how to reconstruct the three-dimensional spatial structure from images only [27].

To estimation the conditional pdf we use the Bayes equation to switch the variables:

$$P(\mathbf{x}, \mathbf{y}|\mathbf{z}, \mathbf{u}) = \frac{P(\mathbf{z}, \mathbf{u}|\mathbf{x}, \mathbf{y})\, P(\mathbf{x}, \mathbf{y})}{P(\mathbf{z}, \mathbf{u})} \propto \underbrace{P(\mathbf{z}, \mathbf{u}|\mathbf{x}, \mathbf{y})}_{\text{likehood}} \underbrace{P(\mathbf{x}, \mathbf{y})}_{\text{prior}}. \tag{6.5}$$

The left side is called **posterior probability**, and $P(\mathbf{z}|\mathbf{x})$ on the right is called **likelihood** (or likehood), and the other part is $P(\mathbf{x})$ is called **prior**. **It is normally difficult to find the posterior distribution directly (in nonlinear systems), but it is feasible to find an optimal point which maximize the posterior** (Maximize a Posterior, MAP):

$$(\mathbf{x}, \mathbf{y})^*_{\text{MAP}} = \arg\max P(\mathbf{x}, \mathbf{y}|\mathbf{z}, \mathbf{u}) = \arg\max P(\mathbf{z}, \mathbf{u}|\mathbf{x}, \mathbf{y}) P(\mathbf{x}, \mathbf{y}). \tag{6.6}$$

Please note that the denominator part of Bayes' rule has nothing to do with the state $\mathbf{x}, \mathbf{y}$ to be estimated, so it can be just ignored. Bayes' rule tells us that solving the maximum posterior probability is **equivalent to the estimate the product of maximum likelihood and a priori**. Further, we can also say "I'm sorry, I don't know in prior where the robot pose or the map points are", then there is no **prior**. Then, you can solve **Maximize Likelihood Estimation** (MLE):

$$(\mathbf{x}, \mathbf{y})^*_{\text{MLE}} = \arg\max P(\mathbf{z}, \mathbf{u}|\mathbf{x}, \mathbf{y}). \tag{6.7}$$

Intuitively speaking, likelihood refers to "what observation data may be generated in the current pose". Since we know the observation data, the maximum likelihood estimation can be understood as: "**under what state is it most likely to produce the data currently observed**".

6.1.2 Introduction to Least Squares

Now we have a formulated the state estimation problem into a MAP/MLE problem, and the next question is how to solve it. Under the assumption of Gaussian distribution, we can have a simpler form of the maximum likelihood problem. Looking back at the observation model, for a certain kind of observation, we have:

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

Since we assume the noise item is gaussian, which means $\mathbf{v}_k \sim \mathcal{N}\left(\mathbf{0}, \mathbf{Q}_{k,j}\right)$, so the conditional probability of the observation data is:

$$P(\mathbf{z}_{j,k}|\mathbf{x}_k, \mathbf{y}_j) = N\left(h(\mathbf{y}_j, \mathbf{x}_k), \mathbf{Q}_{k,j}\right),$$

which is, of course, still a Gaussian distribution. Now let's consider solving the maximum likelihood estimation under this single observation.

We can rewrite this maximum problem into a **minimium of negative logarithm** one for Gaussian distributions since they have better mathematical forms under negative logarithm. Consider an arbitrary multi-dimensional Gaussian distribution $\mathbf{x} \sim \mathcal{N}(\mu, \mathbf{\Sigma})$, its probability density function expansion form is:

$$P\left(\mathbf{x}\right) = \frac{1}{\sqrt{\left(2\pi\right)^N \det(\mathbf{\Sigma})}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}} \mathbf{\Sigma}^{-1} \left(\mathbf{x}-\mu\right)\right). \tag{6.8}$$

Taking the negative logarithm of both sides:

$$-\ln\left(P\left(\mathbf{x}\right)\right) = \frac{1}{2}\ln\left(\left(2\pi\right)^N \det\left(\mathbf{\Sigma}\right)\right) + \frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}} \mathbf{\Sigma}^{-1} \left(\mathbf{x}-\mu\right). \tag{6.9}$$

Because the logarithm function is monotonically increasing, maximizing the original function is equivalent to minimizing the negative logarithm. When minimizing $\mathbf{x}$ in the above formula, the first term has nothing to do with $\mathbf{x}$ and can be omitted. Therefore, as long as the quadratic term on the right is minimized, the maximum likelihood estimate of the state is obtained. Substituting into the SLAM observation model, it is equivalent to find such a solution:

$$\begin{aligned}\left(\mathbf{x}_k, \mathbf{y}_j\right)^* &= \arg\max \mathcal{N}(h(\mathbf{y}_j, \mathbf{x}_k), \mathbf{Q}_{k,j}) \\ &= \arg\min\left(\left(\mathbf{z}_{k,j} - h\left(\mathbf{x}_k, \mathbf{y}_j\right)\right)^{\mathrm{T}} \mathbf{Q}_{k,j}^{-1}\left(\ bmz_{k,j} - h\left(\mathbf{x}_k, \mathbf{y}_j\right)\right)\right).\end{aligned} \tag{6.10}$$

We found that this equation is equivalent to a quadratic form that minimizes the noise term (i.e., the error). This quadratic form is called **Mahalanobis distance**. It can also be regarded as the Euclidean distance ($\mathcal{L}_2$-norm) weighted by $\mathbf{Q}_{k,j}^{-1}$, where $\mathbf{Q}_{k,j}^{-1}$ is also called the **information matrix**, which is extacly the **inverse** of the Gaussian **covariance matrix**.

Now we put all the observations together. It is usually assumed that the inputs and observations at each time are independent of each other, which means that each

input is independent, each observation is independent, and input and observation are also independent. So we can factorize the joint distribution like this:

$$P\left(\mathbf{z},\mathbf{u}|\mathbf{x},\mathbf{y}\right)=\prod_{k}P\left(\mathbf{u}_{k}|\mathbf{x}_{k-1},\mathbf{x}_{k}\right)\prod_{k,j}P\left(\mathbf{z}_{k,j}|\mathbf{x}_{k},\mathbf{y}_{j}\right), \tag{6.11}$$

It shows that we can handle the movement and observation at each moment independently. Let's define the error between the model and real data:

$$\begin{aligned}\mathbf{e}_{u,k}&=\mathbf{x}_{k}-f\left(\mathbf{x}_{k-1},\mathbf{u}_{k}\right)\\ \mathbf{e}_{z,j,k}&=\mathbf{z}_{k,j}-h\left(\mathbf{x}_{k},\mathbf{y}_{j}\right),\end{aligned} \tag{6.12}$$

Then, minimizing the Mahalanobis distance between the estimated value at all times and the measurements from sensors is equivalent to finding the maximum likelihood estimation. The negative logarithm allows us to turn the product into a summation:

$$\min J(\mathbf{x},\mathbf{y})=\sum_{k}\mathbf{e}_{u,k}^{\mathrm{T}}\mathbf{R}_{k}^{-1}\mathbf{e}_{u,k}+\sum_{k}\sum_{j}\mathbf{e}_{z,k,j}^{\mathrm{T}}\mathbf{Q}_{k,j}^{-1}\mathbf{e}_{z,k,j}. \tag{6.13}$$

In this way, a **least square problem** is obtained with the same solution as the MLE problem. Intuitively speaking, due to the presence of noise, when we substitute the estimated trajectory and map into the SLAM motion and observation models, they will not be perfectly fit. What shall we do at this time? We perform **fine-tuning** on the estimated value of the state, so that the overall error is reduced. Of course, finally we will reach a (local) **minimum value**. This is a typical nonlinear optimization process.

Observing the formula (6.13) carefully, we find that the least squares problem in SLAM has some specific structures:

- First, the objective function of the whole problem consists of many (weighted) error quadratic forms. Although the dimensionality of the overall state variable is very high, each error term is simple and is only related to one or two state variables. For example, the motion error is only related to $\mathbf{x}_{k-1},\mathbf{x}_{k}$, and the observation error is only related to $\mathbf{x}_{k},\mathbf{y}_{j}$. This relationship will give a **sparse** least square problem, which we will investigate further in the backend chapter.

- Secondly, if you use Lie algebra to represent the increment, the problem is the least squares problem of **unconstrained**. However, if the rotation matrix/transformation matrix is used to describe the pose, the constraint of the rotation matrix itself will be introduced, that is, s.t. $\mathbf{R}^{\mathrm{T}}\mathbf{R}=\mathbf{I}$ and $\det(\mathbf{R})=1$ need to be added to the problem. Additional constraints can make optimization more difficult.

- Finally, we used a quadratic metric error, which is exactly the $\mathcal{L}_2-$norm. The information matrix is used as the weights of each elemnents. For example, if an observation is very accurate, the covariance matrix will be "small" and the information matrix will be "large", so this error term will have a higher weight than the others in the whole problem. We will see some drawbacks of the $\mathcal{L}_2$ error later.

We introduce how to solve this least-squares problem, which requires some **basic knowledge of nonlinear optimization**. In particular, we want to discuss how to solve such a general unconstrained nonlinear least-squares problem. In the following lectures, we will make extensive use of this lecture's results and discuss in detail its application in the front and back ends of SLAM.

6.1.3 Example: Batch state estimation

Maybe it's better to give an example here. Consider a very simple discrete-time system:

$$\begin{array}{ll} x_k &= x_{k-1} + u_k + w_k, \qquad w_k \sim \mathcal{N}(0, Q_k) \\ z_k &= x_k + n_k, \qquad n_k \sim \mathcal{N}(0, R_k) \end{array}, \tag{6.14}$$

which can express a car moving forward or backward along the x axis. The first formula is the motion model, where u_k is the input, and w_k is the noise; the second formula is the observation model, where z_k is the measurement of the car position. We set the time $k = 1, \ldots, 3$, and want to estimate the states based on the existing v, y. Suppose the initial state x_0 is known. Let's derive the maximum likelihood estimation of the batch state estimation.

First, let the batch state variable be $\mathbf{x} = [x_0, x_1, x_2, x_3]^{\mathrm{T}}$, and the batch observation be $\mathbf{z} = [z_1, z_2, z_3]^{\mathrm{T}}$, define $\mathbf{u} = [u_1, u_2, u_3]^{\mathrm{T}}$ in the same way. According to the previous derivation, we know that the maximum likelihood estimate is:

$$\begin{aligned} \mathbf{x}^*_{\mathrm{map}} &= \arg\max P(\mathbf{x}|\mathbf{u}, \mathbf{z}) = \arg\max P(\mathbf{u}, \mathbf{z}|\mathbf{x}) \\ &= \prod_{k=1}^{3} P(u_k|x_{k-1}, x_k) \prod_{k=1}^{3} P(z_k|x_k), \end{aligned} \tag{6.15}$$

For each item, such as the equation of motion, we have:

$$P(u_k|x_{k-1}, x_k) = \mathcal{N}(x_k - x_{k-1}, Q_k). \tag{6.16}$$

The observation equation is similar:

$$P(z_k|x_k) = \mathcal{N}(x_k, R_k). \tag{6.17}$$

According to the previous statements, the error variable can be constructed as:

$$e_{u,k} = x_k - x_{k-1} - u_k, \quad e_{z,k} = z_k - x_k, \tag{6.18}$$

Then the objective function of the least squares is:

$$\min \sum_{k=1}^{3} e_{u,k}^{\mathrm{T}} Q_k^{-1} e_{u,k} + \sum_{k=1}^{3} e_{z,k}^{\mathrm{T}} R_k^{-1} e_{z,k}. \tag{6.19}$$

In addition, since this system is a linear one, we can easily write the equations in the vector/matrix form. Define the vector $\mathbf{y} = [\mathbf{u}, \mathbf{z}]^{\mathrm{T}}$, then the error can be defined as:

$$\mathbf{y} - \mathbf{H}\mathbf{x} = \mathbf{e} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}), \tag{6.20}$$

where

$$\mathbf{H} = \left[\begin{array}{cccc} 1 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right], \tag{6.21}$$

and $\mathbf{\Sigma} = \mathrm{diag}(Q_1, Q_2, Q_3, R_1, R_2, R_3)$. The whole question can be written as:

$$\mathbf{x}^*_{\mathrm{map}} = \arg\min \mathbf{e}^{\mathrm{T}} \mathbf{\Sigma}^{-1} \mathbf{e}, \tag{6.22}$$

Later we will see that this problem has a unique solution:

$$\mathbf{x}^*_{\mathrm{map}} = (\mathbf{H}^{\mathrm{T}} \mathbf{\Sigma}^{-1} \mathbf{H})^{-1} \mathbf{H}^{\mathrm{T}} \mathbf{\Sigma}^{-1} \mathbf{y}, \tag{6.23}$$

since the $(\mathbf{H}^{\mathrm{T}} \mathbf{\Sigma}^{-1} \mathbf{H})^{-1}$ is invertible.

6.2 Nonlinear least squares

First consider a simple least squares problem:

$$\min_{\mathbf{x}} F(\mathbf{x}) = \frac{1}{2}\|f(\mathbf{x})\|_2^2. \tag{6.24}$$

Among them, the status variable is $\mathbf{x} \in \mathbb{R}^n$, and f is any scalar nonlinear function $f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}$. Note that the coefficient $\frac{1}{2}$ here is not important, some literature have this coefficient and some not. It will not affect the subsequent conclusions. Obviously, if f is a mathematically simple function, then the problem can be solved in analytical form. Let the derivative of the objective function be zero, and then find the optimal value of $\mathbf{x}$, just like finding the extreme value of a scalar function:

$$\frac{\mathrm{d}F}{\mathrm{d}\mathbf{x}} = \mathbf{0}. \tag{6.25}$$

We reach the minimum, maximum, or saddle points by solving this equation (or, intuitively, by letting the derivative be zero). But is this equation easy to solve? Well, it depends on the form of the derivative function of f. If f is just a simple linear function, then the problem is only a simple linear least squares problem. Still, some derivative functions may be complicated in form, making the equation difficult to solve. Solving this equation requires us to know the **global property** of the objective function, which is usually not possible. For the least squares problem that is inconvenient to solve directly, we can use **iterated methods** to start from an initial value and continuously update the current estimations to reduce the objective function. The specific steps can be listed as follows:

1. Give an initial value $\mathbf{x}_0$.
2. For k-th iteration, we find an incremental value of $\Delta\mathbf{x}_k$, such that the object function $\|f(\mathbf{x}_k + \Delta\mathbf{x}_k)\|_2^2$ reaches a smaller value.
3. If $\Delta\mathbf{x}_k$ is small enough, stop the algorithm.
4. Otherwise, let $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$ and return to step 2.

Now things get much simpler. We turn the problem of solving **the derivative function equals zero** into a problem of **looking for decreasing increments** $\Delta\mathbf{x}_k$. We will see that since the objective function can be linearly approximated at the current estimation, the increment calculation will be simpler *. When the function decreases until the increment becomes very small, it is considered that the algorithm converges, and the objective function reaches a minimum value. In this process, the problem is how to find the increment at each iteration point, which is a local problem. We only need to be concerned about the local properties of f at the iteration value rather than the global properties. Such methods are widely used in optimization, machine learning, and other fields.

Next, we examine how to find this increment $\Delta\mathbf{x}_k$. This part of knowledge belongs to the field of numerical optimization. Let's take a quick look at some widely used results.

* Linear cases are always the easiest ones.

6.2.1 First and Second-order Method

Now consider the k-th iteration. Suppose we are at $\mathbf{x}_k$ and want to find the increment $\Delta\mathbf{x}_k$, then the most intuitive way is to make the Taylor expansion of the objective function in $\mathbf{x}_k$:

$$F(\mathbf{x}_k + \Delta\mathbf{x}_k) \approx F(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)^{\mathrm{T}} \Delta\mathbf{x}_k + \frac{1}{2}\Delta\mathbf{x}_k^{\mathrm{T}}\mathbf{H}(\mathbf{x}_k)\Delta\mathbf{x}_k, \tag{6.26}$$

where $\mathbf{J}(\mathbf{x}_k)$ is the first derivative of $F(\mathbf{x})$ with respect to $\mathbf{x}$ (also called gradient, **Jacobi** Matrix, or **Jacobian**) *, $\mathbf{H}$ is the second-order derivative (or **Hessian**), which are all taken at $\mathbf{x}_k$. Readers should be familiar with them in the undergraduate course like multivariate calculus. We can choose to keep the first-order or second-order terms of the Taylor expansion, and the corresponding solution is called the first-order or the second-order method. In the simplest way, if we only keep the first-order one, then taking the increment at the minus gradient direction will ensure that the function decreases:

$$\Delta\mathbf{x}^* = -\mathbf{J}(\mathbf{x}_k). \tag{6.27}$$

Of course, this is only a direction, usually we have to compute another step length parameter, say, λ. The step length can be calculated according to certain conditions [28]. There are also some empirical methods in machine learning, but we will not discuss them. This method is called **steepest descent method**. Its intuitive meaning is very simple, as long as we move along the reverse gradient direction, the objective function must decrease if the first-order (linear) approximation still holds.

Note that the above discussion was carried out during the k-th iteration and did not involve any information about k. To simplify the notation, we will omit the subscript k later and think that these discussions are valid for any iterations.

On the other hand, we can choose to keep the second step information, and the increment equation is:

$$\Delta\mathbf{x}^* = \arg\min\left(F(\mathbf{x}) + \mathbf{J}(\mathbf{x})^{\mathrm{T}}\Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^{\mathrm{T}}\mathbf{H}\Delta\mathbf{x}\right). \tag{6.28}$$

The right side only contains the zero-order, first-order and quadratic terms of $\Delta\mathbf{x}$. Finding the derivative of $\Delta\mathbf{x}$ on the right side of the equation and setting it to zero leads to: †

$$\mathbf{J} + \mathbf{H}\Delta\mathbf{x} = \mathbf{0} \Rightarrow \mathbf{H}\Delta\mathbf{x} = -\mathbf{J}. \tag{6.29}$$

We can also solve this linear equation and get the increment. This method is also called **Newton's method**.

We have seen that both the first-order and second-order methods are very intuitive, as long as we can calculate Taylor expansion of f and find the increments. We say, hey, the function looks like a linear or quadratic one. We can use the approximated function's minimum value to guess the minimum value of the original function. As long as the original objective function really looks like a first-order or quadratic function locally, this type of algorithm is always valid (this is also most cases in reality). However, these two methods also have their drawbacks. The steepest descent method is too greedy and easy to lead a zigzag way and increases the number of iterations. However, Newton's method needs to calculate the $\mathbf{H}$ matrix of the objective function, which is very time-expensive when the problem is

* We write $\mathbf{J}(\mathbf{x})$ as a column vector, then it can be inner product with $\Delta\mathbf{x}$ to get a scalar.

† For students who are not familiar with matrix derivation, please refer to Appendix B.

large, and we usually tend to avoid the calculation of $\mathbf{H}$. For general problems, some quasi-Newton methods can get better results, and for least squares problems, there are several more practical methods: the **Gauss-Newton's method** and the (Levernburg-Marquardt's method).

6.2.2 The Gauss-Newton Method

The Gauss-Newton method is one of the simplest methods in optimization algorithms. Its idea is to carry out a first-order Taylor expansion of $f(\mathbf{x})$. Please note that this is not the objective function $F(\mathbf{x})$ but the lower case $f(\mathbf{x})$, otherwise it is same as the Newton's method.

$$f\left(\mathbf{x}+\Delta\mathbf{x}\right) \approx f\left(\mathbf{x}\right)+\mathbf{J}\left(\mathbf{x}\right)^{\mathrm{T}}\Delta\mathbf{x}. \tag{6.30}$$

Here $\mathbf{J}(\mathbf{x})$ is the derivative of $f(\mathbf{x})$ with respect to $\mathbf{x}$, which is a column vector of $n\times 1$. According to the previous framework, the current goal is to find the increment $\Delta\mathbf{x}$ such that $\left\|f\left(\mathbf{x}+\Delta\mathbf{x}\right)\right\|^2$ reached the minimum. In order to find $\Delta\mathbf{x}$, we need to solve a linear least square problem:

$$\Delta\mathbf{x}^{*}=\arg\min_{\Delta\mathbf{x}}\frac{1}{2}\left\|f\left(\mathbf{x}\right)+\mathbf{J}\left(\mathbf{x}\right)^{\mathrm{T}}\Delta\mathbf{x}\right\|^{2}. \tag{6.31}$$

What's the difference from before? According to the extreme conditions, we set the derivative with $\Delta\mathbf{x}$ to zero to reach the extreme value. To do this, let's first expand the square term of the objective function:

$$\begin{aligned}\frac{1}{2}\left\|f\left(\mathbf{x}\right)+\mathbf{J}\left(\mathbf{x}\right)^{\mathrm{T}}\Delta\mathbf{x}\right\|^{2} &= \frac{1}{2}\left(f\left(\mathbf{x}\right)+\mathbf{J}\left(\mathbf{x}\right)^{\mathrm{T}}\Delta\mathbf{x}\right)^{\mathrm{T}}\left(f\left(\mathbf{x}\right)+\mathbf{J}\left(\mathbf{x}\right)^{\mathrm{T}}\Delta\mathbf{x}\right)\\ &= \frac{1}{2}\left(\|f(\mathbf{x})\|_2^2+2f\left(\mathbf{x}\right)\mathbf{J}(\mathbf{x})^{\mathrm{T}}\Delta\mathbf{x}+\Delta\mathbf{x}^{\mathrm{T}}\mathbf{J}(\mathbf{x})\mathbf{J}(\mathbf{x})^{\mathrm{T}}\Delta\mathbf{x}\right).\end{aligned}$$

Find the derivative of the above formula with respect to $\Delta\mathbf{x}$ and set it to zero:

$$\mathbf{J}(\mathbf{x})f\left(\mathbf{x}\right)+\mathbf{J}(\mathbf{x})\mathbf{J}^{\mathrm{T}}\left(\mathbf{x}\right)\Delta\mathbf{x}=\mathbf{0}.$$

The following equations can be obtained:

$$\underbrace{\mathbf{J}(\mathbf{x})\mathbf{J}^{\mathrm{T}}}_{\mathbf{H}(\mathbf{x})}\left(\mathbf{x}\right)\Delta\mathbf{x}=\underbrace{-\mathbf{J}(\mathbf{x})f\left(\mathbf{x}\right)}_{\mathbf{g}(\mathbf{x})}. \tag{6.32}$$

This equation is a **linear equation** of the variable $\Delta\mathbf{x}$. We call it **normal equation** or **Gauss-Newton equation**. We define the coefficients on the left as $\mathbf{H}$ and the coefficient on the right as $\mathbf{g}$, then the above formula becomes:

$$\mathbf{H}\Delta\mathbf{x}=\mathbf{g}. \tag{6.33}$$

It makes sense to mark the left side as $\mathbf{H}$ here. Compared with Newton's method 6.29, Gauss-Newton's method uses $\mathbf{J}\mathbf{J}^{\mathrm{T}}$ **as the approximation** of the second-order Hessian matrix in Newton's method, thus omitting the calculation of $\mathbf{H}$. Please note that solving the normal equation is the core of the entire optimization problem. If we can get the $\Delta\mathbf{x}$ in each iteration, then the algorithm of Gauss-Newton method can be written as:

1. Set it initial value as $\mathbf{x}_0$.
2. For k-th iteration calculate the Jacobian $\mathbf{J}(\mathbf{x}_k)$ and residual $f(\mathbf{x}_k)$.
3. Solve the normal equation: $\mathbf{H}\Delta\mathbf{x}_k = \mathbf{g}$.
4. If $\Delta\mathbf{x}_k$ is small enough, stop the algorithm. Otherwise let $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$ and return to step 2.

It can be seen from the algorithm steps that the solution of the incremental equation occupies a major position. As long as we can solve the increment smoothly, we can ensure that the objective function decreases correctly.

In order to solve the incremental equation, we need to solve $\mathbf{H}^{-1}$, which requires the $\mathbf{H}$ matrix to be invertible, but the calculated $\mathbf{J}\mathbf{J}^{\mathrm{T}}$ is only semi-positive definite. If $\mathbf{J}$ has null-spaces, which is to say, we can find non-zero $\Delta\mathbf{x}$ such that $\mathbf{J}\Delta\mathbf{x} = \mathbf{0}$, then we can not determine which $\Delta\mathbf{x}$ can really makes the objective function really decreases. In that cases, the algorithm is probably not converging and we may obtain erroneous results. Intuitively, the local approximation of the original function at this point is not like a quadratic function. Furthermore, even if we assume that $\mathbf{H}$ is not singular or ill-conditioned, but if we take a very large step size $\Delta\mathbf{x}$, the result can still be bad since linear approximation is not accurate enough at that point. So in practice, we can't guarantee that the Gauss-Newton method converges. Sometimes they can reach even larger objective functions values that the initial one.

Although the Gauss-Newton method has these shortcomings, it is still a simple and effective method for nonlinear optimization, and it is worth learning. In nonlinear optimization, quite a few algorithms can be taken as the variants of the Gauss-Newton method. These algorithms all use the idea of the Gauss-Newton method and correct their shortcomings through their own improvements. For example, some **line search method** adds an extra step size α. After determining $\Delta\mathbf{x}$, we may further find the α to make $\|f(\mathbf{x} + \alpha\Delta\mathbf{x})\|^2$ is minimized, instead of simply making $\alpha = 1$.

The Levenberg-Marquardt method corrects these problems to a certain extent. It is generally considered to be more robust than the Gauss-Newton method, but its convergence rate may be slower than the Gauss-Newton method. Such a kind of method is also called as **damped Newton Method**.

6.2.3 The Levernberg-Marquatdt Method

The approximate second-order Taylor expansion used in the Gauss-Newton method can only have a good approximation effect near the expansion point, so we naturally thought that a range should be added to $\Delta\mathbf{x}$, called **trust region**. This range defines under what circumstances the second-order approximation is valid. This type of method is also called **trust region method**. We think the approximation is valid only in the trusted region; otherwise, it may go wrong if the approximation goes outside.

So how to determine the scope of this trust region? A good method is to determine it based on the difference between our approximate model and the real object function: if the difference is small, it means that the approximation is good, and we may expand the trust region; conversely, if the difference is large, we will reduce the range of approximation. We define an indicator ρ to describe the degree of

approximation:

$$\rho = \frac{f(\mathbf{x} + \Delta\mathbf{x}) - f(\mathbf{x})}{\mathbf{J}(\mathbf{x})^{\mathrm{T}} \Delta\mathbf{x}}. \tag{6.34}$$

The numerator of ρ is the decreasing value of the real object function, and the denominator is the decreasing value of the approximate model. If ρ is close to 1, the approximation is good. If ρ is too small, indicating that the actual reduced value is far less than the approximate reduced value, the approximation is considered poor, and the trust region needs to be reduced. Conversely, if ρ is relatively large, it means that the actual decline is greater than expected, and we can enlarge the approximate range.

Therefore, we build an improved version of the nonlinear optimization framework, which will have a better effect than the Gauss-Newton method:

1. Give the initial valude $\mathbf{x}_0$ and initial trust region radius μ.
2. For k-th iteration, we solve a linear problem based on Gauss-Newton's method added with a trust region:

$$\min_{\Delta\mathbf{x}_k} \frac{1}{2} \left\| f(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)^{\mathrm{T}} \Delta\mathbf{x}_k \right\|^2, \quad \text{s.t.} \quad \|\mathbf{D}\Delta\mathbf{x}_k\|^2 \leqslant \mu, \tag{6.35}$$

   where μ is the radius and $\mathbf{D}$ is a coefficient matrix, which will be discussed in below.
3. Compute ρ using equation (6.34).
4. If $\rho > \frac{3}{4}$, set $\mu = 2\mu$.
5. Otherwise, if $\rho < \frac{1}{4}$, set $\mu = 0.5\mu$.
6. If ρ is larger than a given threshold, set $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$.
7. Go back to step 2 if not converged, otherwise return the result.

Here, the multiples and thresholds of the approximate range expansion are empirical values and can be replaced with other values. In the formula (6.35), we limit the increment to a sphere with a radius of μ, and think that it is valid only in this sphere. After bringing $\mathbf{D}$, this sphere can be seen as an ellipsoid. In the optimization method proposed by Levenberg, taking $\mathbf{D}$ into the unit matrix $\mathbf{I}$ is equivalent to directly constraining $\Delta\mathbf{x}_k$ in a sphere. Subsequently, Marquardt proposed to take $\mathbf{D}$ as a non-negative diagonal matrix-in practice, the square root of the diagonal elements of $\mathbf{J}^T\mathbf{J}$ is usually used so that The constraint range is larger on the dimension with small gradient.

In any case, in Levenberg-Marquardt optimization, we need to solve a subproblem like (6.35) to obtain the gradient. This sub-problem is an optimization problem with inequality constraints. We use Lagrangian multipliers to put the constraints in the objective function to form the Lagrangian function:

$$\mathcal{L}(\Delta\mathbf{x}_k, \lambda) = \frac{1}{2} \left\| f(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)^{\mathrm{T}} \Delta\mathbf{x}_k \right\|^2 + \frac{\lambda}{2} \left(\|\mathbf{D}\Delta\mathbf{x}_k\|^2 - \mu \right), \tag{6.36}$$

where λ is the Lagrange multiplier. Similar to the method in the Gauss-Newton, let the derivative of the Lagrangian function with respect to $\Delta\mathbf{x}$ be zero, and we still

need to solve the linear equation for calculating the increment:

$$\left(\mathbf{H} + \lambda \mathbf{D}^{\mathrm{T}} \mathbf{D}\right) \Delta \mathbf{x}_k = \mathbf{g}. \tag{6.37}$$

It can be seen that the incremental equation has an extra $\lambda \mathbf{D}^T \mathbf{D}$ compared to the Gauss-Newton method. If you consider its simplified form, that is, $\mathbf{D} = \mathbf{I}$, then it is equivalent to solving *:

$$(\mathbf{H} + \lambda \mathbf{I}) \Delta \mathbf{x}_k = \mathbf{g}.$$

We can see that when the parameter λ is relatively small, $\mathbf{H}$ dominates, which shows that the quadratic approximation model is better in this range. The Levenberg-Marquardt method is more Close to the Gauss-Newton method. On the other hand, when λ is relatively large, $\lambda \mathbf{I}$ occupies a dominant position, and the Levenberg-Marquardt method is closer to the one-step descent method (that is, the steepest descent). This shows that the nearby quadratic approximation is not good enough. The solution method of the Levenberg-Marquardt method can avoid the non-singular and ill-conditioned problems of the coefficient matrix of linear equations to a certain extent and provide more stable and accurate increments $\Delta \mathbf{x}$.

In practice, there are also many other ways to solve the increment, such as Dog-Leg [29] and other methods. We have introduced here is just the most common and basic method, and it is also the most used method in visual SLAM. We usually choose one of the Gauss-Newton method or Levenberg-Marquardt method as the gradient descent strategy in practical problems. When the problem is well-formed, Gauss-Newton is used. Otherwise, in ill-formed problems, we use the Levenberg-Marquardt method.

6.2.4 Conclusion

Since I don't want this book to become a headache-inducing mathematics textbook, only two of the most common nonlinear optimization schemes are listed here: the Gauss-Newton method and the Levenberg-Marquardt method. We avoided many discussions of mathematical properties. If readers are interested in optimization, you can read further books dedicated to numerical optimization (this is a big topic)[29]. The optimization methods represented by the Gauss-Newton method and the Levenberg-Marquardt method have been implemented and provided to users in many open-source optimization libraries. We will conduct experiments below. Optimization is a basic mathematical tool for dealing with many practical problems. It plays a central role in not only visual SLAM but also one of the core methods for solving problems in other fields such as deep learning (the amount of deep learning data is substantial, with a First-order method is the main). We hope that readers can learn more about optimization algorithms based on their own capabilities.

Perhaps you have discovered that whether it is the Gauss-Newton method or the Levenberg-Marquardt method, the variable's initial value needs to be provided when doing the optimization calculation. You may ask, can this initial value be set at will?

---

* Strict readers may not be satisfied with the description here. In addition to the Lagrangian function derivation to zero, the KKT condition also has some other constraints: $\lambda > 0$, and $\lambda(\|\mathbf{D}\Delta\mathbf{x}\|^2 - \mu) = 0$. But in the L-M iteration, we might as well regard it as a penalty term (Argumented Lagrangian) with λ as the weight on the objective function of the original problem. After each iteration, if it is found that the trust-region condition is not satisfied, or the objective function increases, the weight of λ is increased until the trust-region condition is finally satisfied. Therefore, there are different interpretations of the L-M algorithm in theory, but in practice, we only care about whether it works smoothly.

Of course not. In fact, alliterative solutions of nonlinear optimization require users to provide a good initial value. Because the objective function is too complicated, it is difficult to predict the solution space change. Providing different initial values for the problem often leads to different calculation results. This situation is a common problem of nonlinear optimization: most algorithms easily fall into local minima. Therefore, no matter what kind of scientific problem, we should provide the initial value carefully. For example, in the visual SLAM problem, we will use ICP, PnP, and other algorithms to provide the optimized initial value. In short, a good initial value is significant for optimization problems!

Perhaps readers will also have questions about the optimization mentioned above: how to solve the linear incremental equations? We have only mentioned that the incremental equation is linear, but does it require a lot of calculations to invert the coefficient matrix directly? Of course not. In the visual SLAM algorithm, the dimension of $\Delta\mathbf{x}$ is often as large as hundreds or thousands. If you are doing large-scale visual 3D reconstruction, you will often find that this dimension can be easily achieved at hundreds of thousands or even higher levels. Most processors cannot afford the inversion computation of such a large matrix, so there are many numerical solutions for linear equations. There are different solutions in different fields, but there is almost no way to find the coefficient matrix's inverse directly. We will use matrix decomposition to solve linear equations, such as QR, Cholesky, and other decomposition methods. These methods can usually be found in textbooks, such as the matrix theory, and we will not introduce them.

Fortunately, this matrix in visual SLAM often has a specific sparse form, which can solve optimization problems in real-time. We will introduce its principle in detail in Lecture 9. Using the sparse form of elimination, decomposition, and finally solving increment will greatly improve the solution's efficiency. In many open source optimization libraries, variables with a dimension of more than 10,000 can be solved in a few seconds or less on a general PC. The reason is that more advanced mathematical tools are used. The visual SLAM algorithm can now be implemented in real-time, thanks to the coefficient matrix is sparse. If the matrix is dense, I am afraid that optimization of this kind of visual SLAM algorithm will not be widely adopted by the academic community [30?, 31].

6.3 Practice: Curve Fitting

6.3.1 Curve Fitting with Gauss-Newton

Next we use a simple example to illustrate how to solve the least squares problem. We will demonstrate how to write Gauss-Newton method by hand, and then introduce how to use the optimization library to solve this problem. For the same problem, these implementations will get the same result because their core algorithms are the same.

Consider a curve that satisfies the following equation:

$$y = \exp(ax^2 + bx + c) + w,$$

where a, b, c are the parameters of the curve, and w is Gaussian noise, satisfying $w \sim (0, \sigma^2)$. We deliberately chose such a nonlinear model so that the problem is not too simple. Now, suppose we have N observation data points about $xandy$, and want to find the parameters of the curve based on these data points. Then, the

following least-squares problem can be solved to estimate the curve parameters:

$$\min_{a,b,c} \frac{1}{2} \sum_{i=1}^{N} \left\| y_i - \exp\left(ax_i^2 + bx_i + c\right) \right\|^2. \tag{6.38}$$

Please note that in this question, the variables to be estimated are a, b, c, not x. In our program, the true value of x, y is generated according to the model, and then Gaussian noise is added to the true value. Subsequently, the Gauss-Newton method was used to fit a parametric model from the noisy data. Define the error as:

$$e_i = y_i - \exp\left(ax_i^2 + bx_i + c\right), \tag{6.39}$$

Then we can find the derivative of each error term with respect to the state variable:

$$\begin{aligned} \frac{\partial e_i}{\partial a} &= -x_i^2 \exp\left(ax_i^2 + bx_i + c\right) \\ \frac{\partial e_i}{\partial b} &= -x_i \exp\left(ax_i^2 + bx_i + c\right) \\ \frac{\partial e_i}{\partial c} &= -\exp\left(ax_i^2 + bx_i + c\right) \end{aligned} \tag{6.40}$$

So $\mathbf{J}_i = \left[\frac{\partial e_i}{\partial a}, \frac{\partial e_i}{\partial b}, \frac{\partial e_i}{\partial c}\right]^{\mathrm{T}}$, and the normal equation of Gauss-Newton method is:

$$\left(\sum_{i=1}^{100} \mathbf{J}_i (\sigma^2)^{-1} \mathbf{J}_i^{\mathrm{T}} \right) \Delta \mathbf{x}_k = \sum_{i=1}^{100} -\mathbf{J}_i (\sigma^2)^{-1} e_i, \tag{6.41}$$

Of course, we can also choose to arrange all $\mathbf{J}_i$ in a row and write this equation in matrix form, but its meaning is consistent with the summation form. The following code demonstrates how this process works.

Listing 6.1: slambook2/ch6/gaussNewton.cpp

```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <Eigen/Core>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;

int main(int argc, char **argv) {
    double ar = 1.0, br = 2.0, cr = 1.0;         // ground-truth values
    double ae = 2.0, be = -1.0, ce = 5.0;        // initial estimation
    int N = 100;                                 // num of data points
    double w_sigma = 1.0;                        // sigma of the noise
    double inv_sigma = 1.0 / w_sigma;
    cv::RNG rng;                                 // Random number generator

    vector<double> x_data, y_data;      // the data
    for (int i = 0; i < N; i++) {
        double x = i / 100.0;
        x_data.push_back(x);
        y_data.push_back(exp(ar * x * x + br * x + cr) + rng.gaussian(w_sigma *
            w_sigma));
    }

    // start Gauss-Newton iterations
    int iterations = 100;
    double cost = 0, lastCost = 0;

    chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
```

```
    for (int iter = 0; iter < iterations; iter++) {
        Matrix3d H = Matrix3d::Zero();  // Hessian = J^T W^{-1} J in Gauss-Newton
        Vector3d b = Vector3d::Zero();  // bias
        cost = 0;

        for (int i = 0; i < N; i++) {
            double xi = x_data[i], yi = y_data[i];  // the i-th data
            double error = yi - exp(ae * xi * xi + be * xi + ce);
            Vector3d J; // jacobian
            J[0] = -xi * xi * exp(ae * xi * xi + be * xi + ce);  // de/da
            J[1] = -xi * exp(ae * xi * xi + be * xi + ce);  // de/db
            J[2] = -exp(ae * xi * xi + be * xi + ce);  // de/dc

            H += inv_sigma * inv_sigma * J * J.transpose();
            b += -inv_sigma * inv_sigma * error * J;

            cost += error * error;
        }

        // solve Hx=b
        Vector3d dx = H.ldlt().solve(b);
        if (isnan(dx[0])) {
            cout << "result is nan!" << endl;
            break;
        }

        if (iter > 0 && cost >= lastCost) {
            cout << "cost: " << cost << ">= last cost: " << lastCost << ", break." <<
                endl;
            break;
        }

        ae += dx[0];
        be += dx[1];
        ce += dx[2];

        lastCost = cost;

        cout << "total cost: " << cost << ", \t\tupdate: " << dx.transpose() <<
            "\t\testimated params: " << ae << "," << be << "," << ce << endl;
    }

    chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
    chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double
        >>(t2 - t1);
    cout << "solve time cost = " << time_used.count() << " seconds. " << endl;
    cout << "estimated abc = " << ae << ", " << be << ", " << ce << endl;
    return 0;
}
```

In this example, we demonstrate how to optimize a simple fitting problem iteratively. Through our handwritten code, it is easy to see the entire optimization process. The program outputs the objective function value and updates the amount of each iteration, as follows:

Listing 6.2: terminal output:

```
/home/xiang/Code/slambook2/ch6/cmake-build-debug/gaussNewton
total cost: 3.19575e+06,    update: 0.0455771  0.078164 -0.985329   estimated params:
     2.04558,-0.921836,4.01467
total cost: 376785,     update:  0.065762  0.224972 -0.962521   estimated params:
     2.11134,-0.696864,3.05215
total cost: 35673.6,    update: -0.0670241   0.617616  -0.907497    estimated params:
     2.04432,-0.0792484,2.14465
total cost: 2195.01,    update: -0.522767   1.19192 -0.756452   estimated params:
     1.52155,1.11267,1.3882
total cost: 174.853,    update: -0.537502  0.909933 -0.386395   estimated params:
     0.984045,2.0226,1.00181
total cost: 102.78,     update: -0.0919666   0.147331 -0.0573675    estimated params:
     0.892079,2.16994,0.944438
total cost: 101.937,    update: -0.00117081  0.00196749 -0.00081055   estimated params
     : 0.890908,2.1719,0.943628
```

```
total cost: 101.937,    update:   3.4312e-06 -4.28555e-06  1.08348e-06    estimated
     params: 0.890912,2.1719,0.943629
total cost: 101.937,    update: -2.01204e-08  2.68928e-08 -7.86602e-09    estimated
     params: 0.890912,2.1719,0.943629
cost: 101.937>= last cost: 101.937, break.
solve time cost = 0.000212903 seconds.
estimated abc = 0.890912, 2.1719, 0.943629
```

It is easy to see that the objective function of the whole problem approaches convergence after 9 iterations, and the updated amount approaches zero. The final estimated value is close to the true value, and the function image is shown in Figure 6-1. On my machine (my CPU is i7-8700), the optimization takes about 0.2 milliseconds. Let's try to use the optimized library to accomplish the same task.

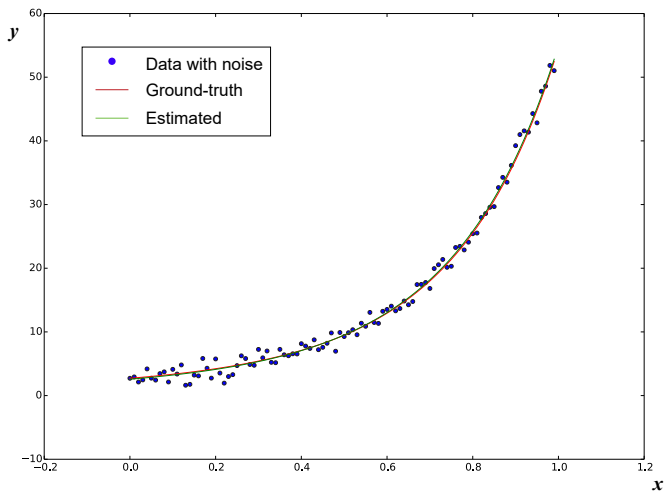


Figure 6-1: Estimated curve after fitting when the noise $\sigma = 1$.

6.3.2 Curve Fitting with Google Ceres

In the next two sections we introduce two C++ optimization libraries: the Ceres library [32] from Google and the g2o library [33] based on graph optimization. Since in g2o, we still need some knowledge about **graph** optimization, we first go through the Ceres here, then introduce some graph optimization theories, and finally talk about g2o. Since the optimization algorithms will still appear in the later "visual odometry" and "backend" chapters, please make sure you understand the meaning of the optimization algorithm and the content of the program.

Introduction to Ceres

Google Ceres is a widely used optimization library for least-squares problems. In Ceres, as users, we only need to define the optimization problem to be solved according to certain steps and then hand it over to the solver for calculation. The most general form of the least-squares problem solved by Ceres is as follows (kernel

function least squares with boundary):

$$\begin{aligned} \min_{x} \quad & \frac{1}{2}\sum_{i}\rho_i\left(\left\|f_i\left(x_{i_1},\cdots x_{i_n}\right)\right\|^2\right) \\ \text{s.t.} \quad & l_j \leqslant x_j \leqslant u_j. \end{aligned} \tag{6.42}$$

In this question, $x_1,\cdots,x_n$ are optimized variables, also called **parameter blocks** (Parameter blocks), f_i is called **cost function** (Cost function), also called residual Residual blocks can also be understood as error terms in SLAM. l_j and u_j are the upper and lower limits of the jth optimization variable. In the simplest case, take $l_j = -\infty, u_j = \infty$ (do not limit the boundary of the optimization variable). At this time, the objective function is composed of many square terms after a **kernel function**$\rho(\cdot)$ and then the sum of the *. Similarly, you can take ρ as the identity function. The objective function is the sum of the squares of many terms, and we get the unconstrained least squares problem, which is consistent with the previously introduced theory.

In order to tell Ceres the definition of the problem, we need to do the following things:

1. defines each parameter block. The parameter block is usually a trivial vector, but it can also be defined as a special structure such as quaternion and Lie algebra in SLAM. If it is a vector, we need to allocate a double array for each parameter block to store the variable's value.

2. Then, define the calculation method of the residual block. The residual block is usually associated with several parameter blocks, performs some custom calculations on them, and then returns the residual value. After Ceres sums the squares of them, it is used as the value of the objective function.

3. The residual block often also needs to define the Jacobian calculation method. In Ceres, you can use the "automatic derivative" function it provides, or you can manually specify the Jacobian calculation process. If you want to use automatic derivation, then the residual block needs to be written in a specific way: the residual calculation should be a bracketed operator with a template. We will illustrate this point through an example.

4. Finally, add all the parameter blocks and residual blocks to the Problem object defined by Ceres and call the Solve function to solve it. Before solving, we can pass some configuration information, such as the number of iterations, termination conditions, etc., or use the default configuration.

Next, let's actually operate Ceres to solve the curve fitting problem and understand the optimization process.

Install Ceres

In order to use Ceres, we first need to compile and install it. Ceres' GitHub address is `https://github.com/ceres-solver/ceres-solver`, and you can also directly use Ceres in our 3rdparty directory of the code repository so that you will use the same version as mine.

Like the libraries encountered before, Ceres is also a CMake project. Before compiling it, we need to install the dependencies first. You can install them with apt-get in Ubuntu, mainly some logging and testing tools used by Google itself:

* kernel function is discussed in Chapter 9.

Listing 6.3: terminal input:

```
sudo apt-get install liblapack-dev libsuitesparse-dev libcxsparse3 libgflags-dev
    libgoogle-glog-dev libgtest-dev
```

Then, enter the Ceres library directory, use cmake to compile and install it. We have done this process many times, so I won't repeat it here. After the installation is complete, find the Ceres header file under /usr/local/include/ceres, and find the library file named libceres.a under /usr/local/lib/. With these files, we are able to include the Ceres headers and do the optimization calculations.

Use Ceres for Curve Fitting

The following code demonstrates how to use Ceres to solve the same problem.

Listing 6.4: slambook/ch6/ceresCurveFitting.cpp

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <ceres/ceres.h>
#include <chrono>

using namespace std;

// residual
struct CURVE_FITTING_COST {
    CURVE_FITTING_COST(double x, double y) : _x(x), _y(y) {}

    // implement operator () to compute the error
    template<typename T>
    bool operator()(
        const T *const abc, // the estimated variables, 3D vector
        T *residual) const {
        // y-exp(ax^2+bx+c)
        residual[0] = T(_y) - ceres::exp(abc[0] * T(_x) * T(_x) + abc[1] * T(_x) + abc
            [2]);
        return true;
    }

    const double _x, _y;    // x,y data
};

int main(int argc, char **argv) {
    // same as before
    double ar = 1.0, br = 2.0, cr = 1.0;         // ground-truth values
    double ae = 2.0, be = -1.0, ce = 5.0;        // initial estimation
    int N = 100;                                 // num of data points
    double w_sigma = 1.0;                        // sigma of the noise
    double inv_sigma = 1.0 / w_sigma;
    cv::RNG rng;                                 // Random number generator

    vector<double> x_data, y_data;      // the data
    for (int i = 0; i < N; i++) {
        double x = i / 100.0;
        x_data.push_back(x);
        y_data.push_back(exp(ar * x * x + br * x + cr) + rng.gaussian(w_sigma *
            w_sigma));
    }

    double abc[3] = {ae, be, ce};

    // construct the problem in ceres
    ceres::Problem problem;
    for (int i = 0; i < N; i++) {
        problem.AddResidualBlock(     // add i-th residual into the problem
            // use auto-diff, template params: redisual type, output dimension, input
                dimension
            // shoule be same as the struct written before
            new ceres::AutoDiffCostFunction<CURVE_FITTING_COST, 1, 3>(
                new CURVE_FITTING_COST(x_data[i], y_data[i])
            ),
```

```
52              nullptr,            // kernel function, don't use here
53              abc                 // estimated variables
54          );
55      }
56
57      // set the solver options
58      ceres::Solver::Options options;     // actually there're lots of params can be
            adjusted
59      options.linear_solver_type = ceres::DENSE_NORMAL_CHOLESKY;  // use cholesky to
            solve the normal equation
60      options.minimizer_progress_to_stdout = true;   // print to cout
61
62      ceres::Solver::Summary summary;
63      chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
64      ceres::Solve(options, &problem, &summary);  // do optimization!
65      chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
66      chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double
            >>(t2 - t1);
67      cout << "solve time cost = " << time_used.count() << " seconds. " << endl;
68
69      // get the outpus
70      cout << summary.BriefReport() << endl;
71      cout << "estimated a,b,c = ";
72      for (auto a:abc) cout << a << " ";
73      cout << endl;
74
75      return 0;
76  }
```

The code is quite self-explained because we write a lot of comments here. As you can see, we used OpenCV's noise generator to generate 100 data with Gaussian noise and then used Ceres for fitting. The Ceres usage demonstrated here has the following steps:

1. Defines the class of the residual block. The method is to write a class (or structure), and define the () operator with template parameters in the class so that the class becomes a **Functor** *. This way of definition allows Ceres to call the <double>() method for an instance of this class like a function. In fact, Ceres will pass the Jacobian matrix as a type parameter to this function to realize the function of automatic derivation (auto-diff, which is one of the best feature of Ceres).

2. The double array abc[3] in the program is the parameter block, and for the residual block, we construct a CURVE_FITTING_COST object for each data and then call AddResidualBlock to add the error term to the objective function. Since optimization requires gradients, we have several options: (1) Use Ceres's Auto Diff feature; (2) Use Numeric Diff (Numeric Diff) †; (3) Derive the analytical derivative form by yourself and provide it to Ceres. Because automatic derivation is the most convenient in coding, we use automatic derivation here.

3. Automatic derivation requires specifying the dimensions of the error term and optimization variable. The error here is a scalar with a dimension of 1; the optimized three quantities are a, b, c with 3. Therefore, the variable dimensions are set to 1, 3 in the template parameter of the auto-derivation class AutoDiffCostFunction.

---

* A C++ term, such a class can be called as if it were a function because the operator () is overloaded.

† Automatic derivation is also implemented with numerical derivatives, but since it is a template operation, it runs faster.

4. After setting the problem, call the Solve function to solve it. You can configure (very detailed) optimization options in Ceres. For example, you can choose to use Line Search or Trust Region, the number of iterations, step size, and so on. Readers can check the definition of Options to see what optimization methods are available. Of course, the default configuration can be used for a wide range of problems.

Finally let's see the experimental results by calling build/ceresCurveFitting:

Listing 6.5: terminal output:

```
iter      cost      cost_change  |gradient|   |step|    tr_ratio  tr_radius
    ls_iter  iter_time  total_time
0  1.597873e+06    0.00e+00    3.52e+06   0.00e+00   0.00e+00  1.00e+04        0
        2.10e-05    7.92e-05
1  1.884440e+05    1.41e+06    4.86e+05   9.88e-01   8.82e-01  1.81e+04        1
        5.60e-05    1.05e-03
2  1.784821e+04    1.71e+05    6.78e+04   9.89e-01   9.06e-01  3.87e+04        1
        2.00e-05    1.09e-03
3  1.099631e+03    1.67e+04    8.58e+03   1.10e+00   9.41e-01  1.16e+05        1
        6.70e-05    1.16e-03
4  8.784938e+01    1.01e+03    6.53e+02   1.51e+00   9.67e-01  3.48e+05        1
        1.88e-05    1.19e-03
5  5.141230e+01    3.64e+01    2.72e+01   1.13e+00   9.90e-01  1.05e+06        1
        1.81e-05    1.22e-03
6  5.096862e+01    4.44e-01    4.27e-01   1.89e-01   9.98e-01  3.14e+06        1
        1.79e-05    1.25e-03
7  5.096851e+01    1.10e-04    9.53e-04   2.84e-03   9.99e-01  9.41e+06        1
        1.81e-05    1.28e-03
solve time cost = 0.00130755 seconds.
Ceres Solver Report: Iterations: 8, Initial cost: 1.597873e+06, Final cost:
    5.096851e+01, Termination: CONVERGENCE
estimated a,b,c = 0.890908 2.1719 0.943628
```

The final optimized value is basically the same as our experimental result in the previous section, but Ceres is relatively slow in running speed. Ceres used about 1.3 milliseconds on my machine, which is about six times slower than the handwritten Gauss-Newton method.

I hope readers have a general understanding of how to use Ceres through this simple example. Its advantage is that it provides an automatic derivation tool, making it unnecessary to calculate the cumbersome Jacobian matrix. Ceres's automatic derivation is realized through template elements, and the automatic derivation can be completed at compile-time, but it is still a numerical derivative. Most of the time in this book, I will still introduce the calculation of the Jacobian matrix because it is more helpful to understand the problem, and there are fewer problems in the optimization. In addition, Ceres' optimization process configuration is also very rich, making it suitable for a wide range of least squares optimization problems, including various problems other than SLAM.

6.3.3 Curve Fitting with g2o

The second practice part of this lecture is about another optimization library (widely used mainly in the SLAM field): g2o (General Graphic Optimization, G^2O). It is a library based on **graphic optimization**. Graph optimization is a theory that combines nonlinear optimization with graph theory, so before using it, let's spend a little space to introduce graph optimization theory.

Introduction to Graph Optimization Theory

We have introduced the solution of nonlinear least squares. The least square problem in SLAM is normally composed of the sum of many small error terms. However, if we only treat them as variables and residuals, it would be hard to tell the **relationship** between them. For example, how many error terms are related to a certain optimization variable x_j? If we adjust some variables, does the overall cost function changes or keep the same? Furthermore, we hope to visually see the what optimization problem **looks like**. Therefore, the graph optimization is involved.

Graph optimization is a way to express the optimization problem as **Graph**. A graph consists of a number of **vertices** and **edges** connecting these vertices. A **vertex** is used to represent an **optimization variable**, and a **edge** is used to represent an **error term**. Therefore, for any of the above-mentioned nonlinear least squares problem, we can construct a corresponding **graph**. We can simply call it **graph**, or use the definition in the probability graph, call it **Bayesian graph** or **factor graph**. Sometimes they are also called as **hyper graph** because an edge can be connect to more than two variables, e.g., where an error term is related to more than two variables.

Figure 6-2 is a simple graph optimization example. We use triangles to represent the camera poses, and circles to represent landmark points, which constitute the vertices of the graph optimization; at the same time, the solid line represents the camera's motion model, and the dashed line represents the observation model, which form the edges of the graph optimization. At this point, although the mathematical form of the entire problem is still like (6.13), now we can intuitively see the **structure** of the problem. If you want, you can also make improvements like **remove isolated vertices** or **preferentially optimize vertices with more edges (or in graph theory terms, greater degrees)**. But the most basic graph optimization is just to use a graph model to express a nonlinear least-squares optimization problem. And we can use some properties of the graph model to do better optimization.

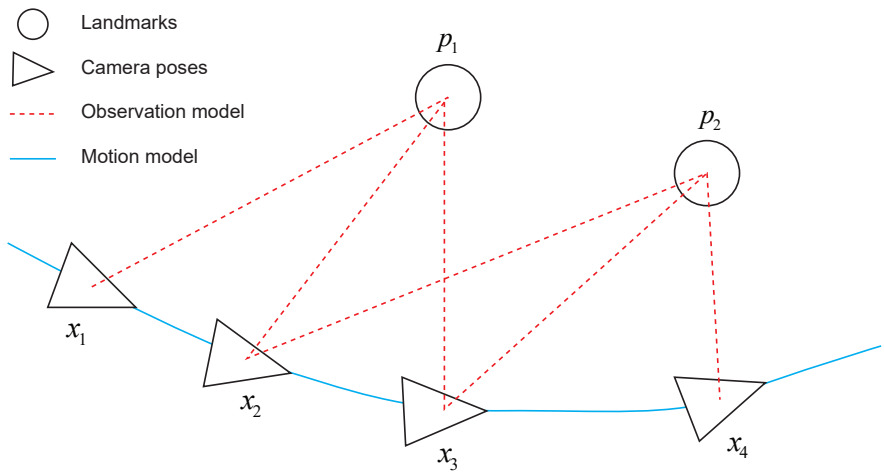


Figure 6-2: An example of the graph optimization.

The g2o is a **general** graph optimization library. "General" means that you

can solve any least squares problem that can be expressed as graph optimization, obviously including the curve fitting problem discussed above. Let us demonstrate how to do that.

Compilation and Installation of G2o

Just like the Ceres section, we should first compile and install the g2o library. Readers should have experienced this process many times, and they are basically the same. Regarding g2o, readers can download it from GitHub: `https://github.com/RainerKuemmerle/g2o`, or obtain it from the third-party code library provided in this book. Since g2o is still being updated, I suggest you use g2o under 3rdparty to ensure that the version is the same as mine.

g2o is also a cmake project. Let's install its dependencies first (some dependencies overlap with Ceres):

Listing 6.6: terminal input:

```
sudo apt-get install qt5-qmake qt5-default libqglviewer-dev-qt5 libsuitesparse-dev
    libcxsparse3 libcholmod3
```

Then, compile and install g2o according to the cmake method. The description of the process is omitted here. After the installation is complete, the header files of g2o will be located under /usr/local/g2o, and the library files will be located under /usr/local/lib/. Now, we reconsider the curve fitting experiment in the Ceres routine, and do that experiment again in g2o.

Curve Fitting with g2o

In order to use g2o, the curve fitting problem must first be constructed into graph optimization. In this process, just remember that **nodes are the optimization variables, and edges are the error terms**. The graph optimization problem of curve fitting can be drawn in the form of Figure 6-3 .

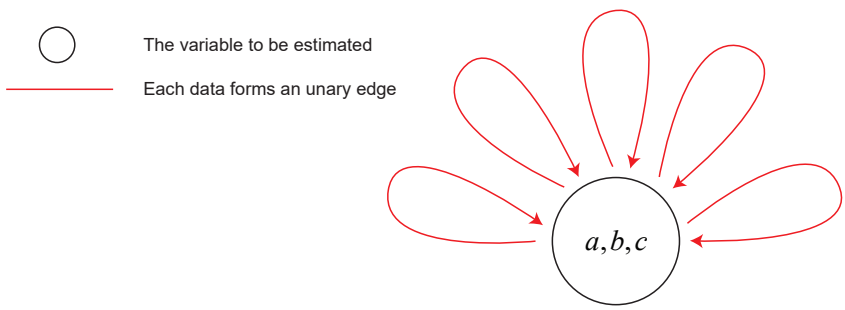


Figure 6-3: Graph model of the curve fitting problem.

The entire problem has only one vertex in the curve fitting problem: the parameters a, b, c of the curve model; Each noisy data point constitutes an error term, which is the edge of the graph optimization. But the edges here are not the same as we usually think. They are **Unary Edge** (Unary Edge), that is, **connect only one vertex**-because the entire graph has only one vertex. So in Figure 6-3 , we can only draw it as if it is connected to itself. In fact, an edge in graph optimization can connect one, two or more vertices, which mainly reflects how many optimization

variables each error is related to. In a slightly mysterious way, we call it **Hyper Edge**, and the whole graph is called **Hyper Graph** *.

After clarifying the graph model, the next step is to build the model in g2o for optimization. As a user of g2o, what we need to do mainly includes the following steps:

1. Define the type of vertices and edges.
2. Build the graph.
3. Select optimization algorithm.
4. Call g2o to optimize and get the result.

This part is very similar to Ceres, of course there will be some differences in coding. Let's demonstrate the program.

Listing 6.7: slambook/ch6/g2oCurveFitting.cpp

```
#include <iostream>
#include <g2o/core/g2o_core_api.h>
#include <g2o/core/base_vertex.h>
#include <g2o/core/base_unary_edge.h>
#include <g2o/core/block_solver.h>
#include <g2o/core/optimization_algorithm_levenberg.h>
#include <g2o/core/optimization_algorithm_gauss_newton.h>
#include <g2o/core/optimization_algorithm_dogleg.h>
#include <g2o/solvers/dense/linear_solver_dense.h>
#include <Eigen/Core>
#include <opencv2/core/core.hpp>
#include <cmath>
#include <chrono>

using namespace std;

// vertex: 3d vector
class CurveFittingVertex : public g2o::BaseVertex<3, Eigen::Vector3d> {
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW

    // override the reset function
    virtual void setToOriginImpl() override {
        _estimate << 0, 0, 0;
    }

    // override the plus operator, just plain vector addition
    virtual void oplusImpl(const double *update) override {
        _estimate += Eigen::Vector3d(update);
    }

    // the dummy read/write function
    virtual bool read(istream &in) {}
    virtual bool write(ostream &out) const {}
};

// edge: 1D error term, connected to exactly one vertex
class CurveFittingEdge : public g2o::BaseUnaryEdge<1, double, CurveFittingVertex> {
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW

    CurveFittingEdge(double x) : BaseUnaryEdge(), _x(x) {}

    // define the error term computation
    virtual void computeError() override {
        const CurveFittingVertex *v = static_cast<const CurveFittingVertex *> (
            _vertices[0]);
        const Eigen::Vector3d abc = v->estimate();
```

* I personally would better call it just as a graph to avoid being mysterious.

```
48          _error(0, 0) = _measurement - std::exp(abc(0, 0) * _x * _x + abc(1, 0) * _x +
                abc(2, 0));
49      }
50
51      // the jacobian
52      virtual void linearizeOplus() override {
53          const CurveFittingVertex *v = static_cast<const CurveFittingVertex *> (
                _vertices[0]);
54          const Eigen::Vector3d abc = v->estimate();
55          double y = exp(abc[0] * _x * _x + abc[1] * _x + abc[2]);
56          _jacobianOplusXi[0] = -_x * _x * y;
57          _jacobianOplusXi[1] = -_x * y;
58          _jacobianOplusXi[2] = -y;
59      }
60
61      virtual bool read(istream &in) {}
62      virtual bool write(ostream &out) const {}
63  public:
64      double _x;  // x data, note y is given in _measurement
65  };
66
67  int main(int argc, char **argv) {
68      // ... we omit the data sampling code, same as before
69      typedef g2o::BlockSolver<g2o::BlockSolverTraits<3, 1>> BlockSolverType;  // bolck
            solver
70      typedef g2o::LinearSolverDense<BlockSolverType::PoseMatrixType> LinearSolverType;
            // linear solver
71
72      // choose the optimizatoin method from GN, LM, DogLeg
73      auto solver = new g2o::OptimizationAlgorithmGaussNewton(
74          g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>()));
75      g2o::SparseOptimizer optimizer;     // graph optimizer
76      optimizer.setAlgorithm(solver);     // set the algorithm
77      optimizer.setVerbose(true);         // print the results
78
79      // add vertex
80      CurveFittingVertex *v = new CurveFittingVertex();
81      v->setEstimate(Eigen::Vector3d(ae, be, ce));
82      v->setId(0);
83      optimizer.addVertex(v);
84
85      // add edges
86      for (int i = 0; i < N; i++) {
87          CurveFittingEdge *edge = new CurveFittingEdge(x_data[i]);
88          edge->setId(i);
89          edge->setVertex(0, v);                // connect to the vertex
90          edge->setMeasurement(y_data[i]);      // measurement
91          edge->setInformation(Eigen::Matrix<double, 1, 1>::Identity() * 1 / (w_sigma *
                w_sigma)); // set the information matrix
92          optimizer.addEdge(edge);
93      }
94
95      // carry out the optimization
96      cout << "start optimization" << endl;
97      chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
98      optimizer.initializeOptimization();
99      optimizer.optimize(10);
100     chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
101     chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double
            >>(t2 - t1);
102     cout << "solve time cost = " << time_used.count() << " seconds. " << endl;
103
104     // print the results
105     Eigen::Vector3d abc_estimate = v->estimate();
106     cout << "estimated model: " << abc_estimate.transpose() << endl;
107
108     return 0;
109 }
```

In this program, we derive graph optimization vertices and edges for curve fitting from g2o: CurveFittingVertex and CurveFittingEdge, which essentially expands the use of g2o. These two classes are derived from BaseVertex and BaseUnaryEdge, respectively. In the derived classes, we have rewritten some important virtual func-

tions:

1. Vertex update function: oplusImpl. We know that the most important thing in the optimization process is the calculation of incremental $\Delta\mathbf{x}$, and this function deals with $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}$ process.

   Readers may think this is not something worth mentioning because it is just simple addition. Why doesn't g2o help us complete it? In the curve fitting process, since the optimization variables (curve parameters) are located in plain **vector space**, this update calculation is indeed nothing but just a simple addition. However, when the optimization variable is not in the vector space, for example, for $\mathbf{x}$ is the camera pose, it does not necessarily have an addition operation. At this time, it is necessary to redefine the behavior of **how the increment is added to the existing estimate**. According to the explanation in Lecture 4, we may use the left-multiplication update or the right-multiplication update instead of direct addition.

2. Vertex reset function: setToOriginImpl. This is trivial, and we just set the estimate to zero.

3. The error calculation function: computeError. This function needs to take out the current estimated value of the vertex connected by the edge and compare it with its observed value according to the curve model. This is consistent with the error model in the least squares problem.

4. The Jacobian calculation function: linearizeOplus. In this function, we calculate the Jacobian of each edge relative to the vertex.

5. Save and read functions: read, write. Since we do not want to perform read/write operations, leave it blank.

After defining the vertices and edges, we declare a graph model in the main function, then add vertices and edges to the graph model according to the generated noise data, and finally call the optimization function for optimization. g2o will give the optimized result:

Listing 6.8: terminal output:

```
start optimization
iteration= 0   chi2= 376785.128234   time= 3.3299e-05  cumTime= 3.3299e-05   edges= 100  schur= 0
iteration= 1   chi2= 35673.566018  time= 1.3789e-05  cumTime= 4.7088e-05   edges= 100  schur= 0
iteration= 2   chi2= 2195.012304   time= 1.2323e-05  cumTime= 5.9411e-05   edges= 100  schur= 0
iteration= 3   chi2= 174.853126  time= 1.3302e-05  cumTime= 7.2713e-05   edges= 100  schur= 0
iteration= 4   chi2= 102.779695  time= 1.2424e-05  cumTime= 8.5137e-05   edges= 100  schur= 0
iteration= 5   chi2= 101.937194  time= 1.2523e-05  cumTime= 9.766e-05  edges= 100  schur= 0
iteration= 6   chi2= 101.937020  time= 1.2268e-05  cumTime= 0.000109928  edges= 100  schur= 0
iteration= 7   chi2= 101.937020  time= 1.2612e-05  cumTime= 0.00012254   edges= 100  schur= 0
iteration= 8   chi2= 101.937020  time= 1.2159e-05  cumTime= 0.000134699  edges= 100  schur= 0
iteration= 9   chi2= 101.937020  time= 1.2688e-05  cumTime= 0.000147387  edges= 100  schur= 0
solve time cost = 0.000919301 seconds.
estimated model: 0.890912   2.1719 0.943629
```

We use the Gauss-Newton method for gradient descent, and after 9 iterations, the optimization result is obtained, which is almost the same as the Ceres and handwritten Gauss-Newton method. From the running speed perspective, our experimental conclusion is that handwriting is faster than g2o, and g2o is faster than Ceres. This is a generally intuitive experience that versatility and efficiency are often contradictory. However, in this experiment, Ceres uses automatic derivation, and the solver configuration is not completely consistent with Gauss-Newton, so it seems slower.

6.4 Summary

This section introduces a nonlinear optimization problem often encountered in SLAM: the least squares problem consisting of the sum of squares of many error terms. We introduced its definition and solution and discussed two main gradient descent methods: the Gauss-Newton method and the Levenberg-Marquardt method. In the practice part, two optimization libraries of the handwritten Gauss-Newton method, Ceres and g2o, were used to solve the same curve fitting problem and found that they gave similar results.

Since Bundle Adjustment has not been discussed in detail, we have chosen a simple but representative example of curve fitting in the practice part to demonstrate the general nonlinear least squares solution method. In particular, if you use g2o to fit a curve, you must first convert the problem to graph optimization and define new vertices and edges. This approach is somewhat roundabout—the main purpose of g2o is not here. In contrast, the usage of Ceres is much more natural because it is designed to be a general optimization library. However, the more problem in SLAM is how to solve an optimization problem with many camera poses and many spatial points. In particular, when the camera pose is represented by Lie algebra, calculating the derivative of the camera pose in the error term will be worthy of detailed discussion. We will find in the follow-up content that g2o provides a large number of ready-to-use vertices and edges, which is very convenient for the camera pose estimation problem. In Ceres, we have to implement each Cost Function ourselves, which has some inconveniences.

In the two programs of the practical part, we did not calculate the derivative of the curve model with respect to the three parameters but used the numerical derivative of the optimization library, which made the theory and code simpler. The Ceres library provides automatic derivation based on template elements and numerical derivation at runtime, while g2o only provides numerical derivation method at runtime. However, for most problems, if you can derive the analytical form of the Jacobian matrix and tell the optimization library, you can avoid many problems in numerical derivation.

Finally, I hope readers can adapt to Ceres and g2o's extensive use of template programming. It may seem scary at first (especially the parenthesis operator for Ceres to set the residual block and the code for the g2o initialization part), but once you are familiar with it, you will feel that this method is natural and easy to extend. We will continue to discuss issues such as sparsity, kernel functions, and pose graphs in the SLAM backend lecture.

Exercises

1. Prove that the linear equation $\mathbf{Ax} = \mathbf{b}$ when the coefficient matrix $\mathbf{A}$ is overdetermined, the least square solution is $\mathbf{x} = (\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{b}$.

2. Investigate the advantages and disadvantages of the steepest descent method, Newton method, Gauss-Newton method, and Levenberg-Marquardt method. In addition to the Ceres library and g2o library we cited, what other commonly used optimization libraries are there? (You may find some libraries on MATLAB.)

3. Why the coefficient matrix of the Gauss-Newton method's incremental equation may not be positive definite? What is the geometric meaning of indefinite? Why is the solution unstable in this case?

4. What is DogLeg? What are the similarities and differences between it and the Gauss-Newton method and the Levenberg-Marquardt method? Please search for relevant materials *.

5. Read Ceres' teaching materials (`http://ceres-solver.org/tutorial.html`) to grasp its usage better.

6. Read the documentation that comes with g2o, can you understand it? If you still can't fully understand it, please come back after lectures 10 and 11.

7.* Please change the curve model in the curve fitting experiment, and use Ceres and g2o to optimize the experiment. For example, write an example with more parameters and more complex models.

* e.g.`http://www.numerical.rl.ac.uk/people/nimg/course/lectures/raphael/lectures/lec7slides.pdf`.

Chapter 7

Visual Odometry: Part 1

Goal of Study

1. Understand the meaning of image feature points, and learn how to extract feature points from a single image and match feature points in multiple images.
2. Understand the principle of epipolar geometry and use epipolar constraints to recover the camera's 3D motion between two images.
3. Study how to solve the PNP problem and use the known correspondence between the 3D structure and the 2D image to solve the camera's 3D motion.
4. Understand the ICP problem and use the point clouds matching to solve the three-dimensional motion of the camera.
5. Understand how to obtain the three-dimensional structure of corresponding points on the two-dimensional image through triangulation.

In the previous chapters, we introduced the details of motion and observation equations, and explained nonlinear optimization methods to solve those equations. From this chapter, we finish the introduction to the fundamental knowledge and move on to the main topic: in the order of Chapter 2, we will introduce visual odometry, optimization of backend, loop detection, and map reconstruction. This chapter and the next chapter mainly focus on two commonly used methods in visual odometry: feature point method and optical flow method. In this chapter, we will introduce what feature points are, how to extract and match them, and how to estimate camera motion based on matched feature points.

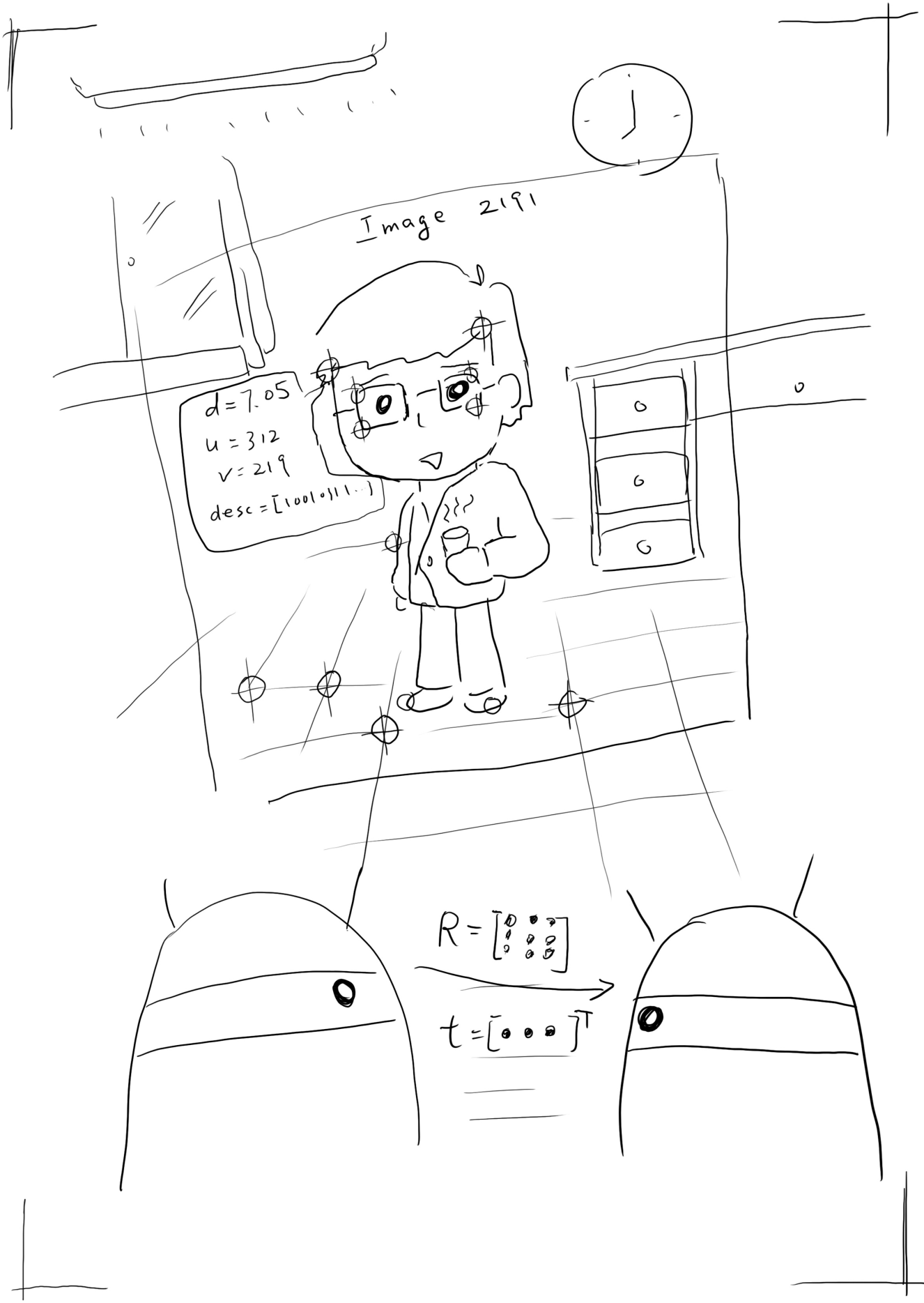


Image 2191
d = 7.05
u = 312
v = 219
desc = [10010111...]
R = [ ]
t = [ ]^T

7.1 Feature Point Method

In the chapter 2, we said that a SLAM system can be divided into front-end and back-end, where the front-end is also called visual odometry (VO). VO estimates the rough camera movement based on the information of the consecutive images, and provides a good initial value for the back end. VO algorithms are mainly divided into two categories: **feature point method** and **direct method**. The front end based on the feature point method has been considered the mainstream method of visual odometry for a long time (even until now). It performs well thanks to its stability and insensitivity to lighting and dynamic objects. It is relatively mature at present. In this chapter, we will start with the feature point method, learn how to extract and match image feature points, and then estimate the camera motion and scene structure between two frames to realize a visual odometer between two frames. This type of algorithm is sometimes called Two-View geometry.

7.1.1 Feature Point

The core problem of VO is **how to estimate camera motion based on images**. However, the image itself is a numerical matrix encoding brightness and color. It is abstract and very difficult to consider motion estimation directly from the matrix level. Therefore, it is more convenient to do this: First, select some **representative (feature) points** from the image. These points will remain the same after a small change in the camera's angle of view, so that we can find the same points in each image. Then, on the basis of these points, the problem of camera pose estimation and the positioning of these points are discussed. In the classic SLAM model, we call these points **landmark**. In visual SLAM, they are always referred as image features.

On the Wikipedia, image features are defined as a set of information related to computing tasks. The computing tasks depend on the specific application [34]. Briefly speaking, **feature is another digital expression of image information**. A good set of features is crucial to the final performance on the specified task, so researchers have comprehensive work on the features for many years. Digital images are stored in a computer as a gray value matrix, so at the simplest, a single image pixel can be also considered a "feature". However, in visual odometry, we hope that **feature points remain stable after the camera moves**, and the gray value is badly affected by illumination, deformation, and object material. It varies greatly between different images and is not stable enough. Ideally, when the scene and camera angle of view change slightly, the algorithm can also determine from the images which places refer to the same point. Therefore, the gray value alone is not feasible, we need to extract feature points from the image.

We can say that the feature points are some **special places** in the image. Taking Figure 7-1 as an example, we can see the corners, edges and blocks as representative places in the image. It is easy for us to correctly point out that the same corner point appears in two images; whereas pointing out the same edge is slightly more difficult, because the image patches are similar when moving along the edge; it is even harder as for the same block. We found that the corners and edges in the image are more "special" than pixels, and they are more distinguished between different images. Therefore, an intuitive way to extract features is to identify corners between different images and then determine their correspondence. In this approach, the corners are the so-called features. There are many corner extraction algorithms,

such as Harris corner [35], FAST corner [36], GFTT corner point [37], etc. Most of them were proposed before 2000.

However, in the majority of applications, a single corner still cannot meet our needs. For example, a place that appears to be a corner point from a long distance may not be viewed as a corner when the camera steps in. Or, when the camera rotates, the appearance of the corner points will change, and it is not easy for us to recognize that they are the same corner point. For this reason, researchers in the field of computer vision have designed many more stable local image features during years of research, such as the SIFT[38], SURF[39] , ORB[40], etc. Compared with simple corner points, these handcrafted features should have the following properties:

1. *Repeatability*: The same feature can be found in different images.
2. *Distinctiveness*: Different features have different expressions.
3. *Efficiency*: In the same image, the number of feature points should be far smaller than the number of pixels.
4. *Locality*: The feature is only related to a small image area.

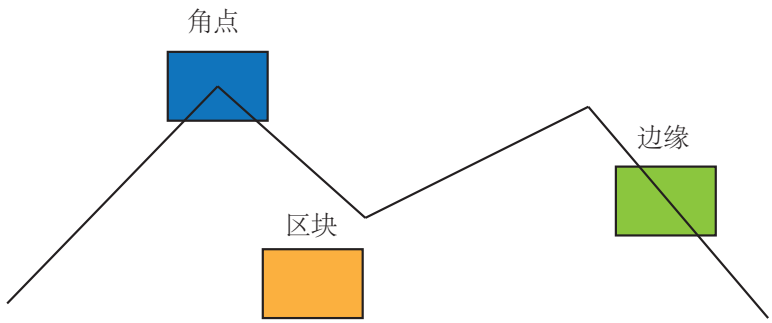


Figure 7-1: corners, edges, blocks can be used as image features.

A feature point is composed of two parts: **Key-point** and **Descriptor**. For example, when we say "calculate SIFT feature points in an image", we mean "extract SIFT key points and calculate SIFT descriptors". The key point refers to the position of the feature point in the image, and some types of feature points also hold information such as orientation and size. The descriptor is usually a vector, describing the information of the pixels around the key point according to some handcrafted rules. The descriptor should be designed according to the principle of "**features with similar appearance should have similar descriptors**". Therefore, as long as the descriptors of two feature points are close in vector space, they can be considered as the same feature point.

Historically, researchers put forward many image features. Some of them are very accurate and robust. They still have similar expressions under camera movement and lighting changes, and consequentially they might require a large amount of calculation. Among them, SIFT (Scale-Invariant Feature Transform) is one of the most classic. To fully consider the changes in illumination, scale, and rotation during the image transformation, it comes with a huge amount of calculation. The extraction and matching of image features is only a part compared to the entire SLAM process. Until now (2016), CPUs equipped on PCs cannot achieve real time

to calculate SIFT features for localization and mapping *. So we rarely use this "luxury" image feature in SLAM.

Some other features exchange some accuracy and robustness for the calculation speed increase. For example, the FAST key point is a feature point that is extremely fast to calculate (note the expression of "key point" here, which means that it has no descriptor), while the ORB (Oriented FAST and Rotated BRIEF) feature is currently widely used for real-time image feature extraction. It solves the problem that the FAST detector [36] does not have directionality, and uses the extremely fast binary descriptor BRIEF[41] to make the whole image feature extraction process accelerate greatly. According to the author's experiment in the paper, extracting about 1000 feature points in the same image, it takes about 15.3ms for ORB, 217.3ms for SURF, and 5228.7ms for SIFT. It can be seen that ORB made a significant boost in speed while maintaining the features of rotation and scale invariance. It is a good choice for SLAM with high real-time requirements.

Most feature extractions have good parallelism and can be accelerated by GPU and other devices. SIFT accelerated by GPU can meet real-time requirements. However, the inclusion of GPU will increase the cost of the entire SLAM system. Whether the resulting performance improvement is sufficient to offset the computational cost requires careful consideration by the system designer.

Obviously, there are a large number of feature points in the field of computer vision, and we cannot introduce them one by one in the book. In the current SLAM scheme, ORB is a good tradeoff between quality and performance. Therefore, we take ORB as an example to introduce the entire process of extracting features. If readers are interested in feature extraction and matching algorithms, we recommend reading related books in this area[42].

7.1.2 ORB Feature

ORB features are also composed of two parts: **key points** and **descriptor**. Its key point is called "Oriented FAST", which is an improved FAST corner point. We will introduce what is FAST corner point below. Its descriptor is called BRIEF (Binary Robust Independent Elementary Feature). Therefore, the extraction of ORB features is divided into the following two steps:

1. FAST corner point extraction: find the "corner point" in the image. Compared with the original FAST, the main direction of the feature points is calculated in ORB, which makes the subsequent BRIEF descriptor rotation-invariant.

2. BRIEF descriptor: describe the surrounding image area where the feature points were extracted in the previous step. ORB has made some improvements to BRIEF, mainly referring to utilizing the previously calculated direction.

Then we will introduce FAST and BRIEF respectively.

FAST key points

FAST is a kind of corner point, which mainly detects the obvious grayscale changes locally, and is known for its fast speed. Its main idea is: if a pixel is very different from the neighboring pixels (too bright or too dark), then it is more likely to be a corner point. Compared with other corner detection algorithms, FAST only needs

* here refers Real-time speed as of 30Hz.

to compare the brightness of the pixels, which is very fast. Its entire procedure is as follows (see Figure 7-2):

1. Select pixel p in the image assuming its brightness as I_p
2. Set a threshold T (for example, 20% of I_p).
3. Take the pixel p as the center, and select the 16 pixels on a circle with a radius of 3.
4. If there are consecutive N points on the selected circle whose brightness is greater than I_p+T or less than I_p-T, then the central pixel p can be considered a feature point (N usually takes 12, which is FAST-12. Other commonly used N values are 9 and 11, which are called FAST-9 and FAST-11 respectively).
5. Iterate through the above four steps on each pixel.

In the FAST-12 algorithm, to speed up, checking the brightness of the 1, 5, 9 and 13-th pixels on the circle for each pixel can quickly exclude a lot of pixels that are not corner points. Only when 3 of these 4 pixels are greater than $I_p + T$ or less than $I_p - T$, the current pixel may potentially be a corner point, otherwise it should be directly excluded. Such 'pre-processing' operation greatly accelerates FAST corner detection. In addition, the original FAST corners are often "clustered", meaning a lot of FAST corners present in the same area. Therefore, after the initial detection, non-maximal suppression is required. Only corner points with maximum response in a certain area will be retained to avoid the corners concentrating.

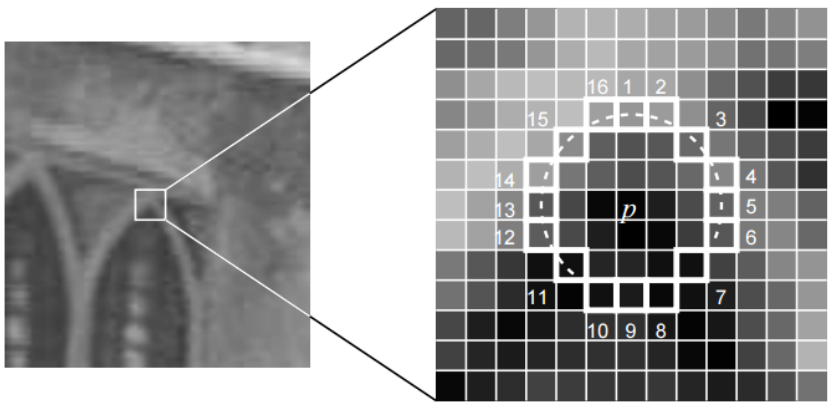


Figure 7-2: FAST key points[36]

The calculation of FAST feature points only compares the brightness difference between pixels, thus the speed is very fast, but it suffers from bad repeatability and uneven distribution. Moreover, FAST corner points do not include direction information. At the same time, because it fixed the radius of circle as 3, there is also a scaling problem: a place that looks like a corner from a distance may not be a corner when it comes close. To solve those, ORB adds the description of scale and rotation. The scale invariance is achieved by the image pyramid *, and detect

* pyramid refers to the downsampling of images at different levels to obtain images with different resolutions.

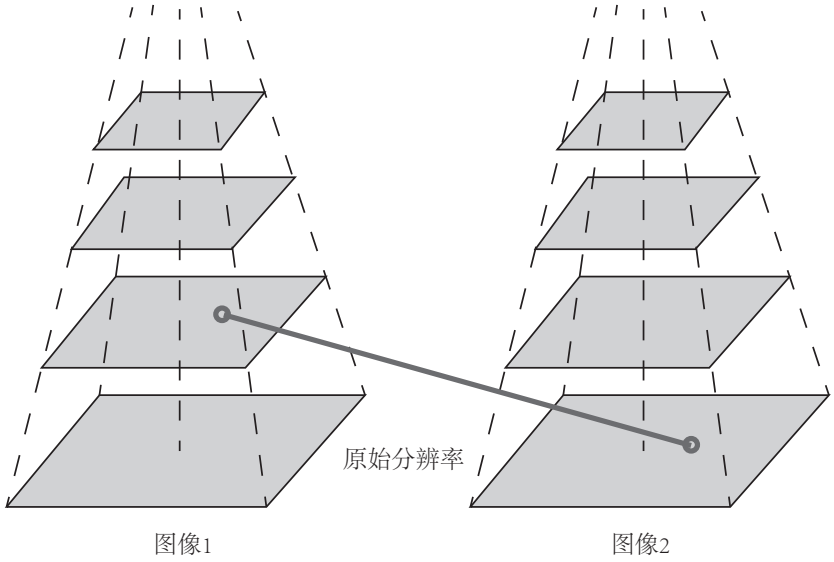


Figure 7-3: Use pyramids to match images at different resolutions.

corner points on each layer of the pyramid. The rotation of features is realized by the Intensity Centroid method.

Pyramid is a common approach in computer vision. For a schema, see Figure 7-3. The bottom of the pyramid is the original image. For each layer up, the image is scaled with a fixed ratio, so that we have images of different resolutions. The smaller image can be seen as a scene viewed from a distance. In the feature matching algorithm, we can match images on different layers to achieve scale invariance. For example, if the camera is moving backwards, then we should be able to find a match in the upper layer of the previous image pyramid and the lower layer of the next image.

In terms of rotation, we calculate the gray centroid of the image near the feature point. The so-called centroid refers to the gray value of the image block as the center of weight. The specific steps are as follows [43]

1. In a small image block B, define the moment of the image block as

$$m_{pq} = \sum_{x,y \in B} x^p y^q I(x, y), \quad p, q = \{0, 1\}.$$

2. calculate the centroid of the image block by the moment:

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right).$$

3. Connect the geometric center O and the centroid C of the image block to get a direction vector $\overrightarrow{OC}$, so the direction of the feature point can be defined as

$$\theta = \arctan(m_{01}/m_{10}).$$

Now, FAST corner points have a description of scale and rotation, which greatly improves the robustness of their representation between different images. This improved FAST is called Oriented FAST in ORB.

BRIEF descriptor

After extracting the Oriented FAST key points, we calculate the descriptor for each point. ORB uses an improved BRIEF feature description. Let's first introduce what BRIEF is.

BRIEF is a **binary** descriptor. Its description vector consists of many 0s and 1s, where 0s and 1s encode the size relationship between two random pixels near the key point (such as p and q) : If p is greater than q, then take 1, otherwise take 0. If we take 128 such p, q pairs, we will finally get a 128-dimensional vector [41] consisting of 0s and 1s. BRIEF implements the comparison of randomly selected points, which is very fast, and since it expresses in binary, it is also very convenient to store and suitable for real-time image matching. The original BRIEF descriptor does not have rotation invariance, so it is easy to get lost when the image is rotated. The ORB calculates the direction of the key points in the FAST feature point extraction stage, so the direction information can be used to calculate the "Steer BRIEF" feature after the rotation, so that the ORB descriptor has better rotation invariance.

Due to the consideration of rotation and scaling, the ORB still performs well under the translation, rotation and scaling. Meanwhile, the combination of FAST and BREIF is very efficient, which makes ORB features very popular in real-time SLAM. In Figure 7-4 , we show the result of extracting ORB from an image. In the following, we will move on to feature matching between different images.

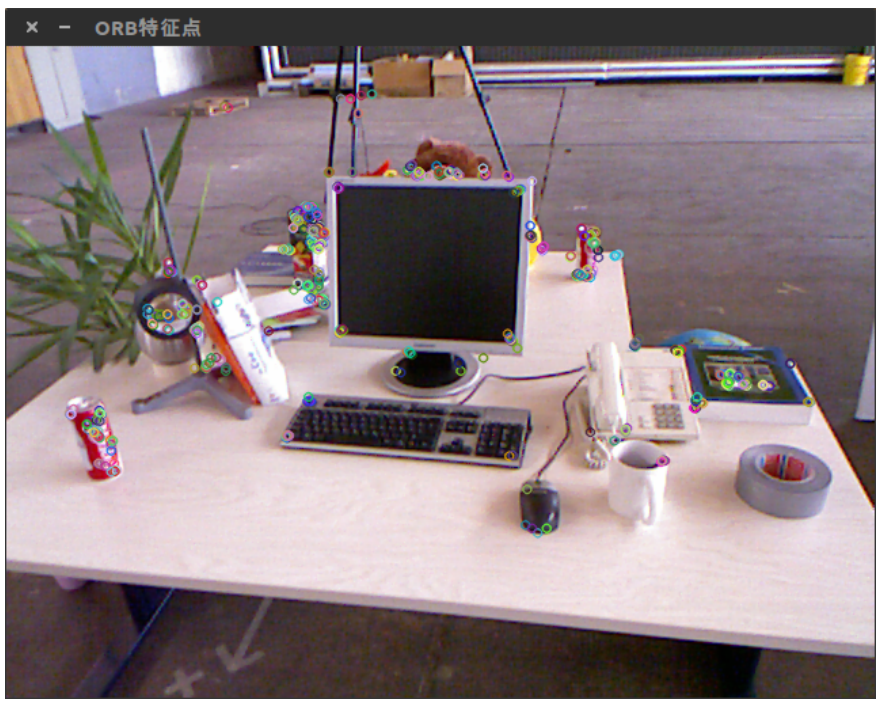


Figure 7-4: ORB feature point detection by OpenCV.

7.1.3 Feature Matching

Feature matching (as shown in Figure 7-5) is a key step in visual SLAM. Broadly speaking, feature matching solves the data association problem in SLAM, that is, to determine the current view correspondence between the landmarks (feature points) and the landmarks (feature points) seen before. By accurately matching the descriptors between images or between image and map, we can reduce a lot of work for subsequent pose estimation and optimization operations. However, due to the locality of image features, mismatches are common and have not been effectively resolved for a long time. It has become a major bottleneck to improve the performance of visual SLAM. It is somehow due to repeated textures in the scene, making the feature descriptions very similar. Under this circumstance, it is hard to resolve the mismatch by using local features only.

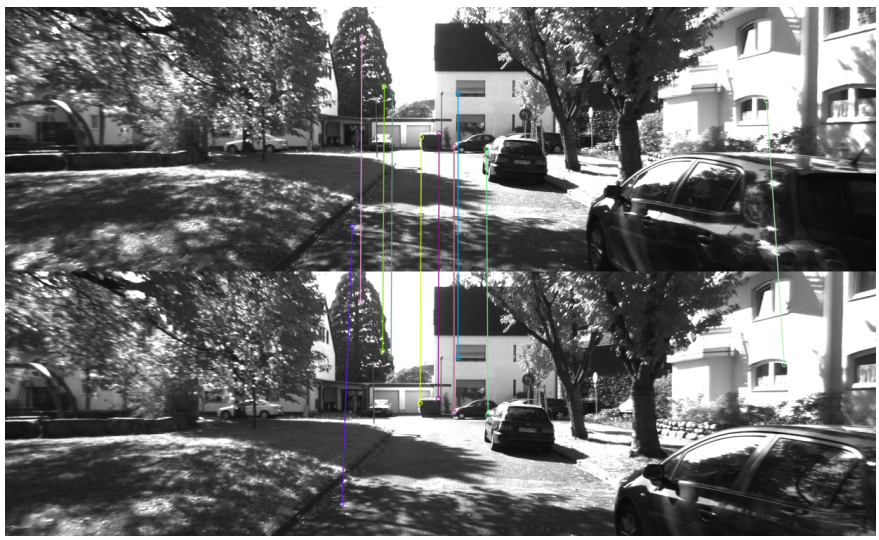


Figure 7-5: Feature matching between two images.

However, let's look at the correct matching first, and then go back to discuss the mismatch problem. Consider images at two timestamps, if the feature points $x_t^m, m = 1, 2, ..., M$ are extracted in the image I_t, then the feature points are also extracted in the image I_{t+1} as $x_{t+1}^n, n = 1, 2, ..., N$. How to find the correspondence between these two set elements? The simplest feature matching method is **Brute-Force Matcher**. That is, measure the distance between each feature point x_t^m and all x_{t+1}^n descriptors, and then sort, and take the closest one as the matching point. The descriptor distance indicates the **degree of similarity** between two features. In practice, different distance metric norms can be used. For descriptors of floating-point type, use Euclidean distance to measure. For binary descriptors (such as BRIEF), we often use Hamming distance as a metric - the Hamming distance between two binary strings refers to the number of **different digits**.

However, when the number of feature points is very large, the computational complexity of the brute force matching method will become large, especially when you want to match a certain frame and a map. This does not meet our real-time requirements in SLAM. To solve this, the **Fast Approximate Nearest Neighbor (FLANN)** algorithm is more suitable. Since the theory of these matching algorithms is mature and the implementation has been already integrated into OpenCV,

the technical details will not be described here. Interested readers can refer to the literature [44].

7.2 Practice: Feature extraction and matching

Figure 7-6: Two frames of images used in the practice.

OpenCV has integrated most mainstream image features, and we can easily use them by function calls. Let's complete two exercise: In the first one, we demonstrate the use of OpenCV for feature matching of ORB; in the second, we demonstrate how to write a simple ORB feature from scratch based on the principles introduced. Through the practice, readers will have a deeper and more clear understanding of the ORB calculation process. Then other features can be done analogously.

7.2.1 ORB features in OpenCV

First we call OpenCV to extract and match ORB. I prepared two images for this practice, 1.png and 2.png under slambook2/ch7/, as shown in Figure 7-6 . They are two images from the public data set [21]. We can see a slight movement of the camera. The procedure in this section demonstrates how to extract ORB features and perform matching. In the next section, we will demonstrate how to use matching results to estimate camera motion.

The following program demonstrates how to use ORB:

Listing 7.1: slambook2/ch7/orb_cv.cpp

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <chrono>

using namespace std;
using namespace cv;

int main(int argc, char **argv) {
    if (argc != 3) {
        cout << "usage: feature_extraction img1 img2" << endl;
        return 1;
    }
    //-- load the image
    Mat img_1 = imread(argv[1], CV_LOAD_IMAGE_COLOR);
    Mat img_2 = imread(argv[2], CV_LOAD_IMAGE_COLOR);
    assert(img_1.data != nullptr && img_2.data != nullptr);

    //-- initialization
```

```
21      std::vector<KeyPoint> keypoints_1, keypoints_2;
22      Mat descriptors_1, descriptors_2;
23      Ptr<FeatureDetector> detector = ORB::create();
24      Ptr<DescriptorExtractor> descriptor = ORB::create();
25      Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce-Hamming");
26
27      //-- Step 1: calculate Oriented FAST keypoints
28      chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
29      detector->detect(img_1, keypoints_1);
30      detector->detect(img_2, keypoints_2);
31
32      //-- Step 2: calculate BRIEF descriptors based on the position of Oriented FAST
            keypoints
33      descriptor->compute(img_1, keypoints_1, descriptors_1);
34      descriptor->compute(img_2, keypoints_2, descriptors_2);
35      chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
36      chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double
            >>(t2 - t1);
37      cout << "extract ORB cost = " << time_used.count() << " seconds. " << endl;
38
39      Mat outimg1;
40      drawKeypoints(img_1, keypoints_1, outimg1, Scalar::all(-1), DrawMatchesFlags::
            DEFAULT);
41      imshow("ORB features", outimg1);
42
43      //-- Step 3: match BRIEF descriptors of the two images using Hamming distance
44      vector<DMatch> matches;
45      t1 = chrono::steady_clock::now();
46      matcher->match(descriptors_1, descriptors_2, matches);
47      t2 = chrono::steady_clock::now();
48      time_used = chrono::duration_cast<chrono::duration<double>>(t2 - t1);
49      cout << "match ORB cost = " << time_used.count() << " seconds. " << endl;
50
51      //-- Step 4: select correct matching (filtering)
52      // calculate the max & min distances
53      auto min_max = minmax_element(matches.begin(), matches.end(),
54          [](const DMatch &m1, const DMatch &m2) { return m1.distance < m2.distance; });
55      double min_dist = min_max.first->distance;
56      double max_dist = min_max.second->distance;
57
58      printf("-- Max dist : %f \n", max_dist);
59      printf("-- Min dist : %f \n", min_dist);
60
61    // When the distance between the descriptors is greater than 2 times the min
          distance, we treat the matching as wrong.
62    // But sometimes the min distance could be very small, set an experience value of 30
          as the lower bound.
63      std::vector<DMatch> good_matches;
64      for (int i = 0; i < descriptors_1.rows; i++) {
65          if (matches[i].distance <= max(2 * min_dist, 30.0)) {
66              good_matches.push_back(matches[i]);
67          }
68      }
69
70      //-- Step 5: visualize the matching result
71      Mat img_match;
72      Mat img_goodmatch;
73      drawMatches(img_1, keypoints_1, img_2, keypoints_2, matches, img_match);
74      drawMatches(img_1, keypoints_1, img_2, keypoints_2, good_matches, img_goodmatch);
75      imshow("all matches", img_match);
76      imshow("good matches", img_goodmatch);
77      waitKey(0);
78
79      return 0;
80  }
```

Run this program (you need to enter the paths of two image manually), the screen output should be like:

Listing 7.2:

```
1  % build/orb_cv 1.png 2.png
2  extract ORB cost = 0.0229183 seconds.
3  match ORB cost = 0.000751868 seconds.
```

```
-- Max dist : 95.000000
-- Min dist : 4.000000
```

Figure 7-7 shows the results of the example. We see a large number of false matches before the filtering. After one pass of screening, though the number of matches was reduced a lot, most of the remaining matches were correct. Here, we followed an empirical rule in engineering **Hamming distance is less than twice the minimum distance** to carry out the filtering, and it may not have a theoretical explanation. However, although we can select out the correct matches in the example image, we still cannot guarantee that the matches obtained in all other images are correct. Therefore, during the motion estimation step, it is necessary to remove mismatches. On my machine, ORB extraction took 22.9 milliseconds (two images), and matching took 0.75 milliseconds. It can be seen that most of the calculation is spent on feature extraction.

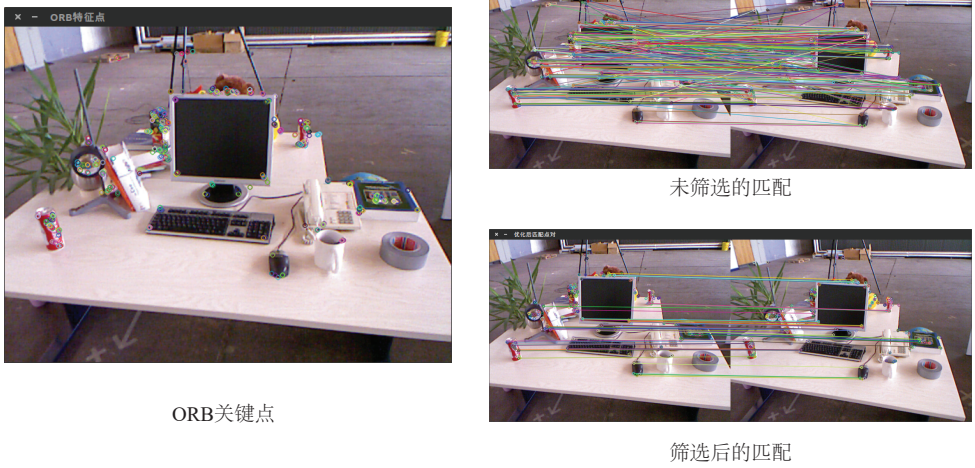


Figure 7-7: Feature extraction and matching results.

7.2.2 Coding ORB features

Below we show the method of coding ORB features from scratch. There are a branch of codes in this section. Only a snippet of the code is shown here. Readers are recommended to obtain the rest from the GitHub code base.

Listing 7.3: slambook2/ch7/orb_self.cpp

```
typedef vector<uint32_t> DescType;
// ... omit some image loading and testing code
// compute the descriptor
void ComputeORB(const cv::Mat &img, vector<cv::KeyPoint> &keypoints, vector<DescType>
     &descriptors) {
    const int half_patch_size = 8;
    const int half_boundary = 16;
    int bad_points = 0;
    for (auto &kp: keypoints) {
        if (kp.pt.x < half_boundary || kp.pt.y < half_boundary ||
        kp.pt.x >= img.cols - half_boundary || kp.pt.y >= img.rows - half_boundary) {
            // outside
            bad_points++;
            descriptors.push_back({});
            continue;
        }
```

```
16
17          float m01 = 0, m10 = 0;
18          for (int dx = -half_patch_size; dx < half_patch_size; ++dx) {
19              for (int dy = -half_patch_size; dy < half_patch_size; ++dy) {
20                  uchar pixel = img.at<uchar>(kp.pt.y + dy, kp.pt.x + dx);
21                  m01 += dx * pixel;
22                  m10 += dy * pixel;
23              }
24          }
25
26          // angle should be arc tan(m01/m10);
27          float m_sqrt = sqrt(m01 * m01 + m10 * m10);
28          float sin_theta = m01 / m_sqrt;
29          float cos_theta = m10 / m_sqrt;
30
31          // compute the angle of this point
32          DescType desc(8, 0);
33          for (int i = 0; i < 8; i++) {
34              uint32_t d = 0;
35              for (int k = 0; k < 32; k++) {
36                  int idx_pq = i * 8 + k;
37                  cv::Point2f p(ORB_pattern[idx_pq * 4], ORB_pattern[idx_pq * 4 + 1]);
38                  cv::Point2f q(ORB_pattern[idx_pq * 4 + 2], ORB_pattern[idx_pq * 4 +
                        3]);
39
40                  // rotate with theta
41                  cv::Point2f pp = cv::Point2f(cos_theta * p.x - sin_theta * p.y,
                        sin_theta * p.x + cos_theta * p.y) + kp.pt;
42                  cv::Point2f qq = cv::Point2f(cos_theta * q.x - sin_theta * q.y,
                        sin_theta * q.x + cos_theta * q.y) + kp.pt;
43                  if (img.at<uchar>(pp.y, pp.x) < img.at<uchar>(qq.y, qq.x)) {
44                      d |= 1 << k;
45                  }
46              }
47              desc[i] = d;
48          }
49          descriptors.push_back(desc);
50      }
51
52      cout << "bad/total: " << bad_points << "/" << keypoints.size() << endl;
53  }
54
55  // brute-force matching
56  void BfMatch(
57      const vector<DescType> &desc1, const vector<DescType> &desc2, vector<cv::DMatch> &
            matches) {
58      const int d_max = 40;
59
60      for (size_t i1 = 0; i1 < desc1.size(); ++i1) {
61          if (desc1[i1].empty()) continue;
62          cv::DMatch m{i1, 0, 256};
63          for (size_t i2 = 0; i2 < desc2.size(); ++i2) {
64              if (desc2[i2].empty()) continue;
65              int distance = 0;
66              for (int k = 0; k < 8; k++) {
67                  distance += _mm_popcnt_u32(desc1[i1][k] ^desc2[i2][k]);
68              }
69              if (distance < d_max && distance < m.distance) {
70                  m.distance = distance;
71                  m.trainIdx = i2;
72              }
73          }
74          if (m.distance < d_max) {
75              matches.push_back(m);
76          }
77      }
78  }
```

We only show the ORB calculation and matching code. In the calculation, we use 256-bit binary description, which corresponds to 8 32-bit unsigned int data, which is expressed as DescType with typedef. Then, we calculate the angle of the FAST feature point according to the principle introduced above, and then use the angle to

calculate the descriptor. To accelerated, some complicated calculations, such as of arctan, sin, and cos, are worked around by the principle of trigonometric functions. In the BfMatch function, we also use the _mm_popcnt_u32 function in the SSE instruction set to calculate the number of 1s in an unsigned int, which is used to achieve the effect of calculating the Hamming distance. The result of this program is as follows, and the matching result is shown in Figure 7-8:

Listing 7.4:

```
bad/total: 43/638
bad/total: 8/595
extract ORB cost = 0.00390721 seconds.
match ORB cost = 0.000862984 seconds.
matches: 51
```

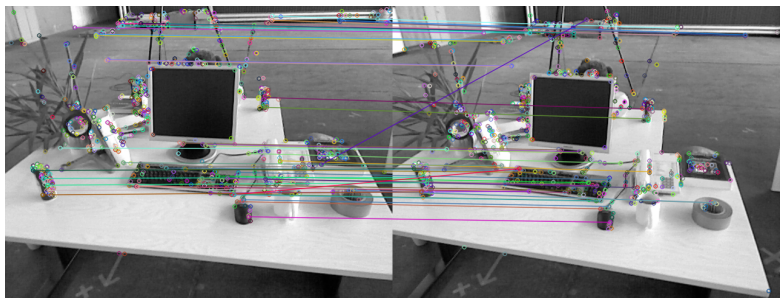


Figure 7-8: Matching Result

In this program, the extraction of ORB takes 3.9 milliseconds, and the matching takes only 0.86 milliseconds. Through some simple modifications in the algorithm implementation, we accelerated the extraction of ORB by 5.8 times. Keep in mind that compiling this program requires your CPU to support the SSE instruction set, which should already be supported on most modern home CPUs. If we can further parallelize the feature extraction, the algorithm can be further accelerated.

7.2.3 Calculate camera motion

Now, we have key points matching. In the next step, we need to estimate the camera's motion based on the matching. It is totally different for different camera settings (or different information available for the calculation):

1. When the camera is monocular, we only know the 2D pixel coordinates, so the problem is to estimate the motion according to **two sets of 2D points**. This problem is solved by **epipolar geometry**.

2. When the camera is binocular, RGB-D, or the distance is obtained by some method, then the problem is to estimate the motion according to **two sets of 3D points**. This problem is usually solved by ICP.

3. If one set is 3D and one set is 2D, that is, we get some 3D points and their projection positions on the camera, and we can also estimate the movement of the camera. This problem is solved by **PnP**.

The following sections will introduce camera motion estimation in these three situations. We will start from the 2D-2D case with the least information and see how it is dealt with and what are the troublesome problems.

7.3 2D—2D: epipolar geometry

7.3.1 epipolar constraints

Suppose we have a pair of matched feature points from two images, as shown in Figure 7-9 . If there are several pairs of such matching points, the camera motion between the two frames can be recovered through the correspondence between these two-dimensional image points. How many pairs do we need? We will introduce it later. Let's first take a look at the geometric relationship between the matching points in the two images.

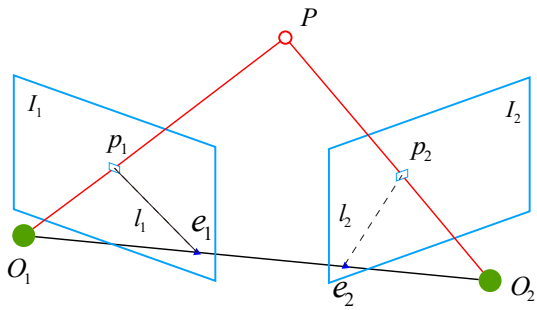


Figure 7-9: epipolar constraints.

Taking Figure 7-9 as an example, our goal is to find the motion between two frames of image I_1, I_2. Define the motion from the first frame to the second frame as $\mathbf{R}, \mathbf{t}$, and the centers of the two cameras as O_1, O_2 (Origin). Now, consider that there is a feature point p_1 in I_1, which corresponds to the feature point p_2 in I_2, which are obtained through feature matching. If the matching is correct, it means that they are indeed **projection of the same point in space on two image planes**. Now we need some terms to describe the geometric relationship between them. First, the connection $\overrightarrow{O_1p_1}$ and the connection $\overrightarrow{O_2p_2}$ will intersect at the point P in the three-dimensional space. The three points $O_1, O_2, andP$ can determine a plane, and it is called **Epipolar plane**. The intersection of the line of O_1O_2 and the image plane I_1, I_2 is e_1, e_2 respectively. e_1, e_2 is called **Epipoles**, and O_1O_2 is called **Baseline**. We call the intersecting line l_1, l_2 between the polar plane and the two image planes I_1, I_2 as **Epipolar line**.

From the first frame, the ray $\overrightarrow{O_1p_1}$ represents **spatial locations where a pixel may appear**, since all points on the ray will be projected to the same pixel. Meanwhile, if we don't know the location of P, when we look at the second image, the connection $\overrightarrow{e_2p_2}$ (i.e. the epipolar line in the second image) is possible projected positions of the point P, as well as the projection of the ray $\overrightarrow{O_1p_1}$ in the second image. Now, since we have determined the pixel location of p_2 through feature point matching, we can infer the spatial location of P and the movement of the camera, as long as the feature matching is correct. If there is no feature matching, we can't determine where the p_2 is on the epipolar line. At that time, you must search on the epipolar line l_2 to get the correct match, which will be discussed in Lecture 12.

Now, let's look at the geometric relationship algebraically. Define the spatial position of P in the first frame to be

$$\mathbf{P} = [X, Y, Z]^{\mathrm{T}}.$$

According to the pinhole camera model introduced in Lecture 5, we know that the pixel positions of the two pixels $\mathbf{p}_1, \mathbf{p}_2$ are

$$s_1\mathbf{p}_1 = \mathbf{K}\mathbf{P}, \quad s_2\mathbf{p}_2 = \mathbf{K}\left(\mathbf{R}\mathbf{P} + \mathbf{t}\right). \tag{7.1}$$

where $\mathbf{K}$ is the camera intrinsic parameters matrix, and $\mathbf{R}, \mathbf{t}$ are the camera motions between two camera frames. Specifically, they are $\mathbf{R}_{21}$ and $\mathbf{t}_{21}$, i.e. transformation from the first frame to the second. We can also write them in Lie algebra form.

Sometimes, we use homogeneous coordinates to represent pixels. When using homogeneous coordinates, a vector will be equal to itself multiplied by any non-zero constant. This is usually used to express a projection relationship. For example, $s_1\mathbf{p}_1$ and $\mathbf{p}_1$ form a projection relationship, and they are equal in the sense of homogeneous coordinates. We call this **equal up to a scale** (equal up to a scale), denoted as:

$$s\mathbf{p} \simeq \mathbf{p}. \tag{7.2}$$

Then, the relationship between two projections can be written as:

$$\mathbf{p}_1 \simeq \mathbf{K}\mathbf{P}, \quad \mathbf{p}_2 \simeq \mathbf{K}\left(\mathbf{R}\mathbf{P} + \mathbf{t}\right). \tag{7.3}$$

Now, let

$$\mathbf{x}_1 = \mathbf{K}^{-1}\mathbf{p}_1, \quad \mathbf{x}_2 = \mathbf{K}^{-1}\mathbf{p}_2. \tag{7.4}$$

Here, $\mathbf{x}_1, \mathbf{x}_2$ are the coordinates on the normalized plane of two pixels. Substituting to the above equation, we get:

$$\mathbf{x}_2 \simeq \mathbf{R}\mathbf{x}_1 + \mathbf{t}. \tag{7.5}$$

Left multiply both sides by $\mathbf{t}^\wedge$. Recalling the definition of $^\wedge$, this is equivalent to the outer product of both sides with $\mathbf{t}$:

$$\mathbf{t}^\wedge\mathbf{x}_2 \simeq \mathbf{t}^\wedge\mathbf{R}\mathbf{x}_1. \tag{7.6}$$

Then, left multiply $\mathbf{x}_2^{\mathrm{T}}$ on both sides:

$$\mathbf{x}_2^{\mathrm{T}}\mathbf{t}^\wedge\mathbf{x}_2 \simeq \mathbf{x}_2^{\mathrm{T}}\mathbf{t}^\wedge\mathbf{R}\mathbf{x}_1. \tag{7.7}$$

On the left side, $\mathbf{t}^\wedge\mathbf{x}_2$ is a vector perpendicular to both $\mathbf{t}$ and $\mathbf{x}_2$. When its inner product with $\mathbf{x}_2$ will get 0. Since the left side of the equation is strictly zero, it is also zero after multiplying by any non-zero constant, so we can write $\simeq$ as the usual equal sign. Therefore, we have a concise equation:

$$\mathbf{x}_2^{\mathrm{T}}\mathbf{t}^\wedge\mathbf{R}\mathbf{x}_1 = 0. \tag{7.8}$$

Substituting $\mathbf{p}_1, \mathbf{p}_2$ again, we have:

$$\mathbf{p}_2^{\mathrm{T}}\mathbf{K}^{-\mathrm{T}}\mathbf{t}^\wedge\mathbf{R}\mathbf{K}^{-1}\mathbf{p}_1 = 0. \tag{7.9}$$

Both equations are called **antipolar constraint**, which is famous for its conciseness. Geometically, it means O_1, P, O_2 are coplanar. The epipolar constraint encodes both translation and rotation. We denote two matrices: Fundamental Matrix $\mathbf{F}$ and Essential Matrix $\mathbf{E}$, so the epipolar constraint can be further simplified:

$$\mathbf{E} = \mathbf{t}^\wedge\mathbf{R}, \quad \mathbf{F} = \mathbf{K}^{-\mathrm{T}}\mathbf{E}\mathbf{K}^{-1}, \quad \mathbf{x}_2^{\mathrm{T}}\mathbf{E}\mathbf{x}_1 = \mathbf{p}_2^{\mathrm{T}}\mathbf{F}\mathbf{p}_1 = 0. \tag{7.10}$$

The epipolar constraint gives the spatial relationship of two matching points concisely. Therefore, the camera pose estimation problem can be summarized as the following two steps:

1. Find $\mathbf{E}$ or $\mathbf{F}$ based on the pixel positions of the matched points.
2. Find $\mathbf{R}, \mathbf{t}$ based on $\mathbf{E}$ or $\mathbf{F}$.

Since $\mathbf{E}$ and $\mathbf{F}$ only differ from the camera internal parameters, and the internal parameters are usually known in SLAM problem *, so the simpler form $\mathbf{E}$ is often used in practice. Let's take $\mathbf{E}$ as an example to introduce how to solve the above two problems.

7.3.2 Essential Matrix

By definition, the essential matrix $\mathbf{E} = \mathbf{t}^\wedge\mathbf{R}$. It is a matrix of 3×3 with 9 unknown variables. So, can any matrix of the size 3×3 be an essential matrix? From the structure of $\mathbf{E}$, there are the following points worth noting:

- The essential matrix is defined by the epipolar constraint. Since the epipolar constraint is the constraint of an **equal-to-zero equation**, after multiplying $\mathbf{E}$ by any non-zero constant, **polar constraint is still satisfied**. We call this $\mathbf{E}$'s equivalence under different scales.
- According to $\mathbf{E} = \mathbf{t}^\wedge\mathbf{R}$, it can be proved that [2], the singular value of the essential matrix $\mathbf{E}$ must be in the form of $[\sigma, \sigma, 0]^\mathrm{T}$. This is called **intrinsic properties of essential matrix**.
- On the other hand, since translation and rotation each have 3 degrees of freedom, $\mathbf{t}^\wedge\mathbf{R}$ has 6 degrees of freedom. But due to the equivalence of scales, $\mathbf{E}$ actually has 5 degrees of freedom.

The fact that $\mathbf{E}$ has 5 degrees of freedom indicates that we can use at least 5 pairs of points to solve $\mathbf{E}$. However, the intrinsic property of $\mathbf{E}$ is non-linear, which could cause trouble in the estimation. Therefore, it is also possible to consider only its **scale equivalence** and use 8 pairs of matched points to estimate $\mathbf{E}$——This is the classic **Eight-point-algorithm**[45, 46]. The eight-point method only uses the linear properties of $\mathbf{E}$, so it can be solved under the framework of linear algebra. Let's take a look at how the eight-point method works.

Consider a pair of matched points, their normalized coordinates are $\mathbf{x}_1 = [u_1, v_1, 1]^\mathrm{T}$, $\mathbf{x}_2 = [u_2, v_2, 1]^\mathrm{T}$. According to the polar constraints, we have:

$$(u_2, v_2, 1) \begin{pmatrix} e_1 & e_2 & e_3 \\ e_4 & e_5 & e_6 \\ e_7 & e_8 & e_9 \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix} = 0. \tag{7.11}$$

We rewrite the matrix $\mathbf{E}$ in the vector form:

$$\mathbf{e} = [e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9]^\mathrm{T},$$

Then the epipolar constraint can be written in a linear form related to $\mathbf{e}$:

$$[u_2u_1, u_2v_1, u_2, v_2u_1, v_2v_1, v_2, u_1, v_1, 1] \cdot \mathbf{e} = 0. \tag{7.12}$$

* in SfM research, it may be unknown and need to be estimated.

Analogously, we stack all the points into one equation and obtain a linear equation system (where u^i, v^i represent the ith feature point):

$$\begin{pmatrix} u_2^1 u_1^1 & u_2^1 v_1^1 & u_2^1 & v_2^1 u_1^1 & v_2^1 v_1^1 & v_2^1 & u_1^1 & v_1^1 & 1 \\ u_2^2 u_1^2 & u_2^2 v_1^2 & u_2^2 & v_2^2 u_1^2 & v_2^2 v_1^2 & v_2^2 & u_1^2 & v_1^2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ u_2^8 u_1^8 & u_2^8 v_1^8 & u_2^8 & v_2^8 u_1^8 & v_2^8 v_1^8 & v_2^8 & u_1^8 & v_1^8 & 1 \end{pmatrix} \begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \\ e_9 \end{pmatrix} = 0. \tag{7.13}$$

These 8 equations form a linear equation system. Its coefficient matrix is composed of 2D feature point positions, and its size is 8×9. $\mathbf{e}$ is located in the null space of this matrix. If the coefficient matrix is of full rank (i.e. 8), then its null space dimension is 1, meaning that $\mathbf{e}$ forms a line. This is consistent with the scale equivalence of $\mathbf{e}$. If the matrix composed of 8 pairs of matching points meets the condition of rank 8, then the elements of $\mathbf{E}$ can be solved uniquely by the above equation.

The next question is how to recover the movement of the camera $\mathbf{R}, \mathbf{t}$ according to the estimated essential matrix $\mathbf{E}$. This process is obtained by singular value decomposition (SVD). Let the SVD decomposition of $\mathbf{E}$ be

$$\mathbf{E} = \mathbf{U}\ \mathbf{V}^{\mathrm{T}}, \tag{7.14}$$

where $\mathbf{U}, \mathbf{V}$ are orthogonal matrices, and are singular value matrices. According to the intrinsic properties of $\mathbf{E}$, we know that $= \mathrm{diag}(\sigma, \sigma, 0)$. In SVD decomposition, for any $\mathbf{E}$, there are two possible $\mathbf{t}, \mathbf{R}$:

$$\begin{aligned} \mathbf{t}_1^\wedge &= \mathbf{U}\mathbf{R}_Z\left(\tfrac{\pi}{2}\right)\mathbf{U}^{\mathrm{T}}, \quad \mathbf{R}_1 = \mathbf{U}\mathbf{R}_Z^{\mathrm{T}}\left(\tfrac{\pi}{2}\right)\mathbf{V}^{\mathrm{T}} \\ \mathbf{t}_2^\wedge &= \mathbf{U}\mathbf{R}_Z\left(-\tfrac{\pi}{2}\right)\mathbf{U}^{\mathrm{T}}, \quad \mathbf{R}_2 = \mathbf{U}\mathbf{R}_Z^{\mathrm{T}}\left(-\tfrac{\pi}{2}\right)\mathbf{V}^{\mathrm{T}}. \end{aligned} \tag{7.15}$$

Among them, $\mathbf{R}_Z\left(\frac{\pi}{2}\right)$ represents the rotation matrix obtained by rotating 90° along the Z axis. Since $-\mathbf{E}$ is equivalent to $\mathbf{E}$, taking the minus sign for any $\mathbf{t}$ will also get the same result. Therefore, when decomposing from $\mathbf{E}$ to $\mathbf{t}, \mathbf{R}$, there are a total of **4** possible solutions.

Figure 7-10 shows the 4 solutions obtained by decomposing the essential matrix. We know the projection (red point) of the space point on the camera (blue line), and want to solve the camera's motion. In the case of keeping the red point unchanged, 4 possible situations can be drawn. Fortunately, only in the first solution, P has positive depths in both cameras. Therefore, we can substitute any points into the four solutions and check the depth of the point under two cameras, to determine which solution is correct.

If you use the intrinsic properties of $\mathbf{E}$, then it has only 5 degrees of freedom. So at least 5 pairs of matched points can be used to solve the camera motion [47, 48]. However, this approach is more complicated. For engineering realization, since there are usually dozens or even hundreds of matching points, it is often not helpful to reduce from 8 pairs to 5 pairs. To keep it simple, we only introduce the basic eight-point method here.

There is one remaining problem: $\mathbf{E}$ solved according to linear equations may not satisfy the intrinsic properties of $\mathbf{E}$, i.e. its singular value is not necessarily in the

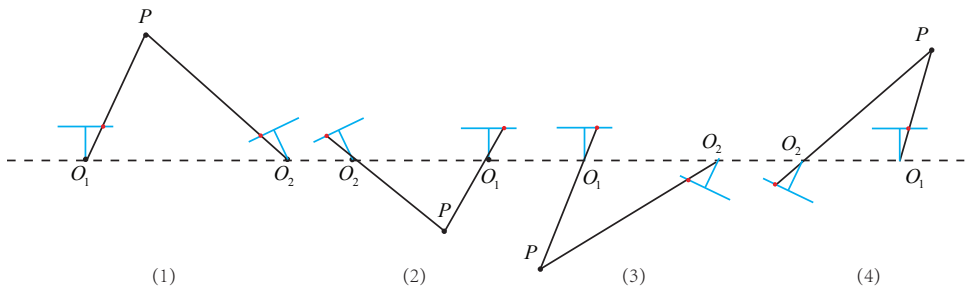


Figure 7-10: 4 4

form of $\sigma, \sigma, 0$. At this time, we will deliberately adjust the matrix to look like the above. The usual procedure is to perform SVD decomposition on the $\mathbf{E}$ obtained by the eight-point method, and the singular value matrix $= \text{diag}(\sigma_1, \sigma_2, \ sigma_3)$, might as well set $\sigma_1 \geqslant \sigma_2 \geqslant \sigma_3$. take:

$$\mathbf{E} = \mathbf{U}\text{diag}(\frac{\sigma_1 + \sigma_2}{2}, \frac{\sigma_1 + \sigma_2}{2}, 0)\mathbf{V}^{\mathrm{T}}. \tag{7.16}$$

This is equivalent to projecting the calculated essential matrix onto the manifold where $\mathbf{E}$ is located. A simpler approach is to take the singular value matrix as $\text{diag}(1, 1, 0)$, due to $\mathbf{E}$'s scale equivalence, so it is also reasonable.

7.3.3 Homography

In addition to the fundamental matrix and the essential matrix, there is another common matrix in two-view geometry: the homography matrix $\mathbf{H}$, which describes the mapping relationship between two planes. If the feature points in the scene all fall on the same plane (such as walls, ground, etc.), motion can be estimated through homography. This situation is more common in top-view cameras carried by drones or sweepers.

The homography matrix usually describes the transformation between some points on a common plane between two images. Consider that there is a pair of matched feature points p_1 and p_2 in the images I_1 and I_2. These feature points fall on the plane P. Let this plane satisfy the equation:

$$\mathbf{n}^{\mathrm{T}}\mathbf{P} + d = 0. \tag{7.17}$$

Rearrange it:

$$-\frac{\mathbf{n}^{\mathrm{T}}\mathbf{P}}{d} = 1. \tag{7.18}$$

(7.1)

$$\begin{aligned}
\mathbf{p}_2 &\simeq \mathbf{K}(\mathbf{R}\mathbf{P} + \mathbf{t}) \\
&\simeq \mathbf{K}\left(\mathbf{R}\mathbf{P} + \mathbf{t} \cdot (-\frac{\mathbf{n}^{\mathrm{T}}\mathbf{P}}{d})\right) \\
&\simeq \mathbf{K}\left(\mathbf{R} - \frac{\mathbf{t}\mathbf{n}^{\mathrm{T}}}{d}\right)\mathbf{P} \\
&\simeq \mathbf{K}\left(\mathbf{R} - \frac{\mathbf{t}\mathbf{n}^{\mathrm{T}}}{d}\right)\mathbf{K}^{-1}\mathbf{p}_1.
\end{aligned}$$

So, we got a direct description of the transformation between $\mathbf{p}_1$ and $\mathbf{p}_2$, denoting the middle part as $\mathbf{H}$, so:

$$\mathbf{p}_2 \simeq \mathbf{H}\mathbf{p}_1. \tag{7.19}$$

Its definition is related to the parameters of rotation, translation and the plane. Similar to the fundamental matrix $\mathbf{F}$, the homography matrix $\mathbf{H}$ is also a matrix of 3×3. Solving this is similar to that of $\mathbf{F}$. Calculate $\mathbf{H}$ based on the matching points, and then decompose it to find rotation and translation. Expand the above formula, get:

$$\begin{pmatrix} u_2 \\ v_2 \\ 1 \end{pmatrix} \simeq \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix}. \tag{7.20}$$

Note that the equal sign here is still $\simeq$, not the ordinary equal sign, so the $\mathbf{H}$ matrix can also be multiplied by any non-zero constant. We can make $h_9 = 1$ in practice (when it takes a non-zero value). Then according to the third row, remove this non-zero factor. So we have:

$$\begin{aligned} u_2 &= \frac{h_1 u_1 + h_2 v_1 + h_3}{h_7 u_1 + h_8 v_1 + h_9} \\ v_2 &= \frac{h_4 u_1 + h_5 v_1 + h_6}{h_7 u_1 + h_8 v_1 + h_9}. \end{aligned}$$

Rearrange:

$$\begin{aligned} h_1 u_1 + h_2 v_1 + h_3 - h_7 u_1 u_2 - h_8 v_1 u_2 &= u_2 \\ h_4 u_1 + h_5 v_1 + h_6 - h_7 u_1 v_2 - h_8 v_1 v_2 &= v_2. \end{aligned}$$

A pair of matching point can construct two constraints (in fact, there are three constraints, but considering their linear correlation, only the first two are taken), so the homography matrix with 8 degrees of freedom can be estimated by 4 pairs of matching feature points (In the case of non-degenerate, that is, these feature points cannot have three collinear points), that is, to solve the following linear equations (when $h_9 = 0$, the right side is zero):

$$\begin{pmatrix} u_1^1 & v_1^1 & 1 & 0 & 0 & 0 & -u_1^1 u_2^1 & -v_1^1 u_2^1 \\ 0 & 0 & 0 & u_1^1 & v_1^1 & 1 & -u_1^1 v_2^1 & -v_1^1 v_2^1 \\ u_1^2 & v_1^2 & 1 & 0 & 0 & 0 & -u_1^2 u_2^2 & -v_1^2 u_2^2 \\ 0 & 0 & 0 & u_1^2 & v_1^2 & 1 & -u_1^2 v_2^2 & -v_1^2 v_2^2 \\ u_1^3 & v_1^3 & 1 & 0 & 0 & 0 & -u_1^3 u_2^3 & -v_1^3 u_2^3 \\ 0 & 0 & 0 & u_1^3 & v_1^3 & 1 & -u_1^3 v_2^3 & -v_1^3 v_2^3 \\ u_1^4 & v_1^4 & 1 & 0 & 0 & 0 & -u_1^4 u_2^4 & -v_1^4 u_2^4 \\ 0 & 0 & 0 & u_1^4 & v_1^4 & 1 & -u_1^4 v_2^4 & -v_1^4 v_2^4 \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \end{pmatrix} = \begin{pmatrix} u_2^1 \\ v_2^1 \\ u_2^2 \\ v_2^2 \\ u_2^3 \\ v_2^3 \\ u_2^4 \\ v_2^4 \end{pmatrix}. \tag{7.21}$$

This approach regards the $\mathbf{H}$ matrix as a vector, and estimates the $\mathbf{H}$ by solving the linear equation of the vector, also known as the Direct Linear Transform. Similar to the essential matrix, the homography matrix needs to be decomposed after obtaining the homography matrix to get the corresponding rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$. The decomposition method includes numerical method [49, 50] and analytical method [51]. Similar to the decomposition of the essential matrix, the decomposition of the homography matrix will also return 4 sets of rotation matrices and translation vectors, and meanwhile the normal vectors of the planes where the corresponding scene points located. If the depths of the projected map points are all positive (that is, in front of the camera), then two sets of solutions can be excluded.

In the end, only two sets of solutions are left, and more prior information is needed to make a final decision. Usually we can solve it by assuming the normal vector of the known scene plane. If the scene plane is parallel to the camera plane, then the theoretical value of the normal vector $\mathbf{n}$ is $\mathbf{1}^{\mathrm{T}}$.

Homography is of great significance in SLAM. When the feature points are coplanar or the camera experiences purely rotation, the degree of freedom of the fundamental matrix decreases, which causes the degeneration. The actual data is always noisy. If you stick to the eight-point method to solve the fundamental matrix, the redundant freedom of the fundamental matrix will be mainly determined by the noise. In order to avoid the effects of degradation, we usually estimate the fundamental matrix $\mathbf{F}$ and the homography matrix $\mathbf{H}$ at the same time, and choose the one with the smaller reprojection error to estimate the motion.

7.4 Practice: Solving camera motion with epipolar constraints

In the following, let's solve the camera motion through the essential matrix. The program in the practice part of the previous section provides feature matching, and here we use the matching feature points to calculate $\mathbf{E}, \mathbf{F}$ and $\mathbf{H}$, and then decompose $\mathbf{E}$ to get $\mathbf{R}and\mathbf{t}$. The whole program is solved using the algorithm provided by OpenCV. We encapsulate the feature extraction in the previous section into functions for later use. This section only shows the code for the pose estimation.

Listing 7.5: slambook2/ch7/pose_estimation_2d2d.cpp

```
void pose_estimation_2d2d(std::vector<KeyPoint> keypoints_1,
    std::vector<KeyPoint> keypoints_2,
    std::vector<DMatch> matches,
    Mat &R, Mat &t) {
    // Camera Intrinsics,TUM Freiburg2
    Mat K = (Mat_<double>(3, 3) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1);

    //-- Convert the matching point to the form of vector<Point2f>
    vector<Point2f> points1;
    vector<Point2f> points2;

    for (int i = 0; i < (int) matches.size(); i++) {
        points1.push_back(keypoints_1[matches[i].queryIdx].pt);
        points2.push_back(keypoints_2[matches[i].trainIdx].pt);
    }

    //-- Calculate fundamental matrix
    Mat fundamental_matrix;
    fundamental_matrix = findFundamentalMat(points1, points2, CV_FM_8POINT);
    cout << "fundamental_matrix is " << endl << fundamental_matrix << endl;

    //-- Calculate essential matrix
    Point2d principal_point(325.1, 249.7);  // camera principal point, calibrated in
         TUM dataset
    double focal_length = 521;      // camera focal length, calibrated in TUM dataset
    Mat essential_matrix;
    essential_matrix = findEssentialMat(points1, points2, focal_length,
         principal_point);
    cout << "essential_matrix is " << endl << essential_matrix << endl;

    //-- Calculate homography matrix
    //-- But the scene is not planar, and calculating the homography matrix here is of
          little significance
    Mat homography_matrix;
    homography_matrix = findHomography(points1, points2, RANSAC, 3);
    cout << "homography_matrix is " << endl << homography_matrix << endl;

```

```
    //-- Recover rotation and translation from the essential matrix.
    recoverPose(essential_matrix, points1, points2, R, t, focal_length,
         principal_point);
    cout << "R is " << endl << R << endl;
    cout << "t is " << endl << t << endl;
}
```

This function shows how to solve the camera motion from the feature point, and then, we call it in the main function to get the camera motion:

Listing 7.6: slambook2/ch7/pose_estimation_2d2d.cpp

```
int main( int argc, char** argv ){
    if (argc != 3) {
        cout << "usage: pose_estimation_2d2d img1 img2" << endl;
        return 1;
    }
    //-- Fetch images
    Mat img_1 = imread(argv[1], CV_LOAD_IMAGE_COLOR);
    Mat img_2 = imread(argv[2], CV_LOAD_IMAGE_COLOR);
    assert(img_1.data && img_2.data && "Can not load images!");

    vector<KeyPoint> keypoints_1, keypoints_2;
    vector<DMatch> matches;
    find_feature_matches(img_1, img_2, keypoints_1, keypoints_2, matches);
    cout << "In total, we get " << matches.size() << " set of feature points" << endl;

    //-- Estimate the motion between two frames
    Mat R, t;
    pose_estimation_2d2d(keypoints_1, keypoints_2, matches, R, t);

    //-- Check E=t^R*scale
    Mat t_x =
        (Mat_<double>(3, 3) << 0, -t.at<double>(2, 0), t.at<double>(1, 0),
        t.at<double>(2, 0), 0, -t.at<double>(0, 0),
        -t.at<double>(1, 0), t.at<double>(0, 0), 0);
    cout << "t^R=" << endl << t_x * R << endl;

    //-- Check epipolar constraints
    Mat K = (Mat_<double>(3, 3) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1);
    for (DMatch m: matches) {
        Point2d pt1 = pixel2cam(keypoints_1[m.queryIdx].pt, K);
        Mat y1 = (Mat_<double>(3, 1) << pt1.x, pt1.y, 1);
        Point2d pt2 = pixel2cam(keypoints_2[m.trainIdx].pt, K);
        Mat y2 = (Mat_<double>(3, 1) << pt2.x, pt2.y, 1);
        Mat d = y2.t() * t_x * R * y1;
        cout << "epipolar constraint = " << d << endl;
    }
    return 0;
}
```

We get the values of $\mathbf{E}, \mathbf{F}$ and $\mathbf{H}$ in the function, and then verify whether the epipolar constraint is satisfied, and $\mathbf{t}^\wedge\mathbf{R}$ and $\mathbf{E}$ are equivalent to non-zero numbers. Now, execute this program to see the output result:

Listing 7.7: Terminal Input

```
% build/pose_estimation_2d2d 1.png 2.png
-- Max dist : 95.000000
-- Min dist : 4.000000
In total, we get 79 set of feature points
fundamental_matrix is
[4.844484382466111e-06, 0.0001222601840188731, -0.01786737827487386;
-0.0001174326832719333, 2.122888800459598e-05, -0.01775877156212593;
0.01799658210895528, 0.008143605989020664, 1]
essential_matrix is
[-0.0203618550523477, -0.4007110038118445, -0.03324074249824097;
0.3939270778216369, -0.03506401846698079, 0.5857110303721015;
-0.006788487241438284, -0.5815434272915686, -0.01438258684486258]
homography_matrix is
[0.9497129583105288, -0.143556453147626, 31.20121878625771;
0.04154536627445031, 0.9715568969832015, 5.306887618807696;
```

```
-2.81813676978796e-05, 4.353702039810921e-05, 1]
R is
[0.9985961798781875, -0.05169917220143662, 0.01152671359827873;
0.05139607508976055, 0.9983603445075083, 0.02520051547522442;
-0.01281065954813571, -0.02457271064688495, 0.9996159607036126]
t is
[-0.8220841067933337;
-0.03269742706405412;
0.5684264241053522]

t^R=
[0.02879601157010516, 0.5666909361828478, 0.04700950886436416;
-0.5570970160413605, 0.0495880104673049, -0.8283204827837456;
0.009600370724838804, 0.8224266019846683, 0.02034004937801349]
epipolar constraint = [0.002528128704106625]
epipolar constraint = [-0.001663727901710724]
epipolar constraint = [-0.00080090884108841O2]
......
```

It can be seen from the output of the program that the accuracy of meeting the epipolar constraints is around the order of 10^{-3}. According to the previous discussion, there are 4 possibilities for $\mathbf{R}, \mathbf{t}$ obtained by decomposition. In this program, OpenCV will use triangulation to detect whether the depth of the detected point is positive to select the correct solution.

Discussion

From demo program that the resulted $\mathbf{E}$ and $\mathbf{F}$ are different by a camera internal parameter matrix. Although it is not numerically intuitive, their mathematical relationship can be verified. Motion can be decomposed from $\mathbf{E}, \mathbf{F}$ and $\mathbf{H}$, but $\mathbf{H}$ needs to assume that the feature points are on the same plane. For the data of this experiment, this assumption is not true, so we mainly use $\mathbf{E}$ to find the motion.

It is worth mentioning that since $\mathbf{E}$ itself has scale equivalence, the $\mathbf{t}, \mathbf{R}$ also have scale equivalence. And $\mathbf{R} \in \mathrm{SO}(3)$ has its own constraints, so we think that $\mathbf{t}$ has a **scale**. In other words, in the decomposition process, if $\mathbf{t}$ is multiplied by any non-zero constant, the decomposition is still valid. Therefore, we usually **normalize t** to make its length equal to 1.

Scale Ambiguity

The normalization of $\mathbf{t}$ directly leads to **Scale Ambiguity of monocular vision**. For example, the first dimension of $\mathbf{t}$ output in the program is about 0.822. We are unsure that it refers to 0.822 meters or 0.822 centimeters. Because after multiplying $\mathbf{t}$ by any constant, the epipolar constraint is still valid. In other words, in monocular SLAM, the trajectory and map are simultaneously zoomed at any multiple, and the image we get is still the same. This has already been introduced in the Chapter 2.

In monocular vision, the normalization of $\mathbf{t}$ for two images is equivalent to **fixing scale**. Although we don't know its actual length, we use $\mathbf{t}$ as the unit 1 to calculate the camera motion and the 3D position of the feature points. This is called **initialization** of monocular SLAM. After initialization, 3D-2D can be used to calculate camera motion. The unit of the trajectory and map after initialization is the scale fixed during initialization. Therefore, monocular SLAM has an inevitable **initialization** step. The two initialized images must have a certain amount of translation, and then the unit of trajectory and map will be determined by this translation.

In addition to normalizing $\mathbf{t}$, another method is to set the average depth of all the feature points during initialization to 1, or a fixed scale. Compared to set the length

of $\mathbf{t}$ to 1, normalizing the depth of the feature points can control the scale of the scene and make the calculation more numerically stable. But there is no theoretical difference.

The problem of initialization caused by pure rotation

In the decomposition of $\mathbf{E}$ to get $\mathbf{R}, \mathbf{t}$, if the camera is purely rotated, causing $\mathbf{t}$ to be zero, then $\mathbf{E}$ will also be zero, which will make it impossible for us to solve $\mathbf{R}$. However, at this time we can rely on $\mathbf{H}$ to find the rotation, but when only the rotation is performed, we cannot use triangulation to estimate the spatial position of the feature points (this will be introduced later), so we can conclude that **Monocular initialization cannot only be pure rotation, it must have a certain amount of translation**. If there is no translation, the monocular can not be initialized. In practice, if the translation is too small during initialization, it will make the pose estimation and triangulation results unstable and even failed. In contrast, if the camera is moved left and right instead of rotating in place, it is easy to initialize the monocular SLAM. Therefore, experienced SLAM researchers often choose to move the camera left and right to smoothly initialize in the case of monocular SLAM.

More than 8 pairs of matched points

When we get more than 8 pairs of matched points (for example, we found 79 pairs of matches), we can calculate a least squares solution. Recalling the linearized epipolar constraint in (7.13), we denote the coefficient matrix on the left as $\mathbf{A}$:

$$\mathbf{Ae} = \mathbf{0}. \tag{7.22}$$

For the eight-point method, the size of $\mathbf{A}$ is 8×9. If the given matching points are more than 8, the equation constitutes an overdetermined equation, that is, $\mathbf{e}$ does not necessarily exist to make the above formula true. Therefore, it can be solved by minimizing in a quadratic form:

$$\min_{\mathbf{e}} \|\mathbf{Ae}\|_2^2 = \min_{\mathbf{e}} \mathbf{e}^{\mathrm{T}} \mathbf{A}^{\mathrm{T}} \mathbf{Ae}. \tag{7.23}$$

So the $\mathbf{E}$ matrix in the sense of least squares is obtained. However, due to potential mismatches, we prefer to use **Random Sample Concensus (RANSAC)** instead of least squares to solve the above problem. RANSAC is general, applicable to many cases with incorrect data, and can handle data with incorrect matching.

7.5 Triangulation

In the previous two sections, we introduced using the epipolar constraint to estimate the camera motion and discussed its limitations. After estimating the motion, the next step is to use the camera motion to estimate the spatial positions of the feature points. In monocular SLAM, the depths of the pixels cannot be obtained by a single image. We need to estimate the depths of the map points by the method of **Triangulation**, shown in **??**.

Triangulation refers to observing the same landmark point at different locations and determining the distance of the landmark point from the observed locations. Triangulation was first proposed by Gauss and used in metrology. It can be also applied to astronomy and geography. For example, we can estimate the distance to

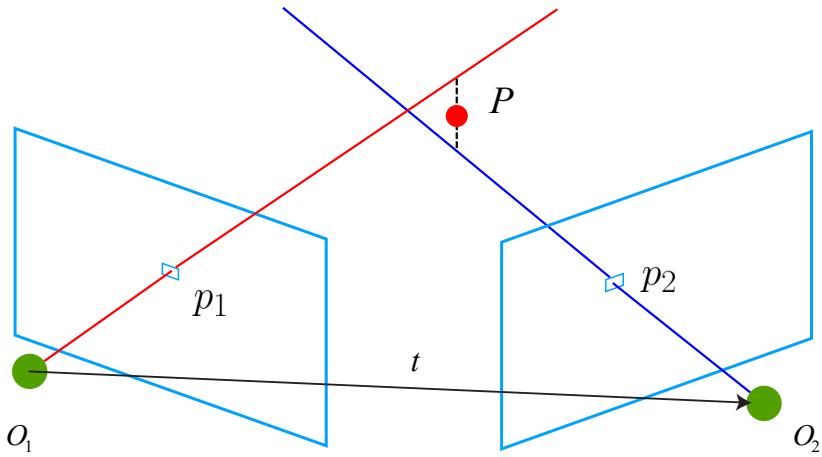


Figure 7-11:

us from the angle of the star observed in different seasons. In SLAM, we mainly use triangulation to estimate the distance of pixels.

Similar to the previous section, consider the images I_1 and I_2, with the left image as a reference, and the transformation matrix to the right image is $\mathbf{T}$. The principal points of the camera are O_1 and O_2. There is a feature point p_1 in I_1, which corresponds to a feature point p_2 in I_2. In theory, the straight line O_1p_1 and O_2p_2 will intersect at a point P in the scene, which is the map point corresponding to the two feature points in the 3D scene. However, due to the noise, these two lines often fail to exactly intersect. Therefore, it can be solved in the sense of least square.

According to the definition in epipolar geometry, let $\mathbf{x}_1, \mathbf{x}_2$ be the normalized coordinates of two feature points, then they satisfy:

$$s_2\mathbf{x}_2 = s_1\mathbf{R}\mathbf{x}_1 + \mathbf{t}. \tag{7.24}$$

Now we know $\mathbf{R}$ and $\mathbf{t}$, we want to find the depth s_1 and s_2 of two feature points. Geometically, you can find a 3D point on the ray O_1p_1 to make its projection close to $\mathbf{p}_2$. Similarly, you can also find it on O_2p_2, or in the middle of two lines . Different strategies correspond to different calculation methods, but they return similar results. For example, if we want to calculate s_1, we first multiply both sides of the above formula by $\mathbf{x}_2^\wedge$ to get:

$$s_2\mathbf{x}_2^\wedge\mathbf{x}_2 = 0 = s_1\mathbf{x}_2^\wedge\mathbf{R}\mathbf{x}_1 + \mathbf{x}_2^\wedge\mathbf{t}. \tag{7.25}$$

The left side of this equation is zero, and the right side can be regarded as an equation of s_1, and s_1 can be obtained directly from it. With s_1, s_2 is also very easy to calculate. Thus, we get the depth of the points under the two frames and determine their spatial coordinates. Definitely, due to the existence of noise, our estimated $\mathbf{R}, \mathbf{t}$ may not exactly make the equation (7.25) zero, so a more common approach is to find the least square solution rather than a direct solution.

7.6 Practice: Triangulation

7.6.1 Triangulation Program

In the following, we demonstrate how to obtain the spatial positions of the feature points in the previous section through triangulation based on the camera pose previously solved by epipolar geometry. We will use the triangulation function provided by OpenCV.

Listing 7.8: slambook2/ch7/triangulation.cpp

```
void triangulation(
  const vector<KeyPoint> &keypoint_1,
  const vector<KeyPoint> &keypoint_2,
  const std::vector<DMatch> &matches,
  const Mat &R, const Mat &t,
  vector<Point3d> &points) {
  Mat T1 = (Mat_<float>(3, 4) <<
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0);
  Mat T2 = (Mat_<float>(3, 4) <<
    R.at<double>(0, 0), R.at<double>(0, 1), R.at<double>(0, 2), t.at<double>(0, 0),
    R.at<double>(1, 0), R.at<double>(1, 1), R.at<double>(1, 2), t.at<double>(1, 0),
    R.at<double>(2, 0), R.at<double>(2, 1), R.at<double>(2, 2), t.at<double>(2, 0)
  );

  Mat K = (Mat_<double>(3, 3) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1);
  vector<Point2f> pts_1, pts_2;
  for (DMatch m:matches) {
    // Convert pixel coordinates to camera coordinates
    pts_1.push_back(pixel2cam(keypoint_1[m.queryIdx].pt, K));
    pts_2.push_back(pixel2cam(keypoint_2[m.trainIdx].pt, K));
  }

  Mat pts_4d;
  cv::triangulatePoints(T1, T2, pts_1, pts_2, pts_4d);

  // Convert to non-homogeneous coordinates
  for (int i = 0; i < pts_4d.cols; i++) {
    Mat x = pts_4d.col(i);
    x /= x.at<float>(3, 0); // □ □ □
    Point3d p(
      x.at<float>(0, 0),
      x.at<float>(1, 0),
      x.at<float>(2, 0)
    );
    points.push_back(p);
  }
}
```

Meanwhile, we can add the triangulation part to the main function, and then draw the depth of each point. Readers can run this program to view the triangulation results.

7.6.2 Discussion

Regarding triangulation, there is one more thing that must be noted.

Triangulation is caused by **translation**. Only when there is enough amount of translation, triangles in the epipolar geometry can be formed, and only then can triangulation be implemented. Therefore, triangulation cannot be used for pure rotation, because the epipolar constraint will always be satisfied. Definitely, the actual data is often not completely equal to zero. In the presence of translation, we should also concern about the uncertainty of triangulation, which will lead to a **triangulation contradiction**.

As shown in Figure 7-12 , when the translation is small, the uncertainty on the pixel will result in a larger depth uncertainty. That is to say, if the feature point moves by one pixel δx, so that the line of sight angle changes by an angle $\delta\theta$, then the measured depth will experience a change of δd. It can be seen from the geometric relationship that when $\mathbf{t}$ is larger, δd will be significantly smaller, meaning that when the translation is larger, the triangulation measurement will be more accurate at the same camera resolution. The quantitative analysis of the process can be proceeded by using the law of sine.

Therefore, to improve the accuracy of triangulation, one is to improve the accuracy of feature points extraction, that means to increase the image resolution, but this will cause the image to become larger and increase the computational cost. Another way is to increase the amount of translation. However, this will cause obvious changes in the **appearance** of the image, for example, the side of the box that was originally blocked can be exposed, or the lighting of the object changes, etc. Changes in appearance will make feature extraction and matching more difficult. In a nutshell, increasing the translation may lead to failure of matching; and if the translation is too small, the accuracy of the triangulation is insufficient. This is the contradiction of triangulation. We call this problem "parallax".

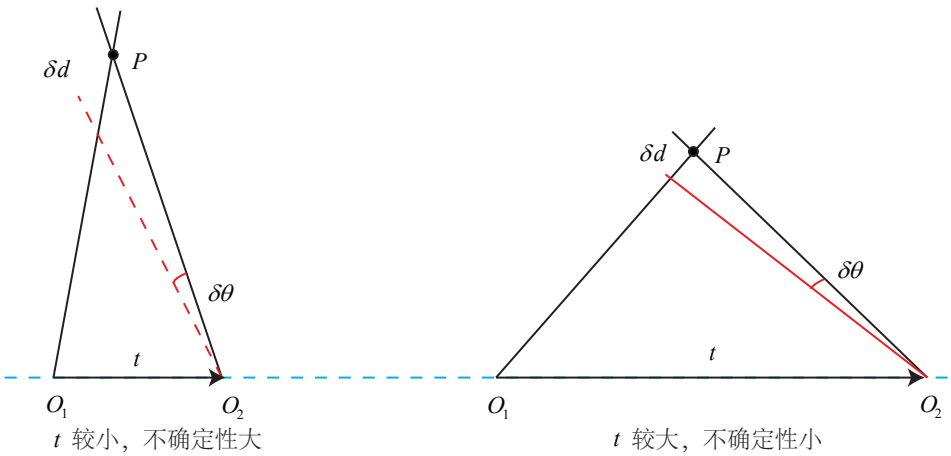


Figure 7-12:

In monocular vision, since the image has no depth information, we have to wait for the feature points to be tracked for a few frames, and then use triangulation to determine the depth of the new feature points. This is also called delayed triangulation [52]. However, if the camera rotates in place, causing the parallax to be small, it is difficult to estimate the depth of the newly observed feature points. This situation is more common in robotics, as rotation is a common command for robots. In this case, monocular vision may suffer from tracking failures and incorrect scales.

Although this section only introduces the depth estimation of triangulation, we can also quantitatively calculate the **location** and **uncertainty** of each feature point. Therefore, if we assume that the feature point obeys the Gaussian distribution and continuously observe it, with the correct information, we can expect **its variance will continue to decrease and even converge**. This can be referred to as **Depth Filter**. However, since its principle is more complicated, we will discuss it in detail later. Next, we will discuss the estimation of camera motion from 3D-2D

matching points, as well as 3D-3D estimation methods.

7.7 3D—2D PnP

PnP (Perspective-n-Point) is a method to solve 3D to 2D paired point motion. It describes how to estimate the pose of the camera when the n 3D space points and their projection positions are known. As mentioned earlier, the 2D-2D epipolar geometry method requires 8 or more point pairs (take the eight-point method as an example), and there have problems with initialization, pure rotation, and scale. However, if the 3D position of one of the feature points in the two images is known, then we need at least 3 point pairs (and at least one additional point to verify the result) to estimate the camera motion. The 3D position of the feature point can be determined by triangulation or the depth map of an RGB-D camera. Therefore, in binocular or RGB-D visual odometry, we can directly use PnP to estimate camera motion. In the monocular visual odometry, initialization must be conducted before using PnP. The 3D-2D method does not require epipolar constraints, and can obtain better motion estimation in a few matching points. It is the most important pose estimation method.

There are many ways to solve PnP problems, for example, P3P[53], direct linear transformation (DLT), EPnP (Efficient PnP)[?], UPnP[54], etc. In addition, the method of **non-linear optimization** can be used to construct a least-square problem and iteratively solve it, which is commonly called the Bundle Adjustment. Let's look at DLT first, and then we will explain Bundle Adjustment.

7.7.1 Direct Linear Transformation

Consider such a problem: the positions of a set of 3D points and their projection positions in a certain camera are known, find the pose of the camera. This problem can also be used to solve the problem of camera pose when a given map and image are given. If the 3D point is regarded as a point in another camera coordinate system, it can also be used to solve the relative motion problem of two cameras. We will start from simple questions.

Consider a 3D spatial point P, its homogeneous coordinates are $\mathbf{P} = (X, Y, Z, 1)^{\mathrm{T}}$. In the image I_1, it is projected to the feature point $\mathbf{x}_1 = (u_1, v_1, 1)^{\mathrm{T}}$(expressed in the normalized plane homogeneous coordinates). At this time, the pose of the camera $\mathbf{R}, \mathbf{t}$ is unknown. Similar to the solution of the homography matrix, we define the 3×4 augmented matrix $[\mathbf{R}|\mathbf{t}]$, encoding rotation and translation information *. We will write its expanded form as follows:

$$s \begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix} = \begin{pmatrix} t_1 & t_2 & t_3 & t_4 \\ t_5 & t_6 & t_7 & t_8 \\ t_9 & t_{10} & t_{11} & t_{12} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}. \tag{7.26}$$

Eliminate s with the last row to get two constraints:

$$u_1 = \frac{t_1 X + t_2 Y + t_3 Z + t_4}{t_9 X + t_{10} Y + t_{11} Z + t_{12}}, \quad v_1 = \frac{t_5 X + t_6 Y + t_7 Z + t_8}{t_9 X + t_{10} Y + t_{11} Z + t_{12}}.$$

* this is different from the transformation matrix $\mathbf{T}$ in SE(3).

To simplify the representation, define $\mathbf{T}$ as a row vector:

$$\mathbf{t}_1 = (t_1, t_2, t_3, t_4)^{\mathrm{T}}, \mathbf{t}_2 = (t_5, t_6, t_7, t_8)^{\mathrm{T}}, \mathbf{t}_3 = (t_9, t_{10}, t_{11}, t_{12})^{\mathrm{T}},$$

Now we have:

$$\mathbf{t}_1^{\mathrm{T}}\mathbf{P} - \mathbf{t}_3^{\mathrm{T}}\mathbf{P}u_1 = 0,$$

and

$$\mathbf{t}_2^{\mathrm{T}}\mathbf{P} - \mathbf{t}_3^{\mathrm{T}}\mathbf{P}v_1 = 0.$$

Please note that $\mathbf{t}$ is the variable to be found. As you can see, each feature point provides two linear constraints on $\mathbf{t}$. Assuming there are a total of N feature points, the following linear equations can be constructed:

$$\begin{pmatrix} \mathbf{P}_1^{\mathrm{T}} & 0 & -u_1\mathbf{P}_1^{\mathrm{T}} \\ 0 & \mathbf{P}_1^{\mathrm{T}} & -v_1\mathbf{P}_1^{\mathrm{T}} \\ \vdots & \vdots & \vdots \\ \mathbf{P}_N^{\mathrm{T}} & 0 & -u_N\mathbf{P}_N^{\mathrm{T}} \\ 0 & \mathbf{P}_N^{\mathrm{T}} & -v_N\mathbf{P}_N^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} \mathbf{t}_1 \\ \mathbf{t}_2 \\ \mathbf{t}_3 \end{pmatrix} = 0. \tag{7.27}$$

Since $\mathbf{t}$ has a total dimension of 12, the linear solution of matrix $\mathbf{T}$ can be achieved by at least 6 pairs of matching points. This method is called Direct Linear Transform (DLT). When the matching points are greater than 6 pairs, methods such as SVD can also be used to find the least square solution of the overdetermined equation.

In the DLT solution, we directly regard the $\mathbf{T}$ matrix as 12 unknowns, ignoring the correlation between them. Because the rotation matrix $\mathbf{R} \in \mathrm{SO}(3)$, the solution obtained by DLT does not necessarily satisfy the constraint, it is a general matrix. The translation vector is easier to handle, it belongs to the vector space. For the rotation matrix $\mathbf{R}$, we must look for the best rotation matrix to approximate the matrix block of 3×3 on the left of $\mathbf{T}$ estimated by DLT. This can be done by QR decomposition [2, 55], or it can be calculated like this [5, 56]:

$$\mathbf{R} \leftarrow \left(\mathbf{R}\mathbf{R}^{\mathrm{T}}\right)^{-\frac{1}{2}}\mathbf{R}. \tag{7.28}$$

This can be seen as reprojecting the result from the matrix space onto the SE(3) manifold and converting it into two parts: rotation and translation.

What needs to be mentioned is that our $\mathbf{x}_1$ here uses normalized plane coordinates and neglects the influence of the intrinsic parameter matrix $\mathbf{K}$, this is because the intrinsic parameter $\mathbf{K}$ is usually assumed to be known in SLAM. Even if the intrinsic parameters are unknown, PnP can be used to estimate the three quantities $\mathbf{K}, \mathbf{R}, \mathbf{t}$. However, due to the increase in the number of unknown variables, the quantity of the result may be worse.

7.7.2 P3P

P3P is another way to solve PnP. It only uses 3 pairs of matching points and requires less data (this part refers to the literature [57]).

P3P requires to establish geometric relationships of the given 3 points. Its input data is 3 pairs of 3D-2D matching points. Define 3D points as A, B, C, 2D points as a, b, c, where the point represented by the lowercase letter is the projection of the point on the camera image plane represented by the corresponding uppercase letter, as shown in **??** . In addition, P3P also needs to a pair of verification points

to select the correct one from the possible solutions (similar to the case of epipolar geometry). Denote the verification point pair as $D-d$ and the camera principal point as O. Suppose that A, B, C are in **the world coordinate frame**, not **camera coordinate**. Once the coordinates of the 3D point in the camera coordinate system can be calculated, we get the 3D−3D corresponding point and convert the PnP problem to the ICP problem.

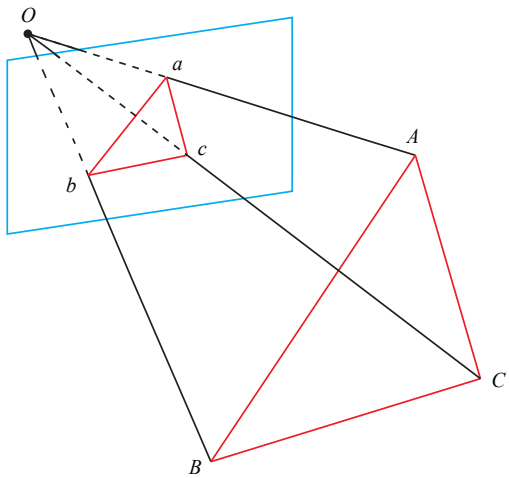


Figure 7-13: P3P

Obviously, there is a relationship between triangles:

$$\Delta Oab - \Delta OAB, \quad \Delta Obc - \Delta OBC, \quad \Delta Oac - \Delta OAC. \tag{7.29}$$

Consider the relationship between Oab and OAB. Using the law of cosines, there are:

$$OA^2 + OB^2 - 2OA \cdot OB \cdot \cos\langle a, b\rangle = AB^2. \tag{7.30}$$

The other two triangles have similar properties, so we further get:

$$\begin{aligned} OA^2 + OB^2 - 2OA \cdot OB \cdot \cos\langle a, b\rangle &= AB^2 \\ OB^2 + OC^2 - 2OB \cdot OC \cdot \cos\langle b, c\rangle &= BC^2 \\ OA^2 + OC^2 - 2OA \cdot OC \cdot \cos\langle a, c\rangle &= AC^2. \end{aligned} \tag{7.31}$$

Divide all the above three equations by OC^2 on both sides, and let $x = OA/OC, y = OB/OC$, we get:

$$\begin{aligned} x^2 + y^2 - 2xy \cos\langle a, b\rangle &= AB^2/OC^2 \\ y^2 + 1^2 - 2y \cos\langle b, c\rangle &= BC^2/OC^2 \\ x^2 + 1^2 - 2x \cos\langle a, c\rangle &= AC^2/OC^2. \end{aligned} \tag{7.32}$$

Let $v = AB^2/OC^2, uv = BC^2/OC^2, wv = AC^2/OC^2$ we then have

$$\begin{aligned} x^2 + y^2 - 2xy \cos\langle a, b\rangle - v &= 0 \\ y^2 + 1^2 - 2y \cos\langle b, c\rangle - uv &= 0 \\ x^2 + 1^2 - 2x \cos\langle a, c\rangle - wv &= 0. \end{aligned} \tag{7.33}$$

Move v in the first equation to the right side, and combine it with the other two equations, we have:

$$\begin{aligned} (1-u)\, y^2 - ux^2 - \cos\langle b, c\rangle\, y + 2uxy \cos\langle a, b\rangle + 1 &= 0 \\ (1-w)\, x^2 - wy^2 - \cos\langle a, c\rangle\, x + 2wxy \cos\langle a, b\rangle + 1 &= 0. \end{aligned} \tag{7.34}$$

Please distinguish the known from the unknown quantities in these equations. Since we know the positions of the 2D points in the image, the 3 cosine angles $\cos\langle a,b\rangle$, $\cos\langle b,c\rangle$, $\cos\langle a,c\rangle$ can be calculated. Meanwhile, $u = BC^2/AB^2, w = AC^2/AB^2$ can be calculated by the coordinates of A, B, C in the world frame, transforming to the camera frame does not change the ratio. The x and y are unknown and will change as the camera moves. Therefore, the system of equations is a quadratic equation (polynomial equation) about two unknowns x, y. Solving the equations analytically is complicated and requires Wu's elimination method. It will not be introduced here, interested readers could refer to the literature [53]. Analogous to the case of decomposing $\mathbf{E}$, this equation may get 4 solutions at most, but we can use the verification points to select the most probable solution, and get the 3D of A, B, C in the camera frame. Then, based on the 3D−3D point pair, the camera movement $\mathbf{R}, \mathbf{t}$ can be calculated, which will be introduced in section 7.9.

From the principle of P3P to solve PnP, we use the similarity of triangles to solve the 3D coordinates of the projection points a, b, c in the camera frame, and finally convert the problem into a 3D to 3D pose estimation problem. As we will see later, it is easy to solve the 3D-3D pose with matching information, so this idea is very effective. Some other methods, such as EPnP, also adopted this idea. However, P3P also has some deficiencies:

1. P3P only includes the information of 3 points. When the given matched points are more than 3, it is difficult to use more information.
2. If the 3D point or 2D point is affected by noise, or there is a mismatch, the algorithm goes into trouble.

People also proposed many other methods, such as EPnP, UPnP and so on. They use more information and optimize the camera pose in an iterative way to eliminate the effects of noise as much as possible. However, compared to P3P, the principles are more complicated, so we recommend that readers read the original papers or understand the PnP process by practice. In SLAM, the usual approach is to first estimate the camera pose using methods such as P3P/EPnP, and then construct a least squares optimization problem to adjust the estimated values (Bundle Adjustment). When the camera motion is sufficiently continuous, or you can also assume that the camera does not move or move at a constant speed, you can use the estimated values as the initial values for optimization. Next we look at the PnP problem from the perspective of nonlinear optimization.

7.7.3 Solve PnP by minimizing reprojection error

Other than the linear method, we can also construct the PnP problem as a nonlinear least-square problem about reprojection errors. This will use the knowledge from the 4 and 5 chapters of this book. The linear method mentioned above is often **a first look for the camera pose and the position of the points in space**, while nonlinear optimization treats them as optimization variables and optimizes them together. This is a very general solution method, we can use it to optimize the results given by PnP or ICP. This type of problems, **putting the camera and 3D points together to minimize**, are generally referred to as Bundle Adjustment*.

* The meaning of BA in different documents and contexts are not exactly the same. Some scholars only refer to the problem of minimizing reprojection errors as BA, while others have a broader definition of BA. Even if the BA has only one camera or other similar sensors, it

We can build a Bundle Adjustment problem in PnP to optimize the camera pose. If the camera is moving continuously (as in the majority of SLAM processes), you can also use BA directly to solve the camera pose. In this section, we will give the basic form of this problem in two views, and then discuss the larger-scale BA problem in Lecture 9.

Considering n 3D space points P and their projection p, we want to calculate the pose $\mathbf{R}, \mathbf{t}$ of the camera, and its Lie group is expressed as $\mathbf{T}$. Suppose the coordinates of a point in space are $\mathbf{P}_i = [X_i, Y_i, Z_i]^{\mathrm{T}}$, and their projected pixel coordinates are $\mathbf{u}_i = [u_i, v_i]^{\mathrm{T}}$. According to 5, the relationship between the 2D pixel position and the 3D spatial position is:

$$s_i \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \mathbf{KT} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix}. \tag{7.35}$$

In matrix form, it is:

$$s_i \mathbf{u}_i = \mathbf{KTP}_i.$$

This equation includes a conversion from homogeneous coordinates to non-homogeneous coordinates implicitly. Otherwise, according to the matrix multiplication, the dimension is wrong *. Now, due to the unknown camera pose and the noise of the observation points, there is a residual in the equation. Therefore, we sum up the residuals, construct a least-square problem, and then minimize it to find the most reliable camera pose:

$$\mathbf{T}^* = \arg\min_{\mathbf{T}} \frac{1}{2} \sum_{i=1}^{n} \left\| \mathbf{u}_i - \frac{1}{s_i} \mathbf{KTP}_i \right\|_2^2. \tag{7.36}$$

The residual term of this problem is the difference between the projection position of the 3D points and the observation positions, so it is called **reprojection error**. When using homogeneous coordinates, this error has 3 dimensions. However, since the last dimension of $\mathbf{u}$ is 1, the error of this dimension is always zero, so more often we use non-homogeneous coordinates, thus the error is only 2 dimensions. As shown in Figure 7-14 , we know that p_1 and p_2 are projections of the same space point P through feature matching, but we don't know the pose of the camera. In the initial value, there is a certain distance between the projection of $P\hat{p}_2$ and the actual p_2. So we adjusted the pose of the camera to make this distance smaller. However, since this adjustment needs to consider many points, the goal is to reduce the overall error, and the error of each point usually can not be exactly zero.

We have already discussed the least-square optimization problem in the Chapter 6. Using Lie algebra, unconstrained optimization problems can be constructed, which can be easily solved by optimization algorithms such as Gauss Newton method and Levenberg-Marquardt method. However, before using the Gauss-Newton method and Levenberg-Marquardt method, we need to find the derivative of each error term with respect to the optimization variable, which is **linearization**:

$$\mathbf{e}(\mathbf{x} + \Delta\mathbf{x}) \approx \mathbf{e}(\mathbf{x}) + \mathbf{J}^{\mathrm{T}} \Delta\mathbf{x}. \tag{7.37}$$

---

can be called BA. I personally prefer the broader definition, so the method of calculating PnP here is also called BA.

* The result of $\mathbf{TP}_i$ is 4×1, and the $\mathbf{K}$ on the left is 3×3, so the first three dimensions of $\mathbf{TP}_i$ must be taken out and changed into three dimension non-homogeneous coordinates. Or, using $\mathbf{RP} + \mathbf{t}$

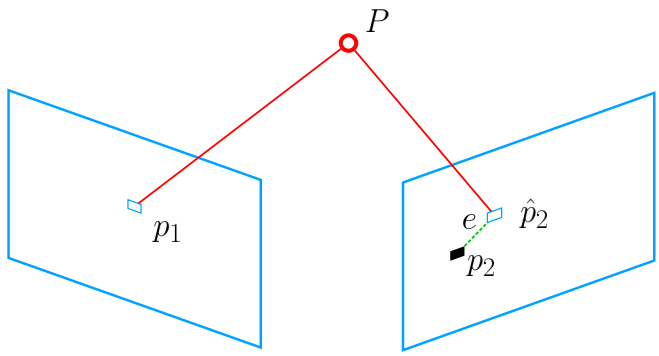


Figure 7-14:

The form of $\mathbf{J}^{\mathrm{T}}$ is worth discussing, and it can even be said to be the key. Definitely, we can use numerical derivatives, but if we can derive an analytical form, we will prefer to analytical derivatives. Now, $\mathbf{e}$ is the pixel coordinate error (2-dimensional) and $\mathbf{x}$ is the camera pose (6-dimensional), $\mathbf{J}^{\mathrm{T}}$ is a matrix of 2×6. Let's derive the form of $\mathbf{J}^{\mathrm{T}}$.

We have introduced how to use the perturbation model to find the derivative of Lie algebra. First, define the coordinates of the space point in the camera frame as $\mathbf{P}'$, and take out the first 3 dimensions:

$$\mathbf{P}' = (\mathbf{TP})_{1:3} = [X', Y', Z']^{\mathrm{T}}. \tag{7.38}$$

Then, the camera projection model with respect to $\mathbf{P}'$ is:

$$s\mathbf{u} = \mathbf{KP}'. \tag{7.39}$$

Expand:

$$\begin{bmatrix} su \\ sv \\ s \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix}. \tag{7.40}$$

Use the third row to eliminate s (actually it is the distance of $\mathbf{P}'$), we get:

$$u = f_x \frac{X'}{Z'} + c_x, \quad v = f_y \frac{Y'}{Z'} + c_y. \tag{7.41}$$

This is consistent with the camera model described in 5. When we find the error, we can compare the u, v here with the actual measured value to find the difference. After defining the intermediate variables, we left multiply $\mathbf{T}$ by a disturbance quantity $\delta\boldsymbol{\xi}$, and then consider the derivative of the change of $\mathbf{e}$ with respect to the disturbance quantity. Using the chain rule, it is:

$$\frac{\partial \mathbf{e}}{\partial \delta\boldsymbol{\xi}} = \lim_{\delta\boldsymbol{\xi} \to 0} \frac{\mathbf{e}\left(\delta\boldsymbol{\xi} \oplus \boldsymbol{\xi}\right) - \mathbf{e}(\boldsymbol{\xi})}{\delta\boldsymbol{\xi}} = \frac{\partial \mathbf{e}}{\partial \mathbf{P}'} \frac{\partial \mathbf{P}'}{\partial \delta\boldsymbol{\xi}}. \tag{7.42}$$

Here $\oplus$ refers to the disturbance left multiplication in Lie algebra. The first item is the derivative of the error with respect to the projection point. The relationship

between the variables is in the equation (7.41), and it is easy to get:

$$\frac{\partial \mathbf{e}}{\partial \mathbf{P}'} = -\begin{bmatrix} \frac{\partial u}{\partial X'} & \frac{\partial u}{\partial Y'} & \frac{\partial u}{\partial Z'} \\ \frac{\partial v}{\partial X'} & \frac{\partial v}{\partial Y'} & \frac{\partial v}{\partial Z'} \end{bmatrix} = -\begin{bmatrix} \frac{f_x}{Z'} & 0 & -\frac{f_x X'}{Z'^2} \\ 0 & \frac{f_y}{Z'} & -\frac{f_y Y'}{Z'^2} \end{bmatrix}. \tag{7.43}$$

The second term is the derivative of the transformed point with respect to the Lie algebra. According to the section 4.3.5, we get:

$$\frac{\partial (\mathbf{TP})}{\partial \delta \boldsymbol{\xi}} = (\mathbf{TP})^{\odot} = \begin{bmatrix} \mathbf{I} & -\mathbf{P}'^{\wedge} \\ \mathbf{0}^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} \end{bmatrix}. \tag{7.44}$$

In the definition of $\mathbf{P}'$, we took out the first 3 dimensions, so we get:

$$\frac{\partial \mathbf{P}'}{\partial \delta \boldsymbol{\xi}} = [\mathbf{I}, -\mathbf{P}'^{\wedge}]. \tag{7.45}$$

Multiply these two items together, we get the 2×6 Jacobian matrix:

$$\frac{\partial \mathbf{e}}{\partial \delta \boldsymbol{\xi}} = -\begin{bmatrix} \frac{f_x}{Z'} & 0 & -\frac{f_x X'}{Z'^2} & -\frac{f_x X' Y'}{Z'^2} & f_x + \frac{f_x X'^2}{Z'^2} & -\frac{f_x Y'}{Z'} \\ 0 & \frac{f_y}{Z'} & -\frac{f_y Y'}{Z'^2} & -f_y - \frac{f_y Y'^2}{Z'^2} & \frac{f_y X' Y'}{Z'^2} & \frac{f_y X'}{Z'} \end{bmatrix}. \tag{7.46}$$

This Jacobian matrix describes the first-order derivative of the reprojection error with respect to the Lie algebra of the camera pose. We keep the negative sign in front of it because the error is defined by **observed value minus predicted value**. It can also be reversed and defined in the form of "predicted value minus observed value". In that case, just remove the negative sign in front. In addition, if the definition of $\mathfrak{se}(3)$ is rotation followed by translation, just swap the first 3 columns and the last 3 columns of this Jacobian matrix.

On top of optimizing the pose, we want to optimize the spatial position of the feature points. Therefore, we also need to discuss the derivative of $\mathbf{e}$ with respect to the space point $\mathbf{P}$. Fortunately, this derivative matrix is relatively easy. Still using the chain rule, there are:

$$\frac{\partial \mathbf{e}}{\partial \mathbf{P}} = \frac{\partial \mathbf{e}}{\partial \mathbf{P}'} \frac{\partial \mathbf{P}'}{\partial \mathbf{P}}. \tag{7.47}$$

The first item has been deduced before, and the second item is defined as:

$$\mathbf{P}' = (\mathbf{TP})_{1:3} = \mathbf{RP} + \mathbf{t},$$

We find that after $\mathbf{P}'$ differentiates $\mathbf{P}$, only $\mathbf{R}$ is left. then:

$$\frac{\partial \mathbf{e}}{\partial \mathbf{P}} = -\begin{bmatrix} \frac{f_x}{Z'} & 0 & -\frac{f_x X'}{Z'^2} \\ 0 & \frac{f_y}{Z'} & -\frac{f_y Y'}{Z'^2} \end{bmatrix} \mathbf{R}. \tag{7.48}$$

Now, we derived the two Jacobian matrices of the observation camera equation with respect to the camera pose and feature points. They are **very important** to provide gradient directions in the optimization and guide the iteration of optimization.

7.8 Practice: Solving PnP

7.8.1 Use EPnP to solve pose

In the following, we will have a deeper understand of the PnP process through practice. First, we demonstrate how to use OpenCV's EPnP to solve the PnP problem, and then solve it again through nonlinear optimization. In the second edition of the book, we also add a handwriting optimization practice. Since PnP needs to use 3D points, in order to avoid the trouble of initialization, we use the depth map (1_depth.png) in the RGB-D camera as the 3D position of the feature points. First look at the PnP function provided by OpenCV:

Listing 7.9: slambook2/ch7/pose_estimation_3d2d.cpp

```
int main( int argc, char** argv ) {
    Mat r, t;
   solvePnP(pts_3d, pts_2d, K, Mat(), r, t, false); // Call OpenCV's PnP, you can
        choose from EPNP, DLS and other methods
   Mat R;
   cv::Rodrigues(r, R); // r is in the form of rotation vector, and converted to a
        rotation matrix by Rodrigues formula
   cout << "R=" << endl << R << endl;
   cout << "t=" << endl << t << endl;
}
```

In the example, after obtaining the matched feature points, we look for their depth in the depth map of the first image and find their spatial position. Taking this spatial position as a 3D point, and then taking the pixel position of the second image as a 2D point, call EPnP to solve the PnP problem. The program output is as follows:

Listing 7.10:

```
% build/pose_estimation_3d2d 1.png 2.png d1.png d2.png
-- Max dist : 95.000000
-- Min dist : 4.000000
In total, we get 79 set of feature points
3d-2d pairs: 76
R=
[0.9978662025826269, -0.05167241613316376, 0.03991244360207524;
0.0505958915956335, 0.998339762771668, 0.02752769192381471;
-0.04126860182960625, -0.025449547736074, 0.998823919929363]
t=
[-0.1272259656955879;
-0.007507297652615337;
0.06138584177157709]
```

Readers can compare $\mathbf{R}, \mathbf{t}$ solved in the previous 2D-2D case to see the difference. It can be seen that when 3D information is involved, the estimated $\mathbf{R}$ is almost the same, while the $\mathbf{t}$ is quite different. This is due to the inclusion of depth information. However, since the depth map collected by Kinect has some noise, the 3D points here are not accurate. In a larger-scale BA, we would like to optimize the pose and all three-dimensional feature points at the same time.

7.8.2 Handwriting pose estimation

The following demonstrates how to use nonlinear optimization to calculate the camera pose. We first write a PnP of Gauss-Newton method, and then demonstrate how to solve it by g2o.

Listing 7.11: slambook2/ch7/pose_estimation_3d2d.cpp

```
void bundleAdjustmentGaussNewton(
const VecVector3d &points_3d,
const VecVector2d &points_2d,
const Mat &K,
Sophus::SE3d &pose) {
  typedef Eigen::Matrix<double, 6, 1> Vector6d;
  const int iterations = 10;
  double cost = 0, lastCost = 0;
  double fx = K.at<double>(0, 0);
  double fy = K.at<double>(1, 1);
  double cx = K.at<double>(0, 2);
  double cy = K.at<double>(1, 2);

  for (int iter = 0; iter < iterations; iter++) {
    Eigen::Matrix<double, 6, 6> H = Eigen::Matrix<double, 6, 6>::Zero();
    Vector6d b = Vector6d::Zero();

    cost = 0;
    // compute cost
    for (int i = 0; i < points_3d.size(); i++) {
      Eigen::Vector3d pc = pose * points_3d[i];
      double inv_z = 1.0 / pc[2];
      double inv_z2 = inv_z * inv_z;
      Eigen::Vector2d proj(fx * pc[0] / pc[2] + cx, fy * pc[1] / pc[2] + cy);
      Eigen::Vector2d e = points_2d[i] - proj;
      cost += e.squaredNorm();
      Eigen::Matrix<double, 2, 6> J;
      J << -fx * inv_z,
      0,
      fx * pc[0] * inv_z2,
      fx * pc[0] * pc[1] * inv_z2,
      -fx - fx * pc[0] * pc[0] * inv_z2,
      fx * pc[1] * inv_z,
      0,
      -fy * inv_z,
      fy * pc[1] * inv_z,
      fy + fy * pc[1] * pc[1] * inv_z2,
      -fy * pc[0] * pc[1] * inv_z2,
      -fy * pc[0] * inv_z;

      H += J.transpose() * J;
      b += -J.transpose() * e;
    }

    Vector6d dx;
    dx = H.ldlt().solve(b);

    if (isnan(dx[0])) {
      cout << "result is nan!" << endl;
      break;
    }

    if (iter > 0 && cost >= lastCost) {
      // cost increase, update is not good
      cout << "cost: " << cost << ", last cost: " << lastCost << endl;
      break;
    }

    // update your estimation
    pose = Sophus::SE3d::exp(dx) * pose;
    lastCost = cost;

    cout << "iteration " << iter << " cost=" << cout.precision(12) << cost << endl;
    if (dx.norm() < 1e-6) {
      // converge
      break;
    }
  }

  cout << "pose by g-n: \n" << pose.matrix() << endl;
}
```

In this function, we implement a simple Gauss-Newton iteration optimization

based on the previous theoretical derivation. Then we will compare the efficiency of OpenCV, handwritten implementation and g2o implementation.

7.8.3 BA optimization by g2o

After handwriting the optimization process, let's look at how to achieve the same functionality using the library g2o (in fact, it is completely similar with Ceres). The basic knowledge of g2o has been introduced in Lecture 6. Before using g2o, we have to model the problem as a graph optimization problem, as shown in Figure 7-15 .

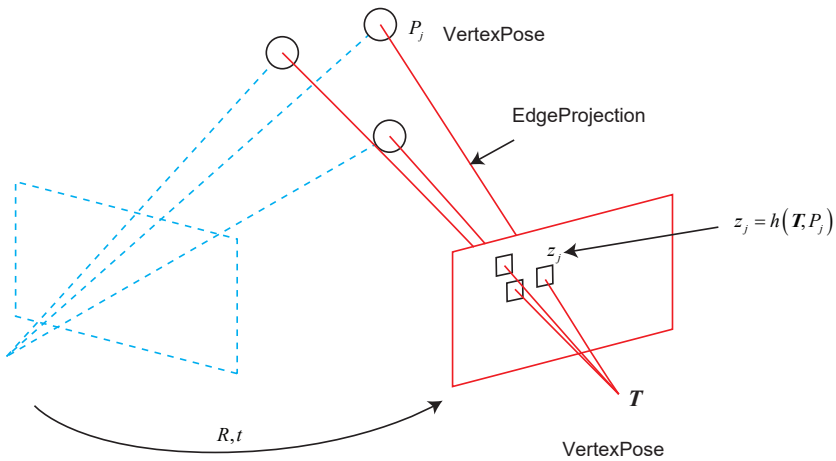


Figure 7-15: PnP Bundle Adjustment

In this graph optimization, the nodes and edges are defined as follows:

1. **Node**: The pose of the second camera $\mathbf{T} \in \mathrm{SE}(3)$.
2. **Edge**: The projection of each 3D point in the second camera, described by the observation equation:

$$\mathbf{z}_j = h(\mathbf{T}, \mathbf{P}_j).$$

Since the pose of the first camera is fixed to zero, we excluded it from the optimization variables, but in the normal occasions, we will consider estimations of a lot of camera poses. Now we estimate the pose of the second camera based on a set of 3D points and the 2D projection in the second image. We drew the first camera as a dotted line to indicate that we don't want to consider it.

g2o provides many nodes and edges about BA. For example, g2o/types/sba/types_six_dof_expmap.h provides nodes and edges expressed by Lie algebra. In the second edition of the book, we implement a VertexPose vertex and EdgeProjection edge ourselves, as follows:

Listing 7.12: slambook2/ch7/pose_estimation_3d2d.cpp

```
/// vertex and edges used in g2o ba
class VertexPose : public g2o::BaseVertex<6, Sophus::SE3d> {
  public:
  EIGEN_MAKE_ALIGNED_OPERATOR_NEW;

  virtual void setToOriginImpl() override {
```

```
    _estimate = Sophus::SE3d();
  }

  /// left multiplication on SE3
  virtual void oplusImpl(const double *update) override {
    Eigen::Matrix<double, 6, 1> update_eigen;
    update_eigen << update[0], update[1], update[2], update[3], update[4], update[5];
    _estimate = Sophus::SE3d::exp(update_eigen) * _estimate;
  }

  virtual bool read(istream &in) override {}

  virtual bool write(ostream &out) const override {}
};

class EdgeProjection : public g2o::BaseUnaryEdge<2, Eigen::Vector2d, VertexPose> {
  public:
  EIGEN_MAKE_ALIGNED_OPERATOR_NEW;

  EdgeProjection(const Eigen::Vector3d &pos, const Eigen::Matrix3d &K) : _pos3d(pos),
      _K(K) {}

  virtual void computeError() override {
    const VertexPose *v = static_cast<VertexPose *> (_vertices[0]);
    Sophus::SE3d T = v->estimate();
    Eigen::Vector3d pos_pixel = _K * (T * _pos3d);
    pos_pixel /= pos_pixel[2];
    _error = _measurement - pos_pixel.head<2>();
  }

  virtual void linearizeOplus() override {
    const VertexPose *v = static_cast<VertexPose *> (_vertices[0]);
    Sophus::SE3d T = v->estimate();
    Eigen::Vector3d pos_cam = T * _pos3d;
    double fx = _K(0, 0);
    double fy = _K(1, 1);
    double cx = _K(0, 2);
    double cy = _K(1, 2);
    double X = pos_cam[0];
    double Y = pos_cam[1];
    double Z = pos_cam[2];
    double Z2 = Z * Z;
    _jacobianOplusXi
    << -fx / Z, 0, fx * X / Z2, fx * X * Y / Z2, -fx - fx * X * X / Z2, fx * Y / Z,
    0, -fy / Z, fy * Y / (Z * Z), fy + fy * Y * Y / Z2, -fy * X * Y / Z2, -fy * X / Z;
  }

  virtual bool read(istream &in) override {}

  virtual bool write(ostream &out) const override {}

  private:
  Eigen::Vector3d _pos3d;
  Eigen::Matrix3d _K;
};
```

This implements vertex update and edge error calculation. The following is to combine them into a graph optimization problem:

Listing 7.13: slambook2/ch7/pose_estimation_3d2d.cpp

```
void bundleAdjustmentG2O(
const VecVector3d &points_3d,
const VecVector2d &points_2d,
const Mat &K,
Sophus::SE3d &pose) {
  // Build graph optimization, first let's define g2o
  typedef g2o::BlockSolver<g2o::BlockSolverTraits<6, 3>> BlockSolverType;  // pose is
      6, landmark is 3
  typedef g2o::LinearSolverDense<BlockSolverType::PoseMatrixType> LinearSolverType;
  // Gradient descent method, you can choose from GN, LM, DogLeg
  auto solver = new g2o::OptimizationAlgorithmsGaussNewton(
    g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>()));
  g2o::SparseOptimizer optimizer;     // Graph model
```

```
  optimizer.setAlgorithm(solver);   // Set up the solver
  optimizer.setVerbose(true);       // Turn on verbose output for debugging

  // vertex
  VertexPose *vertex_pose = new VertexPose(); // camera vertex_pose
  vertex_pose->setId(0);
  vertex_pose->setEstimate(Sophus::SE3d());
  optimizer.addVertex(vertex_pose);

  // K
  Eigen::Matrix3d K_eigen;
  K_eigen <<
  K.at<double>(0, 0), K.at<double>(0, 1), K.at<double>(0, 2),
  K.at<double>(1, 0), K.at<double>(1, 1), K.at<double>(1, 2),
  K.at<double>(2, 0), K.at<double>(2, 1), K.at<double>(2, 2);

  // edges
  int index = 1;
  for (size_t i = 0; i < points_2d.size(); ++i) {
    auto p2d = points_2d[i];
    auto p3d = points_3d[i];
    EdgeProjection *edge = new EdgeProjection(p3d, K_eigen);
    edge->setId(index);
    edge->setVertex(0, vertex_pose);
    edge->setMeasurement(p2d);
    edge->setInformation(Eigen::Matrix2d::Identity());
    optimizer.addEdge(edge);
    index++;
  }

  chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
  optimizer.setVerbose(true);
  optimizer.initializeOptimization();
  optimizer.optimize(10);
  chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
  chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double
      >>(t2 - t1);
  cout << "optimization costs time: " << time_used.count() << " seconds." << endl;
  cout << "pose estimated by g2o =\n" << vertex_pose->estimate().matrix() << endl;
  pose = vertex_pose->estimate();
}
```

The program is similar to g2o in lecture 6. We first declare the g2o graph optimizer, and configure the optimization solver and gradient descent method. Then based on the estimated feature points, put the pose and spatial points into the graph. Finally, the optimization function is called. The partial output of the run is as follows:

Listing 7.14:

```
./build/pose_estimation_3d2d 1.png 2.png 1_depth.png 2_depth.png
-- Max dist : 95.000000
-- Min dist : 4.000000
In total, we get 79 set of feature points
3d-2d pairs: 76
solve pnp in opencv cost time: 0.000332991 seconds.
R=
[0.9978662025826269, -0.05167241613316376, 0.03991244360207524;
0.0505958915956335, 0.998339762771668, 0.02752769192381471;
-0.04126860182960625, -0.025449547736074, 0.998823919929363]
t=
[-0.1272259656955879;
-0.007507297652615337;
0.06138584177157709]
calling bundle adjustment by gauss newton
iteration 0 cost=645538.1857253
iteration 1 cost=12750.239874896
iteration 2 cost=12301.774589343
iteration 3 cost=12301.427574651
iteration 4 cost=12301.426806652
pose by g-n:
0.99786618832  -0.0516873580423    0.039893448423   -0.127218696289
```

```
0.0506143671126    0.998340854865   0.0274540224544 -0.00738695798083
-0.0412462852904  -0.0253762590968    0.998826706403   0.0617019263823
0                 0                  0                1
solve pnp by gauss newton cost time: 0.000159492 seconds.
calling bundle adjustment by g2o
iteration= 0   chi2= 413.390599  time= 2.7291e-05  cumTime= 2.7291e-05   edges= 76
     schur= 0  lambda= 79.000412   levenbergIter= 1
iteration= 1   chi2= 301.367030  time= 1.47e-05  cumTime= 4.1991e-05   edges= 76
     schur= 0  lambda= 26.333471   levenbergIter= 1
iteration= 2   chi2= 301.365779  time= 1.7794e-05  cumTime= 5.9785e-05   edges= 76
     schur= 0  lambda= 17.555647   levenbergIter= 1
iteration= 3   chi2= 301.365779  time= 1.4875e-05  cumTime= 7.466e-05  edges= 76
     schur= 0  lambda= 11.703765   levenbergIter= 1
iteration= 4   chi2= 301.365779  time= 1.3132e-05  cumTime= 8.7792e-05   edges= 76
     schur= 0  lambda= 7.802510  levenbergIter= 1
iteration= 5   chi2= 301.365779  time= 2.0379e-05  cumTime= 0.000108171  edges= 76
     schur= 0  lambda= 41.613386   levenbergIter= 3
iteration= 6   chi2= 301.365779  time= 3.4186e-05  cumTime= 0.000142357  edges= 76
     schur= 0  lambda= 2859650082279.672363  levenbergIter= 8
optimization costs time: 0.000763649 seconds.
pose estimated by g2o =
0.997866202583  -0.0516724161336   0.0399124436024   -0.127225965696
0.050595891596    0.998339762772   0.0275276919261 -0.00750729765631
-0.04126860183  -0.0254495477384    0.998823919929   0.0613858417711
0                 0                  0                1
solve pnp by g2o cost time: 0.000923095 seconds.
```

Those three results are basically the same. In terms of efficiency, the Gauss-Newton method implemented by ourselves ranked first with 0.15 milliseconds, followed by OpenCV's PnP, and finally the implementation of g2o. Nonetheless, the time for the three is within 1 millisecond, which shows that the pose estimation algorithm does not really consume computational effort.

Bundle Adjustment is common. It may not be limited to two images. We can put the poses and spatial points matched by multiple images for iterative optimization, and even put the entire SLAM process in. That approach is large in scale and mainly used in the back-end. We will deal with this problem again in Lecture 10. At the front end, we usually consider a small Bundle Adjustment problem regarding local camera poses and feature points, aiming to solve and optimize it in real time.

7.9 3D−3D Iterative Closest Point (ICP)

In the end, we will introduce the 3D-3D pose estimation problem. Suppose we have a set of matched 3D points (for example, we matched two RGB-D images):

$$\mathbf{P} = \{\mathbf{p}_1, \cdots, \mathbf{p}_n\}, \quad \mathbf{P}' = \{\mathbf{p}'_1, \cdots, \mathbf{p}'_n\},$$

Now, we want to find an Euclidean transformation $\mathbf{R}, \mathbf{t}$, which makes *:

$$\forall i, \mathbf{p}_i = \mathbf{R}\mathbf{p}'_i + \mathbf{t}.$$

This problem can be solved by Iterative Closest Point (ICP). The camera model does not appear in the 3D−3D pose estimation, meaning that when only the transformation between two sets of 3D points is considered, it has nothing to do with the camera. Therefore, ICP is also feasible in laser SLAM, but since the features in laser data are not rich enough, it is hard to know the **matching relationship** between the two point sets, and we can only consider the two closest points to be the same. So this method is called iterative closest point. In vision, the feature points provide

* slightly different from the symbols in the previous two chapters. You can consider $\mathbf{p}_i$ as the data in the second image, and $\mathbf{p}'_i$ as the data in the first image, and consistently geting $\mathbf{R}, \mathbf{t}$.

us with a better matching relationship, so the whole problem becomes simpler. In RGB-D SLAM, the camera pose can be estimated in this way. In the following, we use ICP to refer to the motion estimation problem between the two sets of **matched** points.

Similar to PnP, the solution to ICP can be divided into two ways: using linear algebra (mainly SVD), and using nonlinear optimization (similar to Bundle Adjustment). They will be introduced separately below.

7.9.1 Using linear algebra (SVD)

First look at SVD, on behalf of the algebraic method. According to the ICP problem described above, we first define the error term for the point i as:

$$\mathbf{e}_i = \mathbf{p}_i - (\mathbf{R}\mathbf{p}_i' + \mathbf{t}). \tag{7.49}$$

Then, construct a least-square problem to find the $\mathbf{R}, \mathbf{t}$ by minimization of sum of the squared errors:

$$\min_{\mathbf{R},\mathbf{t}} \frac{1}{2} \sum_{i=1}^{n} \|(\mathbf{p}_i - (\mathbf{R}\mathbf{p}_i{}' + \mathbf{t}))\|_2^2. \tag{7.50}$$

First, define the centroids of the two sets of points as:

$$\mathbf{p} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{p}_i), \quad \mathbf{p}' = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{p}_i'). \tag{7.51}$$

Note that the centroid is not subscripted. Then, in the error function:

$$\begin{aligned}
\tfrac{1}{2}\sum_{i=1}^{n} \|\mathbf{p}_i - (\mathbf{R}\mathbf{p}_i{}' + \mathbf{t})\|^2 &= \tfrac{1}{2}\sum_{i=1}^{n} \left\|\mathbf{p}_i - \mathbf{R}\mathbf{p}_i{}' - \mathbf{t} - \mathbf{p} + \mathbf{R}\mathbf{p}' + \mathbf{p} - \mathbf{R}\mathbf{p}'\right\|^2 \\
&= \tfrac{1}{2}\sum_{i=1}^{n} \left\|(\mathbf{p}_i - \mathbf{p} - \mathbf{R}(\mathbf{p}_i{}' - \mathbf{p}')) + (\mathbf{p} - \mathbf{R}\mathbf{p}' - \mathbf{t})\right\|^2 \\
&= \tfrac{1}{2}\sum_{i=1}^{n} (\|\mathbf{p}_i - \mathbf{p} - \mathbf{R}(\mathbf{p}_i{}' - \mathbf{p}')\|^2 + \left\|\mathbf{p} - \mathbf{R}\mathbf{p}' - \mathbf{t}\right\|^2 + \\
&\quad 2(\mathbf{p}_i - \mathbf{p} - \mathbf{R}(\mathbf{p}_i{}' - \mathbf{p}'))^T (\mathbf{p} - \mathbf{R}\mathbf{p}' - \mathbf{t})).
\end{aligned}$$

Since $(\mathbf{p}_i - \mathbf{p} - \mathbf{R}(\mathbf{p}_i{}' - \mathbf{p}'))$ is zero after the summation, so the optimization objective function can be simplified to

$$\min_{\mathbf{R},\mathbf{t}} J = \frac{1}{2}\sum_{i=1}^{n} \|\mathbf{p}_i - \mathbf{p} - \mathbf{R}(\mathbf{p}_i{}' - \mathbf{p}')\|^2 + \left\|\mathbf{p} - \mathbf{R}\mathbf{p}' - \mathbf{t}\right\|^2. \tag{7.52}$$

Carefully observe those two terms, we find that the first term is only related to the rotation matrix $\mathbf{R}$, while the second has both $\mathbf{R}$ and $\mathbf{t}$, but only related to the centroid. As long as we get $\mathbf{R}$, we can get $\mathbf{t}$ by making the second term zero. Therefore, ICP can be solved in the following three steps:

1. Calculate the centroid positions of the two groups of points $\mathbf{p}, \mathbf{p}'$, and then calculate the **de-centroid coordinates** of each point:

$$\mathbf{q}_i = \mathbf{p}_i - \mathbf{p}, \quad \mathbf{q}_i' = \mathbf{p}_i' - \mathbf{p}'.$$

2. The rotation matrix is calculated according to the following optimization

problem:

$$\mathbf{R}^* = \arg\min_{\mathbf{R}} \frac{1}{2} \sum_{i=1}^{n} \left\| \mathbf{q}_i - \mathbf{R}\mathbf{q}_i' \right\|^2. \tag{7.53}$$

3. Calculate $\mathbf{t}$ according to $\mathbf{R}$ in step 2:

$$\mathbf{t}^* = \mathbf{p} - \mathbf{R}\mathbf{p}'. \tag{7.54}$$

We find that once the rotation between the two sets of points is found, and the translation is easy to obtain. So we focus on the calculation of $\mathbf{R}$. Expand the error term about $\mathbf{R}$, get:

$$\frac{1}{2} \sum_{i=1}^{n} \left\| \mathbf{q}_i - \mathbf{R}\mathbf{q}_i' \right\|^2 = \frac{1}{2} \sum_{i=1}^{n} \mathbf{q}_i^{\mathrm{T}} \mathbf{q}_i + \mathbf{q}_i'^{\mathrm{T}} \mathbf{R}^{\mathrm{T}} \mathbf{R} \mathbf{q}_i' - 2\mathbf{q}_i^{\mathrm{T}} \mathbf{R} \mathbf{q}_i'. \tag{7.55}$$

The first item is not relevant to $\mathbf{R}$. The second item can also be ignored since $\mathbf{R}^{\mathrm{T}}\mathbf{R} = \mathbf{I}$. Therefore, the actual optimization objective function becomes the simplest form of

$$\sum_{i=1}^{n} -\mathbf{q}_i^{\mathrm{T}} \mathbf{R} \mathbf{q}_i' = \sum_{i=1}^{n} -\mathrm{tr}\left(\mathbf{R}\mathbf{q}_i' \mathbf{q}_i^{\mathrm{T}} \right) = -\mathrm{tr}\left(\mathbf{R} \sum_{i=1}^{n} \mathbf{q}_i' \mathbf{q}_i^{\mathrm{T}} \right). \tag{7.56}$$

Next, we introduce how to solve the optimal $\mathbf{R}$ in the above problem through SVD. The proof of optimality is more complicated, and interested readers can refer to the literature [58, 59]. To find $\mathbf{R}$, first define the matrix:

$$\mathbf{W} = \sum_{i=1}^{n} \mathbf{q}_i \mathbf{q}_i'^{\mathrm{T}}. \tag{7.57}$$

$\mathbf{W}$ is a 3×3 matrix. Performing SVD decomposition on $\mathbf{W}$, we get:

$$\mathbf{W} = \mathbf{U}\, \mathbf{V}^{\mathrm{T}}. \tag{7.58}$$

is a diagonal matrix composed of singular values, the diagonal elements are arranged from large to small, and $\mathbf{U}$ and $\mathbf{V}$ are diagonal matrices. When $\mathbf{W}$ is of full rank, $\mathbf{R}$ is

$$\mathbf{R} = \mathbf{U}\mathbf{V}^{\mathrm{T}}. \tag{7.59}$$

Once solving $\mathbf{R}$, use the equation (7.54) to solve $\mathbf{t}$. If the determinant of $\mathbf{R}$ is negative, then $-\mathbf{R}$ is taken as the optimal value.

7.9.2 Using non-linear optimization

Another way to solve ICP is to use non-linear optimization to find the optimal value iteratively. This method is similar to the PnP we described earlier. When expressing the poses in Lie algebra, the objective function can be written as

$$\min_{\boldsymbol{\xi}} = \frac{1}{2} \sum_{i=1}^{n} \left\| \left(\mathbf{p}_i - \exp\left(\boldsymbol{\xi}^{\wedge} \right) \mathbf{p}_i' \right) \right\|_2^2. \tag{7.60}$$

The derivative of a single error term with respect to the pose has been derived above, using the Lie algebra perturbation model:

$$\frac{\partial \mathbf{e}}{\partial \delta \boldsymbol{\xi}} = -\left(\exp\left(\boldsymbol{\xi}^{\wedge} \right) \mathbf{p}_i{}' \right)^{\odot}. \tag{7.61}$$

Therefore, in nonlinear optimization, it only needs to iterate continuously to find the minimum value. Moreover, it can be proved that [5], the ICP problem has a unique solution or an infinite number of solutions. In the case of a unique solution, as long as the minimum value solution can be found, then **this minimum value is the global optimal value**. So the local minimum is always the global minimum. This also means that the initial value of the ICP solution can be arbitrarily selected. This is a big advantage of solving ICP when the points have been matched already.

It should be noted that the ICP we are talking about here refers to the problem of pose estimation when the matching is given by the image features. In the case of known matching, this least squares problem actually has an analytical solution [60–62], so iterative optimization is not necessary. ICP researchers tend to be more concerned about the unknown matching situation. So, why do we introduce optimization-based ICP? This is because, in some cases, such as in RGB-D SLAM, the depth of a pixel may or may not be measured, so we can combine PnP and ICP optimization: For feature points with known depth, model their 3D-3D errors; for feature points with unknown depths, model 3D-2D reprojection errors. Therefore, all errors can be considered in the same problem, making the solution more convenient.

7.10 Practice: Solving ICP

7.10.1 Using SVD

Let's demonstrate how to use SVD and nonlinear optimization to solve ICP. In this section, we use two RGB-D images to obtain two sets of 3D points through feature matching, and finally use ICP to calculate their pose transformation. Since OpenCV does not currently have a method to calculate two sets of ICPs with matching points, and its principle is not complicated, we will implement an ICP by ourselves.

Listing 7.15: slambook2/ch7/pose_estimation_3d3d.cpp

```
void pose_estimation_3d3d(
const vector<Point3f> &pts1,
const vector<Point3f> &pts2,
Mat &R, Mat &t) {
  Point3f p1, p2;     // center of mass
  int N = pts1.size();
  for (int i = 0; i < N; i++) {
    p1 += pts1[i];
    p2 += pts2[i];
  }
  p1 = Point3f(Vec3f(p1) / N);
  p2 = Point3f(Vec3f(p2) / N);
  vector<Point3f> q1(N), q2(N); // remove the center
  for (int i = 0; i < N; i++) {
    q1[i] = pts1[i] - p1;
    q2[i] = pts2[i] - p2;
  }

  // compute q1*q2^T
  Eigen::Matrix3d W = Eigen::Matrix3d::Zero();
  for (int i = 0; i < N; i++) {
    W += Eigen::Vector3d(q1[i].x, q1[i].y, q1[i].z) * Eigen::Vector3d(q2[i].x, q2[i].y
        , q2[i].z).transpose();
  }
  cout << "W=" << W << endl;

  // SVD on W
  Eigen::JacobiSVD<Eigen::Matrix3d> svd(W, Eigen::ComputeFullU | Eigen::ComputeFullV);
  Eigen::Matrix3d U = svd.matrixU();
```

```
  Eigen::Matrix3d V = svd.matrixV();

  cout << "U=" << U << endl;
  cout << "V=" << V << endl;

  Eigen::Matrix3d R_ = U * (V.transpose());
  if (R_.determinant() < 0) {
    R_ = -R_;
  }
  Eigen::Vector3d t_ = Eigen::Vector3d(p1.x, p1.y, p1.z) - R_ * Eigen::Vector3d(p2.x,
      p2.y, p2.z);

  // convert to cv::Mat
  R = (Mat_<double>(3, 3) <<
    R_(0, 0), R_(0, 1), R_(0, 2),
    R_(1, 0), R_(1, 1), R_(1, 2),
    R_(2, 0), R_(2, 1), R_(2, 2)
  );
  t = (Mat_<double>(3, 1) << t_(0, 0), t_(1, 0), t_(2, 0));
}
```

The implementation of ICP is consistent with the previous theoritical part. We call Eigen for SVD, and then calculate the $\mathbf{R}, \mathbf{t}$ matrix. We output the matched result, but please note that since the previous derivation is based on $\mathbf{p}_i = \mathbf{R}\mathbf{p}'_i + \mathbf{t}$, here is $\mathbf{R}, \mathbf{t}$ is the transformation from the second frame to the first frame, which is the opposite of the previous theoritical part. So in the output result, we also printed the inverse transform:

Listing 7.16:

```
./build/pose_estimation_3d3d 1.png 2.png 1_depth.png 2_depth.png
-- Max dist : 95.000000
-- Min dist : 4.000000
In total, we get 79 set of feature points
3d-3d pairs: 74
W=  11.9404 -0.567258   1.64182
-1.79283   4.31299  -6.57615
3.12791  -6.55815   10.8576
U=  0.474144  -0.880373 -0.0114952
-0.460275  -0.258979   0.849163
0.750556   0.397334   0.528006
V=  0.535211  -0.844064 -0.0332488
-0.434767  -0.309001    0.84587
0.724242   0.438263   0.532352
ICP via SVD results:
R = [0.9972395977366739, 0.05617039856770099, -0.04855997354553433;
-0.05598345194682017, 0.9984181427731508, 0.005202431117423125;
0.0487753812298326, -0.002469515369266572, 0.9988067198811421]
t = [0.1417248739257469;
-0.05551033302525193;
-0.03119093188273858]
R_inv = [0.9972395977366739, -0.05598345194682017, 0.0487753812298326;
0.05617039856770099, 0.9984181427731508, -0.002469515369266572;
-0.04855997354553433, 0.005202431117423125, 0.9988067198811421]
t_inv = [-0.1429199667309695;
0.04738475446275858;
0.03832465717628181]
```

Readers can compare the difference between ICP and PnP, and the motion estimation results of epipolar geometry. We can conclude that we are using more and more information (no depth - the depth of one image - the depth of two images). Therefore, when the depth is accurate, the estimates will be more and more accurate. However, due to the noise in Kinect's depth map and the possibility of data loss, we have to discard some feature points without depth data. This may cause the estimation of ICP to be inaccurate, and if too many feature points are discarded, it may cause a situation where motion estimation cannot be performed due to too few feature points.

7.10.2 Using non-linear optimization

Now consider using nonlinear optimization to calculate ICP. We still use Lie algebra to optimize the camera pose. The RGB-D camera can observe the 3D position of the landmarks every time, thereby generating 3D observation data. We use the VertexPose in the previous practice, and then define the unary edges of 3D-3D:

Listing 7.17: slambook2/ch7/pose_estimation_3d3d.cpp

```
1
2  /// g2o edge
3  class EdgeProjectXYZRGBDPoseOnly : public g2o::BaseUnaryEdge<3, Eigen::Vector3d,
       VertexPose> {
4    public:
5    EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
6
7    EdgeProjectXYZRGBDPoseOnly(const Eigen::Vector3d &point) : _point(point) {}
8
9    virtual void computeError() override {
10     const VertexPose *pose = static_cast<const VertexPose *> ( _vertices[0] );
11     _error = _measurement - pose->estimate() * _point;
12   }
13
14   virtual void linearizeOplus() override {
15     VertexPose *pose = static_cast<VertexPose *>(_vertices[0]);
16     Sophus::SE3d T = pose->estimate();
17     Eigen::Vector3d xyz_trans = T * _point;
18     _jacobianOplusXi.block<3, 3>(0, 0) = -Eigen::Matrix3d::Identity();
19     _jacobianOplusXi.block<3, 3>(0, 3) = Sophus::SO3d::hat(xyz_trans);
20   }
21
22   bool read(istream &in) {}
23
24   bool write(ostream &out) const {}
25
26   protected:
27   Eigen::Vector3d _point;
28 };
```

They are unary edges, written similar to the g2o::EdgeSE3ProjectXYZ mentioned earlier, but the observation has changed from 2 to 3 dimensions, there is no camera model involved, and only one node is related. Please pay attention to the form of the Jacobian matrix here, it must be consistent with our previous derivation. The Jacobian matrix gives the derivative of the camera pose and is 3×6.

The code for using g2o for optimization is similar, we just set the nodes and edges for graph optimization. Readers are suggested check the source file for this part of the code, which is not listed here. Now, let's take a look at the results of the optimization:

Listing 7.18:

```
1  iteration= 0   chi2= 1.811539  time= 1.7046e-05  cumTime= 1.7046e-05   edges= 74
       schur= 0
2  iteration= 1   chi2= 1.811051  time= 1.0422e-05  cumTime= 2.7468e-05   edges= 74
       schur= 0
3  iteration= 2   chi2= 1.811050  time= 9.589e-06   cumTime= 3.7057e-05   edges= 74
       schur= 0□ □ □
4  ...
5  iteration= 9   chi2= 1.811050  time= 9.113e-06   cumTime= 0.000100604  edges= 74
       schur= 0
6  optimization costs time: 0.000559208 seconds.
7
8  after optimization:
9  T=
10 0.99724  0.0561704   -0.04856   0.141725
11 -0.0559834   0.998418 0.00520242 -0.0555103
12 0.0487754 -0.0024695   0.998807 -0.0311913
13 0          0          0          1
```

We found that the overall error has become stable after only one iteration, indicating that the algorithm has converged after only one iteration. From the result of the pose, it can be seen that it is almost the same as the pose calculated by the previous SVD, which shows that SVD has already given an analytical solution to the optimization problem. Therefore, in this practice, it can be considered that the result given by SVD is the optimal value of the camera pose.

It should be noted that in the practice of ICP, we used feature points that have depth readings in both images. However, in fact, as long as the depth of one of the images is determined, we can use errors similar to PnP to add them to the optimization. In addition to the camera pose, considering the spatial points as optimization variables is also a way to solve the problem. We should be clear that the actual solution is very flexible and does not need to be bound to a certain fixed form. If you consider points and cameras at the same time, the whole problem becomes **more flexible**, and you may get other solutions. For example, you can make the camera rotate less and move the point more. This reflects that in Bundle Adjustment, we would like to have as many constraints as possible, because multiple observations will bring more information and enable us to estimate each variable more accurately.

7.11 Summary

This chapter introduces several important issues in visual odometry based on feature points. include:

1. How the feature points are extracted and matched.
2. How to estimate camera motion through 2D-2D feature points.
3. How to estimate the spatial position of a point from a 2D−2D match.
4. 3D−2D PnP problem, its linear solution and Bundle Adjustment solution.
5. 3D−3D ICP problem, its linear solution and Bundle Adjustment solution.

This chapter is complicated in content and combines the basic knowledge of the previous lectures. If readers find it difficult to understand, they can go back to review the previous knowledge. It is best to do the practice yourself to entirely understand the content of the motion estimation.

What needs to be mentioned here is we have omitted a lot of discussion about some special situations. For example, what happens if the given feature points are coplanar during the solution of the epipolar geometry (this is mentioned in the homography matrix $\mathbf{H}$)? What happens to collinear? Given such a solution in PnP and ICP, what will happen? Can the solution algorithm recognize these special cases and report that the resulting solution may be unreliable? Can you give the estimated uncertainty of $\mathbf{T}$? Although they are all worthy of research and exploration, the discussion on them is more suitable in specific papers. The goal of this book is extensive knowledge coverage and basic knowledge. We will not expand on these issues for now. At the same time, these situations rarely occur in engineering. If you care about these rare situations, you can read papers such as [2].

Exercises

1. In addition to the ORB feature points introduced in this book, what other feature points do you know? Please elaborate the principles of SIFT or SURF, and compare their advantages and disadvantages with ORB.

2. Design a program to call other types of feature points in OpenCV. Compare their time spent on your machine when extracting 1000 feature points.

3.* We found that the ORB feature points provided by OpenCV are not evenly distributed in the image. Can you find or propose a way to make the distribution of feature points more evenly?

4. Investigate why FLANN can quickly handle matching problems. In addition to FLANN, what other ways to accelerate matching?

5. Substitute the EPnP used in the demo program with other PnP methods and investigate their working principles.

6. In PnP optimization, taking the observation of the first camera into consideration, how should the problem/program be formed? How will the final result change?

7. In the ICP program, if the spatial point is also considered as an optimization variable, how should the program be written? How will the final result change?

8.* In the feature point matching, mismatches will inevitably be encountered. What happens if we put the wrong match into PnP or ICP? What methods can you think of to avoid mismatches?

9.* Use the SE3 class in Sophus to design the nodes and edges of g2o by yourself to implement the optimization of PnP and ICP.

10.* Implement the optimization of PnP and ICP in Ceres.

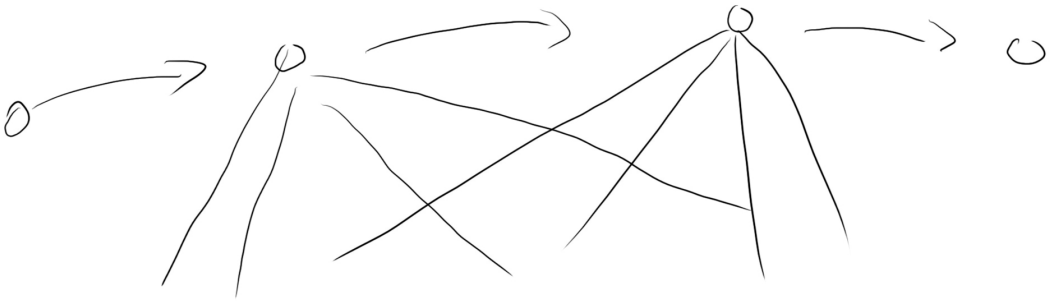
Chapter 8

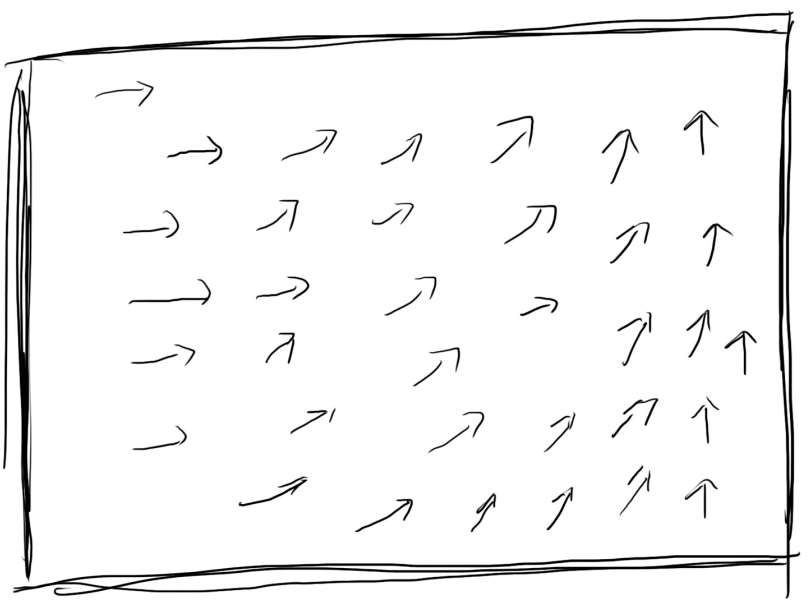
Visual Odometry: Part 2

Goal of Study

1. Understand the principle of optical flow to track feature points.
2. Understand how the direct method estimates the camera pose.
3. Use g2o for direct method.

Different from feature point method, direct method is another main stream of visual odometry. Despite that it has not yet become the mainstream of VO, after recent years of development, the direct method can compete with feature point method to some extent. In this chapter, we will introduce the principle of the direct method and implement its core part.





$$\min \| I_1(p) - I_2(K(Rp+t)) \|_2$$

$$J = \frac{\partial I}{\partial u} \frac{\partial u}{\partial \xi} \frac{\partial \xi}{\partial \xi}$$

8.1 Origin of the direct method

In the last chapter, we introduced using feature points to estimate camera motion. Although the feature point method plays a key role in visual odometry, researchers still believe that it has at least the following shortcomings:

1. The extraction of key points and the calculation of descriptors are very time-consuming. In practice, SIFT currently cannot be calculated in real time on the CPU, and ORB also requires nearly 20ms of calculation. If the entire SLAM runs at a speed of 30 milliseconds per frame, more than half of the time will be spent on feature points calculation.

2. When using feature points, all information except feature points is ignored. An image has hundreds of thousands of pixels, but only a few hundred feature points. Using only feature points discards most of the **possibly useful** image information.

3. The camera sometimes moves to places **lack of feature**, where there is often no obvious texture information. For example, sometimes we will face a white wall or an empty corridor. The number of feature points in these scenes will be significantly reduced, and we may not find enough matching points to calculate camera motion.

Now we see that there are indeed some problems with using feature points. Is there any way to overcome these shortcomings? We have the following ideas:

- Keep feature points, but discard their descriptors. At the same time, use **Optical Flow** to track the motion of feature points. This can avoid the time brought by the calculation and matching of the descriptor, and the time spent on calculating optical flow itself is less than the descriptor calculation and matching.

- Only calculate key points, not descriptors. At the same time, use **Direct Method** to calculate the position of the feature point in the image at the next timestamp. This can also save the time spent on the calculation of the descriptor as well as the the calculation of optical flow.

The first method still uses feature points, but substitutes the descriptor matching with optical flow tracking, and still uses epipolar geometry, PnP or ICP algorithms to estimate camera motion. This still requires that the extracted keypoints are distinguishable, that is, we need to extract the corner points. In the direct method, we will estimate the camera motion and the projection of the points at the same time according to the **pixel gray information** of the image, and the extracted points to be corner points is not longer a hard prerequisite. As you will see later, they can even be randomly selected points.

When using the feature point method to estimate camera motion, we regard feature points as fixed points in three-dimensional space. According to their projection position in the camera, the camera motion is optimized by **minimize reprojection error**. In this process, we need to know exactly the pixel position of the spatial point after the projection of the two cameras-this is why we need to match or track the features. Meanwhile, computing and matching features requires a lot of computation. In contrast, in the direct method, we do not need to know the correspondence between points in advance, but find it by minimizing **Photometric error**.

We will focus on the direct method in this chapter. It is to overcome the shortcomings of the feature point method listed above. The direct method estimates the camera motion based on the brightness information of the pixels, and can completely eliminate the calculation of keypoints and descriptors. Therefore, it not only saves the calculation time of features, but also solves the problems caused by lacking features. As long as there are brightness changes in the scene (it can be a gradual change without forming a local image gradient), the direct method will work. According to the number of pixels used, the direct method can be categorized into sparse, semi-dense and dense. Compared with the feature point method that can only reconstruct sparse feature points (sparse map), the direct method also has the capability to restore semi-dense or dense structures.

Historically, there were also early uses of the direct method [63]. With the emergence of some open source projects that use the direct method, such as SVO[64], LSD-SLAM[65], DSO[66], etc. Direct method became a more and more important part of the visual odometry.

8.2 2D Optical Flow

Direct method was inspired by the optical flow. They are similar and use the same assumptions. Optical flow describes the motion of pixels in the image, and the direct method is accompanied by a camera motion model. Before the direct method, we will introduce optical flow first.

Optical flow is a method of describing the movement of pixels between images, as shown in Figure 8-1 . The same pixel will move in the image over time, and we want to track its movement. The calculation of motion of a portion of pixels is called **sparse optical flow**, and the calculation of all pixels in an image is called **dense optical flow**. A well-known sparse optical flow method is called Lucas-Kanade optical flow [67]. It can be used to track the position of feature points in SLAM. Dense optical flow is represented by Horn-Schunck optical flow [68]. This section mainly introduces Lucas-Kanade optical flow, also known as LK optical flow.

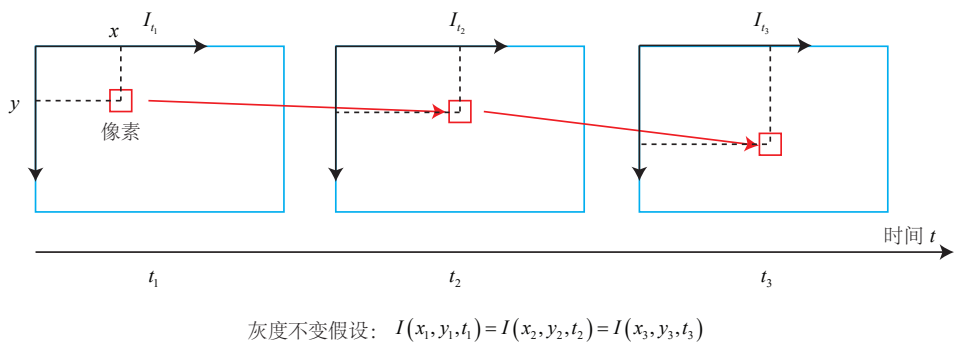


Figure 8-1: workflow of LK method

Lucas-Kanade optical flow

In the LK optical flow, we think that the image from the camera changes over time. The image can be regarded as a function of time $\mathbf{I}(t)$. Then, for a pixel at (x, y) at

time t, its grayscale can be written as

$$\mathbf{I}(x, y, t).$$

In this way, the image is regarded as a function of position and time, and its range is the grayscale of the pixels in the image. Now consider a fixed point in space, its pixel coordinates at time t are x, y. Due to the movement of the camera, its image coordinates will change. We want to estimate the position of this space point in the image at other times. How to estimate it? Here we will introduce the basic assumptions of the optical flow method.

Constant Brightness The pixel grayvalue of the same space point is constant in each image.

For the pixel at (x, y) at time t, suppose it moves to $(x + \mathrm{d}x, y + \mathrm{d}y)$ at time $t + \mathrm{d}t$. Since the grayscale is unchanged, we have:

$$\mathbf{I}(x + \mathrm{d}x, y + \mathrm{d}y, t + \mathrm{d}t) = \mathbf{I}(x, y, t). \tag{8.1}$$

Note that in the most of time in practice the assumption of constant brightness is not true. In fact, due to the different materials of the objects, the pixels will have highlights and shadows; sometimes, the camera will automatically adjust its exposure parameters to make the overall image brighter or darker. At these times, the assumption of constant brightness is invalid, so the result of optical flow is not necessarily reliable. However, on the other hand, all algorithms work under certain assumptions. If we do not make any assumptions, we cannot design practical algorithms. So, let us consider this assumption to be true for now and see how to calculate the motion of the pixels.

Carry out Taylor expansion on the left side and only keep the first-order term, we have:

$$\mathbf{I}\left(x + \mathrm{d}x, y + \mathrm{d}y, t + \mathrm{d}t\right) \approx \mathbf{I}\left(x, y, t\right) + \frac{\partial \mathbf{I}}{\partial x}\mathrm{d}x + \frac{\partial \mathbf{I}}{\partial y}\mathrm{d}y + \frac{\partial \mathbf{I}}{\partial t}\mathrm{d}t. \tag{8.2}$$

Because we assume that the brightness does not change, the brightness at the next timestamp is equal to the current one, thus:

$$\frac{\partial \mathbf{I}}{\partial x}\mathrm{d}x + \frac{\partial \mathbf{I}}{\partial y}\mathrm{d}y + \frac{\partial \mathbf{I}}{\partial t}\mathrm{d}t = 0. \tag{8.3}$$

Divide both sides by $\mathrm{d}t$:

$$\frac{\partial \mathbf{I}}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial \mathbf{I}}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} = -\frac{\partial \mathbf{I}}{\partial t}. \tag{8.4}$$

Where $\mathrm{d}x/\mathrm{d}t$ is the speed of the pixel on the x axis, and $\mathrm{d}y/\mathrm{d}t$ is the speed on the y axis, denoting them as u, v. At the same time, $\partial \mathbf{I}/\partial x$ is the gradient of the image in the x direction at this point, and the other is the gradient in the y direction, denoted as $\mathbf{I}_x, \mathbf{I}_y$. Denote the change of the image brightness with respect to time as $\mathbf{I}_t$. They can be written in a matrix form as:

$$\begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = -\mathbf{I}_t. \tag{8.5}$$

What we want is to calculate the motion u, v of the pixel, but this formula is a linear equation with two variables, and we cannot find u, v by itself. Therefore,

additional constraints are needed to calculate u, v. In LK optical flow, we assume **pixels in a certain window have the same motion**.

Consider a window of size $w \times w$, which contains w^2 pixels. Since the pixels in this window are assumed to have the same motion, we have a total of w^2 equations:

$$\begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix}_k \begin{bmatrix} u \\ v \end{bmatrix} = -\mathbf{I}_{tk}, \quad k = 1, \ldots, w^2. \tag{8.6}$$

Stacking them:

$$\mathbf{A} = \begin{bmatrix} [\mathbf{I}_x, \mathbf{I}_y]_1 \\ \vdots \\ [\mathbf{I}_x, \mathbf{I}_y]_k \end{bmatrix}, \mathbf{b} = \begin{bmatrix} \mathbf{I}_{t1} \\ \vdots \\ \mathbf{I}_{tk} \end{bmatrix}. \tag{8.7}$$

The whole equation is:

$$\mathbf{A} \begin{bmatrix} u \\ v \end{bmatrix} = -\mathbf{b}. \tag{8.8}$$

This is an overdetermined linear equation about u, v. And we can find its least square solution.

$$\begin{bmatrix} u \\ v \end{bmatrix}^* = -\left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{b}. \tag{8.9}$$

In this way, the speed u, v of pixels between images is obtained. When t takes discrete moments instead of continuous time, we can estimate the position of a block of pixels in several images. Since the pixel gradient is only valid locally, if one iteration does not produce a reasonable result, we will iterate this calculation several times. In SLAM, LK optical flow is often used to track the motion of corner points. We can have a deeper understanding of it through the program.

8.3 Practice: LK Optical Flow

8.3.1 Use LK optical flow

In the practice, we will use several sample images to track their feature points with OpenCV optical flow. At the same time, we will also manually implement a LK optical flow for a comprehensive understanding. We use two sample images from the Euroc dataset, extract the corner points in the first image, and then use optical flow to track their position in the second image. First, let's use the LK optical flow in OpenCV:

Listing 8.1: slambook2/ch8/optical_flow.cpp (snippet))

```
// use opencv's flow for validation
vector<Point2f> pt1, pt2;
for (auto &kp: kp1) pt1.push_back(kp.pt);
vector<uchar> status;
vector<float> error;
cv::calcOpticalFlowPyrLK(img1, img2, pt1, pt2, status, error);
```

The optical flow in OpenCV is very simple to use. You only need to call the cv::calcOpticalFlowPyrLK function, provide two images and the corresponding feature points, you can get the tracked points, as well as the status and error of each point. We can determine whether the corresponding point is tracked correctly according to whether the status variable is 1. This function also has some optional parameters, but in the demonstration we only use the default parameters. We omit

other codes that mention features and draw results here, which have been shown in the previous code snippet.

8.3.2 Implement optical flow with Gauss Newton method

Single-layer optical flow

Optical flow can also be seen as an optimization problem: by minimizing the grayscale error, the optimal pixel shift is estimated. Therefore, similar to those various Gauss-Newton methods previously implemented, we now also implement an optical flow based on the Gauss-Newton method.

Listing 8.2: slambook2/ch8/optical_flow.cpp (snippet))

```
class OpticalFlowTracker {
public:
  OpticalFlowTracker(
    const Mat &img1_,
    const Mat &img2_,
    const vector<KeyPoint> &kp1_,
    vector<KeyPoint> &kp2_,
    vector<bool> &success_,
    bool inverse_ = true, bool has_initial_ = false) :
    img1(img1_), img2(img2_), kp1(kp1_), kp2(kp2_), success(success_), inverse(
        inverse_),
    has_initial(has_initial_) {}

  void calculateOpticalFlow(const Range &range);

private:
  const Mat &img1;
  const Mat &img2;
  const vector<KeyPoint> &kp1;
  vector<KeyPoint> &kp2;
  vector<bool> &success;
  bool inverse = true;
  bool has_initial = false;
};

void OpticalFlowSingleLevel(
  const Mat &img1,
  const Mat &img2,
  const vector<KeyPoint> &kp1,
  vector<KeyPoint> &kp2,
  vector<bool> &success,
  bool inverse, bool has_initial) {
  kp2.resize(kp1.size());
  success.resize(kp1.size());
  OpticalFlowTracker tracker(img1, img2, kp1, kp2, success, inverse, has_initial);
  parallel_for_(Range(0, kp1.size()),
    std::bind(&OpticalFlowTracker::calculateOpticalFlow, &tracker, placeholders::_1));
}

void OpticalFlowTracker::calculateOpticalFlow(const Range &range) {
  // parameters
  int half_patch_size = 4;
  int iterations = 10;
  for (size_t i = range.start; i < range.end; i++) {
    auto kp = kp1[i];
    double dx = 0, dy = 0; // dx,dy need to be estimated
    if (has_initial) {
      dx = kp2[i].pt.x - kp.pt.x;
      dy = kp2[i].pt.y - kp.pt.y;
    }

    double cost = 0, lastCost = 0;
    bool succ = true; // indicate if this point succeeded

    // Gauss-Newton iterations
    Eigen::Matrix2d H = Eigen::Matrix2d::Zero();    // hessian
```

```
56      Eigen::Vector2d b = Eigen::Vector2d::Zero();    // bias
57      Eigen::Vector2d J;  // jacobian
58      for (int iter = 0; iter < iterations; iter++) {
59        if (inverse == false) {
60          H = Eigen::Matrix2d::Zero();
61          b = Eigen::Vector2d::Zero();
62        } else {
63          // only reset b
64          b = Eigen::Vector2d::Zero();
65        }
66
67        cost = 0;
68
69        // compute cost and jacobian
70        for (int x = -half_patch_size; x < half_patch_size; x++)
71        for (int y = -half_patch_size; y < half_patch_size; y++) {
72          double error = GetPixelValue(img1, kp.pt.x + x, kp.pt.y + y) -
73            GetPixelValue(img2, kp.pt.x + x + dx, kp.pt.y + y + dy);;  // Jacobian
74          if (inverse == false) {
75            J = -1.0 * Eigen::Vector2d(
76              0.5 * (GetPixelValue(img2, kp.pt.x + dx + x + 1, kp.pt.y + dy + y) -
77                GetPixelValue(img2, kp.pt.x + dx + x - 1, kp.pt.y + dy + y)),
78              0.5 * (GetPixelValue(img2, kp.pt.x + dx + x, kp.pt.y + dy + y + 1) -
79                GetPixelValue(img2, kp.pt.x + dx + x, kp.pt.y + dy + y - 1))
80            );
81          } else if (iter == 0) {
82            // in inverse mode, J keeps same for all iterations
83            // NOTE this J does not change when dx, dy is updated, so we can store it
                   and only compute error
84            J = -1.0 * Eigen::Vector2d(
85              0.5 * (GetPixelValue(img1, kp.pt.x + x + 1, kp.pt.y + y) -
86                GetPixelValue(img1, kp.pt.x + x - 1, kp.pt.y + y)),
87              0.5 * (GetPixelValue(img1, kp.pt.x + x, kp.pt.y + y + 1) -
88                GetPixelValue(img1, kp.pt.x + x, kp.pt.y + y - 1))
89            );
90          }
91          // compute H, b and set cost;
92          b += -error * J;
93          cost += error * error;
94          if (inverse == false || iter == 0) {
95            // also update H
96            H += J * J.transpose();
97          }
98        }
99
100       // compute update
101       Eigen::Vector2d update = H.ldlt().solve(b);
102
103       if (std::isnan(update[0])) {
104         // sometimes occurred when we have a black or white patch and H is
                irreversible
105         cout << "update is nan" << endl;
106         succ = false;
107         break;
108       }
109
110       if (iter > 0 && cost > lastCost) {
111         break;
112       }
113
114       // update dx, dy
115       dx += update[0];
116       dy += update[1];
117       lastCost = cost;
118       succ = true;
119
120       if (update.norm() < 1e-2) {
121         // converge
122         break;
123       }
124     }
125
126     success[i] = succ;
127
128     // set kp2
```

```
129         kp2[i].pt = kp.pt + Point2f(dx, dy);
130     }
131 }
```

We have implemented a single-layer optical flow function in the OpticalFlowSingleLevel function, in which cv::parallel_for_ is called in parallel to call OpticalFlowTracker::calculateOpticalFlow, which calculates the optical flow of feature points within a specified range. This parallel for loop is internally implemented by the Intel tbb library. We only need to define the function body according to its interface, and then pass the function to it as a std::function object.

In the implementation of calculateOpticalFlow, we solve such a problem:

$$\min_{\Delta x, \Delta y} \|\mathbf{I}_1(x, y) - \mathbf{I}_2(x + \Delta x, y + \Delta y)\|_2^2 . \tag{8.10}$$

Therefore, the residual is the part inside the brackets, and the corresponding Jacobian is the gradient of the second image at $x + \Delta x, y + \Delta y$. In addition, according to [69], the gradient can also be replaced by the gradient $\mathbf{I}_1(x, y)$ of the first image. This is called **Inverse** optical flow method. In inverse optical flow, the gradient of $\mathbf{I}_1(x, y)$ remains unchanged, so we can use the result calculated in the first iteration in the subsequent iterations. When the Jacobian remains unchanged, the $\mathbf{H}$ matrix is unchanged, and only the residual is calculated for each iteration, which can save a lot of calculation.

Multi-layer optical flow

Since we write optical flow as an optimization problem, we must assume that the initial value of optimization is close to the optimal value to ensure the convergence of the algorithm. Therefore, if the camera moves faster and the difference between the two images is obvious, the single-layer image optical flow method can be easily stuck at a local minimum. While it can be resolved to some extent by image pyramids.

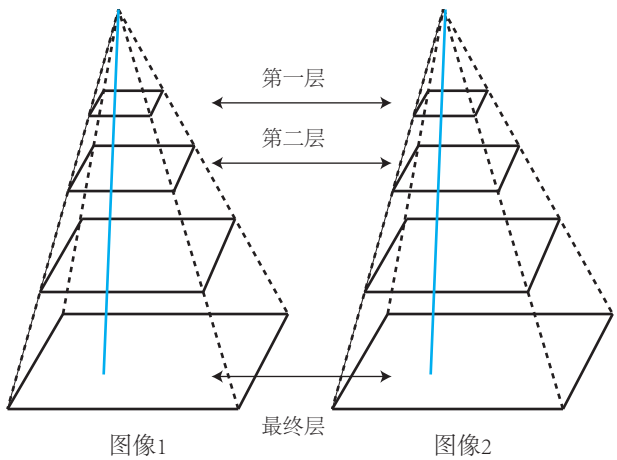


Figure 8-2: Image pyramid and coarse-to-fine process.

Image pyramid refers to scaling the image to get samples in different resolutions, as shown in Figure 8-2. The original image is used as the bottom layer of the pyramid. Every time one layer goes up, the lower layer image is scaled to a certain

magnification, and then a pyramid is obtained. When calculating the optical flow, start from the top layer image, and then use the tracking result of the previous layer as the initial value of the optical flow of the next layer. Since the upper layer image is relatively rough, this process is also called **coarse-to-fine** optical flow, which is also the usual process of optical flow in practice.

The advantage of going from coarse to fine is that when the pixel motion of the original image is large, the motion is still within a small range from the image at the top of the pyramid. For example, if the feature points of the original image move by 20 pixels, it is easy for the optimization to be trapped in the minimum value due to the non-convexity of the image. But now suppose there is a pyramid with a zoom magnification of 0.5 times, then in the upper two layers of images, the pixel movement is only 5 pixels, and the result is obviously better than directly optimizing on the original image.

We have implemented multi-layer optical flow in the program, the code is as follows:

Listing 8.3: slambook2/ch8/optical_flow.cpp (snippet)

```
void OpticalFlowMultiLevel(
  const Mat &img1,
  const Mat &img2,
  const vector<KeyPoint> &kp1,
  vector<KeyPoint> &kp2,
  vector<bool> &success,
  bool inverse) {

  // parameters
  int pyramids = 4;
  double pyramid_scale = 0.5;
  double scales[] = {1.0, 0.5, 0.25, 0.125};

  // create pyramids
  vector<Mat> pyr1, pyr2; // image pyramids
  for (int i = 0; i < pyramids; i++) {
    if (i == 0) {
      pyr1.push_back(img1);
      pyr2.push_back(img2);
    } else {
      Mat img1_pyr, img2_pyr;
      cv::resize(pyr1[i - 1], img1_pyr,
      cv::Size(pyr1[i - 1].cols * pyramid_scale, pyr1[i - 1].rows * pyramid_scale));
      cv::resize(pyr2[i - 1], img2_pyr,
      cv::Size(pyr2[i - 1].cols * pyramid_scale, pyr2[i - 1].rows * pyramid_scale));
      pyr1.push_back(img1_pyr);
      pyr2.push_back(img2_pyr);
    }
  }

  // coarse-to-fine LK tracking in pyramids
  vector<KeyPoint> kp1_pyr, kp2_pyr;
  for (auto &kp:kp1) {
    auto kp_top = kp;
    kp_top.pt *= scales[pyramids - 1];
    kp1_pyr.push_back(kp_top);
    kp2_pyr.push_back(kp_top);
  }

  for (int level = pyramids - 1; level >= 0; level--) {
    // from coarse to fine
    success.clear();
    OpticalFlowSingleLevel(pyr1[level], pyr2[level], kp1_pyr, kp2_pyr, success,
        inverse, true);

    if (level > 0) {
      for (auto &kp: kp1_pyr)
      kp.pt /= pyramid_scale;
      for (auto &kp: kp2_pyr)
      kp.pt /= pyramid_scale;
```

```
    }
  }

  for (auto &kp: kp2_pyr)
    kp2.push_back(kp);
}
```

This code constructs a four-layer pyramid with a scaling rate of 0.5, and calls the single-layer optical flow function to achieve the multi-layer optical flow. In the main function, we tested the performance of OpenCV's optical flow, single-layer optical flow, and multi-layer optical flow on two images, and recorded their runtime:

Listing 8.4:

```
./build/optical_flow
build pyramid time: 0.000150349
track pyr 3 cost time: 0.000304633
track pyr 2 cost time: 0.000392889
track pyr 1 cost time: 0.000382347
track pyr 0 cost time: 0.000375099
optical flow by gauss-newton: 0.00189268
optical flow by opencv: 0.00220134
```

In terms of runtime, the multi-layer optical flow method takes roughly the same time as OpenCV. Since the performance of the parallelized program varies from run to run, these numbers will not be exactly the same on the reader's machine. For the result of optical flow, see Figure 8-3. It can be seen that the multi-layer optical flow has the same effect as OpenCV, and the single-layer optical flow performs obviously worser than the multi-layer optical flow.

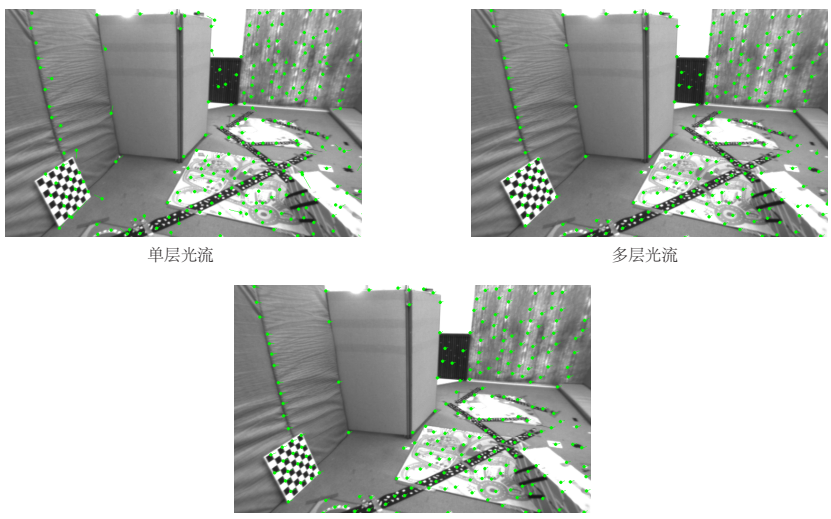


Figure 8-3: Comparison of the results of various optical flows

8.3.3 Summary of optical flow practice

We see that LK optical flow can directly obtain the corresponding relationship of feature points. This correspondence is like the matching of descriptors, except that optical flow requires higher image continuity and light stability. We can use PnP, ICP, or epipolar geometry to estimate the camera motion through the feature points

tracked by optical flow. These methods were introduced in the previous lecture and will not be discussed here.

In terms of runtime, it extracts about 230 feature points in the experiment. OpenCV and multi-layer optical flow need about 2 milliseconds to complete the tracking (the CPU I use is Intel I7-8550U), which is quite fast. If we use keypoints like FAST, then the entire optical flow calculation can be done in about 5 milliseconds, which is very fast compared to feature matching. However, if the position of the corner point is not good, the optical flow is also easy to be lost or give wrong results, which requires the subsequent algorithm to have a certain outlier removal mechanism, we leave the relevant discussion to the later chapter.

In a nutshell, the optical flow method can accelerate the visual odometry calculation method based on feature points by avoiding the process of calculating and matching descriptors, but requires smoother camera movement (or higher collection frequency).

8.4 Direct Method

Next, let's discuss the direct method, which is somehow similar to the optical flow method. We first introduce the principle of the direct method, and then implement the direct method.

8.4.1 Derivation of the direct method

In the optical flow, we will first track the location of feature points, and then determine the camera's movement based on these locations. Then, such a two-step plan is difficult to guarantee the overall optimality. We can ask, can we adjust the result of the previous step in the latter step? For example, if I think that the camera has turned 15 degrees to the right, can the optical flow use this 15-degree motion as the initial value to adjust the calculation of the optical flow? This idea is reflected in the direct method.

As shown in Figure 8-4 , consider a spatial point P and camera at two timestamps. The world coordinates of P are $[X, Y, Z]$, and the pixel coordinates of its imaging on two cameras are $\mathbf{p}_1, \mathbf{p}_2$.

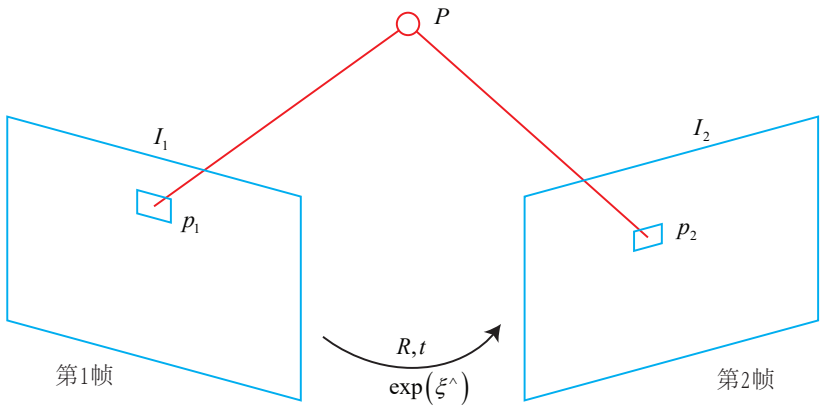


Figure 8-4: the direct method.

Our goal is to find the relative pose transformation from the first camera to the second camera. We take the first camera as the frame of reference, and set the rotation and translation of the second camera as $\mathbf{R}, \mathbf{t}$ (corresponding to the Lie group as $\mathbf{T}$). At the same time, the internal parameters of the two cameras are the same, denoted as $\mathbf{K}$. Let's write down the complete projection equation:

$$\mathbf{p}_1 = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}_1 = \frac{1}{Z_1}\mathbf{K}\mathbf{P},$$
$$\mathbf{p}_2 = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}_2 = \frac{1}{Z_2}\mathbf{K}\left(\mathbf{R}\mathbf{P}+\mathbf{t}\right) = \frac{1}{Z_2}\mathbf{K}\left(\mathbf{T}\mathbf{P}\right)_{1:3}.$$

Where Z_1 is the depth of P, and Z_2 is the depth of P in the second camera frame, which is the third coordinate of $\mathbf{R}\mathbf{P}+\mathbf{t}$. Since $\mathbf{T}$ can only be multiplied with homogeneous coordinates, we need to take out the first 3 elements after multiplying. This is consistent with the content of 5.

Recall that in the feature point method, since we know the pixel positions of $\mathbf{p}_1, \mathbf{p}_2$ through matching descriptors, we can calculate the reprojection position. But in the direct method, since there is no feature matching, we have no way of knowing which $\mathbf{p}_2$ and $\mathbf{p}_1$ correspond to the same point. The idea of the direct method is to find the position of $\mathbf{p}_2$ according to the current camera pose estimation. But if the camera pose is not good enough, the appearance of $\mathbf{p}_2$ and $\mathbf{p}_1$ will be significantly different. Therefore, in order to reduce this difference, we optimize the pose of the camera to find $\mathbf{p}_2$ that is more similar to $\mathbf{p}_1$. This can also be done by solving an optimization problem, but at this time it is not to minimize the reprojection error, but to minimize the **Photometric Error**, which is the brightness error of the two pixels of P:

$$e = \mathbf{I}_1\left(\mathbf{p}_1\right) - \mathbf{I}_2\left(\mathbf{p}_2\right). \tag{8.11}$$

Note that e is a scalar here. Similarly, the optimization is with respect to the second norm of the error, taking the unweighted form for now, as:

$$\min_{\mathbf{T}} J\left(\mathbf{T}\right) = \|e\|^2. \tag{8.12}$$

The optimization is based on **constant brightness assumption**. We assume that the grayscale of a spatial point imaged at various viewing points is constant. If we have many (for example, N) space points P_i, then the whole camera pose estimation problem becomes

$$\min_{\mathbf{T}} J\left(\mathbf{T}\right) = \sum_{i=1}^{N} e_i^{\mathrm{T}} e_i, \quad e_i = \mathbf{I}_1\left(\mathbf{p}_{1,i}\right) - \mathbf{I}_2\left(\mathbf{p}_{2,i}\right). \tag{8.13}$$

The variable to be optimized here is the camera pose $\mathbf{T}$, instead of the motion of each feature point in the optical flow. In order to solve this optimization problem, we are concerned about how the error e changes with the camera pose $\mathbf{T}$, and we need to analyze their derivative relationship. First, define two intermediate variables:

$$\mathbf{q} = \mathbf{T}\mathbf{P},$$
$$\mathbf{u} = \frac{1}{Z_2}\mathbf{K}\mathbf{q}.$$

Here, $\mathbf{q}$ is the coordinates of P in the second camera coordinate system, and $\mathbf{u}$ is its pixel coordinates. Obviously $\mathbf{q}$ is a function of $\mathbf{T}$, and $\mathbf{u}$ is a function of $\mathbf{q}$, and thus is also a function of $\mathbf{T}$. Consider the left perturbation model of Lie algebra, using the first-order Taylor expansion:

$$e(\mathbf{T}) = \mathbf{I}_1(\mathbf{p}_1) - \mathbf{I}_2(\mathbf{u}), \tag{8.14}$$

$$\frac{\partial e}{\partial \mathbf{T}} = \frac{\partial \mathbf{I}_2}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \delta \xi} \delta \xi, \tag{8.15}$$

Where $\delta\xi$ is the left disturbance of $\mathbf{T}$. We see that the first derivative is divided into 3 terms due to the chain rule, and these 3 terms are easy to obtain:

1. $\partial \mathbf{I}_2 / \partial \mathbf{u}$ is the grayscale gradient at pixel $\mathbf{u}$.
2. $\partial \mathbf{u} / \partial \mathbf{q}$ is the derivative of the projection equation with respect to the three-dimensional point in the camera frame. Remember $\mathbf{q} = [X, Y, Z]^{\mathrm{T}}$, according to 7, the derivative is

$$\frac{\partial \mathbf{u}}{\partial \mathbf{q}} = \begin{bmatrix} \frac{\partial u}{\partial X} & \frac{\partial u}{\partial Y} & \frac{\partial u}{\partial Z} \\ \frac{\partial v}{\partial X} & \frac{\partial v}{\partial Y} & \frac{\partial v}{\partial Z} \end{bmatrix} = \begin{bmatrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} \end{bmatrix}. \tag{8.16}$$

3. $\partial \mathbf{q} / \partial \delta \xi$is the derivative of the transformed three-dimensional point with respect to the transformation, which was introduced in the chapter of Lie Algebra:

$$\frac{\partial \mathbf{q}}{\partial \delta \xi} = [\mathbf{I}, -\mathbf{q}^{\wedge}]. \tag{8.17}$$

In practice, the last two items are only related to the three-dimensional point $\mathbf{q}$ irrelevant to the image, we often combine them together:

$$\frac{\partial \mathbf{u}}{\partial \delta \xi} = \begin{bmatrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} & -\frac{f_x XY}{Z^2} & f_x + \frac{f_x X^2}{Z^2} & -\frac{f_x Y}{Z} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} & -f_y - \frac{f_y Y^2}{Z^2} & \frac{f_y XY}{Z^2} & \frac{f_y X}{Z} \end{bmatrix}. \tag{8.18}$$

This 2×6 matrix also appeared in the last chapter. Therefore, we derive the Jacobian of residual with respect to Lie algebra:

$$\mathbf{J} = -\frac{\partial \mathbf{I}_2}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \delta \xi}. \tag{8.19}$$

For the problem of N points, we can use this method to calculate the Jacobian of the optimization problem, and then use the Gauss Newton method or Levenberg-Marquardt method to calculate the increments and iteratively solve it. So far, we have introduced the entire process of the direct method to estimate the camera pose. Let's implement the direct method in a program.

8.4.2 Discussion of Direct Method

In the above derivation, P is a spatial point with a known location. How did it come from? Under the RGB-D camera, we can reproject any pixel into the three-dimensional space, and then project it into the next image. If it is binocular, the pixel depth can also be calculated based on the parallax. If in a monocular camera,

this matter is more difficult, because we must also consider the uncertainty caused by the depth of P. Depth estimation will be elaborated in Chapter 13. Now let's consider the simple case first, i.e. when the depth of P is known.

According to the source of P, we can classify the direct method:

1. P comes from the sparse keypoint, which we call the sparse direct method. Usually we use hundreds to thousands of keypoints, and like L-K optical flow, it is assumed that the surrounding pixels are also unchanged. This sparse direct method does not need to calculate descriptors and only uses hundreds of pixels, so it is the fastest, but it can only calculate sparse reconstruction.

2. P comes from some pixels. We see that in the formula (8.19), if the pixel gradient is zero, the entire Jacobian is zero, which will not contribute to the calculation of the motion increment. Therefore, you can consider only using pixels with gradients and discarding areas where the pixel gradients are not obvious. This is called a semi-dense direct method, which can reconstruct a semi-dense structure.

3. P is all pixels, which is called the dense direct method. Dense reconstruction needs to calculate all pixels (generally hundreds of thousands to several million), so most of them cannot be calculated in real time on the existing CPU and require GPU acceleration. However, as discussed above, the points with inconspicuous pixel gradients will not contribute much in motion estimation, and it will be difficult to estimate the position during reconstruction.

It can be seen that the reconstruction from sparse to dense can be calculated by the direct method. Their computational complexity are gradually increasing. The sparse method can quickly solve the camera pose, while the dense method can build a complete map. Which method to use depends on the objective of the application. In particular, on simple computing platforms, the sparse direct method can achieve very fast results, and is suitable for occasions with high real-time performance and limited computing resources.

[66]

8.5 Practice: Direct method

8.5.1 Single-layer direct method

Now, let's demonstrate how to use the sparse direct method. Since this book does not involve GPU programming, the dense direct method is omitted. Meanwhile, in order to keep the program simple, we use depth data instead of monocular data, so that the monocular depth recovery part can be omitted. The depth recovery based on feature points (i.e. triangulation) has been introduced in the previous chapter, and the depth recovery based on block matching will be introduced later. In this section we will consider the sparse direct method of binocular.

Since solving the direct method is finally equivalent to solving an optimization problem, you can use optimization libraries such as g2o or Ceres to help solve it, or you can implement the Gauss-Newton method yourself. Similar to optical flow, the direct method can also be divided into a single-layer direct method and a pyramid-like multilayer direct method. We also first implement the single-layer direct method, and then extend to multiple layers.

In the single-layer direct method, similar to the parallel optical flow, we can also calculate the error and Jacobian of each pixel in parallel. For this reason, we define a class for calculating Jacobian:

Listing 8.5: slambook2/ch8/direct_method.cpp

```
/// class for accumulator jacobians in parallel
class JacobianAccumulator {
public:
  JacobianAccumulator(
    const cv::Mat &img1_,
    const cv::Mat &img2_,
    const VecVector2d &px_ref_,
    const vector<double> depth_ref_,
    Sophus::SE3d &T21_) :
  img1(img1_), img2(img2_), px_ref(px_ref_), depth_ref(depth_ref_), T21(T21_) {
    projection = VecVector2d(px_ref.size(), Eigen::Vector2d(0, 0));
  }

  /// accumulate jacobians in a range
  void accumulate_jacobian(const cv::Range &range);

  /// get hessian matrix
  Matrix6d hessian() const { return H; }

  /// get bias
  Vector6d bias() const { return b; }

  /// get total cost
  double cost_func() const { return cost; }

  /// get projected points
  VecVector2d projected_points() const { return projection; }

  /// reset h, b, cost to zero
  void reset() {
    H = Matrix6d::Zero();
    b = Vector6d::Zero();
    cost = 0;
  }

private:
  const cv::Mat &img1;
  const cv::Mat &img2;
  const VecVector2d &px_ref;
  const vector<double> depth_ref;
  Sophus::SE3d &T21;
  VecVector2d projection; // projected points

  std::mutex hessian_mutex;
  Matrix6d H = Matrix6d::Zero();
  Vector6d b = Vector6d::Zero();
  double cost = 0;
};

void JacobianAccumulator::accumulate_jacobian(const cv::Range &range) {

  // parameters
  const int half_patch_size = 1;
  int cnt_good = 0;
  Matrix6d hessian = Matrix6d::Zero();
  Vector6d bias = Vector6d::Zero();
  double cost_tmp = 0;

  for (size_t i = range.start; i < range.end; i++) {
    // compute the projection in the second image
    Eigen::Vector3d point_ref =
    depth_ref[i] * Eigen::Vector3d((px_ref[i][0] - cx) / fx, (px_ref[i][1] - cy) / fy,
          1);
    Eigen::Vector3d point_cur = T21 * point_ref;
    if (point_cur[2] < 0)   // depth invalid
      continue;

    float u = fx * point_cur[0] / point_cur[2] + cx, v = fy * point_cur[1] / point_cur
```

```
        [2] + cy;
    if (u < half_patch_size || u > img2.cols - half_patch_size || v < half_patch_size ||
    v > img2.rows - half_patch_size)
      continue;

    projection[i] = Eigen::Vector2d(u, v);
    double X = point_cur[0], Y = point_cur[1], Z = point_cur[2],
    Z2 = Z * Z, Z_inv = 1.0 / Z, Z2_inv = Z_inv * Z_inv;
    cnt_good++;

    // and compute error and jacobian
    for (int x = -half_patch_size; x <= half_patch_size; x++)
    for (int y = -half_patch_size; y <= half_patch_size; y++) {
      double error = GetPixelValue(img1, px_ref[i][0] + x, px_ref[i][1] + y) -
        GetPixelValue(img2, u + x, v + y);
      Matrix26d J_pixel_xi;
      Eigen::Vector2d J_img_pixel;

      J_pixel_xi(0, 0) = fx * Z_inv;
      J_pixel_xi(0, 1) = 0;
      J_pixel_xi(0, 2) = -fx * X * Z2_inv;
      J_pixel_xi(0, 3) = -fx * X * Y * Z2_inv;
      J_pixel_xi(0, 4) = fx + fx * X * X * Z2_inv;
      J_pixel_xi(0, 5) = -fx * Y * Z_inv;

      J_pixel_xi(1, 0) = 0;
      J_pixel_xi(1, 1) = fy * Z_inv;
      J_pixel_xi(1, 2) = -fy * Y * Z2_inv;
      J_pixel_xi(1, 3) = -fy - fy * Y * Y * Z2_inv;
      J_pixel_xi(1, 4) = fy * X * Y * Z2_inv;
      J_pixel_xi(1, 5) = fy * X * Z_inv;

      J_img_pixel = Eigen::Vector2d(
        0.5 * (GetPixelValue(img2, u + 1 + x, v + y) - GetPixelValue(img2, u - 1 + x, v + y)),
        0.5 * (GetPixelValue(img2, u + x, v + 1 + y) - GetPixelValue(img2, u + x, v - 1 + y))
      );

      // total jacobian
      Vector6d J = -1.0 * (J_img_pixel.transpose() * J_pixel_xi).transpose();
      hessian += J * J.transpose();
      bias += -error * J;
      cost_tmp += error * error;
    }
  }

  if (cnt_good) {
    // set hessian, bias and cost
    unique_lock<mutex> lck(hessian_mutex);
    H += hessian;
    b += bias;
    cost += cost_tmp / cnt_good;
  }
}
```

In the accumulate_jacobian function of this class, we calculate the pixel residual and Jacobian according to the previous derivation for the pixels in the specified range, and finally add it to the overall **H** matrix. Then, define a function to iterate this process:

Listing 8.6: slambook2/ch8/direct_method.cpp (snippet)

```
void DirectPoseEstimationSingleLayer(
  const cv::Mat &img1,
  const cv::Mat &img2,
  const VecVector2d &px_ref,
  const vector<double> depth_ref,
  Sophus::SE3d &T21) {
  const int iterations = 10;
  double cost = 0, lastCost = 0;
  JacobianAccumulator jaco_accu(img1, img2, px_ref, depth_ref, T21);
```

```
10
11    for (int iter = 0; iter < iterations; iter++) {
12      jaco_accu.reset();
13      cv::parallel_for_(cv::Range(0, px_ref.size()),
14        std::bind(&JacobianAccumulator::accumulate_jacobian, &jaco_accu, std::
              placeholders::_1));
15      Matrix6d H = jaco_accu.hessian();
16      Vector6d b = jaco_accu.bias();
17
18      // solve update and put it into estimation
19      Vector6d update = H.ldlt().solve(b);;
20      T21 = Sophus::SE3d::exp(update) * T21;
21      cost = jaco_accu.cost_func();
22
23      if (std::isnan(update[0])) {
24        // sometimes occurred when we have a black or white patch and H is irreversible
25        cout << "update is nan" << endl;
26        break;
27      }
28      if (iter > 0 && cost > lastCost) {
29        cout << "cost increased: " << cost << ", " << lastCost << endl;
30        break;
31      }
32      if (update.norm() < 1e-3) {
33        // converge
34        break;
35      }
36
37      lastCost = cost;
38      cout << "iteration: " << iter << ", cost: " << cost << endl;
39    }
40  }
```

This function calculates the corresponding pose updates according to the calculated **H** and **b**, and then updates it to the current estimated value. We have introduced the details clearly in the theoretical part, this part of the code does not seem difficult.

8.5.2 Multi-layer direct method

Then, similar to optical flow, we extend the direct method to the pyramid and use the coarse-to-fine process to calculate relative transformation. This part of the code is also similar to optical flow:

Listing 8.7: slambook2/ch8/direct_method.cpp (snippet)

```
1  void DirectPoseEstimationMultiLayer(
2    const cv::Mat &img1,
3    const cv::Mat &img2,
4    const VecVector2d &px_ref,
5    const vector<double> depth_ref,
6    Sophus::SE3d &T21) {
7    // parameters
8    int pyramids = 4;
9    double pyramid_scale = 0.5;
10   double scales[] = {1.0, 0.5, 0.25, 0.125};
11
12   // create pyramids
13   vector<cv::Mat> pyr1, pyr2; // image pyramids
14   for (int i = 0; i < pyramids; i++) {
15     if (i == 0) {
16       pyr1.push_back(img1);
17       pyr2.push_back(img2);
18     } else {
19       cv::Mat img1_pyr, img2_pyr;
20       cv::resize(pyr1[i - 1], img1_pyr,
21         cv::Size(pyr1[i - 1].cols * pyramid_scale, pyr1[i - 1].rows * pyramid_scale));
22       cv::resize(pyr2[i - 1], img2_pyr,
23         cv::Size(pyr2[i - 1].cols * pyramid_scale, pyr2[i - 1].rows * pyramid_scale));
24       pyr1.push_back(img1_pyr);
25       pyr2.push_back(img2_pyr);
```

```
    }
  }

  double fxG = fx, fyG = fy, cxG = cx, cyG = cy;  // backup the old values
  for (int level = pyramids - 1; level >= 0; level--) {
    VecVector2d px_ref_pyr; // set the keypoints in this pyramid level
    for (auto &px: px_ref) {
      px_ref_pyr.push_back(scales[level] * px);
    }

    // scale fx, fy, cx, cy in different pyramid levels
    fx = fxG * scales[level];
    fy = fyG * scales[level];
    cx = cxG * scales[level];
    cy = cyG * scales[level];
    DirectPoseEstimationSingleLayer(pyr1[level], pyr2[level], px_ref_pyr, depth_ref,
        T21);
  }
}
```

It should be noted that, because the direct method of Jacobian takes the camera's intrinsic parameters, and when the pyramid scales the image, the corresponding internal parameters also need to be multiplied by the corresponding ratio.

8.5.3 Results discussion

Finally, we use some sample pictures to test the results of the direct method. We use several images of the Kitti[70] autonomous driving dataset. First, we read the first image left.png, in the corresponding disparity map disparity.png, calculate the depth corresponding to each pixel, and then use the direct method to calculate the camera poses for the five images 000001.png-000005.png. In order to show the insensitivity of the direct method to the feature points, we randomly select some points in the first image without using any corner points or feature point extraction algorithms.

Listing 8.8: slambook2/ch8/direct_method.cpp (snippet)

```
int main(int argc, char **argv) {

  cv::Mat left_img = cv::imread(left_file, 0);
  cv::Mat disparity_img = cv::imread(disparity_file, 0);

  // let's randomly pick pixels in the first image and generate some 3d points in the
      first image's frame
  cv::RNG rng;
  int nPoints = 2000;
  int boarder = 20;
  VecVector2d pixels_ref;
  vector<double> depth_ref;

  // generate pixels in ref and load depth data
  for (int i = 0; i < nPoints; i++) {
    int x = rng.uniform(boarder, left_img.cols - boarder);  // don't pick pixels close
        to boarder
    int y = rng.uniform(boarder, left_img.rows - boarder);  // don't pick pixels close
        to boarder
    int disparity = disparity_img.at<uchar>(y, x);
    double depth = fx * baseline / disparity; // you know this is disparity to depth
    depth_ref.push_back(depth);
    pixels_ref.push_back(Eigen::Vector2d(x, y));
  }

  // estimates 01~05.png's pose using this information
  Sophus::SE3d T_cur_ref;

  for (int i = 1; i < 6; i++) {  // 1~10
    cv::Mat img = cv::imread((fmt_others % i).str(), 0);
    DirectPoseEstimationMultiLayer(left_img, img, pixels_ref, depth_ref, T_cur_ref);
  }
```

```
  return 0;
}
```

Readers can run this program on your machine, it will output the tracking points on each level of the pyramid of each image, and output the running time. The result of the multi-layer direct method is shown in Figure 8-5. According to the output of the program, you can see that the fifth image is about when the camera moves 3.8 meters forward. It can be seen that even if we randomly select points, the direct method can correctly track most of the pixels and estimate the camera motion. It does not include any feature extraction, matching, or optical flow. In terms of running time, at 2000 points, it takes 1-2 milliseconds for each layer of the direct method to iterate, so the four-layer pyramid takes about 8 milliseconds. In contrast, the optical flow of 2000 points takes about ten milliseconds, excluding the subsequent pose estimation. Therefore, the direct method is usually faster than the traditional feature points and optical flow.

Figure 8-5: Experimental results of the direct method. upper left: original image; upper right: disparity map corresponding to the original image; lower left: fifth tracking image; lower right: tracking result

Below we briefly explain the iterative process of the direct method. Compared with the feature point method, the direct method completely relies on the optimization to solve the camera pose. It can be seen from the formula (8.19) that the pixel gradient guides the direction of optimization. If you want to get the correct optimization results, you must ensure that **most pixel gradients can guide the optimization in the right direction**.

What does it mean? Assume that for the reference image, we measured a pixel with a gray value of 229. And, since we know its depth, we can infer the position of the space point P (Figure 8-6 shown as the grayscale measured in I_1).

Now, we have got a new image and need to estimate its camera pose. This pose is obtained by continuous optimization iterations of an initial value. Assuming that our initial value is relatively poor, under this initial value, the pixel gray value after the projection of the space point P is 126. Therefore, the error of this pixel is $229 - 126 = 103$. In order to reduce this error, we hope to **fine-tune the camera's pose to make the pixels brighter**.

How do I know where to fine-tune the pixels to make them brighter? This requires the use of local pixel gradients. We found in the image that if we take a step forward along the u axis, the gray value at that point becomes 123, that is, 3 is subtracted.

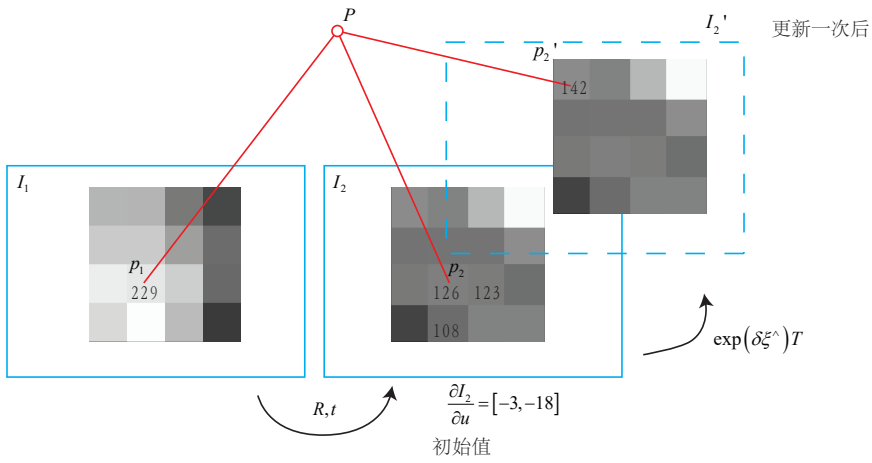


Figure 8-6: workflow of one iteration.

Similarly, if you take a step forward along the v axis, the gray value is reduced by 18 and becomes 108. Around this pixel, we see that the gradient is $[-3, -18]$. In order to increase the brightness, we will suggest optimizing the algorithm to fine-tune the camera so that the image of P moves to **top left**. In this process, we use the local gradient of the pixel to approximate the grayscale distribution near it, but please note that the real image is not smooth, so this gradient is not valid at a distance.

However, the optimization can't just follow the behavior of just one pixel, but also need to get track of other pixels. After considering many pixels, the optimization algorithm chose a place not far from the direction we suggested, and calculated an update amount $\exp(\xi^\wedge)$. After adding the update amount, the image has moved from I_2 to I_2', and the projection position of the pixel has also changed to a brighter place. We see that with this update, **error has become smaller**. Under ideal circumstances, we expect the error to continue to decrease and eventually converge.

But is this actually the case? Do we really only need to walk along the gradient direction to reach an optimal value? Note that the gradient of the direct method is directly determined by the image gradient, so we must ensure that **when walking along the image gradient, the photometric error will continue to decrease**. However, the image is usually a very strong **non-convex function**, as shown in Figure 8-7 . In practice, if we move along the image gradient, it is easy to fall into a local minimum due to the non-convexity (or noise) of the image itself, and we cannot continue to optimize. The direct method can only be established when the camera movement is very small and the gradient in the image will not have strong non-convexity.

In the example, we only calculated the difference of a single pixel, and this difference is obtained by directly subtracting the grayscale. However, a single pixel is not distinguishable, and there are probably many pixels around with similar brightness. Therefore, we sometimes use small patches and use more complex difference measures, such as Normalized Cross Correlation (NCC). For the sake of simplicity, the example uses the sum of squares of errors to maintain consistency with the derivation.

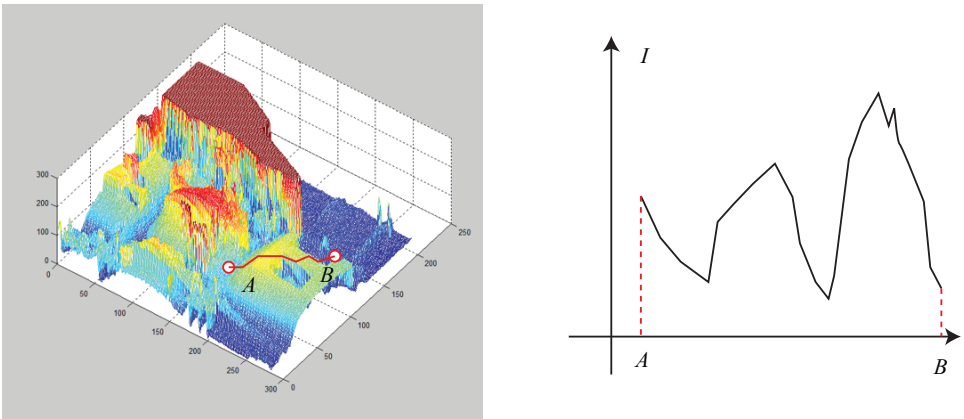


Figure 8-7: Three-dimensional visualization of an image. The path from one point in the image to another point is not necessarily a "straight downhill road", but needs to be "climbing over the mountains" frequently. This reflects the non-convexity of the image itself.

8.5.4 Advantages and disadvantages of the direct method

Finally, we summarize the advantages and disadvantages of the direct method. In general, its advantages are as follows:

- It can save the time of calculating feature points and descriptors.
- Only pixel gradients are required, no feature points are required. Therefore, the direct method can be used in the absence of features. An extreme example is an image with only gradients. It may not be able to extract corner features, but its motion can be estimated by a direct method. In the demonstration experiment, we see that the direct method can also work normally for randomly selected points. This is very important in practice, because practical scenes may not have many corner points to use.
- It is possible to construct semi-dense or even dense maps, which cannot be achieved by the feature point method.

On the other hand, its shortcomings are also obvious:

- **Non-convexity**. The direct method completely relies on gradient search and reduces the objective function to calculate the camera pose. The objective function needs to take the gray value of the pixel, and the image is a strongly non-convex function. This makes the optimization algorithm easy to be stuck at a local minimum, and the direct method can only succeed when the movement is small. Against this, the pyramids can reduce the impact of non-convexity to a certain extent.
- **Single pixel has no discriminativeness**. Many points look alike. So we either calculate image patches or calculate complex correlations. Since each pixel has inconsistent "opinions" about changing the camera movement, only a few obey the majority, and increasing the quantity for better quality. Therefore, the performance of the direct method decreases significantly when there

are fewer selected points. We usually recommend using more than 500 points.
Brightness constant is a strong assumption. If the camera is automatically exposed, when it adjusts the exposure parameters, it will make the overall image brighter or darker. This situation also occurs when the light changes. The feature point method has a certain tolerance to illumination, while the direct method calculates the difference of brightness, and the overall brightness change will destroy the brightness constant assumption and make the algorithm fail. In response to this, the practical direct method will also estimate the camera's exposure parameters [66] so that it can still work when the exposure time changes.

1. In addition to LK optical flow, do you know other optical flow methods? What are their characteristics?
2. In the program to calculate the image gradient, we simply calculate the difference between the brighteness of $u+1$ and $u-1$ divided by 2 as the gradient in the direction of u. What are the disadvantages of this approach? Hint: For features closer together, the changes should be faster; while for features farther away it changes more slowly in the image, can this information be used when calculating the gradient?
3. Can the direct method be implemented in an "inverse" way like optical flow? That is, use the gradient of the original image instead of the gradient of the target image?
4. * Use Ceres or g2o to implement sparse direct method and semi-dense direct method.
5. Compared with the direct method of RGB-D, the monocular direct method is often more complicated. In addition to the unknown matching, the pixel distance is also to be estimated, and we need to use the pixel depth as an optimization variable during optimization. Refer to the literature [65, 71], can you understand its principle?

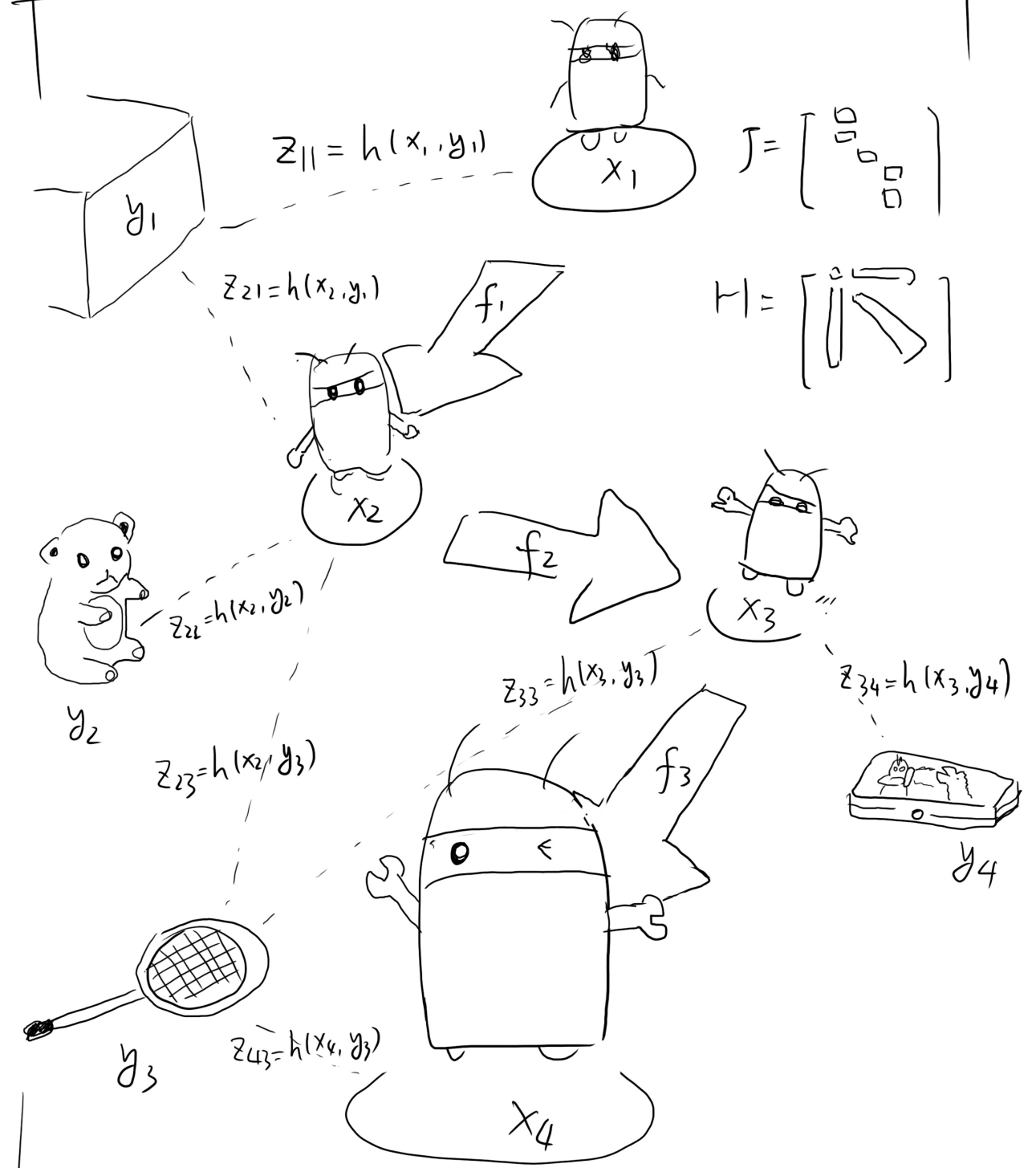
Chapter 9

Backend: Part I

Goal of Study

1. Learn how to formulate the backend problem into a filter or least square optimization problem.
2. Learn how to use the sparse structure in bundle adjustment problem.
3. Solve a BA problem with g2o and Ceres.

From this lecture, we turn to another important module: back-end optimization. We see that the front-end visual odometry can give a short-term trajectory and map. Still, due to the inevitable accumulation of errors, this map is inaccurate for a long time. Therefore, based on visual odometry, we also hope to construct a larger-scale optimization problem to consider the optimal trajectory and map over a long time. However, considering the balance of accuracy and performance, there are many different approaches in practice.


$z_{11} = h(x_1, y_1)$
y_1
x_1
$f =$
$z_{21} = h(x_2, y_1)$
f_1
$H =$
x_2
f_2
x_3
$z_{22} = h(x_2, y_2)$
y_2
$z_{33} = h(x_3, y_3)$
$z_{34} = h(x_3, y_4)$
$z_{23} = h(x_2, y_3)$
f_3
y_4
y_3
$z_{43} = h(x_4, y_3)$
x_4

9.1 Introduction

9.1.1 State Estimation from Probabilistic Perspective

As mentioned in the second lecture, the visual odometry only has a short memory, but we hope that the system can maintain the entire motion trajectory in an optimal state for a long time. We may use the latest knowledge to update an old state. At that time, it seems that future information tells you, "where you should be now." Therefore, in the back-end optimization, we usually consider the problem of state estimation for a longer period of time (or all-time), and not only use the past information to update our current state but also use future information to update ourselves. Such a method might be called "Batch." Otherwise, if the current state is only determined by the past, or even only by the previous moment, it might also be called "Incremental."

We already know that the SLAM process can be described by the motion and observation equations. Suppose in the time from $t = 0$ to $t = N$, we have the poses from $\boldsymbol{x}_0$ to $\boldsymbol{x}_N$ and observation $\boldsymbol{y}_1, \cdots, \boldsymbol{y}_M$. According to the equations in chapter 2, we write them as:

$$\begin{cases} \boldsymbol{x}_k = f\left(\boldsymbol{x}_{k-1}, \boldsymbol{u}_k\right) + \boldsymbol{w}_k \\ \boldsymbol{z}_{k,j} = h\left(\boldsymbol{y}_j, \boldsymbol{x}_k\right) + \boldsymbol{v}_{k,j} \end{cases} \quad k = 1, \ldots, N,\ j = 1, \ldots, M. \tag{9.1}$$

Note that in the SLAM problem we have the following characteristics:

1. In the observation equation, only when $\boldsymbol{x}_k$ sees $\boldsymbol{y}_j$, we will have a real observation equation. In fact, we can usually see only a small part of the landmarks in one location. Moreover, due to a large number of visual SLAM feature points, the number of observation equations in practice will be much larger than that of motion equations.

2. We may not have a device to measure motion, so there may not be a motion equation. In this case, there are several ways to deal with it:

   - Assume that there is really no motion equation.
   - Assume that the camera does not move.
   - Assume that the camera is moving at a constant speed.

   These several methods are all feasible. In the absence of motion equations, the entire optimization problem consists of only observation equations. This is very similar to the SfM (Structure from Motion) problem, which is equivalent to restoring the motion and structure through a set of images. The difference with SfM is that the images in SLAM have a chronological order, while SfM allows the use of completely unrelated images.

We know that every measurement is affected by noise, so the poses $\boldsymbol{x}$ and landmarks $\boldsymbol{y}$ here are regarded as **random variables that obey a certain probability distribution** instead of a single number. Therefore, the question becomes: when I have some motion data $\boldsymbol{u}$ and observation data $\boldsymbol{z}$, how to determine the state $\boldsymbol{x}$ and landmarks $\boldsymbol{y}$'s distribution? Furthermore, if new data is obtained, how to update our estimation? In more common and reasonable cases, we assume that the state quantity and noise terms obey Gaussian distribution-which means that only their mean and covariance matrix need to be stored in the program. The mean can be

regarded as an estimate of the state variable's optimal value, and the covariance matrix measures its uncertainty. Then, the question becomes: when there are some motion and observation data, how do we estimate the Gaussian distribution of the states?

We still put ourselves in the role of a robot. When there is only the equation of motion, it is equivalent to walking blindfolded in an unknown place. Although we know how far we have taken for each step, we will become more and more uncertain about where we are as time grows. This reflects that when the input data is affected by noise, the error is gradually accumulated, and our estimate of the position variance will become larger and larger. However, when we open our eyes, we will become more and more confident because we can continuously observe the external scene, making the uncertainty of position estimation smaller. If we use an ellipse to intuitively express the covariance matrix, then this process is a bit like walking in a mobile phone map software. Taking Figure 9-1 as an example, readers can imagine that when there is no observation data, the circle will become larger and larger with the movement; and if there are correct observations, the circle will shrink to a certain size and keep stable.

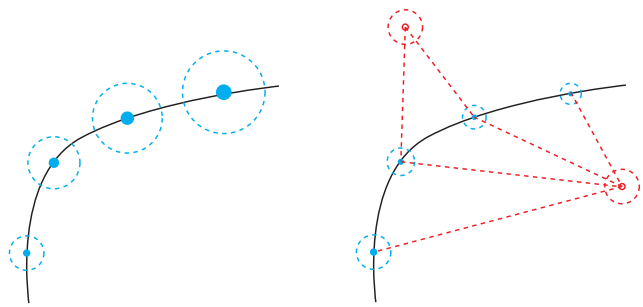


Figure 9-1: An intuitive description of uncertainty. Left: When there is only the motion equation, the pose at the next moment adds noise to the previous moment, so the uncertainty is getting bigger and bigger. Right: When there are road signs, the uncertainty will be significantly reduced. Please note that this is only an intuitive diagram, not actual data.

The above statements explain the problem of state estimation in a metaphorical form. Below we will look at it in a quantitative way. In Lecture 6, we introduced the maximum likelihood estimation where we say that the problem of **batch state estimation** can be transformed into a **maximum likelihood estimation problem and solved by the least square method**. In this section, we will explore how to apply this conclusion to progressive problems and get some classic conclusions. At the same time, we will investigate the special structure of the least square method in visual SLAM.

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

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

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

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

[16] 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.

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

[18] 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.

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

[20] 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.

[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.

[24] H. Hirschmuller, "Stereo processing by semiglobal matching and mutual information," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 30, no. 2, pp. 328–341, 2008.

[25] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International journal of computer vision*, vol. 47, no. 1-3, pp. 7–42, 2002.

[26] S. M. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski, "A comparison and evaluation of multi-view stereo reconstruction algorithms," in *null*, pp. 519–528, IEEE, 2006.

[27] S. Agarwal, N. Snavely, I. Simon, S. M. Seitz, and R. Szeliski, "Building rome in a day," in *2009 IEEE 12th international conference on computer vision*, pp. 72–79, IEEE, 2009.

[28] P. Wolfe, "Convergence conditions for ascent methods," *SIAM review*, vol. 11, no. 2, pp. 226–235, 1969.

[29] J. Nocedal and S. Wright, *Numerical Optimization*. Springer Science & Business Media, 2006.

[30] M. I. Lourakis and A. A. Argyros, "Sba: A software package for generic sparse bundle adjustment," *ACM Transactions on Mathematical Software (TOMS)*, vol. 36, no. 1, p. 2, 2009.

[31] G. Sibley, "Relative bundle adjustment," *Department of Engineering Science, Oxford University, Tech. Rep*, vol. 2307, no. 09, 2009.

[32] S. Agarwal, K. Mierle, and Others, "Ceres solver." `http://ceres-solver.org`.

[33] R. Kummerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "G2o: a general framework for graph optimization," in *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3607–3613, IEEE, 2011.

[34] Wikipedia, "Feature (computer vision)." `"https://en.wikipedia.org/wiki/Feature_(computer_vision)"`, 2016. [Online; accessed 09-July-2016].

[35] C. Harris and M. Stephens, "A combined corner and edge detector.," in *Alvey vision conference*, vol. 15, pp. 10–5244, Citeseer, 1988.

[36] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Computer Vision–ECCV 2006*, pp. 430–443, Springer, 2006.

[37] J. Shi and C. Tomasi, "Good features to track," in *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pp. 593–600, IEEE, 1994.

[38] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[39] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer Vision–ECCV 2006*, pp. 404–417, Springer, 2006.

[40] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: an efficient alternative to sift or surf," in *2011 IEEE International Conference on Computer Vision (ICCV)*, pp. 2564–2571, IEEE, 2011.

[41] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *European conference on computer vision*, pp. 778–792, Springer, 2010.

[42] M. Nixon and A. S. Aguado, *Feature extraction and image processing for computer vision*. Academic Press, 2012.

[43] P. L. Rosin, "Measuring corner properties," *Computer Vision and Image Understanding*, vol. 73, no. 2, pp. 291–307, 1999.

[44] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration.," in *VISAPP (1)*, pp. 331–340, 2009.

[45] R. I. Hartley, "In defense of the eight-point algorithm," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 19, no. 6, pp. 580–593, 1997.

[46] H. C. Longuet-Higgins, "A computer algorithm for reconstructing a scene from two projections," *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms, MA Fischler and O. Firschein, eds*, pp. 61–62, 1987.

[47] H. Li and R. Hartley, "Five-point motion estimation made easy," in *18th International Conference on Pattern Recognition (ICPR'06)*, vol. 1, pp. 630–633, IEEE, 2006.

[48] D. Nistér, "An efficient solution to the five-point relative pose problem," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 6, pp. 756–770, 2004.

[49] O. D. Faugeras and F. Lustman, "Motion and structure from motion in a piecewise planar environment," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 2, no. 03, pp. 485–508, 1988.

[50] Z. Zhang and A. R. Hanson, "3d reconstruction based on homography mapping," *ARPA Image Understanding Workshop*, pp. 1007–1012, 1996.

[51] E. Malis and M. Vargas, *Deeper understanding of the homography decomposition for vision-based control*. PhD thesis, INRIA, 2007.

[52] A. J. Davison, "Real-time simultaneous localisation and mapping with a single camera," in *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pp. 1403–1410, IEEE, 2003.

[53] X.-S. Gao, X.-R. Hou, J. Tang, and H.-F. Cheng, "Complete solution classification for the perspective-three-point problem," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, pp. 930–943, Aug 2003.

[54] A. Penate-Sanchez, J. Andrade-Cetto, and F. Moreno-Noguer, "Exhaustive linearization for robust camera pose and focal length estimation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 10, pp. 2387–2400, 2013.

[55] L. Chen, C. W. Armstrong, and D. D. Raftopoulos, "An investigation on the accuracy of three-dimensional space reconstruction using the direct linear transformation technique," *Journal of Biomechanics*, vol. 27, no. 4, pp. 493–500, 1994.

[56] B. F. Green, "The orthogonal approximation of an oblique structure in factor analysis," *Psychometrika*, vol. 17, no. 4, pp. 429–440, 1952.

[57] iplimage, "P3p(blog)." `"http://iplimage.com/blog/p3p-perspective-point-overview/"`, 2016.

[58] K. S. Arun, T. S. Huang, and S. D. Blostein, "Least-squares fitting of two 3-d point sets," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 5, pp. 698–700, 1987.

[59] F. Pomerleau, F. Colas, and R. Siegwart, "A review of point cloud registration algorithms for mobile robotics," *Foundations and Trends in Robotics (FnTROB)*, vol. 4, no. 1, pp. 1–104, 2015.

[60] O. D. Faugeras and M. Hebert, "The representation, recognition, and locating of 3-d objects," *The International Journal of Robotics Research*, vol. 5, no. 3, pp. 27–52, 1986.

[61] B. K. Horn, "Closed-form solution of absolute orientation using unit quaternions," *JOSA A*, vol. 4, no. 4, pp. 629–642, 1987.

[62] G. C. Sharp, S. W. Lee, and D. K. Wehe, "Icp registration using invariant features," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 1, pp. 90–102, 2002.

[63] G. Silveira, E. Malis, and P. Rives, "An efficient direct approach to visual slam," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 969–979, 2008.

[64] C. Forster, M. Pizzoli, and D. Scaramuzza, "Svo: Fast semi-direct monocular visual odometry," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on* (rs, ed.), pp. 15–22, IEEE, 2014.

[65] J. Engel, T. Schöps, and D. Cremers, "Lsd-slam: Large-scale direct monocular slam," in *Computer Vision–ECCV 2014*, pp. 834–849, Springer, 2014.

[66] J. Engel, V. Koltun, and D. Cremers, "Direct sparse odometry," *arXiv preprint arXiv:1607.02565*, 2016.

[67] B. D. Lucas, T. Kanade, *et al.*, "An iterative image registration technique with an application to stereo vision," 1981.

[68] B. K. Horn and B. G. Schunck, "Determining optical flow," *Artificial intelligence*, vol. 17, no. 1-3, pp. 185–203, 1981.

[69] S. Baker and I. Matthews, "Lucas-kanade 20 years on: A unifying framework," *International journal of computer vision*, vol. 56, no. 3, pp. 221–255, 2004.

[70] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *The International Journal of Robotics Research*, 2013.

[71] J. Engel, J. Sturm, and D. Cremers, "Semi-dense visual odometry for a monocular camera," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1449–1456, 2013.