

14 Lectures on Visual SLAM

From Theory to Practice

Xiang Gao, Tao Zhang, Qinrui Yan and Yi Liu

November 27, 2019

Contents

Preface for English Version

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

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

Xiang

Chapter 1

Preface

1.1 What is this book about?

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

So, what is SLAM?

SLAM stands for **S**imultaneous **L**ocalization and **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[?]. If the sensor referred here is mainly a camera, it is called "**V**isual **S**LAM".

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

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

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

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

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

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

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

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

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

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

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

Well, enough talking, wish you a pleasant journey!

1.2 How to use this book?

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

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

1. Lecture 1: preface (the one you are reading now), introducing the contents and structure of the book.
2. Lecture 2: an overview of a SLAM system. It describes each module of a SLAM system and explains what they do and how they do it. The practice

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

3. Lecture 3: rigid body motion in 3D space. You will learn knowledge about rotation matrices, quaternions, Euler angles, and practice them with the Eigen library.
4. Lecture 4: Lie group and Lie algebra. It doesn't matter if you have never heard of them. You will learn the basics of Lie group, and manipulate them with Sophus.
5. Lecture 5: pinhole camera model and image expression in computer. You will use OpenCV to retrieve camera's intrinsic and extrinsic parameters, and then generate a point cloud using the depth information through PCL (Point Cloud Library).
6. Lecture 6: nonlinear optimization, including state estimation, least squares and gradient descent methods, e.g. Gauss-Newton and Levenburg-Marquardt. You will solve a curve fitting problem using the Ceres and g2o library.

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

7. Lecture 7: feature based visual odometry, which is currently the mainstream in VO. Contents include feature extraction and matching, epipolar geometry calculation, Perspective-n-Point (PnP) algorithm, Iterative Closest Point (ICP) algorithm, and Bundle Adjustment (BA), etc. You will run these algorithms either by calling OpenCV functions or by constructing 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 a same BA problem.
10. Lecture 10: pose graph in the back-end optimization. Pose graph is a more compact representation for BA which marginalizes all map points into constraints between keyframes. You will use g2o and gtsam to optimize a pose graph.
11. Lecture 11: loop closure detection, mainly Bag-of-Word (BoW) based method. You will use dbow3 to train a dictionary from images and detect loops in videos.
12. Lecture 12: map building. We will discuss how to estimate the depth of feature points in monocular SLAM (and show why they are unreliable). Compared with monocular depth estimation, building a dense map with RGB-D cameras is much easier. You will write programs for epipolar line search and patch matching to estimate depth from monocular images, and then build a point cloud map and octagonal tree map from RGB-D data.

13. Lecture 13: a practice chapter for VO. You will build a visual odometer framework by yourself by integrating the previously learned knowledge, and solve problems such as frame and map point management, key frame selection and optimization control.
14. Lecture 14: current open source SLAM projects and future development direction. We believe that after reading the previous chapters, you will be able to understand other people's approaches easily, and be capable to achieve new ideas of your own.

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

1.3 Source code

All source code in this book is hosted on github:

<https://github.com/gaoxiang12/slambook2>

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

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

1.4 Oriented readers

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

- Calculus, Linear Algebra, Probability Theory. These are the fundamental mathematical knowledge that most readers should have learned during undergraduate study. You should at least understand what a matrix and a vector are, and what it means by doing differentiation and integration. For more advanced mathematical knowledge required, we will introduce in this book as we proceed.
- Basic C++ Programming. As we will be using C++ as our major programming language, it is recommended that the readers are at least familiar with its basic concepts and syntax. For example, you should know what a class is, how to use the C++ standard library, how to use template classes, etc. We will try our best to avoid using tricks, but in certain situations we really can not avert. In addition, we will adopt some of C++ 11 standard, but don't worry, they will be explained as they appear.

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

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

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

1.5 Style

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

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

$$\mathbf{y} = \mathbf{Ax}. \quad (1.1)$$

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

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

```

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

```

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

```

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

3. When the code block is too long or contains repeated parts with previously listed code, it is not appropriate to be listed entirely. We will only give **important snippets** and mark it with "Snippet". Therefore, we strongly recommend that readers download all the source code on GitHub and complete the exercises to better understand the book.
4. Due to typographical reasons, the code shown in the book may be slightly different from the code hosted on GitHub. In that case please use the code on GitHub.
5. For each of the libraries we use, it will be explained in details when first appearing, but not repeated in the follow-up. Therefore, it is recommended that readers read this book in order.
6. An abstract will be presented at the beginning of each lecture. A summary and some exercises will be given at the end. The cited references are listed at the end of the book.
7. The chapters with an asterisk mark in front are optional readings, and readers can read them according to their interest. Skipping them will not hinder the understanding of subsequent chapters.
8. Important contents will be marked in **bold** or *italic*, as we are already accustomed to.
9. Most of the experiments we designed are demonstrative. Understanding them does not mean that you are already familiar with the entire library. So we recommend that you spend time on yourselves in further exploring the important libraries frequently used in the book.
10. The book's exercises and optional readings may require you to search for additional materials, so you need to learn to use search engines.

1.6 Acknowledgments

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

1.7 Exercises (self-test questions)

1. There is a linear equation $\mathbf{Ax} = \mathbf{b}$, if \mathbf{A} and \mathbf{b} are known, how to solve for \mathbf{x} ? What are the requirements for \mathbf{A} and \mathbf{b} if we want an unique \mathbf{x} ? (Hint: check the rank of \mathbf{A} and \mathbf{b}).
2. What is a Gaussian distribution? What does it look like in one-dimensional case? How about in high-dimensional case?

3. Do you know what a **class** is in C++? Do you know STL? Have you ever used them?
4. How do you write a C++ program? (It's completely fine if your answer is "using Visual C++ 6.0" * . As long as you have C++ or C programming experience, you are in good hand).
5. Do you know the C++11 standard? Which new features have you heard of or used? Are you familiar with any other standard?
6. Do you know Linux? Have you used at least one flavor (not including Android), such as Ubuntu?
7. What is the directory structure of Linux? What basic commands do you know? (e.g. ls, cat, etc.)
8. How to install a software in Ubuntu (without using the Software Center)? What directories are software usually installed under? If you only know the fuzzy name of a software (for example, you want to install a library with a word "eigen" in its name), how would you do it?
9. *Spend an hour learning Vim, you will be using it sooner or later. You can type "vimtutor" into 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, do not try to turn Vim into an IDE for now, we will only use it for text editing in this book.

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

Chapter 2

First Glance of Visual SLAM

Goal of Study

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

2.1 Introduction

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

2.2 Meet "Little Carrot"

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

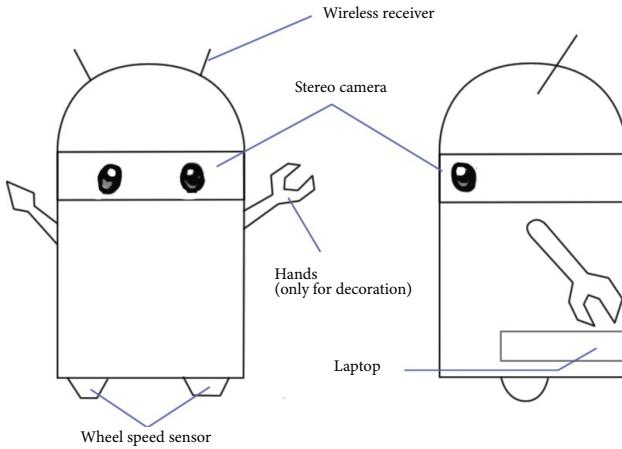


Figure 2-1: The sketch of robot *Little Carrot*

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

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

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

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

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

Localization and *map building*, can be seen as the perception in both inward and outward directions. As a completely autonomous robot, Little Carrot need not only to understand its own *state* (i.e. the location), but also the external *environment* (i.e. the map). Of course, there are many different approaches to solve these two problems. For example, we can lay guiding rails on the floor of the room, or 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. ??) into two classes.

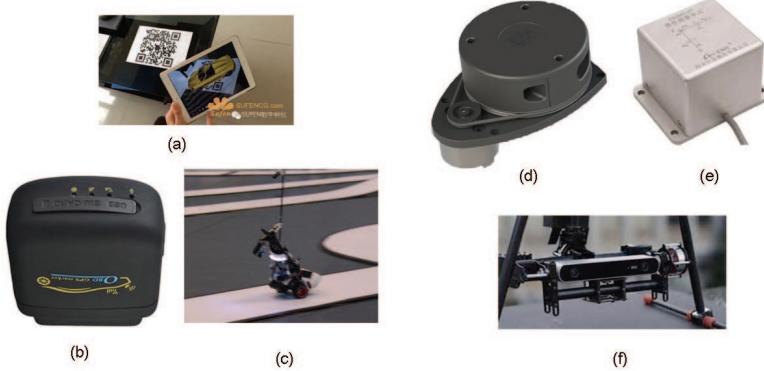


Figure 2-2: Different kinds of sensors: (a) QR code (b) GNSS receiver (c) guiding rails (d) Laser range finder (e) Inertial measurement unit (f) stereo camera

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

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

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

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

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



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

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

Monocular Camera

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

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



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

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

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

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

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

Stereo Camera and RGB-D Camera

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

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

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



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

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

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



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

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

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

2.3 The Classic Visual SLAM Framework

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

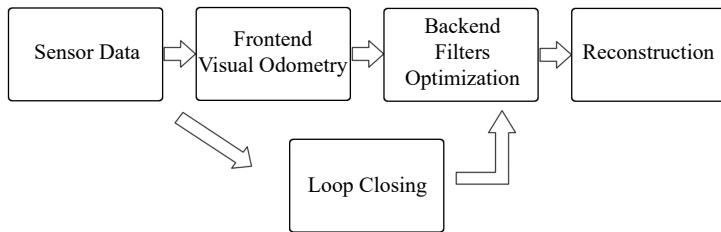


Figure 2-7: The classic visual SLAM framework.

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

1. Sensor data acquisition. In visual SLAM, this mainly refers to for acquisition and preprocessing for camera images. For a mobile robot, this will also include the acquisition and synchronization with motor encoders, IMU sensors, etc.
2. Visual Odometry (VO). The task of VO is to estimate the camera movement between adjacent frames (ego-motion), as well as to generate a rough local map. VO is also known as the *Front End*.
3. Backend filtering/optimization. The back end receives camera poses at different time stamps from VO, as well as results from loop closing, and apply optimization to generate a fully optimized trajectory and map. Because it is connected after the VO, it is also known as the *Back End*.
4. Loop Closing. Loop closing determines whether the robot has returned to its previous position in order to reduce the accumulated drift. If a loop is detected, it will provide information to the back end for further optimization.
5. Reconstruction. It constructs a task specific map based on the estimated camera trajectory.

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

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

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

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

Visual Odometry

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



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

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

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

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

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

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

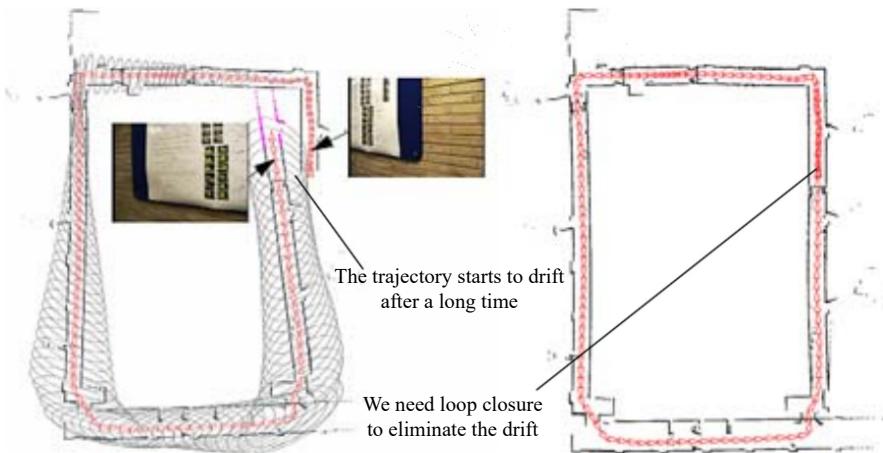


Figure 2-9: Drift will be accumulated if we only have a relative motion estimation.

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

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

Back-end Optimization

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

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

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

Loop Closing

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

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

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

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

Mapping

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

Let's take the domestic cleaning robots as an example. Since they basically move on the ground, a two-dimensional map with marks for open areas and obstacles, built by a single line laser scanner, would be sufficient for navigation for them. And for a camera, we need at least a three-dimensional map for its 6 degrees of freedom movement. Sometimes, we want a smooth and beautiful reconstruction result, not just a set of points, but also with texture of triangular faces. And at other times, we do not care about the map, just need to know things like “point A and point B are connected, while point B and point C are not”, which is a topological way to understand the 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 categories: *metrical map* and *topological map*.

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

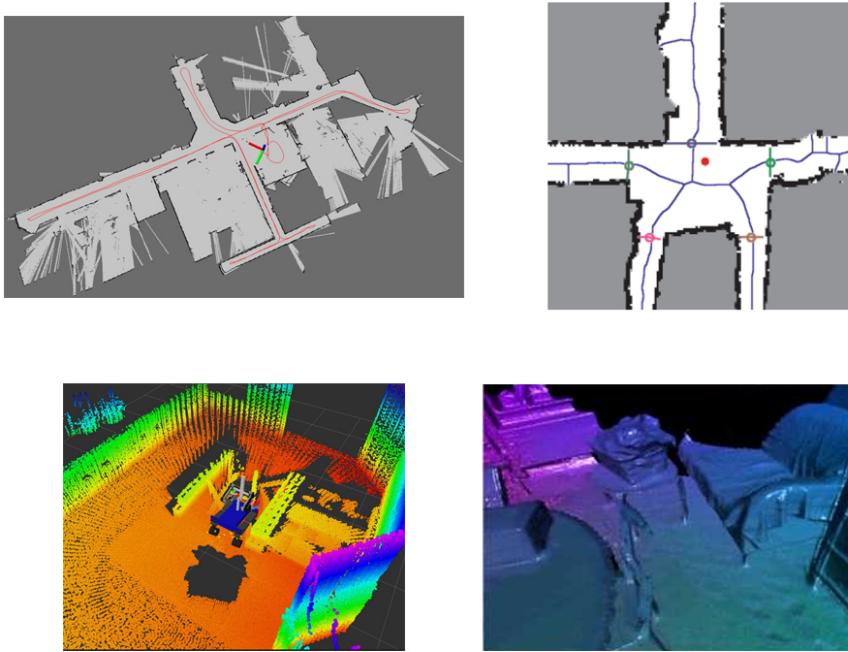


Figure 2-10: Different kinds of maps: 2D grid map, 2D topological map, 3D point clouds and 3D meshes.

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

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

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

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

2.4 Mathematical Formulation of SLAM Problems

Through the previous introduction, readers should have gained an intuitive understanding of the modules in a SLAM system and the main functionality of each module. However, we cannot write 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 some time points, we are only concerned with the locations and map at these moments. This turns a continuous process into discrete time steps, say $1, \dots, k$, 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 $\mathbf{x}_1, \dots, \mathbf{x}_k$, which constitute the trajectory of Little Carrot. In terms of the map, we assume that the map is made up of a number of *landmarks*, and at each time step, the sensors can see a part of the landmarks and record their observations. Assume there are total N landmarks in the map, and we will use $\mathbf{y}_1, \dots, \mathbf{y}_N$ to denote them.

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

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

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

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

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

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

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

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

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

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 $\mathbf{x}_k = [x_1, x_2, \theta]_k^T$, 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: $\mathbf{u}_k = [\Delta x_1, \Delta x_2, \Delta \theta]_k^T$, so the motion equation can be parameterized as:

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

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

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

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

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

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

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

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

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

The solution to the state estimation problem is related to the specific form of the two equations and which distribution the noise obeys. According to whether the motion and observation equations are **linear**, and whether the noise is assumed to be **Gaussian**, it is divided into **linear/nonlinear** and **Gaussian/non-Gaussian** systems. The Linear Gaussian (LG system) is the simplest, and its unbiased optimal estimation can be given by the Kalman Filter (KF). In the complex nonlinear non-Gaussian (NLNG system), we basically rely on two methods: Extended Kalman Filter (EKF) and nonlinear optimization. Until the early 21st century, the EKF-based filter method still dominated SLAM. We will linearize the system at the working point and solve it in two steps: predict-update (see Lecture 10). The earliest real-time visual SLAM system was developed based on EKF [?]. 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 [?]. 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 \mathbf{x}** is. The word **pose** we just said is somewhat vague. Perhaps the reader can understand that a robot moving in the plane can be parameterized by two coordinates plus an angle. However, more robots are more like things in the three-dimensional space. We know that the motion of the three-dimensional space consists of three axes, so the movement of the robot is described by the translation on the three axes and the rotation around the three axes, totally having six Degrees of Freedom (DoF). But wait, does that mean that we can describe it with a vector in \mathbb{R}^6 ? We will find that things are not that simple. For a 6 DoF **pose***, how to express it? How to optimize it? How to describe its mathematical properties? This will be the main content of the third and fourth chapters. Next, we will explain how the **observation equation** is parameterized in the visual SLAM. In other words, how the landmark points in space are projected onto a photo? This requires an explanation of the camera's projection model and distortion, which we will cover in chapter 5. Finally, when you know this information, **how to solve the above state estimation problem?** This requires knowledge of nonlinear optimization and is the content of Lecture 6.

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

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

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

2.5 Practice: Basics

2.5.1 Installing Linux

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

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

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

The first version of this book uses Ubuntu 14.04 as the default development environment. In the second edition, we updated the default version to the newer **Ubuntu 18.04** (??) 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 ??), 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 (binocular cameras, Kinects, etc.), it is recommended that you use dual systems to install Linux.

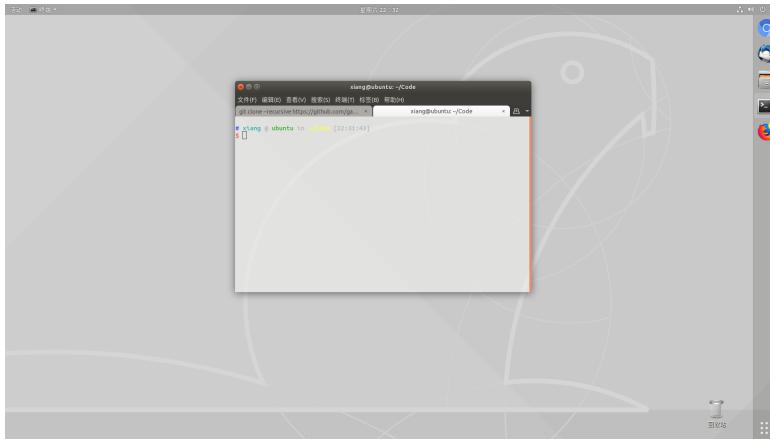


Figure 2-11: Ubuntu 1804 in virtual machine.

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

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

2.5.2 Hello SLAM

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

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

Listing 2.1: slambook2/ch2/helloSLAM.cpp

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

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

```

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

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

Listing 2.2: Terminal input:

```
1 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:

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

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

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

Listing 2.4: terminal input:

```
1 % ./a.out
2 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

```
1 cmake_minimum_required( VERSION 2.8 )
2 project( HelloSLAM )
3 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

```
1 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

```
1 % make
2 Scanning dependencies of target helloSLAM
3 [100%] Building CXX object CMakeFiles/helloSLAM.dir/helloSLAM.cpp.o
4 Linking CXX executable helloSLAM
5 [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:

* 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.

Listing 2.8: Terminal Input

```

1 % ./helloSLAM
2 Hello SLAM!

```

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

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

Listing 2.9: Terminal input

```

1 mkdir build
2 cd build
3 cmake ..
4 make

```

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

2.5.4 Use Libraries

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

A library is often just a collection of many algorithms and programs, and we will be exposed to many libraries in later exercises. For example, the OpenCV library provides many computer vision related algorithms, while the Eigen library provides calculations of matrix algebra. Therefore, we need to learn how to use cmake to generate libraries and use the functions in the library. Now let's demonstrate how to write a library yourself. Write the following libHelloSLAM.cpp file:

Listing 2.10: slambook2/ch2/libHelloSLAM.cpp

```

1 #include <iostream>

```

```

2 using namespace std;
3
4 // Just a function printing hello message
5 void printHello() {
6     cout << "Hello SLAM" << endl;
7 }
```

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

Listing 2.11: `slambook2/ch2/CMakeLists.txt`

```

1 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

```

1 cd build
2 cmake ..
3 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`

```

1 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`

```

1 #ifndef LIBHELLOSLAM_H_
2 #define LIBHELLOSLAM_H_
3
4 // Declares a function in header file
5 void printHello();
6
7 #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

```

1 #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 ?? and ??)*. KDevelop is a free software located in Ubuntu's software repository, meaning you can install it with apt-get; Clion is a paid software, but you can use the student mailbox for free for one year. Both are good C++ development environments, the advantages are listed below:

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

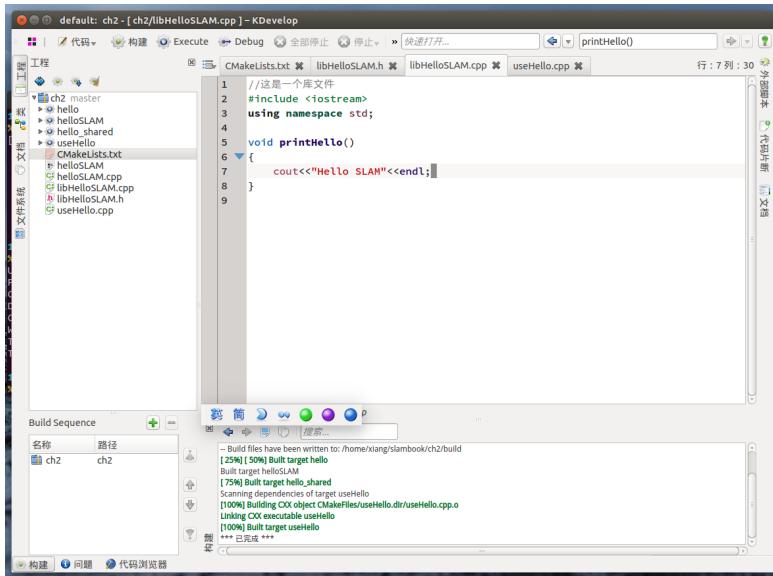


Figure 2-12: Kdevelop in Ubuntu.

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

Use KDE

Kdevelop natively supports the cmake project. To do this, after creating CMakeLists.txt in the terminal, open CMakeLists.txt with “Project → Open/Import Project” in KDevelop. The software will ask you a few questions, and by default create a build folder to help you call the cmake and make commands. These can be done automatically by pressing the shortcut key F8. The following section of ?? 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

* 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.

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 ??, 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 ??.

When debugging, KDevelop will switch to debug mode and the interface will change a bit. At the breakpoint, you can control the operation of the program with single step operation (F10 key), single step follow up (F11 key), and single step jump (F12 key) function. At the same time, you can click the interface on the left to view

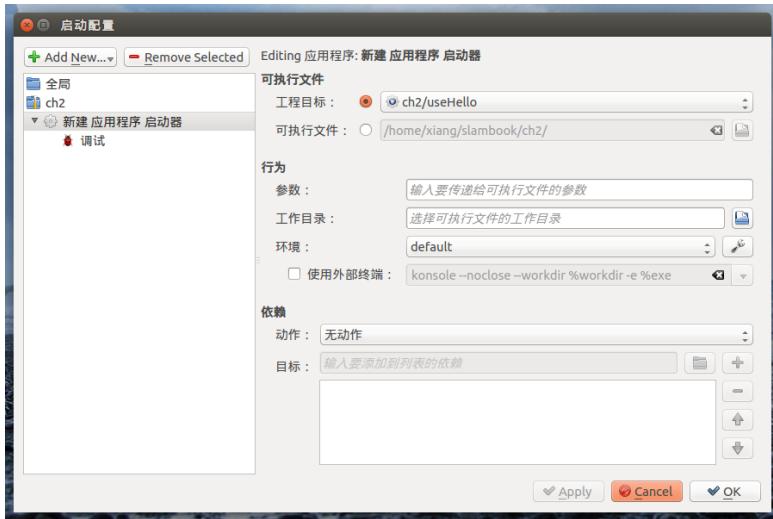


Figure 2-13: Config launches. We can choose a launch target and set parameters here.

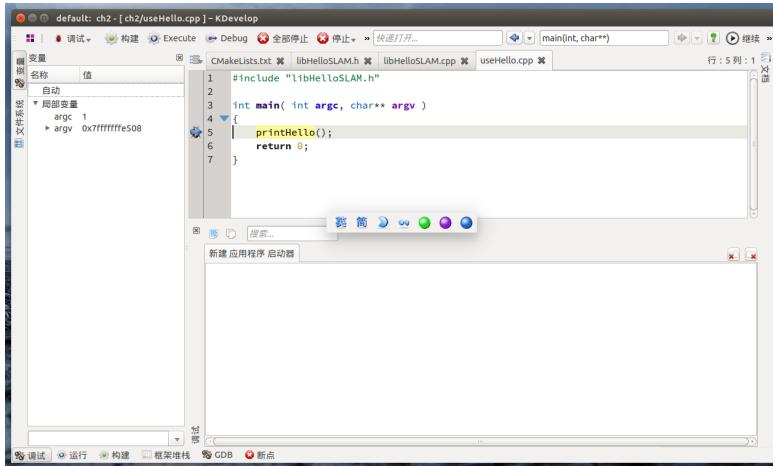


Figure 2-14: Debug interface.

the value of the local variable. Or select the "Stop" button to end debugging. After debugging, KDevelop will return to the normal development interface.

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

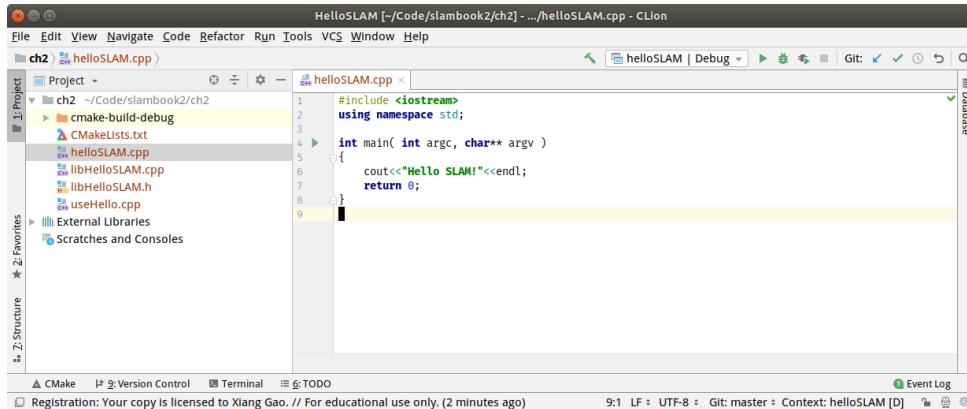


Figure 2-15: Clion interface

Use Clion

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

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

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

Exercises

1. Read the survey SLAM literature [?] and [?], 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 [? ? ? ? ?] and so on. What are the similarities and differences between these papers on SLAM and this book?
3. What are the parameters of the

* 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.

4. g++ command? If I want to change the generated program file name, how should I call g++?
5. Use the build folder to compile your cmake project, then try it in KDevelop.
6. Deliberately add some syntax errors to the code to see what information the build will generate. Can you read the error message of g++?
7. If you forgot to link the library to the executable, will the compiler report an error? What kind of mistakes are reported?
- 8.*Read “cmake practice” (or other cmake materials) to learn about the grammars of cmake.
- 9.*Improve the hello SLAM problem, make it a small library, and install it on your local hard drive. Then, create a new project, use find_package to find the library and call it.
- 10.*Read other cmake instructional materials, such as <https://github.com/TheErk/CMake-tutorial>.
11. Find the official website of KDevelop and see what other features it has. Are you using it?
12. If you learned Vim in the last lecture, please try KDevelop’s/Clion’s Vim editing function.

Chapter 3

3D Rigid Body Motion

Goal of Study

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

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

3.1 Rotation Matrix

3.1.1 Point, Vector and Coordinate System

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

We start with the most basic content: **points** and **vectors**. Points are the basic elements in space, no length, no volume. Connecting the two points forms a vector.

A vector can be thought of as an arrow pointing from one point to another. We need to remind the reader that, please do not confuse the vector with its **coordinates**. A vector is one of the things in space, such as \mathbf{a} . Here \mathbf{a} does not need to be associated with several real numbers. Only when we specify a **coordinate system** in this three-dimensional space can we talk about the coordinates of the vector in this coordinate system, that is, find several real numbers corresponding to this vector.

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

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

Here $(a_1, a_2, a_3)^T$ is called \mathbf{a} 's coordinates[†]. The specific values of the coordinates are related to the vector itself, and also the selection of the bases. In \mathbb{R}^3 , the coordinate system usually consists of 3 orthogonal coordinate axes (although it can also be non-orthogonal, it is rare in practice). For example, given \mathbf{x} and \mathbf{y} axis, the \mathbf{z} axis can be found using the right-hand (or left-hand) rule by $\mathbf{x} \times \mathbf{y}$. According to different definitions, the coordinate system is divided into left-handed and right-handed. The third axis of the left hand system is opposite to the right hand system. Most 3D libraries use right-handed (such as OpenGL, 3D Max, etc.), and some libraries use left-handed (such as Unity, Direct3D, etc.).

Based on basic linear algebra knowledge, we can talk about the operations between vectors/vectors, and vectors/numbers, such as scalar multiplication, vector addition, subtraction, inner product, outer product, and so on. Multiplication, addition and subtraction are fairly basic and intuitive. For example, the result of adding two vectors is to add their respective coordinates, subtraction, and so on. I won't go into details here. Internal and external products may be somewhat unfamiliar to the reader, and their calculations are given here. For $\mathbf{a}, \mathbf{b} \in \mathbb{R}^3$, in the usual sense[‡], the inner product of \mathbf{a}, \mathbf{b} can be written as:

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

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:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \mathbf{b} \triangleq \mathbf{a} \wedge \mathbf{b}. \quad (3.3)$$

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 orthogonal and has unit-length.

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

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

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

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

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

3.1.2 Euclidean Transforms

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

Intuitively, the motion between two coordinate systems consists of a rotation plus a translation, which is called **rigid body motion**. Obviously, the camera movement is a rigid body one. During the rigid body motion, the length and angle of the a vector will not change. Imagine you throw your phone into the air and *, there may only be differences in spatial position and posture, and its own length, angle of each face, etc. will not change. The phone will not be squashed like an eraser or be stretched during this motion. At this point, we say that the phone's motion is a **Euclidean Transform**.

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

[§] Skew-symmetric matrix means \mathbf{A} satisfies $\mathbf{A}^T = -\mathbf{A}$.

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

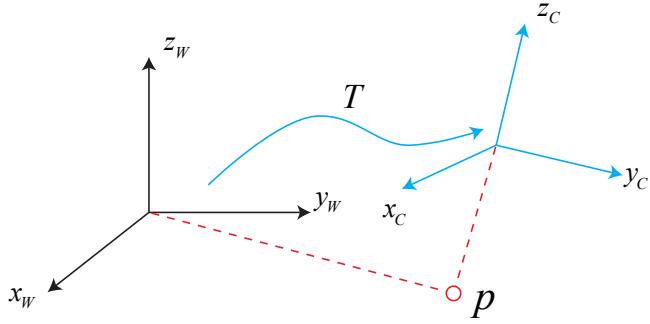


Figure 3-1: Coordinate transformation. For the same vector \mathbf{p} , its coordinates in the world \mathbf{p}_W and coordinates in the camera system \mathbf{p}_C is different. This transformation relationship is described by the transform matrix \mathbf{T} .

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

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

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

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:

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

$\text{SO}(n)$ 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:

$$\mathbf{a}' = \mathbf{R}^{-1}\mathbf{a} = \mathbf{R}^T\mathbf{a}. \quad (3.8)$$

Obviously \mathbf{R}^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:

$$\mathbf{a}' = \mathbf{R}\mathbf{a} + \mathbf{t}. \quad (3.9)$$

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

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

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

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

3.1.3 Transform Matrix and Homogeneous Coordinates

The formula (??) 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 (??):

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

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:

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

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:

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

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

$$\mathbf{T}^{-1} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}. \quad (3.14)$$

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

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

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

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

Let's take a review now. First, we introduce the vector and its coordinate representation, and introduce the operation between the vectors; then, the motion between the coordinate systems is described by the Euclidean transformation, which consists of translation and rotation. The rotation can be described by the rotation matrix $\text{SO}(3)$, 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 $\text{SE}(3)$ 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:

```
1 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:

```
1 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

```
1 #include <iostream>
2 using namespace std;
3
4 #include <ctime>
5 // Eigen core
```

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

```

6 #include <Eigen/Core>
7 // Algebraic operations of dense matrices (inverse, eigenvalues, etc.)
8 #include <Eigen/Dense>
9 using namespace Eigen;
10
11 #define MATRIX_SIZE 50
12
13 /*****
14 * This program demonstrates the use of the basic Eigen type
15 *****/
16
17 int main(int argc, char **argv) {
18     // All vectors and matrices in Eigen are Eigen::Matrix, which is a
19     // template
20     // class. Its first three parameters are: data type, row, column
21     // Declare a 2*3
22     // float matrix
23     Matrix<float, 2, 3> matrix_23;
24
25     // At the same time, Eigen provides many built-in types via typedef,
26     // but the
27     // bottom layer is still Eigen::Matrix. For example, Vector3d is
28     // essentially
29     // Eigen::Matrix<double, 3, 1>, which is a three-dimensional vector.
30     Vector3d v_3d;
31     // This is the same
32     Matrix<float, 3, 1> vd_3d;
33
34     // Matrix3d is essentially Eigen::Matrix<double, 3, 3>
35     Matrix3d matrix_33 = Matrix3d::Zero(); // initialized to zero
36     // If you are not sure about the size of the matrix, you can use a
37     // matrix of
38     // dynamic size
39     Matrix<double, Dynamic, Dynamic> matrix_dynamic;
40     // simpler
41     MatrixXd matrix_x;
42     // There are still many types of this, we doesn't list them one by one.
43
44     // Here is the operation of the Eigen array
45     // input data (initialization)
46     matrix_23 << 1, 2, 3, 4, 5, 6;
47     // output
48     cout << "matrix 2x3 from 1 to 6: \n" << matrix_23 << endl;
49
50     // Use () to access elements in the matrix
51     cout << "print matrix 2x3: " << endl;
52     for (int i = 0; i < 2; i++) {
53         for (int j = 0; j < 3; j++)
54             cout << matrix_23(i, j) << "\t";
55         cout << endl;
56     }
57
58     // The matrix and vector are multiplied (actually still matrices and
59     // matrices)
60     v_3d << 3, 2, 1;
61     vd_3d << 4, 5, 6;
62
63     // But in Eigen you can't mix two different types of matrices, like
64     // this is
65     // wrong Matrix<double, 2, 1> result_wrong_type = matrix_23 * v_3d;
66     // should be
67     // explicitly converted
68     Matrix<double, 2, 1> result = matrix_23.cast<double>() * v_3d;

```

```

61 cout << "[1,2,3;4,5,6]*[3,2,1]" << result.transpose() << endl;
62
63 Matrix<float, 2, 1> result2 = matrix_23 * vd_3d;
64 cout << "[1,2,3;4,5,6]*[4,5,6]" << result2.transpose() << endl;
65
66 // Also you can't misjudge the dimensions of the matrix
67 // Try canceling the comments below to see what Eigen will report.
68 // Eigen::Matrix<double, 2, 3> result_wrong_dimension =
69 // matrix_23.cast<double>() * v_3d;
70
71 // some matrix operations
72 // Four operations are not demonstrated, just use +-*.
73 Matrix_33 = Matrix3d::Random(); // Random Number Matrix
74 cout << "random matrix: \n" << matrix_33 << endl;
75 cout << "transpose: \n" << matrix_33.transpose() << endl;
76 cout << "sum: " << matrix_33.sum() << endl;
77 cout << "trace: " << matrix_33.trace() << endl;
78 cout << "times 10: \n" << 10 * matrix_33 << endl;
79 cout << "inverse: \n" << matrix_33.inverse() << endl;
80 cout << "det: " << matrix_33.determinant() << endl;
81
82 // Eigenvalues
83 // Real symmetric matrix can guarantee successful diagonalization
84 SelfAdjointEigenSolver<Matrix3d> eigen_solver(matrix_33.transpose() *
85 matrix_33);
86 cout << "Eigen values = \n" << eigen_solver.eigenvalues() << endl;
87 cout << "Eigen vectors = \n" << eigen_solver.eigenvectors() << endl;
88
89 // Solving equations
90 // We solve the equation of matrix_NN * x = v_Nd
91 // The size of N is defined in the previous macro, which is generated
92 // by a
93 // random number Direct inversion is the most direct, but the amount of
94 // inverse operations is large.
95
96 Matrix<double, MATRIX_SIZE, MATRIX_SIZE> matrix_NN =
97 MatrixXd::Random(MATRIX_SIZE, MATRIX_SIZE);
98 matrix_NN =
99 matrix_NN * matrix_NN.transpose(); // Guarantee semi-positive definite
100 Matrix<double, MATRIX_SIZE, 1> v_Nd = MatrixXd::Random(MATRIX_SIZE, 1);
101
102 Clock_t time_stt = clock(); // timing
103 // Direct inversion
104 Matrix<double, MATRIX_SIZE, 1> x = matrix_NN.inverse() * v_Nd;
105 cout << "time of normal inverse is "
106 << 1000 * (clock() - time_stt) / (double)CLOCKS_PER_SEC << "ms" << endl
107 ;
108 cout << "x = " << x.transpose() << endl;
109
110 // Usually solved by matrix decomposition, such as QR decomposition,
111 // the speed
112 // will be much faster
113 time_stt = clock();
114 x = matrix_NN.colPivHouseholderQr().solve(v_Nd);
115 cout << "time of QR decomposition is "
116 << 1000 * (clock() - time_stt) / (double)CLOCKS_PER_SEC << "ms" << endl
117 ;
118 cout << "x = " << x.transpose() << endl;
119
120 // For positive definite matrices, you can also use cholesky
121 // decomposition to
122 // solve equations.
123 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     ;
123     cout << "x = " << x.transpose() << endl;
124
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 4 5 6
5 print matrix 2x3:
6 1 2 3
7 4 5 6
8 [1,2,3;4,5,6]*[3,2,1]=10 28
9 [1,2,3;4,5,6]*[4,5,6]: 32 77
10 random matrix:
11 0.680375  0.59688 -0.329554
12 -0.211234  0.823295  0.536459
13 0.566198 -0.604897 -0.444451
14 transpose:
15 0.680375 -0.211234  0.566198
16 0.59688  0.823295 -0.604897
17 -0.329554  0.536459 -0.444451
18 sum: 1.61307
19 trace: 1.05922
20 times 10:
21 6.80375   5.9688 -3.29554
22 -2.11234   8.23295  5.36459
23 5.66198 -6.04897 -4.44451
24 inverse:
25 -0.198521  2.22739   2.8357
26 1.00605 -0.555135 -1.41603
27 -1.62213   3.59308   3.28973
28 it: 0.208598
```

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

1. Please enter the above code by yourself if you are a beginner in C++ (not including comments). At least compile and run the above program for once.
2. Kdevelop may not prompt C++ member operations, which is caused by its incompleteness. Please follow the above to enter, do not care if it prompts an error. Clion will give you a complete hint.
3. The matrix provided by Eigen is very similar to MATLAB, and almost all data is treated as a matrix. However, in order to achieve better efficiency, you need to specify the size and type of the matrix in Eigen. For matrices that know the size at compile time, they are processed faster than dynamically changing matrices. Therefore, data such as rotation matrices and transformation matrices can be determined at compile times by their size and data type.
4. The matrix implementation inside Eigen is more complicated. I won't introduce it here. We hope that you can use Eigen's matrix like the built-in data types like float and double. This should be in line with the original intention of its design.
5. The Eigen matrix does not support automatic type promotion, which is quite different from C++'s built-in data types. In a C++ program, we can add and multiply a float variable and double variable, and **the compiler will automatically cast the data type to the most appropriate one**. In Eigen, for performance reasons, you must **explicitly** convert the matrix type. And if you forget to do this, Eigen will (not very friendly) prompt you with a very long "YOU MIXED DIFFERENT NUMERIC TYPES ..." compilation error. You can try to find out which part of the error message this message appears in. If the error message is too long, it is best to save it to a file and find it.
6. Is the same, in the calculation process also need to ensure the correctness of the matrix dimension, otherwise there will be "YOU MIXED MATRICES OF DIFFERENT SIZES" error. Please don't complain about this kind of error prompting. For C++ template meta-programming, it is very lucky to be able to prompt the information that can be read. Later, if you find some compilation error about Eigen, you can directly look for the uppercase part and figure out what the problem is.
7. Our routines only cover basic matrix operations. You can read more about Eigen by reading the Eigen official website tutorial:
<http://eigen.tuxfamily.org/dox-devel/modules.html> . Only the simplest part is demonstrated here. It is not equal to the fact that you can understand Eigen.

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

3.3 Rotation Vector and Euler Angle

3.3.1 Rotation Vector

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

1. SO(3) has a rotation matrix of 9 quantities, but a 3D rotation only has 3 degrees of freedom. Therefore the matrix expression is redundant. Similarly, the transformation matrix expresses a 6 degree-of-freedom transformation with 16 quantities. So, is there a more compact representation?
2. The rotation matrix itself has constraints: it must be an orthogonal matrix with a determinant of 1. The same is true for the transformation matrix. These constraints make the solution more difficult when you want to estimate or optimize a rotation matrix/transform matrix.

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

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

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

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

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

therefore:

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

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

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

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

$$\mathbf{R}\mathbf{n} = \mathbf{n}. \quad (3.18)$$

Therefore, the axis \mathbf{n} is the eigen vector corresponding to the matrix \mathbf{R} 's eigen-value 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 $\text{SO}(3)$.

3.3.2 Euler Angle

Let's talk about the Euler angle.

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

This uncertainty in the axis orders brings many practical difficulties. Fortunately, in certain research areas, Euler angles usually have a uniform definition. You may have heard the words “pitch angle” and “yaw angle” of an aircraft. One of the most commonly used Euler angles is the yaw-pitch-roll angles. Since it is equivalent to the rotation of the ZYX axis, the ZYX is taken as an example. Suppose the front of a rigid body (toward our direction) is the X axis, the right side is the Y axis, and the top is the Z axis, as shown by [??](#). 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 $[r, p, y]^T$ to describe any rotation. This vector is very intuitive, we can imagine the rotation process from this vector. The other Euler angles are also decomposed into three axes to obtain a three-dimensional vector, but the axes and order maybe different. The *rpy* angle introduced here is a widely used one, and only a few Euler angles have such a popular name as *rpy*. Different Euler angles are referred to in the order of the axes of rotation. For example, the rotation order of the *rpy* angle is ZYX . Similarly, there are Euler angles like XYZ, ZYZ - but they don't have a specific name. It is worth mentioning that most areas have their own coordinate directions and order habits when using Euler angles, not necessarily the same as we said here.

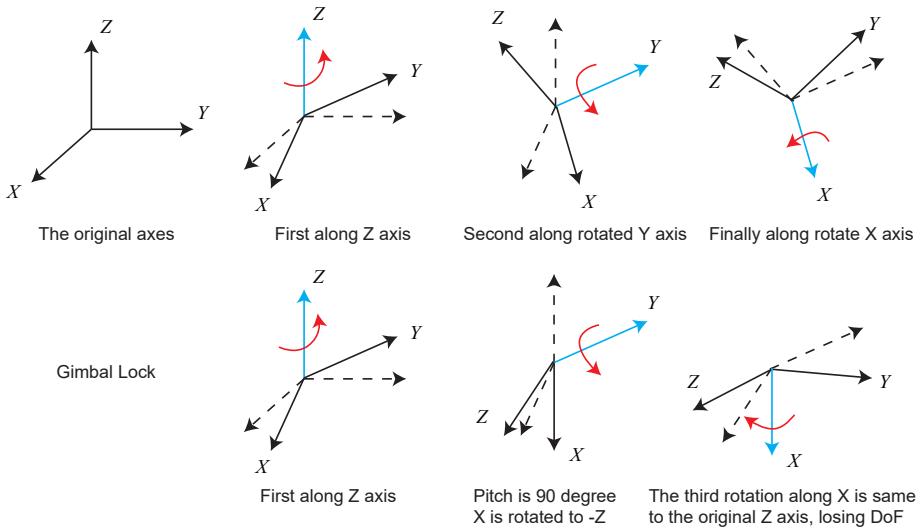


Figure 3-2: Euler angles. The top is defined for the ZYX order (rpy order). The bottoms shows when pitch=90°, the third rotation is using the same axis as the first one, causing the system to lose a degree of freedom. If you don't understand the universal lock, please take a look at the related videos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

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

3.4 Quaternion

The rotation matrix describes the rotation of 3 degrees of freedom with 9 quantities, with redundancy; the Euler angles and the rotation vectors are compact but has

^{*} https://en.wikipedia.org/wiki/Gimbal_lock.

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

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

Recall the complex number that we have studied before. We use the complex set \mathbb{C} to represent the vector on the 2D complex plane, and the complex multiplication with an unit complex number can represent the rotation on the 2D plane: for example, multiplying the complex i is equivalent to rotating a complex vector counterclockwise by 90° . Similarly, when expressing a three-dimensional space rotation, there is also an algebra similar to a complex number: **quaternions**. The quaternion is an extended complex number found by Hamilton. It **is both compact and not singular**. If we must find some shortcomings, the quaternion is not intuitive enough, and its operation is a bit more complicated.

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

$$e^{i\theta} = \cos \theta + i \sin \theta. \quad (3.19)$$

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:

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

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

$$\left\{ \begin{array}{l} i^2 = j^2 = k^2 = -1 \\ ij = k, ji = -k \\ jk = i, kj = -i \\ ki = j, i = -j \end{array} \right. . \quad (3.21)$$

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:

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

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

We can use a **unit quaternion** to represent any rotation in 3D space, but this expression is subtly different from the complex numbers. In the complex, multiplying by i means rotating 90° . Does this mean that in the quaternion, multiplied by i is rotated around the i axis by 90° ? So, does $ij = k$ mean, first rotating around the i by 90° , then around j by 90° , is equivalent to rotating around k by 90° ? Readers can use a cell phone to simulate that, then you will find that this is not the case. The correct situation should be that multiplying i corresponds to rotating 180° , in

order to guarantee the nature of $ij = k$. And $i^2 = -1$ means that after rotating 360° around the i axis, we get an opposite thing. This thing has to be rotated for 720° to be equal to its original appearance.

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

3.4.1 Quaternion Operations

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

$$\mathbf{q}_a \pm \mathbf{q}_b = [s_a \pm s_b, \mathbf{v}_a \pm \mathbf{v}_b]^T. \quad (3.22)$$

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 (??):

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

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:

$$\mathbf{q}_a \mathbf{q}_b = [s_a s_b - \mathbf{v}_a^T \mathbf{v}_b, s_a \mathbf{v}_b + s_b \mathbf{v}_a + \mathbf{v}_a \times \mathbf{v}_b]^T. \quad (3.24)$$

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:

$$\|\mathbf{q}_a\| = \sqrt{s_a^2 + x_a^2 + y_a^2 + z_a^2}. \quad (3.25)$$

It can be verified that the length of the product is the product of the length. This makes the unit quaternion keep unit length when multiplied by another unit quaternion:

$$\|\mathbf{q}_a \mathbf{q}_b\| = \|\mathbf{q}_a\| \|\mathbf{q}_b\|. \quad (3.26)$$

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

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

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

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

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

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

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

$$\mathbf{q} \mathbf{q}^{-1} = \mathbf{q}^{-1} \mathbf{q} = \mathbf{1}. \quad (3.30)$$

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:

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

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

$$k \mathbf{q} = [ks, k\mathbf{v}]^T. \quad (3.32)$$

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

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

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

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:

$$\mathbf{p}' = \mathbf{q} \mathbf{p} \mathbf{q}^{-1}. \quad (3.33)$$

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 $\mathbf{q} = [s, \mathbf{v}]^T$, then define the following symbols $+$ and \oplus as [?]:

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

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

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

We simply get:

$$\mathbf{q}_1 \mathbf{q}_2 = \mathbf{q}_1^+ \mathbf{q}_2 = \mathbf{q}_2^\oplus \mathbf{q}_1. \quad (3.36)$$

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

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

Substituting the matrix corresponding to two symbols, we get:

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

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**:

$$\mathbf{R} = \mathbf{v}\mathbf{v}^T + s^2 \mathbf{I} + 2s\mathbf{v}^\wedge + (\mathbf{v}^\wedge)^2. \quad (3.39)$$

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

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

Also obtained by the formula (??):

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

which is

$$\cos \theta = 2s^2 - 1 = 2 \cos^2 \frac{\theta}{2} - 1, \quad (3.42)$$

and so we have:

$$\theta = 2 \arccos s. \quad (3.43)$$

For the rotation axis, if we replace \mathbf{p} with the imaginary part of \mathbf{q} in the formula (??), 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:

$$\begin{cases} \theta = 2 \arccos q_0 \\ [\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z]^T = [q_1, q_2, q_3]^T / \sin \frac{\theta}{2} \end{cases}. \quad (3.44)$$

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 introduce 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:

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

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 $\text{Sim}(3)$.

2. Affine transformation.

The matrix form of the affine transformation is as follows:

$$\mathbf{T}_A = \begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}. \quad (3.46)$$

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

$$\mathbf{T}_P = \begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{a}^T & v \end{bmatrix}. \quad (3.47)$$

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}^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 than the far away part, it is not even a parallelogram, but an irregular quadrilateral.

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

Table 3-1: comparison of common transformation properties

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

We will introduce later that the transformation from the real world to the camera photo is a perspective transformation. If the focal length of the camera is infinity, then this transformation is an affine transformation. However, before we go into the details of the camera model, let's do some experiments to have a rough impression of these transformations.

3.6 Practice: Eigen Geometry Module

3.6.1 Data demonstration of the Eigen geometry module

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

Listing 3.6: slambook2/ch3/useGeometry/useGeometry.cpp

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 #include <Eigen/Core>
6 #include <Eigen/Geometry>
7
8 using namespace Eigen;
9 // This program demonstrates how to use the Eigen geometry module
10
11 int main(int argc, char **argv) {
12     // The Eigen/Geometry module provides a variety of rotation and
13     // translation representations
14     Matrix3d rotation_matrix = Matrix3d::Identity();
15     // The rotation vector uses AngleAxis, the underlying layer is not
16     // directly Matrix, but the operation can be treated as a matrix (
17     // because the operator is overloaded)
18     AngleAxisd rotation_vector(M_PI / 4, Vector3d(0, 0, 1)); //Rotate 45
19     degrees along the Z axis
20     cout.precision(3);
21     cout << "rotation matrix = \n " << rotation_vector.matrix() << endl; //
22     // convert to matrix with matrix()
23     // can also be assigned directly
24     rotation_matrix = rotation_vector.toRotationMatrix();
25     // coordinate transformation with AngleAxis
26     Vector3d v(1, 0, 0);
27     Vector3d v_rotated = rotation_vector * v;
28     cout << "(1,0,0) after rotation (by angle axis) = " << v_rotated.
29     transpose() << endl;
30     // Or use a rotation matrix
31     v_rotated = rotation_matrix * v;
32     cout << "(1,0,0) after rotation (by matrix) = " << v_rotated.transpose()
33     << endl;
34
35     // Euler angle: You can convert the rotation matrix directly into Euler
36     // angles
37     Vector3d euler_angles = rotation_matrix.eulerAngles(2, 1, 0); // ZYX
38     // order, ie roll pitch yaw order
39     cout << "yaw pitch roll = " << euler_angles.transpose() << endl;
40
41     // Euclidean transformation matrix using Eigen::Isometry
42     Isometry3d T = Isometry3d::Identity(); // Although called 3d, it is
43     // essentially a 4*4 matrix
44     T.rotate(rotation_vector); // Rotate according to rotation_vector
45     T.pretranslate(Vector3d(1, 3, 4)); // Set the translation vector to
46     // (1,3,4)
47     cout << "Transform matrix = \n" << T.matrix() << endl;
48
49     // Use the transformation matrix for coordinate transformation
50     Vector3d v_transformed = T * v; // Equivalent to R*v+v

```

```

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

61 return 0;
}

```

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

- Rotation matrix (3×3): Eigen::Matrix3d.
- Rotation vector (3×1): Eigen::AngleAxisd.
- Euler angle (3×1): Eigen::Vector3d.
- Quaternion (4×1): Eigen::Quaterniond.
- Euclidean transformation matrix (4×4): Eigen::Isometry3d.
- Affine transform (4×4): Eigen::Affine3d.
- Perspective transformation (4×4): Eigen::Projective3d.

This program can be compiled by referring to the corresponding CMakeLists in the code. In this program, I demonstrate how to use the rotation matrix, rotation vectors (AngleAxis), Euler angles, and quaternions in Eigen. We use these rotations to rotate a vector 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 $\mathbf{q}_1 = [0.35, 0.2, 0.3, 0.1]^T$, $\mathbf{t}_1 = [0.3, 0.1, 0.1]^T$. The pose of the robot No. 2 is $\mathbf{q}_2 = [-0.5, 0.4, -0.1, 0.2]^T$, $\mathbf{t}_2 = [-0.1, 0.5, 0.3]^T$. 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 $\mathbf{p}_{R_1} = [0.5, 0, 0.2]^T$, 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

```

1 #include<iostream>
2 #include<vector>
3 #include<algorithm>
4 #include<Eigen/Core>
5 #include<Eigen/Geometry>
6
7 using namespace std;
8 using namespace Eigen;
9
10 int main(int argc, char** argv) {
11     Quaterniond q1(0.35, 0.2, 0.3, 0.1), q2(-0.5, 0.4, -0.1, 0.2);
12     q1.normalize();
13     q2.normalize();
14     Vector3d t1(0.3, 0.1, 0.1), t2(-0.1, 0.5, 0.3);
15     Vector3d p1(0.5, 0, 0.2);
16
17     Isometry3d T1w(q1), T2w(q2);
18     T1w.pretranslate(t1);
19     T2w.pretranslate(t2);
20
21     Vector3d p2 = T2w * T1w.inverse() * p1;
22     cout << endl << p2.transpose() << endl;
23     return 0;
24 }
```

The answer to the program is $[-0.0309731, 0.73499, 0.296108]^T$, and the calculation process is very simple, just by calculating

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

Note that the quaternion needs to be normalized before use.

3.7 Visualization Demo

3.7.1 Plotting Trajectory

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

the rotation and transformation matrix may not be intuitive enough, we can easily draw them in a 3D window.

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

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

where time refers the recording time of this pose, \mathbf{t} is translation, \mathbf{q} is the quaternion, all recorded in the world coordinate system to the robot coordinate system. Below we read these tracks from the file and display them in a window. In principle, if we just talk about "robot pose", then we can use any one of \mathbf{T}_{WR} or \mathbf{T}_{RW} because they are just the inverse of each other. It means that knowing one of them makes it easy to get the other. If we want to store **robot's trajectory**, then saving \mathbf{T}_{WR} or \mathbf{T}_{RW} doesn't make much difference.

When drawing the trajectory, we should draw the "trajectory" as a sequence of points, which is similar to the "trajectory" we imagined. Strictly speaking, this is actually the **the coordinates of the origin of the robot in the world coordinate system**. Consider the origin of the robot coordinate system, i.e., \mathbf{O}_R , then the \mathbf{O}_W at this time is the coordinates of the origin in the world coordinate system:

$$\mathbf{O}_W = \mathbf{T}_{WR} \mathbf{O}_R = \mathbf{t}_{WR}. \quad (3.48)$$

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

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

Listing 3.8: slambook2/ch3/examples/plotTrajectory.cpp

```

1 #include <pangolin/pangolin.h>
2 #include <Eigen/Core>
3 #include <unistd.h>
4
5 using namespace std;
6 using namespace Eigen;
7
8 // path to trajectory file
9 string trajectory_file = "./examples/trajectory.txt";
10
11 void DrawTrajectory(vector<Isometry3d>, Eigen::aligned_allocator<Isometry3d>>);
12
13 int main(int argc, char **argv) {
14     vector<Isometry3d>, Eigen::aligned_allocator<Isometry3d>> poses;
15     ifstream fin(trajectory_file);
16     if (!fin) {

```

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

```

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

```

```

79     glVertex3d(p1.translation()[0], p1.translation()[1], p1.translation()
80                 [2]);
81     glVertex3d(p2.translation()[0], p2.translation()[1], p2.translation()
82                 [2]);
83     glEnd();
84 }
85 pangolin::FinishFrame();
86 usleep(5000);    // sleep 5 ms
87 }
```

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 ??.

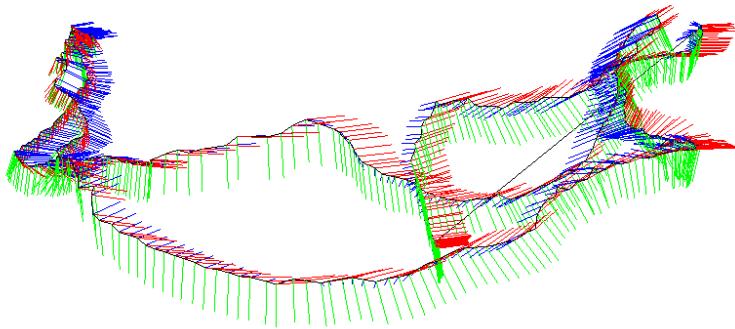


Figure 3-3: Results of pose visualization

3.7.2 Displaying Camera Pose

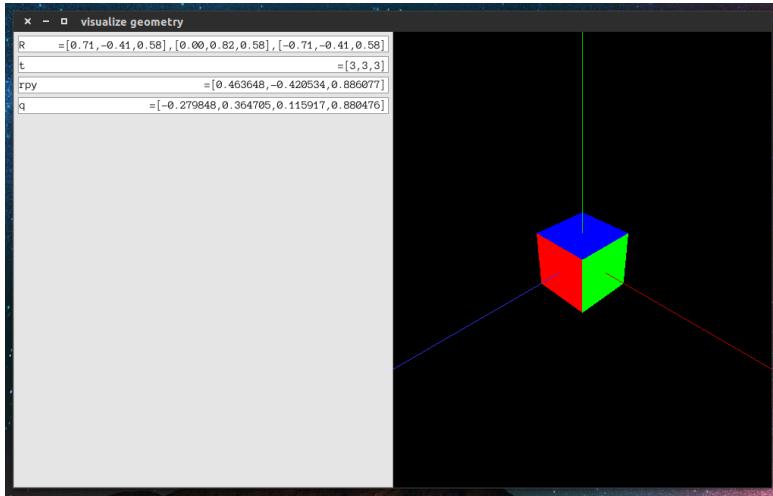


Figure 3-4: Visualization program for rotation matrix, Euler angle, quaternion.

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 ??). 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 ??).
4. Draw a table that summarizes the conversion relationship of the rotation matrix, rotation angle, Euler angle and quaternion.
5. Suppose there is a large Eigen matrix, we want to know the value in the top left 3×3 blocks, and then assign it to $\mathbf{I}_{3 \times 3}$. Please implement it in C++.
6. When does a general linear equation $\mathbf{Ax} = \mathbf{b}$ has an unique solution of \mathbf{x} ? How to solve it numerically? Can you implement it in Eigen?

Chapter 4

Lie Group and Lie Algebra

Main Goal

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

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

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

4.1 Basics of Lie Group and Lie Algebra

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

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

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

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:

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

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:

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

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

4.1.1 Group

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

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

1. Closure: $\forall a_1, a_2 \in A, a_1 \cdot a_2 \in A$.
2. Combination: $\forall a_1, a_2, a_3 \in A, (a_1 \cdot a_2) \cdot a_3 = a_1 \cdot (a_2 \cdot a_3)$.
3. Unit element: $\exists a_0 \in A$, s.t. $\forall a \in A, a \cdot a_0 = a \cdot a_0 = a$.
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 $(\mathbb{Q} \setminus 0, \cdot)$, etc. Common groups in the matrix are:

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

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, $\text{SO}(n)$ and $\text{SE}(n)$ 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 $\text{SO}(3)$ and $\text{SE}(3)$ are especially important for camera pose estimation, we mainly discuss these two Lie groups. However, strictly discussing the concepts of “continuous” and “smooth” requires knowledge of analysis and topology, but we are not mathematics books, so only some important conclusions directly related to SLAM are introduced. If the reader is interested in the theoretical nature of Lie Groups, please refer to the special books like [?].

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

4.1.2 Introduction of the Lie Algebra

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

$$\mathbf{R}\mathbf{R}^T = \mathbf{I}. \quad (4.5)$$

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

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

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

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

Put the second item to right:

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

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

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

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

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

$$\dot{\mathbf{R}}(t)\mathbf{R}(t)^\top = \phi(t)^\wedge.$$

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

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

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

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

We see that ϕ reflects the derivative of \mathbf{R} , so it is called the **Tangent Space** near the origin of $\text{SO}(3)$. Also, if time t is close to t_0 , we assume $\phi(t)$ to close to be a constant $\phi(t_0) = \phi_0$. Then according to the formula (??), we have:

$$\dot{\mathbf{R}}(t) = \phi(t_0)^\wedge \mathbf{R}(t) \approx \phi_0^\wedge \mathbf{R}(t).$$

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

$$\mathbf{R}(t) = \exp(\phi_0^\wedge t). \quad (4.10)$$

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 $\exp(\phi_0^\wedge t)^*$. We see that the rotation matrix \mathbf{R} is associated with another skew-symmetric matrix $\phi_0^\wedge t$ 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 ϕ that describes the local derivative relationship of \mathbf{R} . How are they correlated with each other? We will say that ϕ corresponds to the Lie algebra $\mathfrak{so}(3)$ on $\text{SO}(3)$;
2. Second, when a vector ϕ is given, how is $\exp(\phi^\wedge)$ calculated? Conversely, given \mathbf{R} , is there an opposite operation to calculate ϕ ? 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 its origin point, or in other words, is the tangent space. The general definition of Lie algebra is listed as follows:

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

1. Closure. $\forall \mathbf{X}, \mathbf{Y} \in \mathbb{V}, [\mathbf{X}, \mathbf{Y}] \in \mathbb{V}$.

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

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

3. Reflexive * $\forall \mathbf{X} \in \mathbb{V}, [\mathbf{X}, \mathbf{X}] = \mathbf{0}$.

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

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

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

The previously mentioned ϕ 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 ϕ . According to the previous derivation, each ϕ can generate an skew-symmetric matrix:

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

Under this definition, the two vectors ϕ_1, ϕ_2 's Lie brackets are:

$$[\phi_1, \phi_2] = (\Phi_1 \Phi_2 - \Phi_2 \Phi_1)^\vee. \quad (4.12)$$

The reader can verify that the Lie brackets under this definition satisfy the above properties. Since the vector ϕ 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:

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

Some books also use the symbol $\hat{\phi}$ to represent ϕ^\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 $\text{SO}(3)$ is given by the exponential map:

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

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

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

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

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

We write each $\mathfrak{se}(3)$ element as ξ , 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 ρ ; after the three-dimensional rotation, there is a ϕ , which is essentially a $\mathfrak{so}(3)$ element*. At the same time, we extended the meaning of the $^\wedge$ symbol. In $\mathfrak{se}(3)$, a six-dimensional vector is also converted to a four-dimensional matrix using the $^\wedge$ symbol, but no longer a skew-symmetric one:

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

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 ρ is not the direct translation). Similarly, the Lie algebra $\mathfrak{se}(3)$ also has a Lie bracket similar to $\mathfrak{so}(3)$:

$$[\xi_1, \xi_2] = (\xi_1^\wedge \xi_2^\wedge - \xi_2^\wedge \xi_1^\wedge)^\vee. \quad (4.17)$$

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 $\text{SO}(3)$

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

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

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

* 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 $\phi \in \mathfrak{so}(3)$, we can also define its exponential map in this way:

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

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 ϕ is a three-dimensional vector, we can define its length and direction, denoted as θ and \mathbf{a} , respectively. So we have $\phi = \theta\mathbf{a}$, where \mathbf{a} is a unit-length direction vector, i.e., $\|\mathbf{a}\| = 1$. First, for such a unit-length vector \mathbf{a} , there are two properties:

$$\mathbf{a}^\wedge \mathbf{a}^\wedge = \begin{bmatrix} -a_2^2 - a_3^2 & a_1 a_2 & a_1 a_3 \\ a_1 a_2 & -a_1^2 - a_3^2 & a_2 a_3 \\ a_1 a_3 & a_2 a_3 & -a_1^2 - a_2^2 \end{bmatrix} = \mathbf{aa}^T - \mathbf{I}, \quad (4.20)$$

as well as

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

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

$$\begin{aligned} \exp(\phi^\wedge) &= \exp(\theta\mathbf{a}^\wedge) = \sum_{n=0}^{\infty} \frac{1}{n!} (\theta\mathbf{a}^\wedge)^n \\ &= \mathbf{I} + \theta\mathbf{a}^\wedge + \frac{1}{2!} \theta^2 \mathbf{a}^\wedge \mathbf{Bma}^\wedge + \frac{1}{3!} \theta^3 \mathbf{a}^\wedge \mathbf{a}^\wedge \mathbf{a}^\wedge + \frac{1}{4!} \theta^4 (\mathbf{a}^\wedge)^4 + \dots \\ &= \mathbf{aa}^T - \mathbf{a}^\wedge \mathbf{a}^\wedge + \theta\mathbf{a}^\wedge + \frac{1}{2!} \theta^2 \mathbf{a}^\wedge \mathbf{a}^\wedge - \frac{1}{3!} \theta^3 \mathbf{a}^\wedge - \frac{1}{4!} \theta^4 (\mathbf{a}^\wedge)^2 + \dots \\ &= \mathbf{aa}^T + \underbrace{\left(\theta - \frac{1}{3!} \theta^3 + \text{Frac}15! \theta^5 - \dots \right)}_{\sin \theta} \mathbf{a}^\wedge - \underbrace{\left(1 - \frac{1}{2!} \theta^2 + \frac{1}{4!} \theta^4 - \dots \right)}_{\cos \theta} \mathbf{a}^\wedge \mathbf{a}^\wedge \\ &= \mathbf{a}^\wedge \mathbf{a}^\wedge + \mathbf{I} + \sin \theta \mathbf{a}^\wedge - \cos \theta \mathbf{Bma}^\wedge \mathbf{a}^\wedge \\ &= (1 - \cos \theta) \mathbf{a}^\wedge \mathbf{a}^\wedge + \mathbf{I} + \sin \theta \mathbf{a}^\wedge \\ &= \cos \theta \mathbf{I} + (1 - \cos \theta) \mathbf{aa}^T + \sin \theta \mathbf{a}^\wedge. \end{aligned}$$

Finally, we got a very familiar equation:

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

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

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

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

corresponding Lie algebra according to the rotation matrix, that is, using the formula (??), 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 ϕ for any \mathbf{R} ? Unfortunately, the exponential map is just a surjective map, not injective. This means that for each element in $\text{SO}(3)$ we can find a $\mathfrak{so}(3)$ element corresponding to it; however, there may be multiple $\mathfrak{so}(3)$ elements corresponding to the same $\text{SO}(3)$ 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 $\text{SO}(3)$ 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 $\text{SE}(3)$

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:

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

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

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

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

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

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

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

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

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

$$\mathbf{t} = \mathbf{J}\boldsymbol{\rho}. \quad (4.28)$$

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

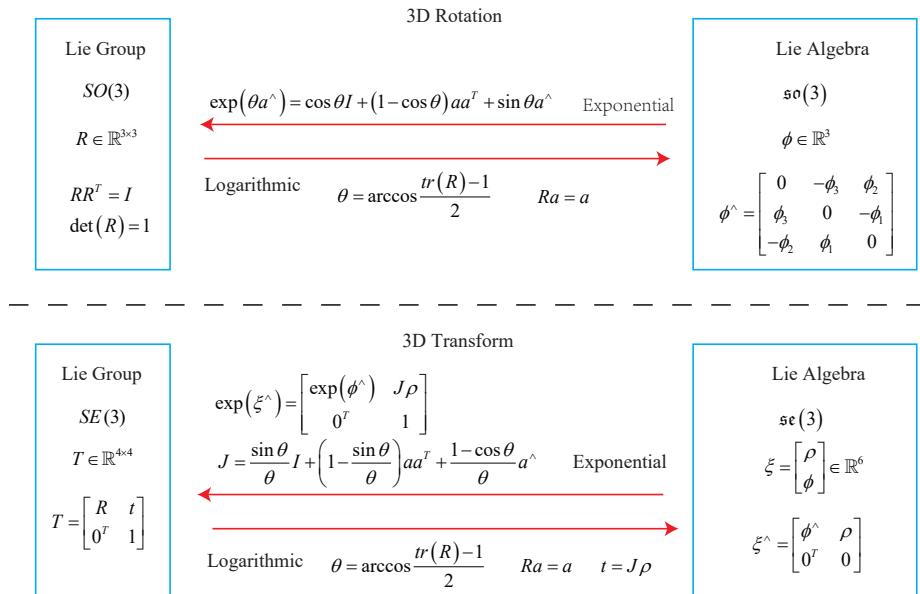


Figure 4-1: The correspondence between $SO(3)$, $SE(3)$, $\mathfrak{so}(3)$, $\mathfrak{se}(3)$.

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:

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

If ϕ_1, ϕ_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:

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

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:

$$\ln(\exp(\mathbf{A})\exp(\mathbf{B})) = \mathbf{A} + \mathbf{B} + \frac{1}{2}[\mathbf{A}, \mathbf{B}] + \frac{1}{12}[\mathbf{A}, [\mathbf{A}, \mathbf{B}]] - \frac{1}{12}[\mathbf{B}, [\mathbf{A}, \mathbf{B}]] + \dots \quad (4.29)$$

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 $\text{SO}(3)$, the $\ln(\exp(\phi_1^\wedge)\exp(\phi_2^\wedge))^\vee$, when ϕ_1 or ϕ_2 is small, small items with more than quadratic can be simply ignored. At this time, BCH has a linear approximation†:

$$\ln(\exp(\phi_1^\wedge)\exp(\phi_2^\wedge))^\vee \approx \begin{cases} \mathbf{J}_l(\phi_2)^{-1}\phi_1 + \phi_2 & \text{when } \phi_1 \text{ is a small amount,} \\ \mathbf{J}_r(\phi_1)^{-1}\phi_2 + \phi_1 & \text{when } \phi_2 \text{ is a small amount.} \end{cases} \quad (4.30)$$

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 ϕ_1 and ϕ_2 , respectively), in $\mathfrak{so}(3)$ it can be approximated by adding a $\mathbf{J}_l(\phi_2)^{-1}\phi_1$ to the original Lie algebra ϕ_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 (??) :

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

Its inverse is:

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

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:

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

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 ϕ . 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 [?] if you are interested.

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

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

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

$$\exp((\phi + \Delta\phi)^\wedge) = \exp((\mathbf{J}_l\Delta\phi)^\wedge) \exp(\phi^\wedge) = \exp(\phi^\wedge) \exp((\mathbf{J}_r\Delta\phi)^\wedge). \quad (4.35)$$

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

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

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

Here the \mathcal{J}_l and \mathcal{J}_r are a more complicated 6×6 matrices. Readers can find its detailed contents in [?]. 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:

$$\mathbf{z} = \mathbf{T}\mathbf{p} + \mathbf{w}, \quad (4.38)$$

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:

$$\mathbf{e} = \mathbf{z} - \mathbf{T}\mathbf{p}. \quad (4.39)$$

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

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

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

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

According to the definition of the derivative, we have:

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

* 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 $d(\mathbf{Ax})/d\mathbf{x} = \mathbf{A}$.

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:

$$\frac{\partial (\mathbf{R}\mathbf{p})}{\partial \phi} = (-\mathbf{R}\mathbf{p})^\wedge \mathbf{J}_l. \quad (4.41)$$

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 φ . Then, for φ , that is:

$$\frac{\partial (\mathbf{R}\mathbf{p})}{\partial \varphi} = \lim_{\varphi \rightarrow 0} \frac{\exp(\varphi^\wedge) \exp(\phi^\wedge) \mathbf{p} - \exp(\phi^\wedge) \mathbf{p}}{\varphi}. \quad (4.42)$$

The derivation of this formula is simpler than the above:

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

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 ξ), and the result is $\mathbf{T}\mathbf{p}^*$. Now, give \mathbf{T} a left perturbation $\Delta\mathbf{T} = \exp(\delta\xi^\wedge)$, whose Lie algebra is $\delta\xi = [\delta\rho, \delta\phi]^T$, then:

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

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

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:

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

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

```

1 #include <iostream>
2 #include <cmath>
3 #include <Eigen/Core>
4 #include <Eigen/Geometry>
5 #include "sophus/se3.hpp"
6
7 using namespace std;
8 using namespace Eigen;
9
10 // This program demonstrates the basic usage of Sophus
11 int main(int argc, char **argv) {
12     // Rotation matrix with 90 degrees along Z axis
13     Matrix3d R = AngleAxisd(M_PI / 2, Vector3d(0, 0, 1)).toRotationMatrix()
14     ;
15     // or quaternion
16     Quaternionnd q(R);                                // Sophus::SO3d can be constructed
17     Sophus::SO3d S03_R(R);                           // from rotation matrix
18     Sophus::SO3d S03_q(q);                            // or quaternion
19     // they are equivalent of course
20     cout << "SO(3) from matrix:\n" << S03_R.matrix() << endl;
21     cout << "SO(3) from quaternion:\n" << S03_q.matrix() << endl;
22     cout << "they are equal" << endl;
23
24     // Use logarithmic map to get the Lie algebra
25     Vector3d so3 = S03_R.log();
26     cout << "so3 = " << so3.transpose() << endl;
27     // hat is from vector to skew-symmetric matrix
28     cout << "so3 hat=\n" << Sophus::SO3d::hat(so3) << endl;
29     // inversely from matrix to vector
30     cout << "so3 hat vee= " << Sophus::SO3d::vee(Sophus::SO3d::hat(so3)).
31         transpose() << endl;
32
33     // update by perturbation model
34     Vector3d update_so3(1e-4, 0, 0); // this is a small update
35     Sophus::SO3d S03_updated = Sophus::SO3d::exp(update_so3) * S03_R;
36     cout << "S03 updated = \n" << S03_updated.matrix() << endl;
37
38     cout << "*****\n";
39     // Similar for SE(3)
40     Vector3d t(1, 0, 0);                      // translation 1 along X
41     Sophus::SE3d SE3_Rt(R, t);                // construction SE3 from R,t
42     Sophus::SE3d SE3_qt(q, t);                // or q,t
43     cout << "SE3 from R,t= \n" << SE3_Rt.matrix() << endl;
44     cout << "SE3 from q,t= \n" << SE3_qt.matrix() << endl;
45     // Lie Algebra is 6d vector, we give a typedef
46     typedef Eigen::Matrix<double, 6, 1> Vector6d;
47     Vector6d se3 = SE3_Rt.log();
48     cout << "se3 = " << se3.transpose() << endl;
49     // The output shows Sophus puts the translation at first in se(3), then
50     // rotation.
51
52     // Save as SO(3) wehave hat and vee

```

```

49     cout << "se3 hat = \n" << Sophus::SE3d::hat(se3) << endl;
50     cout << "se3 hat vee = " << Sophus::SE3d::vee(Sophus::SE3d::hat(se3));
51     transpose() << endl;
52
53     // Finally the update
54     Vector6d update_se3;
55     update_se3.setZero();
56     update_se3(0, 0) = 1e-4d;
57     Sophus::SE3d SE3_updated = Sophus::SE3d::exp(update_se3) * SE3_Rt;
58     cout << "SE3 updated = " << endl << SE3_updated.matrix() << endl;
59
60     return 0;
}

```

The demo is divided into two parts. The first half introduces the operation on SO(3), and the second half is SE(3). We demonstrate how to construct 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 $i = 1, \dots, N$, 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:

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

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 [?], so we can define **Average Translational Error**:

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

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:

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

Similarly, you can only take the translation part:

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

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

```

1 #include <iostream>
2 #include <fstream>
3 #include <unistd.h>
4 #include <pangolin/pangolin.h>
5 #include <sophus/se3.hpp>
6
7 using namespace Sophus;
8 using namespace std;
9
10 string groundtruth_file = "./example/groundtruth.txt";
11 string estimated_file = "./example/estimated.txt";
12
13 typedef vector<Sophus::SE3d, Eigen::aligned_allocator<Sophus::SE3d>>
14     TrajectoryType;
15
16 void DrawTrajectory(const TrajectoryType &gt, const TrajectoryType &est);
17
18 TrajectoryType ReadTrajectory(const string &path);
19
20 int main(int argc, char **argv) {
21     TrajectoryType groundtruth = ReadTrajectory(groundtruth_file);
22     TrajectoryType estimated = ReadTrajectory(estimated_file);
23     assert(!groundtruth.empty() && !estimated.empty());
24 }
```

```

23     assert(groundtruth.size() == estimated.size());
24
25     // compute rmse
26     double rmse = 0;
27     for (size_t i = 0; i < estimated.size(); i++) {
28         Sophus::SE3d p1 = estimated[i], p2 = groundtruth[i];
29         double error = (p2.inverse() * p1).log().norm();
30         rmse += error * error;
31     }
32     rmse = rmse / double(estimated.size());
33     rmse = sqrt(rmse);
34     cout << "RMSE = " << rmse << endl;
35
36     DrawTrajectory(groundtruth, estimated);
37     return 0;
38 }
39
40 TrajectoryType ReadTrajectory(const string &path) {
41     ifstream fin(path);
42     TrajectoryType trajectory;
43     if (!fin) {
44         cerr << "trajectory " << path << " not found." << endl;
45         return trajectory;
46     }
47
48     while (!fin.eof()) {
49         double time, tx, ty, tz, qx, qy, qz, qw;
50         fin >> time >> tx >> ty >> tz >> qx >> qy >> qz >> qw;
51         Sophus::SE3d p1(Eigen::Quaterniond(qx, qy, qz, qw), Eigen::Vector3d
52                         (tx, ty, tz));
53         trajectory.push_back(p1);
54     }
55     return trajectory;
}

```

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

4.5 Similar Transforma Group and its Lie Algebra

Finally, we would like to mention the similar transform group $\text{Sim}(3)$ 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 $\text{SE}(3)$ 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 $\text{SE}(3)$ 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:

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

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 $\text{SO}(3)$, $\text{SE}(3)$, the simi-

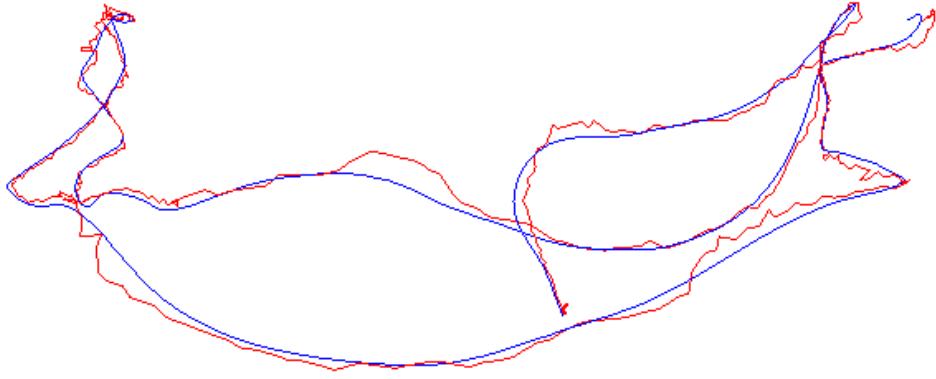


Figure 4-2: Calculates the error between the estimated trajectory and the real trajectory.

larity transform also forms a group on matrix multiplication, called the similarity transform group $\text{Sim}(3)$:

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

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

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

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

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

Where \mathbf{J}_s is:

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

Through exponential mapping, we can find the relationship between Lie algebra and Lie group. For the Lie algebra ζ , its correspondence with Lie group is:

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

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

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

$$\frac{\partial \mathbf{Sp}}{\partial \zeta} = \begin{bmatrix} \mathbf{I} & -\mathbf{q}^\wedge & \mathbf{q} \\ \mathbf{0}^T & \mathbf{0}^T & 0 \end{bmatrix}. \quad (4.53)$$

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

4.6

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:

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

6. Show that:

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

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

$$\mathbf{T} \exp(\boldsymbol{\xi}^\wedge) \mathbf{T}^{-1} = \exp((\text{Ad}(\mathbf{T})\boldsymbol{\xi})^\wedge), \quad (4.54)$$

where

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

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-hole 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 ??.

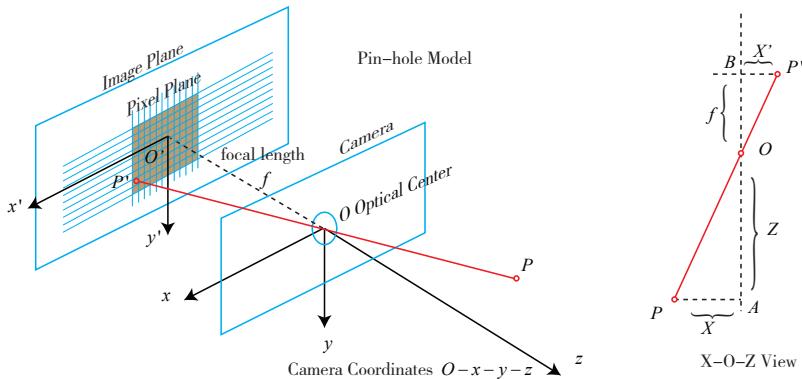


Figure 5-1: Pinhole camera model.

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 $[X, Y, Z]^T$, P' is $[X', Y', Z']^T$, 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:

$$\frac{Z}{f} = -\frac{X}{X'} = -\frac{Y}{Y'}. \quad (5.1)$$

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 ???. This can remove the negative sign in the formula to make the formula more compact:

$$\frac{Z}{f} = \frac{X}{X'} = \frac{Y}{Y'}. \quad (5.2)$$

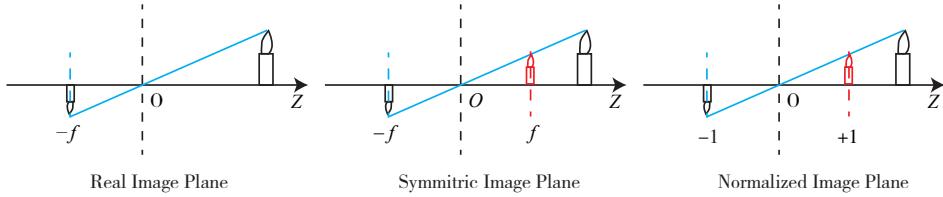


Figure 5-2: The real, symmetric and normalized image plane.

Put X', Y' to the left side:

$$\begin{aligned} X' &= f \frac{X}{Z} \\ Y' &= f \frac{Y}{Z} \end{aligned} \quad (5.3)$$

Readers may ask why 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-hole 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 omit the minus symbol in the pinhole model.

The formula (??) 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: $[u, v]^T$.

The usual definition of 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 $[c_x, c_y]^T$. Then, the relationship between the coordinates of P' and the pixel coordinate $[u, v]^T$ is:

$$\begin{cases} u = \alpha X' + c_x \\ v = \beta Y' + c_y \end{cases} \quad (5.4)$$

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

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

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

where f is the focal length in meters, α , and β 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:

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

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

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

In this equation, we refer to the matrix composed of the middle quantities as the camera's **inner parameter matrix** (or 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 [?]), it will not be introduced here *.

There are internal parameters, and naturally there must be something like "external parameters". In the equation (??), 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:

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

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

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

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

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

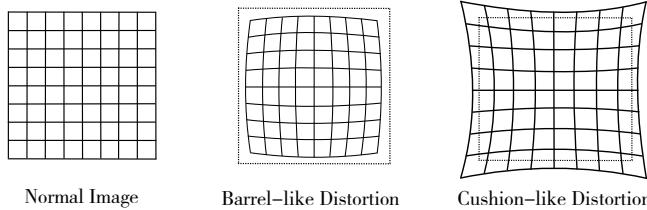


Figure 5-3: The radical distortion.

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

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

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 $[u, v]^T$ 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 ??.

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 ??.

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

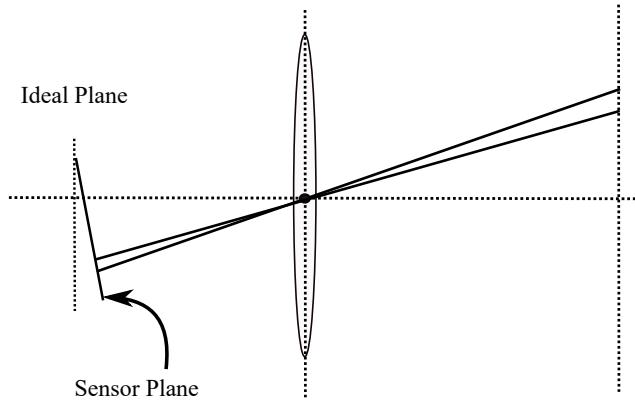


Figure 5-4: Tangential distortion.

whose coordinates are $[x, y]^T$, or $[r, \theta]^T$ 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:

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

where $[x_{\text{distorted}}, y_{\text{distorted}}]^T$ 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:

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

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

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

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 $\mathbf{T} \in \text{SE}(3)$. The camera coordinates for P are $\mathbf{P}_c = \mathbf{R}\mathbf{P}_w + \mathbf{t}$.
3. The \mathbf{P}_c component is X, Y, Z , and they are projected onto the normalized plane $Z = 1$ to get the normalized coordinates: $\mathbf{P}_c = [X/Z, Y/Z, 1]^T$ ^{*}.
4. If the image is distorted, the coordinates of \mathbf{P}_c after distortion are calculated according to the distortion parameters.
5. Finally, the distorted coordinates of P pass through the intrinsics and we find its pixel coordinates: $\mathbf{P}_{uv} = \mathbf{K}\mathbf{P}_c$.

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

* 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.

Chapter 6

nonlinear optimization

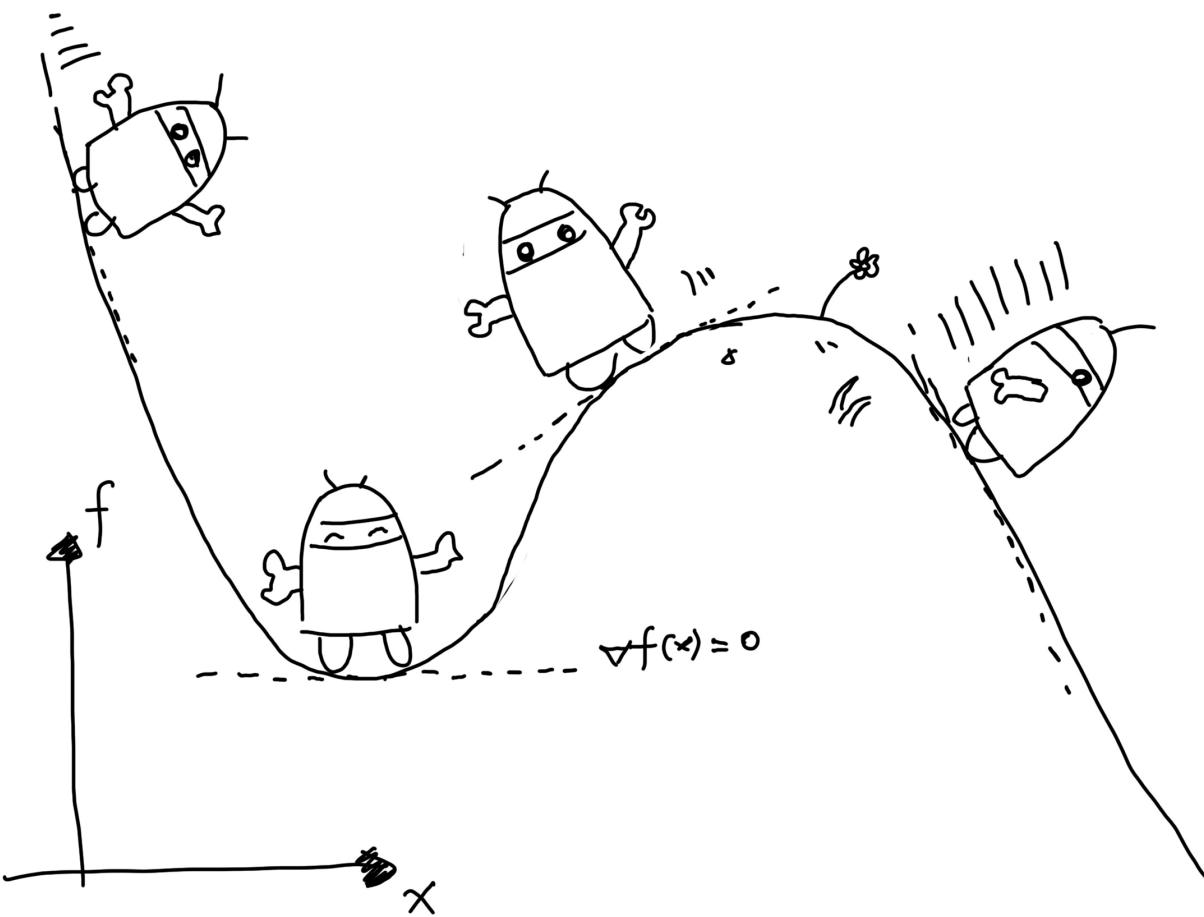
main target

1. understands the meaning and handling of least squares.
2. Understand the descending strategies of Gauss-Newton and Levenburg-Marquadt.
3. Learn the basic usage of the Ceres library and the g2o library.

In the previous few lectures, we introduced the equations of motion and observation equations of the classical SLAM model. Now we know that the poses in the equation can be described by the transformation matrix and then optimized with Lie algebra. The observation equation is given by the camera imaging model, where the internal reference is fixed with the camera and the external reference is the pose of the camera. Thus, we have clarified the specific expression of the classic SLAM model in the visual situation.

However, due to the existence of noise, the equations of the equation of motion and the equation of observation must not be exactly true. Although the camera fits 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, the equations of motion and observation equations can only be approximated. Therefore, instead of assuming that the data must conform to the equation, it is better to discuss how to make an accurate state estimate in the noisy data.

Solving the state estimation problem requires a certain degree of optimization background knowledge. This section introduces the basic unconstrained nonlinear optimization method and introduces how the optimized libraries g2o and Ceres are used.



$$f(x + \Delta x) \approx f(x) + \nabla f(x) \Delta x$$

$$+ \frac{1}{2} \Delta x^T H(x) \Delta x$$

+ ...

6.1 Status estimation problem

6.1.1 Batch state estimation and maximum a posteriori estimate

Following the previous lectures, we review the classic SLAM model discussed in the second lecture. It consists of an equation of motion and an observing equation, as shown by the formula (??):

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

Through the knowledge of Lecture 4, we learned that \mathbf{x}_k is the pose of the camera and can be described by SE(3). As for the observation equation, the fifth lecture has already explained, that is, the pinhole camera model. In order to give readers a deeper impression of them, we may wish to discuss their specific parametric forms. First, the pose variable \mathbf{x}_k can be expressed by $\mathbf{T}_k \in \text{SE}(3)$. Second, the equation of motion is related to the specific form of the input, but there is no particularity in the visual SLAM (as in 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. Suppose that an observation is made on the road sign \mathbf{y}_j at \mathbf{x}_k , corresponding to the pixel position $\mathbf{z}_{k,j}$ on the image, then the observation The equation can be expressed as

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

Where \mathbf{K} is the camera's internal parameter, s is the distance of the pixel, and is the third of $(\mathbf{R}_k \mathbf{y}_j + \mathbf{t}_k)$ Component. If the pose is described using the transformation matrix \mathbf{T}_k , the landmark point \mathbf{y}_j must be described in homogeneous coordinates and converted to non-homogeneous coordinates after the calculation is completed. If you are not familiar with this process, please go back to the previous lecture.

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

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

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

The methods for dealing with this state estimation problem are roughly divided into two types. Since these data are coming over time during the SLAM process, intuitively, we should hold an estimate of the current time and then update it with new data. This method is called **incremental**, or **filter**. 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 "snap" the data together, which is called **batch**. For example, we can put all the input and observation data from 0 to k at the moment, and ask, under such input and observation, how to estimate the trajectory and map from 0 to k ?

These two different approaches lead to many different estimation methods. In general, the incremental method only cares about the state estimate of **current time**, \mathbf{x}_k , and does not consider the previous state; relatively, the batch method

can be larger in **Range** is optimized and is considered superior to the traditional filter [?], becoming the mainstream method of current visual SLAM. In extreme cases, we can let the robot or drone collect data at all times and bring it back to the computing center for unified processing, which is the mainstream practice of SfM (Structure from Motion). Of course, this extreme situation is obviously not **real time**, and does not conform to the SLAM application scenario. So in SLAM, practical methods are usually a compromise. For example, we fixed some historical trajectories and optimized only some trajectories near the current time. This is the **sliding window estimation** method to be mentioned later.

In theory, batch methods are easier to introduce. At the same time, it is understood that the batch method is also easier to understand the incremental method. Therefore, in this section, we focus on the batch optimization method based on nonlinear optimization. The Kalman filter and more in-depth knowledge are left to the back-end chapters. Since the discussion is on a batch approach, consider all the moments from 1 to N and assume that there are M landmark points. Define the robot pose and landmark point coordinates at all times:

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

Similarly, \mathbf{u} without subscripts indicates input at all times, and \mathbf{z} indicates observations at all times. Then we say that the estimation of the state of the robot, from the point of view of probability, is the condition that the input data \mathbf{u} and the observed data \mathbf{z} are known, and the state **Conditional probability distribution for x, y** :

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

In particular, when we don't know the control input, only one image is considered, that is, only the data brought by the observation equation is considered, which is equivalent to estimating $P(\mathbf{x}, \mathbf{y} | \text{The conditional probability distribution of } \mathbf{z})$, also known as Structure from Motion (SfM), is how to reconstruct the three-dimensional structure [?] from many images.

In order to estimate the conditional distribution of state variables, Bayes' rule is used to:

$$P(\mathbf{x}, \mathbf{y} | \mathbf{z}, \mathbf{u}) = \frac{P(\mathbf{z}, \mathbf{u} | \mathbf{x}, \mathbf{y}) P(\mathbf{x}, \mathbf{y})}{P(\mathbf{z}, \mathbf{u})} \propto \underbrace{P(\mathbf{z}, \mathbf{u} | \mathbf{x}, \mathbf{y})}_{\text{liker}} \underbrace{P(\mathbf{x}, \mathbf{y})}_{\text{priority}}. \quad (6.5)$$

Bayes' rule is called **posterior probability** on the left, $P(\mathbf{z} | \mathbf{x})$ on the right is called **Likelihood**, and another part is $P(\mathbf{x})$ is called **priority** (Prior). **It is difficult to directly find the posterior distribution, but to find a state optimal estimate**, so that Maximize a Posterior (MAP) is feasible:

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

Note that the denominator part of Bayes' rule is independent of the state to be estimated \mathbf{x}, \mathbf{y} and can therefore be ignored. The Bayesian rule tells us that solving the maximum posterior probability **is equivalent to maximizing the likelihood and the prior product**. Further, of course we can also say, sorry, I don't know where the robot pose or road sign is, and there is no **priority**. Then, you can solve the **Maximize Likelihood Estimation** (MLE):

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

Intuitively, the likelihood is “what kind of observation data may be produced in the current position”. Since we know the observed data, the maximum likelihood estimate can be understood as: **“what state is most likely to produce the currently observed data”**. This is the intuitive meaning of maximum likelihood estimation.

6.1.2 Least squares lead

So how do you find the maximum likelihood estimate? We say that under the assumption of Gaussian distribution, the maximum likelihood can have a simpler form. Review the observation model for an observation:

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

Since we assumed the noise term $\mathbf{v}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_{k,j})$, so the conditional probability of the observed data is:

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

It is still a Gaussian distribution. Considering the maximum likelihood estimate for a single observation, you can use **minimize the negative logarithm** to find the maximum likelihood of a Gaussian distribution.

We know that the Gaussian distribution has a good mathematical form under the negative logarithm. Consider any high-dimensional Gaussian distribution $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, whose probability density function is expanded as:

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

If you take a negative logarithm, it becomes:

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

Since the logarithmic function is monotonically increasing, maximizing the original function is equivalent to minimizing the negative logarithm. When the \mathbf{x} of the above formula is minimized, the first item has nothing to do with \mathbf{x} and can be omitted. Thus, as long as the quadratic term on the right side is minimized, the maximum likelihood estimate for the state is obtained. Substituting the observation model of SLAM is equivalent to seeking:

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

We have found that this equation is equivalent to a quadratic form that minimizes the noise term (ie, the error). This quadratic type is called **Mahalanobis distance**, also called **Markov distance**. It can also be thought of as the Euclidean distance (two norms) weighted by $\mathbf{Q}_{k,j}^{-1}$, where $\mathbf{Q}_{k,j}^{-1}$ is also called **information matrix**, which is the inverse of the Gaussian distribution covariance matrix.

Now let's consider the data for the batch time. It is usually assumed that the inputs and observations at each moment are independent of each other, which means

that the inputs are independent, the observations are independent, and the inputs and observations are independent. So we can factor the joint distribution:

$$P(\mathbf{z}, \mathbf{u} | \mathbf{x}, \mathbf{y}) = \prod_k P(\mathbf{u}_k | \mathbf{x}_{k-1}, \mathbf{x}_k) \prod_{k,j} P(\mathbf{z}_{k,j} | \mathbf{x}_k, \mathbf{y}_j), \quad (6.11)$$

This shows that we can handle motion and observation at all times independently. Define the error between each input and observation data and the model:

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

Then, minimizing the Mahalanobis distance between all time estimates and real readings is equivalent to finding the maximum likelihood estimate. Negative logarithms allow us to turn the product into a sum:

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

This gives a **Least Square Problem**, whose solution is equivalent to the maximum likelihood estimate of the state. Intuitively, due to the existence of noise, when we substitute the estimated trajectories and maps into the motion and observation equations of SLAM, they are not perfect. What should I do then? We perform **fine tuning** on the estimated state, which causes the overall error to drop. Of course, this decline is also limited, it will generally reach a **minimum value**. This is a typical nonlinear optimization process.

Looking closely at (??), we find that the least squares problem in SLAM has some specific structure:

- First, the objective function of the whole problem consists of a number of error (weighted) quadratic forms. Although the overall state variable has a high dimensionality, 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 make the whole problem have a form of **sparse**, which we will see in the backend chapter.
- Second, if you use Lie algebra to represent increments, the problem is the least squares problem of **unconstrained**. However, if the pose is described by the rotation matrix/transformation matrix, the constraint of the rotation matrix itself is introduced, that is, st $\mathbf{R}^T \mathbf{R}$ is added to the problem. = \mathbf{I} and $\det(\mathbf{R}) = 1$. This is a big condition. Additional constraints make optimization more difficult. This reflects the advantages of Lie algebra.
- Finally, we used a quadratic metric error. The distribution of errors will affect the weight of this item throughout the problem. For example, if a certain observation is very accurate, then the covariance matrix will be "small" and the information matrix will be "large", so this error term will have a higher weight in the whole problem. We will see later that there are some problems, but we will not discuss them at present.

Now, we show how to solve this least squares problem, which requires some **the basic knowledge of nonlinear optimization**. In particular, we will explore how it is solved for such a general unconstrained nonlinear least squares problem. In the following lectures, we will use the results of this lecture extensively to discuss its application in the SLAM front-end and back-end.

6.1.3 Example: Batch status estimation

I find it a better idea to give a simple example here. Consider a very simple discrete time system:

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

This can express a car that moves forward or backward along the x axis. The first formula is the equation of motion, u_k is the input, w_k is the noise, the second is the observation equation, and z_k is the measurement of the position of the car. Take the time $k = 1, \dots, 3$, and now I want to estimate the state based on the existing v, y . Let the initial state x_0 be known. Let's derive the maximum likelihood estimate for the batch state.

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

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

For each specific item, such as the equation of motion, we know that:

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

The observation equations are similar:

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

Based on these methods, we can actually solve the above batch state estimation problem. According to the previous description, the error variable can be constructed:

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

Then the objective function of least squares is:

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

In addition, this system is a linear system, and we can easily write it as a vector form. Define the vector $\mathbf{y} = [\mathbf{u}, \mathbf{z}]^T$, then write the matrix \mathbf{H} so that:

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

Then:

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

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

$$\mathbf{x}_{\text{map}}^* = \arg \min \mathbf{e}^T \Sigma^{-1} \mathbf{e}, \quad (6.22)$$

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

$$\mathbf{x}_{\text{map}}^* = (\mathbf{H}^T \Sigma^{-1} \mathbf{H})^{-1} \mathbf{H}^T \Sigma^{-1} \mathbf{y}. \quad (6.23)$$

6.2 nonlinear least squares

Let's consider a simple least squares problem first:

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

Where the argument $\mathbf{x} \in \mathbb{R}^n$, f is an arbitrary scalar nonlinear function $f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}$. Note that the coefficient $\frac{1}{2}$ is irrelevant. Some documents have this coefficient, and some documents do not. It does not affect the subsequent conclusions. Let's discuss how to solve such an optimization problem. 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 solve the optimal value of \mathbf{x} , just like the extreme value of the binary function:

$$\frac{dF}{d\mathbf{x}} = \mathbf{0}. \quad (6.25)$$

Solving this equation yields an extremum with a derivative of zero. They may be maximal, very small, or values at the saddle point, as long as they compare the size of their function values one by one. But is this equation easy to solve? This depends on the form of the f derivative. If f is a simple linear function, then the problem is a simple linear least squares problem, but some derivatives may be complex in form, making the equation not easy to solve. Solving this equation requires us to know the **global nature** of the objective function, which is usually not possible. For the least squares problem that is inconvenient to solve directly, we can use the **iteration** method to continuously update the current optimization variable from an initial value to make the objective function drop. The specific steps can be listed as follows:

1. gives an initial value of \mathbf{x}_0 .
2. For the k iteration, look for an increment of $\Delta\mathbf{x}_k$, making $\|f(\mathbf{x}_k + \Delta\mathbf{x}_k)\|_2^2$ reaches a minimum value.
3. Stop if $\Delta\mathbf{x}_k$ is small enough.
4. Otherwise, let $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$ return to step 2.

This turns the problem of solving the **derivative zero** is turned into a constant **find drop increment** $\Delta\mathbf{x}_k$ problem, as we will see, since it can be f Linearization, the calculation of the increment will be much simpler. When the function drops until the increment is very small, the algorithm is considered to converge and the objective function reaches a minimum value. In this process, the problem is how to find the increment of each iteration point, and this is a local problem, we only need to care about the local nature of f at the iteration value rather than the global nature. Such methods are widely used in areas such as optimization and machine learning.

Next we look at how to find this increment $\Delta\mathbf{x}_k$. This part of the knowledge is actually in the field of numerical optimization, let's look at some widely used results.

6.2.1 First-order and two-step methods

Now consider the k iteration, assuming we are looking for the increment $\Delta\mathbf{x}_k$ at \mathbf{x}_k , then the most intuitive way is to put the target function at *Taylor expansion near \mathbf{x}_k* :

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

Where $\mathbf{J}(\mathbf{x}_k)$ is $F(\mathbf{x})$ for the first derivative of \mathbf{x} (also called gradient, **Jacobi** Matrix [Jacobian]) *, \mathbf{H} is the second derivative (**Hessian** matrix), they all take values at \mathbf{x}_k , the reader should be in the university undergraduate multivariate calculus Learned in the course. We can choose to retain the first or second order of Taylor expansion, then the corresponding solution method is called a step or two step method. If you keep a step, then take the gradient in the inverse direction to ensure that the function drops:

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

Of course this is just a direction, usually we have to specify another step λ . The step size can be calculated according to certain conditions [?], and there are some empirical methods in machine learning, but we don't talk about it. This method is called **the steepest descent method**. Its intuitive meaning is very simple, as long as we move in the direction of the reverse gradient, the first-order (linear) approximation, the objective function must fall.

Note that the above discussion was made at the k iteration and does not involve other iteration information. So in order to simplify the symbol, we will omit the subscript k later, and think that these discussions are true for any iteration.

On the other hand, we can choose to retain the two-step information, where the incremental equation is:

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

Only zero, one, and quadratic items of $\Delta\mathbf{x}$ are included on the right side. Find the derivative of the right-hand equation about $\Delta\mathbf{x}$ and make it zero. [†], get:

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

Solving this linear equation yields an increment. This method is also known as **Newton method**.

We see that both the first-order and two-step methods are straightforward, as long as the function is Taylor-expanded around the iteration point and minimized for updates. In fact, we approximate the original function with a one- or two-time function, and then use the minimum value of the approximation function to guess the minimum value of the original function. As long as the original objective function locally looks like a first or quadratic function, such an algorithm is true (this is also the case in reality). However, these two methods also have their own problems. The steepest descent method is too greedy, easy to get out of the jagged route, but increases the number of iterations. Newton's law needs to calculate the \mathbf{H} matrix of the objective function, which is very difficult when the problem size is large. We usually tend to avoid the calculation of \mathbf{H} . For the general problem, some quasi-Newton methods can get better results, and for the least squares problem, there are several more practical methods: **Gauss-Newton's method** and **column Levernburg-Marquardt's method**.

* 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 those who are unfamiliar with matrix derivation, please refer to Appendix B.

6.2.2 Gaussian Newton method

The Gauss-Newton method is one of the easiest methods in the optimization algorithm. Its idea is to carry out the first-order Taylor expansion of $f(\mathbf{x})$. Note that this is not the target function $F(\mathbf{x})$ but $f(\mathbf{x})$, otherwise it becomes Newton.

$$f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + \mathbf{J}(\mathbf{x})^T \Delta\mathbf{x}. \quad (6.30)$$

Here $\mathbf{J}(\mathbf{x})^T$ is $f(\mathbf{x})$ for the derivative of \mathbf{x} , $n \times 1$ Column vector. According to the previous framework, the current goal is to find the increment $\Delta\mathbf{x}$, so that $\|f(\mathbf{x} + \Delta\mathbf{x})\|^2$ reaches the minimum. In order to find $\Delta\mathbf{x}$, we need to solve a linear least squares problem:

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

What is the difference between this equation and the previous one? Based on the extreme conditions, the above objective function is derived for $\Delta\mathbf{x}$ and the derivative is zero. To do this, first expand the squared term of the objective function:

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

Find the derivative of $\Delta\mathbf{x}$ and make it zero:

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

The following equations can be obtained:

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

This equation is about the **linear equations** of the variable $\Delta\mathbf{x}$, which we call **incremental equation**, also known as **Gauss Newton's equation** (Gauss-Newton) Equation or **Normal equation**. We define the coefficient on the left as \mathbf{H} and the right as \mathbf{g} , then the above formula becomes:

$$\mathbf{H} \Delta\mathbf{x} = \mathbf{g}. \quad (6.33)$$

It makes sense to write the left side as \mathbf{H} . Compared with the Newton method, the Gauss-Newton method uses $\mathbf{J}\mathbf{J}^T$ as the **approximation of the second-order Hessian matrix** in Newton's method, thus omitting the calculation of **The process of \mathbf{H}** . Solving the incremental equation is at the heart of the overall optimization problem. If we can solve the equation smoothly, the algorithmic steps of the Gauss-Newton method can be written as:

1. gives the initial value \mathbf{x}_0 .
2. For the k iteration, find the current Jacobian matrix $\mathbf{J}(\mathbf{x}_k)$ and the error $f(\mathbf{x}_k)$.
3. solves the incremental equation: $\mathbf{H} \Delta\mathbf{x}_k = \mathbf{g}$.
4. Stop if $\Delta\mathbf{x}_k$ is small enough. Otherwise, let $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$ return to

step 2.

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

To solve the incremental equation, we need to solve for \mathbf{H}^{-1} , which requires the \mathbf{H} matrix to be reversible, but the **\mathbf{J} calculated in the actual data. \mathbf{J}^T** is only semi-positive. That is to say, when using the Gauss-Newton method, it may appear that $\mathbf{J}\mathbf{J}^T$ is a singular matrix or an ill-condition, and the increment is stable. Poor sex, resulting in the algorithm does not converge. Intuitively, the local approximation of the original function at this point is not like a quadratic function. More seriously, even if we assume that \mathbf{H} is not singular, it is not morbid. If we find the step size $\Delta\mathbf{x}$ is too large, it will lead to the local approximation we use. Eqrefeq:approximation is not accurate enough, so we can't even guarantee its iterative convergence, even if it makes the objective function bigger.

Although the Gauss-Newton method has these shortcomings, it is still a simple and effective method for nonlinear optimization, which is worth learning. In the field of nonlinear optimization, quite a few algorithms can be reduced to variants of the Gauss-Newton method. These algorithms all rely on the idea of the Gauss-Newton method and correct its shortcomings through its own improvements. For example, some **line search method** added a step size of α , and after finding $\Delta\mathbf{x}$, further find α to make $\|f(\mathbf{x} + \alpha\Delta\mathbf{x})\|^2$ reaches the minimum, instead of simply making $\alpha = 1$.

The Levenberg-Marquart method corrects these problems to some 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, known as the **damped Newton Method**.

6.2.3 Levinburg-Marquart method

The approximate second-order Taylor expansion used in the Gauss-Newton method can only have a good approximation near the expansion point, so we naturally think that we should add a range to $\Delta\mathbf{x}$, called **Trust Region**. This range defines the circumstances under which a second-order approximation is valid. This type of method is also known as **Trust Region Method**. In the area of trust, we think that the approximation is valid; if this area is out, the approximation may be problematic.

So how do you determine the scope of this trust zone? A better method is based on the difference between our approximation model and the actual function: if the difference is small, the approximation effect is good, we expand the approximation range; conversely, if the difference is large, the approximation range is narrowed. We define an indicator ρ to describe how good or bad the approximation is:

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

The numerator of ρ is the value of the actual function drop, and the denominator is the value of the approximate model drop. If ρ is close to 1, the approximation is good. If ρ is too small, indicating that the actual reduced value is much less than the approximate reduced value, then the approximation is considered to be poor and the approximate range needs to be reduced. Conversely, if ρ is large, the actual decline is larger than expected, and we can zoom in on the approximate range.

So, we build a modified version of the nonlinear optimization framework, which will have a better effect than the Gauss Newton method:

1. gives the initial value \mathbf{x}_0 and the initial optimization radius μ .
2. For the k iteration, add a confidence region based on the Gauss-Newton method to solve:
$$\min_{\Delta\mathbf{x}_k} \frac{1}{2} \left\| f(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)^T \Delta\mathbf{x}_k \right\|^2, \quad \text{st} \quad \|\mathbf{D}\Delta\mathbf{x}_k\|^2 \leq \mu, \quad (6.35)$$

Where μ is the radius of the confidence zone and \mathbf{D} is the coefficient matrix, which will be explained later.
3. calculates ρ by the formula (??).
4. If $\rho > \frac{3}{4}$, set $\mu = 2\mu$.
5. If $\rho < \frac{1}{4}$, set $\mu = 0.5\mu$.
6. If ρ is greater than a certain threshold, it is considered to be approximate. Let $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$.
7. determines if the algorithm converges. If it does not converge, return to step 2, otherwise it ends.

Here, the multiples and thresholds of the approximate range expansion are empirical values and can be replaced with other values. In the formula (??), we limit the increment to a ball with a radius of μ , which is considered valid only in this ball. After taking \mathbf{D} , the ball can be seen as an ellipsoid. In Levinberg's optimization method, taking \mathbf{D} as a unit matrix \mathbf{I} is equivalent to directly constraining $\Delta\mathbf{x}_k$ in a ball. . Subsequently, Marquart proposed to take \mathbf{D} as a non-negative diagonal matrix—in practice, the square root of the diagonal element of $\mathbf{J}^T \mathbf{J}$ is usually used, so that The constraint range is larger in the small gradient.

In any case, in the Levenberg-Marquart optimization, we need to solve the sub-question (??) to get the gradient. This subproblem is an optimization problem with inequality constraints. We use the Lagrangian multiplier to put the constraint into the objective function to form a Lagrangian function:

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

Here λ is the Lagrange multiplier. Similar to the Gaussian Newton method, the Lagrangian function has zero derivative for $\Delta\mathbf{x}$, and its core is still a linear equation for calculating increments:

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

As you can see, the incremental equation has one more $\lambda \mathbf{D}^T \mathbf{D}$ compared to the Gauss-Newton method. If you consider its simplified form, ie $\mathbf{D} = \mathbf{I}$, then the

equivalent of solving *:

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

We see that when the parameter λ is small, \mathbf{H} is dominant, which means that the quadratic approximation model is better in this range, Levinburg-Marquart method is more close to the Gauss Newton method. On the other hand, when λ is large, $\lambda \mathbf{I}$ dominates, and the Levenberg-Marquart method is closer to a step-down method (ie, the steepest drop). This shows that the secondary approximation in the vicinity is not good enough. The Levenberg-Marquart method can solve the non-singular and ill-conditioned problems of the coefficient matrix of linear equations to a certain extent, providing a more stable and accurate increment $\Delta \mathbf{x}$.

In practice, there are many other ways to solve for increments, such as Dog-Leg[?]. What we've introduced here is just the most common and basic method, and the most used method in visual SLAM. In practical problems, we usually choose one of the Gauss Newton method or the Levinburg-Marquart method as the gradient descent strategy. When the problem is of a good nature, use Gauss Newton. If the problem is close to morbidity, use the Levenberg-Marquart method.

6.2.4

Since I don't want this book to become a mathematics textbook that is a headache, here are just two of the most common nonlinear optimization schemes, the Gauss-Newton method and the Levenberg-Marquart method. We have avoided many discussions on the nature of mathematics. If you are interested in optimization, you can read a book that specializes in numerical optimization (this is a big topic)[?]. The optimization methods represented by the Gauss Newton method and the Levenberg-Marquart 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 not only in visual SLAM, but also in other fields like deep learning. It is also one of the core methods for solving problems. The order method is mainly). We hope that readers will be able to understand more optimization algorithms based on their capabilities.

Perhaps you have discovered that both the Gauss Newton method and the Levinburg-Marquardt method need to provide the initial value of the variable when doing the optimization calculation. You may ask, can this initial value be set at will? of course not. In fact, all iterative solution solutions for nonlinear optimization require the user to provide a good initial value. Since the objective function is too complex, the change in the solution space is difficult to predict, and providing different initial values for the problem often leads to different calculation results. This situation is a common problem of nonlinear optimization: most algorithms are prone to fall into local minima. Therefore, no matter what kind of scientific problem, we should provide scientific basis for the initial value. For example, in the visual SLAM problem, we will use the algorithms such as ICP and PnP to provide

* strict readers may not be satisfied with the narrative here. Constraints on the original problem of the trust region In addition to the Lagrangian function deriving zero, the KKT condition 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 consider it as a penalty with λ as the weighting function on the objective function of the original problem. After each iteration, if the trust region condition is found to be unsatisfied, or the objective function increases, the weight of λ is increased until the trust region condition is finally met. Therefore, there are different interpretations of the L-M algorithm in theory, but in practice we only care about whether it works smoothly.

optimized initial values. In short, a good initial value is very important for the optimization problem!

Perhaps the reader will also have questions about the optimization mentioned above: How to solve linear incremental equations? We only talked about the incremental equation is a linear equation, but directly inverting the coefficient matrix is not a lot of calculations? of course not. In the visual SLAM algorithm, the dimension of $\Delta\mathbf{x}$ is often as large as several hundred or thousands. If you are doing large-scale visual 3D reconstruction, you will often find that this dimension can be easily reached. Hundreds of thousands or even higher. Inverting such a large matrix is unaffordable for most processors, so there are many numerical solutions for linear equations. There are different solutions in different fields, but there is almost no way to directly find the inverse of the coefficient matrix. We will use matrix decomposition to solve linear equations, such as QR, Cholesky and other decomposition methods. These methods are usually found in textbooks such as matrix theory, and we will not introduce them.

Fortunately, this matrix in the visual SLAM tends to have a specific sparse form, which provides the possibility to solve the optimization problem in real time. We will explain its principles in detail in Lecture 9. Using the sparse form of the elimination, decomposition, and finally the solution increment, will greatly improve the efficiency of the solution. In many open source optimization libraries, variables with more than 10,000 dimensions can be solved in a few seconds or less on a typical PC, and the reason is to use more advanced mathematical tools. The visual SLAM algorithm can now be implemented in real time, and thanks to the fact that the coefficient matrix is sparse. If the matrix is dense, I am afraid that optimizing such a visual SLAM algorithm will not be widely adopted by the academic community [? ? ?].

6.3 Practice: curve fitting problem

6.3.1 Handwritten Gauss Newton Method

Next we use a simple example to illustrate how to solve the least squares problem. We will demonstrate how to handwrite the Gauss-Newton method and then how to use the optimization library to solve this problem. For the same problem, these implementations will achieve 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 for x, y , and we 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:

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

Note that in this question, the variables to be estimated are a, b, c instead of x . In our program, we first generate the true value of x, y according to the model,

and then add the Gaussian distribution noise to the true value. Subsequently, the Gauss-Newton method is used to fit the parametric model from the noisy data. The definition error is:

$$E_i = y_i - \exp(ax_i^2 + bx_i + c), \quad (6.39)$$

Then we can find the derivative of each error term for the state variable:

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

Then $\mathbf{J}_i = \left[\frac{\partial e_i}{\partial a}, \frac{\partial e_i}{\partial b}, \frac{\partial e_i}{\partial c} \right]^T$, the incremental equation for the Gauss-Newton method is:

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

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

Listing 6.1: slambook2/ch6/gaussNewton.cpp

```

34
35     for (int i = 0; i < N; i++) {
36         double xi = x_data[i], yi = y_data[i]; // 0 0 0 0 i
37         double error = yi - exp(ae * xi * xi + be * xi + ce);
38         Vector3d J; // 0 0 0 0
39         J[0] = -xi * xi * exp(ae * xi * xi + be * xi + ce); // de/da
40         J[1] = -xi * exp(ae * xi * xi + be * xi + ce); // de/db
41         J[2] = -exp(ae * xi * xi + be * xi + ce); // de/dc
42
43         H += inv_sigma * inv_sigma * J * J.transpose();
44         b += -inv_sigma * inv_sigma * error * J;
45
46         cost += error * error;
47     }
48
49     // 0 0 0 0 0 Hx=b
50     Vector3d dx = H.ldlt().solve(b);
51     if (isnan(dx[0])) {
52         cout << "result is nan!" << endl;
53         break;
54     }
55
56     if (iter > 0 && cost >= lastCost) {
57         cout << "cost: " << cost << ">= last cost: " << lastCost << ", "
58             break." << endl;
59         break;
60     }
61
62     ae += dx[0];
63     be += dx[1];
64     ce += dx[2];
65
66     lastCost = cost;
67
68     cout << "total cost: " << cost << ", \t\update: " << dx.transpose
69         () <<
70         "\t\testimated params: " << ae << "," << be << "," << ce << endl;
71
72     chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
73     chrono::duration<double> time_used = chrono::duration_cast<chrono::
74         duration<double>>(t2 - t1);
75     cout << "solve time cost = " << time_used.count() << " seconds. " <<
76         endl;
77     cout << "estimated abc = " << ae << ", " << be << ", " << ce << endl;
78     return 0;
79 }

```

In this example, we demonstrate how to iteratively optimize a simple fitting problem. It's easy to see the entire optimization process with your own handwritten code. The program outputs the target function value and update amount for each iteration of the iteration, as follows:

Listing 6.2: Terminal output:

```

1 /home/xiang/Code/slambook2/ch6/cmake-build-debug/gaussNewton
2 Total cost: 3.19575e+06, update: 0.0455771 0.078164 -0.985329 estimated
   params: 2.04558,-0.921836,4.01467
3 Total cost: 376785, update: 0.065762 0.224972 -0.962521 estimated params:
   2.11134,-0.696864,3.05215
4 Total cost: 35673.6, update: -0.0670241 0.617616 -0.907497 estimated params
   : 2.04432,-0.0792484, 2.14465

```

```

5 Total cost: 2195.01, update: -0.522767 1.19192 -0.756452 estimated params:
  1.52155, 1.11267, 1.3882
6 Total cost: 174.853, update: -0.537502 0.909933 -0.386395 estimated params:
  0.984045, 2.0226, 1.00181
7 Total cost: 102.78, update: -0.0919666 0.147331 -0.0573675 estimated params
  : 0.892079, 2.16994, 0.944438
8 Total cost: 101.937, update: -0.00117081 0.00196749 -0.00081055 estimated
  params: 0.890908, 2.1719, 0.943628
9 Total cost: 101.937, update: 3.4312e-06 -4.28555e-06 1.08348e-06 estimated
  params: 0.890912, 2.1719, 0.943629
10 Total cost: 101.937, update: -2.01204e-08 2.68928e-08 -7.86602e-09
  estimated params: 0.890912, 2.1719, 0.943629
11 Cost: 101.937>= last cost: 101.937, break.
12 Solve time cost = 0.000212903 seconds.
13 Estimated abc = 0.890912, 2.1719, 0.943629

```

The objective function that is easy to see the whole problem approaches convergence after 9 iterations, and the update amount approaches zero. The final estimated value is close to the true value, as shown in ???. On my machine (my CPU is i7-8700), the optimization takes about 0.2 milliseconds. Below we try to use the optimization library to accomplish the same task.

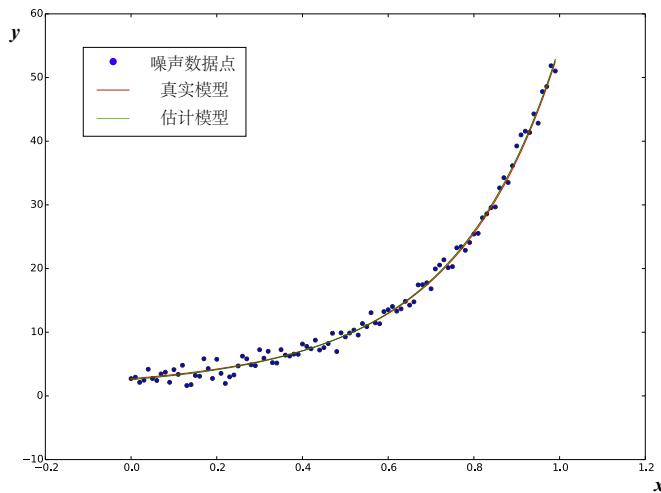


Figure 6-1: The result of curve fitting when noise $\sigma = 1$. The real model is very close to the estimated model.

6.3.2 Curve fitting with Ceres

This section introduces two C++ optimization libraries: the Keres library from Google, [?], and the graph-optimized g2o library [?]. Since the use of g2o also needs to introduce some knowledge about graph optimization, let us introduce Ceres first, then introduce some graph optimization theory, and finally talk about g2o. Since the optimization algorithm will appear in the following "visual odometer" and "backend", the reader must grasp the meaning of the optimization algorithm and understand the content of the program.

Ceres introduction

Google Ceres is a widely used solution library for least squares problems. In Ceres, as a user, we only need to define the optimization problem to be solved according to certain steps, and then give it to the solver for calculation. The most general form of the least squares problem solved by Ceres is as follows (least squares of kernel functions with boundaries):

$$\begin{aligned} \min_x \quad & \frac{1}{2} \sum_i \rho_i \left(\|f_i(x_{i_1}, \dots, x_{i_n})\|^2 \right) \\ \text{s.t.} \quad & l_j \leq x_j \leq u_j. \end{aligned} \quad (6.42)$$

In this problem, x_1, \dots, x_n are optimization variables, also known as **Parameter blocks**, and f_i is called **Cost function**, also known as disability. Residual blocks can also be understood as error terms in SLAM. l_j and u_j are the upper and lower limits of the j optimization variables. In the simplest case, take $l_j = -\infty, u_j = \infty$ (without limiting the boundaries of the optimization variables). At this point, the objective function consists of a number of squared terms after a **nuclear function** $\rho(\cdot)$ and then summed up to form a *. Similarly, ρ can be taken as an identity function, then 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 for Ceres to solve this problem for us, we need to do the following:

1. defines each parameter block. Parameter blocks are usually trivial vectors, but in SLAM they can also be defined as special structures such as quaternions and Lie algebras. If it is a vector, then we need to allocate a double array for each parameter block to store the value of the variable.
2. Then, define how the residual block is calculated. The residual block is usually associated with several parameter blocks, some custom calculations are performed on them, and the residual values are returned. After Ceres squares them, it is the value of the objective function. The
3. residual block also often needs to define the way Jacobi is calculated. In Ceres, you can use the "auto-derivation" feature provided by it, or you can manually specify the calculation process of Jacobi. If automatic derivation is to be used, the residual block needs to be written in a specific way: the calculation of the residual should be a parenthetical operator with a template. This is illustrated by an example.
4. Finally, add all the parameter blocks and residual blocks to the Problem object defined by Ceres, and call Solve function to solve. Before solving, we can pass in some configuration information, such as the number of iterations, termination conditions, etc., or use the default configuration.

Below, let's take a look at Ceres to solve the curve fitting problem and understand the optimization process.

* nuclear function. See ninth lecture for a detailed discussion.

Install Ceres

In order to use Ceres, we need to compile and install it. Ceres' github address is: <https://github.com/ceres-solver/ceres-solver>, but you can also use the Ceres in the 3rdparty directory of this book code directly, so that you will use exactly the same as me. version.

Like the library I encountered before, Ceres is a cmake project. First install its dependencies, you can use apt-get to install in Ubuntu, mainly some of the logs and test tools used by Google itself:

Listing 6.3: terminal input:

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

Then, go to the Ceres library directory, compile and install it using cmake. We have done this process many times, and we won't go into details 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, you can use Ceres for optimization calculations.

Use Ceres to fit curves

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

Listing 6.4: slambook/ch6/cheresCurveFitting.cpp

```
1 #include <iostream>
2 #include <opencv2/core/core.hpp>
3 #include <ceres/ceres.h>
4 #include <chrono>
5
6 Using namespace std;
7
8 // The calculation model of the cost function
9 Struct CURVE_FITTING_COST {
10     CURVE_FITTING_COST(double x, double y) : _x(x), _y(y) {}
11
12     // Calculation of residuals
13     Template<typename T>
14     Bool operator()(
15         Const T *const abc, // model parameter, 3D
16         T *residual) const {
17         // y-exp(ax^2+bx+c)
18         Residual[0] = T(_y) - ceres::exp(abc[0] * T(_x) * T(_x) + abc[1] *
19         T(_x) + abc[2]);
20         Return true;
21     }
22
23     Const double _x, _y; // x,y data
24 };
25
26 Int main(int argc, char **argv) {
27     Double ar = 1.0, br = 2.0, cr = 1.0; // true parameter value
28     Double ae = 2.0, be = -1.0, ce = 5.0; // estimate parameter value
29     Int N = 100; // data points
30     Double w_sigma = 1.0; // noise sigma value
31     Double inv_sigma = 1.0 / w_sigma;
32     Cv::RNG rng; // OpenCV random number generator
```

```

33     Vector<double> x_data, y_data; // data
34     For (int i = 0; i < N; i++) {
35         Double x = i / 100.0;
36         X_data.push_back(x);
37         Y_data.push_back(exp(ar * x * x + br * x + cr) + rng.gaussian(
38             w_sigma * w_sigma));
39     }
40
41     double abc[3] = {ae, be, ce};
42
43     // Build the least squares problem
44     Ceres::Problem problem;
45     for (int i = 0; i < N; i++) {
46         problem.AddResidualBlock( // Add an error term to the question
47             // Use automatic derivation, template parameters: error type,
48             // output dimension, input dimension, dimension must be consistent with
49             // the previous struct
50             New ceres::AutoDiffCostFunction<CURVE_FITTING_COST, 1, 3>(
51                 New CURVE_FITTING_COST(x_data[i], y_data[i])
52             ),
53             Nullpt, // kernel function, not used here, empty
54             Abc // parameter to be estimated
55         );
56     }
57
58     // Configure the solver
59     Ceres::Solver::Options options; // There are a lot of configuration
60     items to fill in
61     Options.linear_solver_type = ceres::DENSE_NORMAL_CHOLESKY; // How to
62     solve the incremental equation
63     Options.minimizer_progress_to_stdout = true; // output to cout
64
65     Ceres::Solver::Summary summary; // Optimization information
66     Chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
67     Ceres::Solve(options, &problem, &summary); // Start optimizing
68     Chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
69     Chrono::duration<double> time_used = chrono::duration_cast<chrono::
70         duration<double>>(t2 - t1);
71     Cout << "solve time cost = " << time_used.count() << " seconds. " <<
72         endl;
73
74     // output result
75     Cout << summary.BriefReport() << endl;
76     Cout << "estimated a,b,c = ";
77     For (auto a:abc) cout << a << " ";
78     Cout << endl;
79
80     Return 0;
81 }
```

The places in the program that need to be explained are commented. As you can see, we used OpenCV's noise generator to generate 100 Gaussian noise data, which was then fitted using Ceres. The Ceres usage demonstrated here is as follows:

1. Defines the class of the residual block. The method is to write a class (or structure) and define the () operator with the template parameter in the class, so that the class becomes a **fiction** (Functor)*. This way of defining makes Ceres call the a<double>() method on an object of that class (such as a)

* C++ terminology The class of the bracket operator is like a function when using the bracket operator.

just like calling a function. In fact, Ceres will pass the Jacobian matrix as a type parameter to this function, thus implementing the automatic derivation function. The double abc[3] in the

2. program is the parameter block, and for the residual block, we construct the CURVE_FITTING_COST object for each data, and then call AddResidualBlock to add the error term to the target function. Since optimization requires gradients, we have several choices: (1) using Ceres' Auto Derivation (Auto Diff); (2) using Numeric Diff*; (3) Deriving the analytical derivative form by itself and providing it to Ceres. Because automatic derivation is the most convenient in coding, we use automatic derivation.
3. Autoderivation requires specifying the dimensions of the error term and the optimization variable. The error here is scalar, the dimension is 1; the optimization is three quantities of a, b, c , and the dimension is 3. Therefore, the variable dimension is set to 1, 3 in the template parameter of the automatic derivation class AutoDiffCostFunction.
4. After setting the problem, call the Solve function to solve. You can configure (very detailed) optimization options in options. For example, you can choose to use Line Search or Trust Region, number of iterations, step size, and more. Readers can look at the definition of Options to see which optimization methods are available, and of course the default configuration is already available for a wide range of issues.

Finally, let's take a look at the experimental results. Call build/cheresCurveFitting to see the optimization results:

```

1 Iter cost cost_change lgradient lstep1 tr_ratio tr_radius ls_iter
   iter_time total_time
2 0 1.597873e+06 0.00e+00 3.52e+06 0.00e+00 0.00e+00 1.00e+04 0 2.10e-05 7.92
   e-05
3 1 1.884440e+05 1.41e+06 4.86e+05 9.88e-01 8.82e-01 1.81e+04 1 5.60e-05 1.05
   e-03
4 2 1.784821e+04 1.71e+05 6.78e+04 9.89e-01 9.06e-01 3.87e+04 1 2.00e-05 1.09
   e-03
5 3 1.099631e+03 1.67e+04 8.58e+03 1.10e+00 9.41e-01 1.16e+05 1 6.70e-05 1.16
   e-03
6 4 8.784938e+01 1.01e+03 6.53e+02 1.51e+00 9.67e-01 3.48e+05 1 1.88e-05 1.19
   e-03
7 5 5.141230e+01 3.64e+01 2.72e+01 1.13e+00 9.90e-01 1.05e+06 1 1.81e-05 1.22
   e-03
8 6 5.096862e+01 4.44e-01 4.27e-01 1.89e-01 9.98e-01 3.14e+06 1 1.79e-05 1.25
   e-03
9 7 5.096851e+01 1.10e-04 9.53e-04 2.84e-03 9.99e-01 9.41e+06 1 1.81e-05 1.28
   e-03
10 Solve time cost = 0.00130755 seconds.
11 Ceres Solver Report: Iterations: 8, Initial cost: 1.597873e+06, Final cost:
   5.096851e+01, Termination: CONVERGENCE
12 Estimated a,b,c = 0.890908 2.1719 0.943628

```

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

* automatic derivation is also done with numerical derivatives, but Because it is a template operation, it runs faster.

I hope that readers can get a general idea of how to use Ceres through this simple example. It has the advantage of providing an automatic derivation tool that eliminates the need to calculate a troublesome Jacobian matrix. Ceres' automatic derivation is implemented by template elements, which can be done automatically at compile time, but still be a numerical derivative. Most of the time, the book 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 issues outside of SLAM.

6.3.3 Curve fitting with g2o

The second practical part of this lecture will introduce another optimized library (mainly in the SLAM field): g2o (General Graphic Optimization, G²O). It is a library based on **map optimization**. Graph optimization is a theory that combines nonlinear optimization with graph theory, so before we use it, let's take a moment to introduce graph optimization theory.

Introduction to graph optimization theory

We have introduced the solution of nonlinear least squares. They are made up of the sum of many error terms. However, there is only one set of optimization variables and many error terms, and we don't know the **association** between them. For example, how many error terms does an optimization variable x_j exist in? Can we guarantee that the optimization of it makes sense? Further, we hope to be able to visually see the optimization problem **what it looks like**. Therefore, it involves the optimization of the graph.

Graph optimization is a way to represent optimization problems as **Graph**. The here is a graph in the sense of graph theory. A graph consists of several **Vertex**, and the **Edge** that connects these vertices. Furthermore, **vertex** is used to represent **optimized variable**, and **edge** is used to denote **error term**. Thus, for any of the above-mentioned forms of nonlinear least squares problem, we can construct a corresponding **graph**. We can simply call it , or use the definition in the probability map, called **Bayesian or factor map**.

?? is a simple example of graph optimization. We use triangles to represent the camera pose nodes, and circles to represent the landmark points, which form the vertices of the graph optimization; at the same time, the solid line represents the motion model of the camera, and the dashed line represents the observation model, which constitute the edge of the graph optimization. At this point, although the mathematical form of the whole problem is still like (??), now we can visually see the **structure** of the problem. If you want, you can also do **remove the isolated vertex** or the **priority of optimizing the number of edges** (or the vertices according to the terminology of the graph theory). But the most basic graph optimization is to use graph models to express a nonlinear least squares optimization problem. And we can use the certain properties of the graph model to make better optimizations.

G2o is a generic graph optimization library. "Universal" means that you can solve any least squares problem that can be represented as a graph optimization in g2o, obviously including the curve fitting problem discussed above. Let us demonstrate

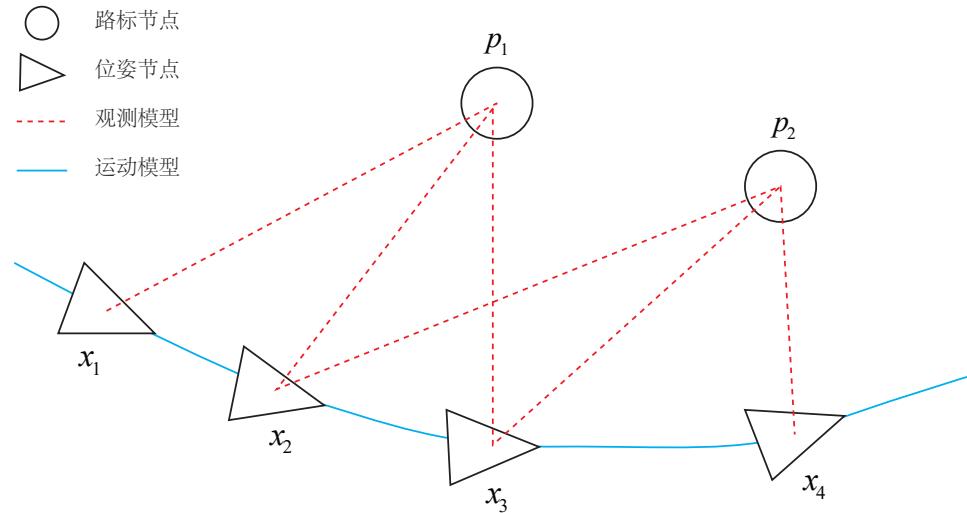


Figure 6-2: Figure optimization example.

this process.

6.3.4 g2o compilation and installation

Before using a library, we need to compile and install it. Readers should have experienced this process many times, and they are basically the same. For g2o, readers can download it from GitHub: <https://github.com/RainerKuemmerle/g2o> or from the third-party codebase provided with this book. Since g2o is still going to be 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 first install its dependencies (some dependencies coincide with Ceres):

Listing 6.5: terminal input:

```
1 Sudo apt-get install qt5-qmake qt5-default libqgviewer-dev-qt5
    libsuitesparse-dev libcxsparse3 libcholmod3
```

Then, compile and install g2o according to cmake. The description of the process is omitted here. After the installation is complete, the g2o header file will be located under /usr/local/g2o and the library file will be located under /usr/local/lib/. Now, let's revisit the curve fitting experiment in the Ceres routine and experiment again in g2o.

6.3.5 Use g2o to fit the curve

In order to use g2o, we first abstract the curve fitting problem into a graph optimization. In this process, just remember that the **node is the optimization variable and the edge is the error term**. The graph optimization problem of curve fitting can be drawn as ??.

In the curve fitting problem, the whole problem has only one vertex: the parameters of the curve model are a, b, c ; and each noisy data point constitutes an error

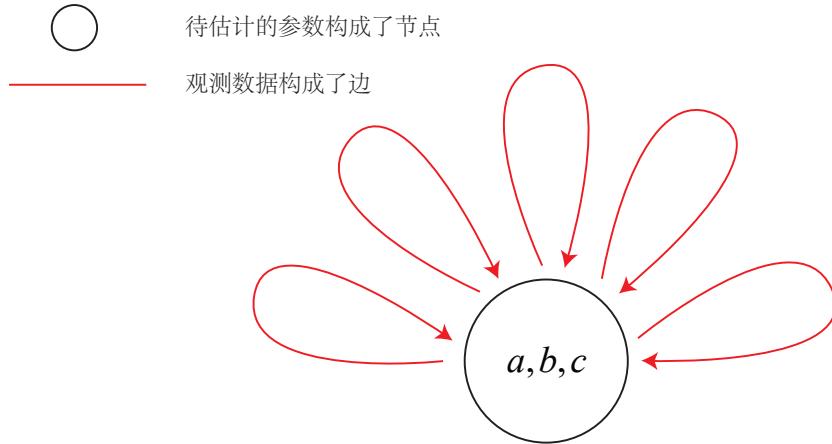


Figure 6-3: The curve is fitted to the corresponding graph optimization model. (There are some signs like Huawei.)

term, that is, the edge of the graph optimization. But the edges here are not the same as the ones we usually think. They are **Unary Edge**, ie **connects only one vertex** - because the whole graph has only one vertex. So in ?? , we can only paint it as if we are connected to ourselves. In fact, one edge of graph optimization can be connected to 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**, the whole picture is called **Hyper Graph***.

After figuring out the graph model, the next step is to build the model in g2o for optimization. As a g2o user, what we have to do mainly consists of the following steps:

1. defines the type of vertex and edge.
2. build diagram.
3. selects the optimization algorithm.
4. calls g2o for optimization and returns the result.

This part is very similar to Ceres, of course, the program will be written differently. Let's demonstrate the program below.

Listing 6.6: slambook/ch6/g2oCurveFitting.cpp

```

1 #include <iostream>
2 #include <g2o/core/g2o_core_api.h>
3 #include <g2o/core/base_vertex.h>
4 #include <g2o/core/base UnaryEdge.h>
5 #include <g2o/core/block_solver.h>
6 #include <g2o/core/optimization_algorithm_levenberg.h>
7 #include <g2o/core/optimization_algorithm_gauss_newton.h>
8 #include <g2o/core/optimization_algorithm_dogleg.h>
9 #include <g2o/solvers/dense/linear_solver_dense.h>
10 #include <Eigen/Core>
11 #include <opencv2/core/core.hpp>
```

* obviously I personally don't like some tricks. I am a naturalist.

```

12 #include <cmath>
13 #include <chrono>
14
15 Using namespace std;
16
17 // Vertex of the curve model, template parameters: optimized variable
18 // dimensions and data types
19 Class CurveFittingVertex : public g2o::BaseVertex<3, Eigen::Vector3d> {
20 Public:
21     EIGEN_MAKE_ALIGNED_OPERATOR_NEW
22
23     // reset
24     Virtual void setToOriginImpl() override {
25         _estimate << 0, 0, 0;
26     }
27
28     // update
29     Virtual void oplusImpl(const double *update) override {
30         _estimate += Eigen::Vector3d(update);
31     }
32
33     // save and read: leave blank
34     Virtual bool read(istream &in) {}
35     Virtual bool write(ostream &out) const {}
36 };
37
38 // Error model Template parameters: observation dimension, type, join
39 // vertex type
40 Class CurveFittingEdge : public g2o::BaseUnaryEdge<1, double,
41     CurveFittingVertex> {
42 Public:
43     EIGEN_MAKE_ALIGNED_OPERATOR_NEW
44
45     CurveFittingEdge(double x) : BaseUnaryEdge(), _x(x) {}
46
47     // Calculate the curve model error
48     Virtual void computeError() override {
49         Const CurveFittingVertex *v = static_cast<const
50             CurveFittingVertex *>(_vertices[0]);
51         Const Eigen::Vector3d abc = v->estimate();
52         _error(0, 0) = _measurement - std::exp(abc(0, 0) * _x * _x +
53             abc(1, 0) * _x + abc(2, 0));
54     }
55
56     // Calculate the Jacobian matrix
57     Virtual void linearizeOplus() override {
58         Const CurveFittingVertex *v = static_cast<const
59             CurveFittingVertex *>(_vertices[0]);
60         Const Eigen::Vector3d abc = v->estimate();
61         Double y = exp(abc[0] * _x * _x + abc[1] * _x + abc[2]);
62         _jacobianOplusXi[0] = -_x * _x * y;
63         _jacobianOplusXi[1] = -_x * y;
64         _jacobianOplusXi[2] = -y;
65     }
66
67     Virtual bool read(istream &in) {}
68     Virtual bool write(ostream &out) const {}
69 Public:
70     Double _x; // x value, y value is _measurement
71 };
72
73 Int main(int argc, char **argv) {
74     // Omit the data generation part of the code

```

```

69 // Build graph optimization, first set g2o
70 TypeDef g2o::BlockSolver<g2o::BlockSolverTraits<3, 1>>
71 BlockSolverType; // Each error term has an optimized variable
72 dimension of 3 and an error value dimension of 1
73 TypeDef g2o::LinearSolverDense<BlockSolverType::PoseMatrixType>
74 LinearSolverType; // Linear solver type
75
76 // Gradient descent method, which can be selected from GN, LM,
77 DogLeg
78 Auto solver = new g2o::OptimizationAlgorithmGaussNewton(
79     G2o::make_unique<BlockSolverType>(g2o::make_unique<
80     LinearSolverType>()));
81 G2o::SparseOptimizer optimizer; // graph model
82 optimizer.setAlgorithm(solver); // Set the solver
83 optimizer.setVerbose(true); // Turn on debug output
84
85 // Add vertices to the graph
86 CurveFittingVertex *v = new CurveFittingVertex();
87 V->setEstimate(Eigen::Vector3d(ae, be, ce));
88 V->setId(0);
89 optimizer.addVertex(v);
90
91 // Add a side to the picture
92 For (int i = 0; i < N; i++) {
93     CurveFittingEdge *edge = new CurveFittingEdge(x_data[i]);
94     Edge->setId(i);
95     Edge->setVertex(0, v); // Set the vertices of the connection
96     Edge->setMeasurement(y_data[i]); // Observe the value
97     Edge->setInformation(Eigen::Matrix<double, 1, 1>::Identity() *
98         1 / (w_sigma * w_sigma)); // Information matrix: the inverse of the
99     covariance matrix
100    optimizer.addEdge(edge);
101 }
102
103 // Perform optimization
104 Cout << "start optimization" << endl;
105 Chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
106 optimizer.initializeOptimization();
107 Optimizer.optimize(10);
108 Chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
109 Chrono::duration<double> time_used = chrono::duration_cast<chrono::
110 duration<double>>(t2 - t1);
111 Cout << "solve time cost = " << time_used.count() << " seconds. "
112 << endl;
113
114 // output optimized value
115 Eigen::Vector3d abc_estimate = v->estimate();
116 Cout << "estimated model: " << abc_estimate.transpose() << endl;
117
118 Return 0;
119 }

```

In this program, we derive the graph-optimized vertices and edges for curve fitting from g2o: CurveFittingVertex and CurveFittingEdge, which essentially extends the way g2o is used. These two classes are derived from the BaseVertex and BaseUnaryEdge classes, respectively. In the derived class, we rewrote important virtual functions:

1. Update function for vertex: oplusImpl. We know that the most important part of the optimization process is the calculation of the incremental Δx , which handles $x_{k+1} = x_k + \Delta T$ he process of x .

Readers may think that this is not something worth mentioning, because it is just a simple addition. Why doesn't g2o help us? In the curve fitting process, since the optimization variable (curve parameter) itself is located in **vector space**, this update calculation is indeed a simple addition. However, when the optimization variable is not in the vector space, for example, \mathbf{x} is the camera pose, it does not necessarily have an addition operation. At this point, you need to redefine the behavior of **incremental increments on existing estimates**. According to the explanation in Lecture 4, we may use left-multiply update or right-multiply update instead of direct addition.

2. The reset function of the vertex: `setToOriginImpl`. This is trivial, we can zero the estimate.
3. The error calculation function of the edge: `computeError`. This function takes the current estimate of the vertex to which the edge is connected and compares it to its observations based on the curve model. This is consistent with the error model in the least squares problem.
4. The Jacobian function of the side: `linearizeOplus`. In this function we calculate the Jacobian of each edge relative to the vertex.
5. Save and read disk functions: `read`, `write`. Since we don't want to do read/write operations, leave it blank.

After defining the vertices and edges, we declare a graph model in the main function, then add the vertices and edges to the graph model according to the generated noise data, and finally call the optimization function to optimize. G2o will give an optimized result:

Listing 6.7: Terminal output:

```

1 Start optimization
2 Iteration= 0 chi2= 376785.128234 time= 3.3299e-05 cumTime= 3.3299e-05 edges=
   = 100 schur= 0
3 Iteration= 1 chi2= 35673.566018 time= 1.3789e-05 cumTime= 4.7088e-05 edges=
   100 schur= 0
4 Iteration= 2 chi2= 2195.012304 time= 1.2323e-05 cumTime= 5.9411e-05 edges=
   100 schur= 0
5 Iteration= 3 chi2= 174.853126 time= 1.3302e-05 cumTime= 7.2713e-05 edges=
   100 schur= 0
6 Iteration= 4 chi2= 102.779695 time= 1.2424e-05 cumTime= 8.5137e-05 edges=
   100 schur= 0
7 Iteration= 5 chi2= 101.937194 time= 1.2523e-05 cumTime= 9.766e-05 edges=
   100 schur= 0
8 Iteration= 6 chi2= 101.937020 time= 1.2268e-05 cumTime= 0.000109928 edges=
   100 schur= 0
9 Iteration= 7 chi2= 101.937020 time= 1.2612e-05 cumTime= 0.00012254 edges=
   100 schur= 0
10 Iteration= 8 chi2= 101.937020 time= 1.2159e-05 cumTime= 0.000134699 edges=
    100 schur= 0
11 Iteration= 9 chi2= 101.937020 time= 1.2688e-05 cumTime= 0.000147387 edges=
    100 schur= 0
12 Solve time cost = 0.000919301 seconds.

```

We use the Gauss-Newton method for gradient descent, and after 9 iterations, we get the optimization results, which is similar to Ceres and handwritten Gauss-Newton method. From the perspective of running speed, our experimental conclusion is that handwriting is faster than g2o, and g2o is faster than Ceres. This is a generally intuitive experience, and versatility and efficiency are often contradictory. However, Ceres used automatic derivation in this experiment, and the solver configuration is not exactly the same as Gauss Newton, so it looks slower.

6.4

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 ways of gradient descent: the Gauss-Newton method and the Levenberg-Machalt method. In the practical part, the handwritten Gauss-Newton method, Ceres and g2o optimization libraries are used to solve the same curve fitting problem, and they are found to give similar results.

Since we haven't talked about the Bundle Adjustment in detail, we chose a simple but representative example of curve fitting in the practice section to demonstrate the general nonlinear least squares solution. In particular, if you use g2o to fit a curve, you must first convert the problem to a graph optimization, defining new vertices and edges. There are some roundabouts in this approach—the main purpose of g2o is not here. In contrast, it is natural for Ceres to define the error term for the curve fitting problem, because it is an optimization library itself. However, more problems in SLAM are how to solve an optimization problem with many camera poses and many spatial points. In particular, when the camera pose is expressed in Lie algebra, how the error term is calculated with respect to the derivative of the camera pose will be a matter worthy of detailed discussion. We will find in subsequent content that g2o provides a large number of ready-made vertices and edges, which is very convenient for camera pose estimation. In Ceres, we have to implement each Cost

Function ourselves, which is inconvenient.

In the two programs in the practice section, we did not calculate the derivative of the curve model for the three parameters, but used the numerical derivation of the optimization library, which makes the theory and code simple. The Ceres library provides automatic derivation and runtime numerical derivation based on template elements, while g2o only provides a way to derive runtime values. 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 that readers will be able to adapt to the extensive use of template programming by Ceres and g2o. Maybe it will look scary at first (especially Ceres sets the parentheses of the residual block and the code for the g2o initialization part), but after familiarity, it feels like this is natural and easy to extend. We will continue to discuss sparsity, kernel functions, Pose Graph and other issues in the SLAM backend.

Exercises

1. proves that the linear equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ when the coefficient matrix \mathbf{A} is overtimed, and the least squares solution is $\mathbf{x} = (\mathbf{B}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$.
2. investigates the advantages and disadvantages of the steepest descent method, the Newton method, the Gauss-Newton method, and the Levenberg-Marquart method. In addition to our Ceres library and g2o library, what other optimization libraries are there? (You may find some libraries on MATLAB.)
3. Why is the Gauss-Newton method's incremental equation coefficient matrix not correct? What is the geometric meaning of uncertainty? Why is the solution unstable in this case? What is
4. DogLeg? What are the similarities and differences between it and the Gauss Newton method and Levinburg-Marquart method? Please search for the relevant material *.
5. Read Ceres' teaching materials (<http://ceres-solver.org/tutorial.html>) to better understand its usage.
6. Read the documentation that comes with g2o. Can you read it? If you still can't fully understand it, please come back after the 10th and 11th lectures.
- 7.*Please change the curve model in the curve fitting experiment and optimize the experiment with Ceres and g2o. For example, more parameters and more complex models can be used.

* for example,<http://www.numerical.rl.ac.uk/people/nimg/course/lectures/raphael/lectures/lec7slides.pdf>.

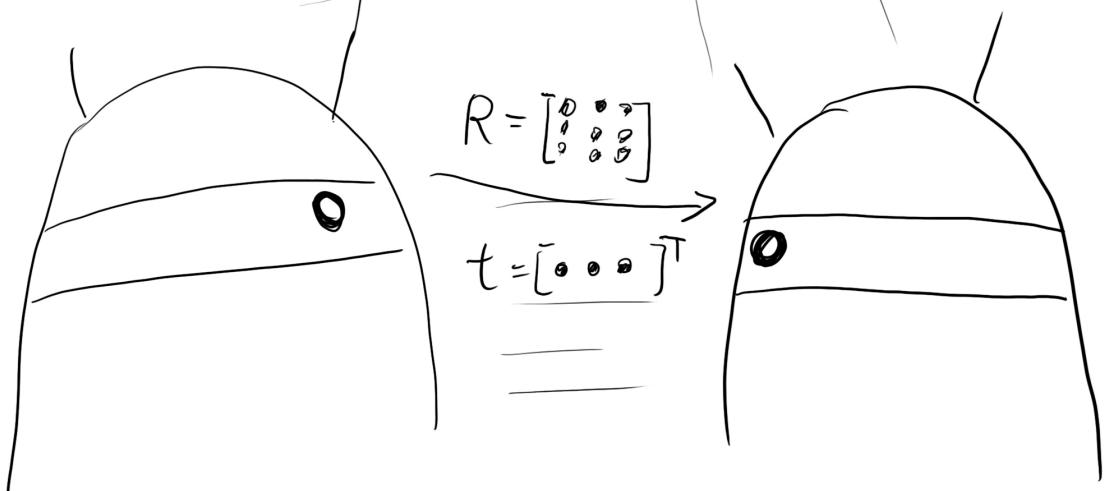
Chapter 7

visual odometer 1

main target

1. understands the meaning of image feature points, and grasps the method of extracting feature points and matching feature points in multiple images in a single image.
2. Understands the principle of polar geometry, using the constraints on polar geometry to recover the three-dimensional motion of the camera between images.
3. understands the PNP problem and solves the three-dimensional motion of the camera using the correspondence between the known three-dimensional structure and the image.
4. understands the ICP problem and uses the matching relationship of the point cloud to solve the three-dimensional motion of the camera.
5. understands how to obtain the three-dimensional structure of the corresponding points on a two-dimensional image by triangulation.

The specific form of the equation of motion and the equation of observation are introduced in the book, and the solution method based on nonlinear optimization is explained. Beginning with this lecture, we conclude the basics and step into the topic: in the order of the second lecture, we introduce four modules: visual odometer, optimized backend, loopback detection and map construction. This lecture and the next lecture mainly introduce the methods commonly used in two types of visual odometers: feature point method and optical flow method. In this lecture, we will introduce what is a feature point, how to extract and match feature points, and how to estimate camera motion based on paired feature points.



7.1 Feature point method

In the second lecture, we say that a SLAM system is divided into a front end and a back end, and the front end is also called a visual odometer (VO). The VO estimates a rough camera motion based on the information of the adjacent image, providing a good initial value to the back end. The VO algorithm is mainly divided into two major categories: **feature point method** and **direct method**. The front end based on the feature point method has long been considered (and until now) the mainstream method of visual odometers. It has the advantages of stability, insensitivity to light and dynamic objects, and is a relatively mature solution. In this lecture, we will start with the feature point method, learn how to extract and match the image feature points, and then estimate the camera motion and scene structure between the two frames to achieve a two-frame visual odometer. This type of algorithm is sometimes referred to as Two-view geometry.

7.1.1 feature point

The core issue with VO is **how to estimate camera motion based on images**. However, the image itself is a matrix of brightness and color, and it would be very difficult to consider motion estimation directly from the matrix level. Therefore, it is convenient to first: select **point** from the image to compare **representative**. These points remain the same after a small change in camera angle, so we can find the same point in each image. Then, based on 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 **roadmap** (Landmark). In visual SLAM, road signs refer to image features.

According to Wikipedia's definition, image features are a set of information related to computing tasks, and the computing tasks depend on the specific application [?]. In short, **features are another form of digital representation of image information**. A good set of features is critical to the ultimate performance on a given task, so researchers have spent a lot of time researching features over the years. Digital images are stored in a computer as a matrix of gray values, so the simplest, single image pixel is also a "feature." However, in visual odometers, we hope that **feature points remain stable after camera motion**, while gray values are severely affected by illumination, deformation, and object material, and vary greatly between images and are not stable enough. Ideally, when there are small changes to the scene and camera perspective, the algorithm can also determine from the image which places are the same point. Therefore, only the gray value is not enough, we need to extract feature points from the image.

The feature points are some **special places** in the image. Take ?? as an example. We can use the corners, edges and blocks in the image as representative places in the image. However, it is easier to pinpoint that the same corner appears in two images; the same edge is slightly more difficult, because the image is similar along the edge; the same block is the most difficult of. We found that the corners and edges in the image are more "special" than the pixel blocks, and the recognition between the different images is stronger. Therefore, an intuitive way to extract features is to identify corner points between different images and determine their correspondence. In this approach, corner points are so-called features. There are many corner extraction algorithms, such as Harris corner point [?], FAST corner point [?], GFTT corner point [?], etc. . Most of them are algorithms proposed

before 2000.

However, in most applications, pure corners still do not meet many of our needs. For example, where it looks like a corner from a distance, when the camera approaches, it may not appear as a corner. Or, when the camera is rotated, the appearance of the corners changes, and it is not easy to recognize that it is the same corner. To this end, researchers in the field of computer vision have designed many more stable local image features in years of research, such as the famous SIFT^[?], SURF^[?], ORB^[?], and so on. Compared to plain corners, these artificially designed feature points can have the following properties:

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

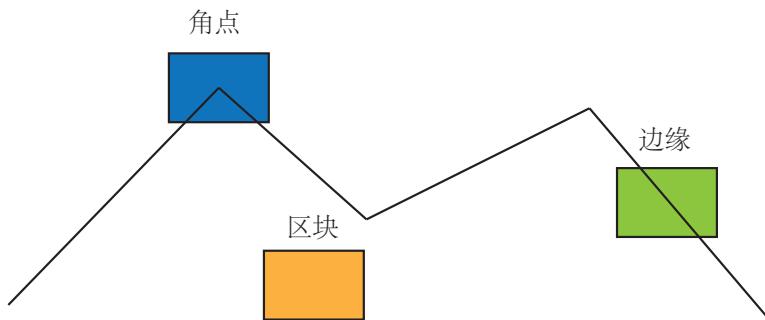


Figure 7-1: can be used as part of the image feature: corner points, edges, blocks.

The feature points consist of two parts: **key-point** and **descriptor**. For example, when we say "calculate SIFT feature points in an image", it means "extracting SIFT key points and calculating SIFT descriptors". The key point refers to the position of the feature point in the image, and some feature points also have information such as orientation, size, and the like. A descriptor is usually a vector that describes the information about the pixels around the key in a way that is artificially designed. The descriptor is designed according to the principle that "**similar appearance features should have similar descriptors**". Therefore, as long as the descriptors of the two feature points have similar distances in the vector space, they can be considered as the same feature points.

Historically, researchers have proposed many image features. They are somewhat accurate and still have similar expressions under camera motion and illumination changes, but correspondingly require a large amount of computation. Among them, SIFT (Scale-Invariant Feature Transform) is the most classic one. It fully considers the changes in illumination, scale, rotation, etc. that occur during image transformation, but it is followed by a large amount of computation. Since the extraction and matching of image features in the whole SLAM process is only one of many

links, up to now (2016), the CPU of ordinary PC can not calculate SIFT features in real time, and locate and map *. So in SLAM we rarely use this "luxury" image feature.

For other features, consider reducing the accuracy and robustness to improve the speed of the calculation. For example, the FAST key points are a feature point that is particularly fast (note the "keypoint" expression here, indicating that it has no descriptors), while the ORB (Oriented FAST and Rotated BRIEF) feature is currently very representative. Real-time image features. It improves the FAST detection [?] without directionality, and uses the extremely fast binary descriptor BURIEF[?] to make the whole image feature extraction greatly accelerate. According to the test described by the author in the paper, in the case of extracting about 1000 feature points simultaneously in the same image, the ORB takes about 15.3 ms, the SURF takes about 217.3 ms, and the SIFT takes about 5228.7 ms. It can be seen that the ORB maintains the characteristics of the rotation and scale invariance, and the speed is obviously improved. It is a good choice for the SLAM with high real-time requirements.

Most feature extractions have good parallelism and can be accelerated by devices such as GPUs. After GPU-accelerated SIFT, real-time computing requirements can be met. However, the introduction of GPU will bring about an increase in the cost of the entire SLAM. Whether the resulting performance improvement is sufficient to offset the computational cost of the system requires careful consideration by the system designer.

Obviously, there are a large number of feature points in the field of computer vision, which we cannot introduce in the book. In the current SLAM scheme, ORB is a good compromise between quality and performance. Therefore, we introduce the whole process of extracting features by ORB. If the reader is interested in feature extraction and matching algorithms, we recommend reading the related book [?].

7.1.2 ORB feature

The ORB feature is also composed of **keypoint** and **descriptor**. Its key point is called "Oriented FAST", which is an improved FAST corner point. We will introduce below on what is a FAST corner point. Its descriptor is called the BRIEF (Binary Robust Independent Elementary Feature). Therefore, extracting the ORB feature is divided into the following two steps:

1. FAST corner extraction: Find the "corner points" in the image. Compared with the original FAST, the main direction of the feature points is calculated in the ORB, and the rotation invariant characteristics are added for the subsequent BRIEF descriptors.
2. BRIEF Descriptor: Describes the surrounding image area from which the feature points were extracted in the previous step. The ORB has made some improvements to the BRIEF, mainly referring to the use of previously calculated direction information in the BRIEF.

The following describes FAST and BRIEF respectively.

* here refers to 30Hz real-time speed.

FAST keypoint

FAST is a kind of corner point, which mainly detects the obvious change of the gray level of local pixels, and is known for its fast speed. The idea is that if a pixel differs greatly from the pixels in the neighborhood (too bright or too dark), then it is more likely to be a corner. Compared to other corner detection algorithms, FAST only needs to compare the brightness of pixels, which is very fast. Its detection process is as follows (see ??):

1. picks the pixel p in the image, assuming its brightness is I_p .
2. sets a threshold of T (for example, 20% of I_p).
3. takes the pixel p as the center and selects 16 pixels on a circle with a radius of 3.
4. If the brightness of consecutive N points on the selected circle is greater than $I_p + T$ or less than $I_p - T$, the pixel p can be considered a feature point (N is usually taken as 12, which is FAST-12. Other commonly used N are 9 and 11, which are called FAST-9 and FAST-11, respectively).
5. loops through the above four steps, performing the same operation for each pixel.

In the FAST-12 algorithm, for more efficiency, a pre-test operation can be added to quickly eliminate most pixels that are not corner points. The specific operation is to directly detect the brightness of the first, fifth, 9, and 13 pixels on the neighborhood circle for each pixel. Only when 3 of these 4 pixels are greater than $I_p + T$ or less than $I_p - T$ at the same time, the current pixel may be a corner point, otherwise it should be directly excluded. Such pre-test operations greatly speed up corner detection. In addition, the original FAST corners often appear to be "stacked". Therefore, after the first pass detection, it is also necessary to use non-maximal suppression to preserve only the corner points of the response maxima in a certain area, thereby avoiding the problem of corner point concentration.

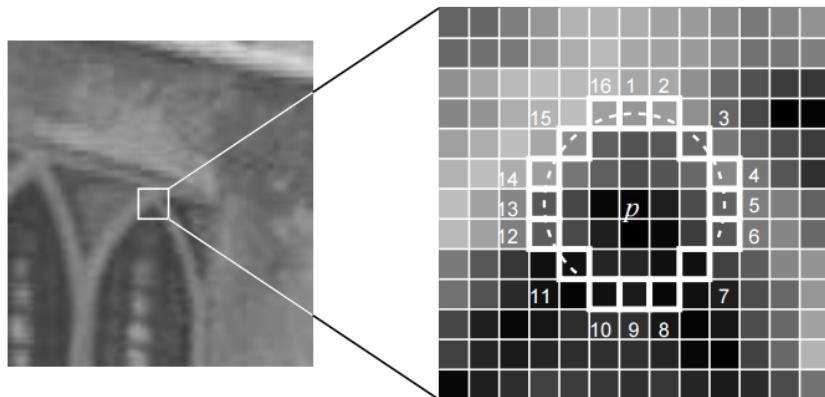


Figure 7-2: FASTFeatures[?].

The calculation of FAST feature points is only to compare the difference in brightness between pixels, so the speed is very fast, but it also has the disadvantages

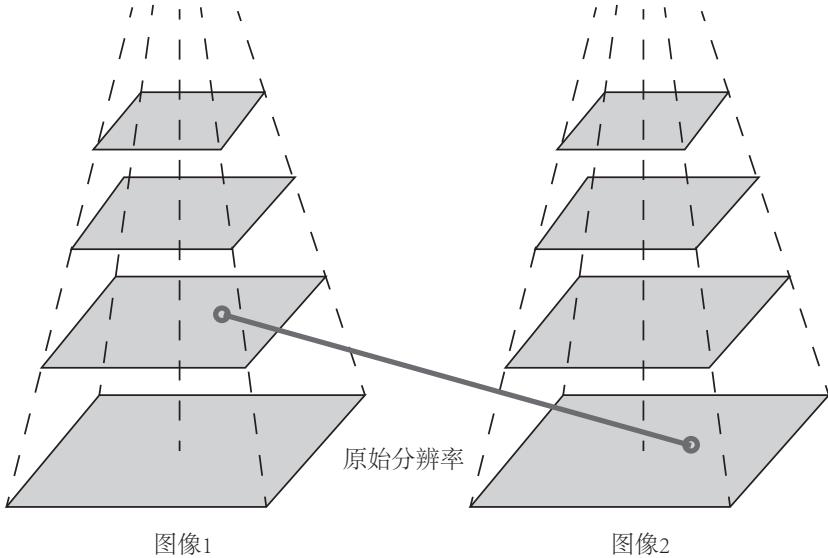


Figure 7-3: Use pyramids to match images at different zoom ratios.

of less repeatability and uneven distribution. In addition, the FAST corner points do not have direction information. At the same time, since it is fixed to take a circle with a radius of 3, there is a scale problem: where it looks like a corner point in the distance, it may not be a corner point when it is close. For the weakness of the FAST corner point without directionality and scale, the ORB adds a description of the scale and rotation. Scale invariance by constructing an image pyramid *, and detect corner points on each layer of the pyramid to achieve. The rotation of the feature is achieved by the Intensity Centroid.

A common method of processing in the calculation of graphs in the pyramid. See ?? for a schematic diagram. The bottom layer of the pyramid is the original image. Each time we go up one level, we scale the image at a fixed magnification so that we have images of different resolutions. Smaller images can be seen as scenes that are seen from a distance. In the feature matching algorithm, we can match the 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 image gray center of mass near the feature point. The so-called centroid refers to the center of the weight of the image block as the weight. The specific steps are as follows: [?]:

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

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

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

2. The moment of the image block can be found by the moment:

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

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

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

Through the above method, the FAST corner point has a description of scale and rotation, which greatly enhances the robustness of its representation between different images. So in the ORB, this improved FAST is called Oriented FAST.

BRIEF descriptor

After extracting the Oriented FAST key points, we calculate their descriptors for each point. The ORB uses a modified BRIEF feature description. Let us first introduce what is the BRIEF.

BRIEF is a **binary** descriptor whose description vector consists of a number of 0s and 1s, where 0 and 1 encode the size relationship of two random pixels near the key (such as p and q). : If p is larger than q , take 1 and vice versa. If we take 128 such p, q , we end up with a 128-dimensional vector of 0,1 [?]. BRIEF uses a random selection of comparisons, very fast, and because of the use of binary expressions, it is also very convenient to store, suitable for real-time image matching. The original BRIEF specifier does not have rotational invariance and is therefore easily 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.

Thanks to rotation and scaling, the ORB still performs well under the transformation of translation, rotation and scaling. At the same time, the combination of FAST and BRIEF is also very efficient, making ORB features very popular in real-time SLAM. We show the result of extracting an ORB from an image in ?? . Here's how to match features between different images.

7.1.3 feature matching

Feature matching (shown in ??) is a critical step in visual SLAM. Broadly speaking, feature matching solves the data association in SLAM, which determines the current view. Correspondence between the road sign and the road sign seen before. By accurately matching the image and the image or the descriptor between the image and the map, we can alleviate a lot of burden for subsequent gesture estimation, optimization and other operations. However, due to the local characteristics of image features, mismatching is widespread and has not been effectively solved for a long time. It has become a major bottleneck in the performance improvement of visual SLAM. Part of the reason is that there are often a large number of repeating textures in the scene, making the descriptions very similar. In this case, it is very difficult to solve the mismatch using only the local features.

However, let's look at the correct match first, and then go back and discuss the mismatch problem after doing the experiment. Consider an image of two moments.

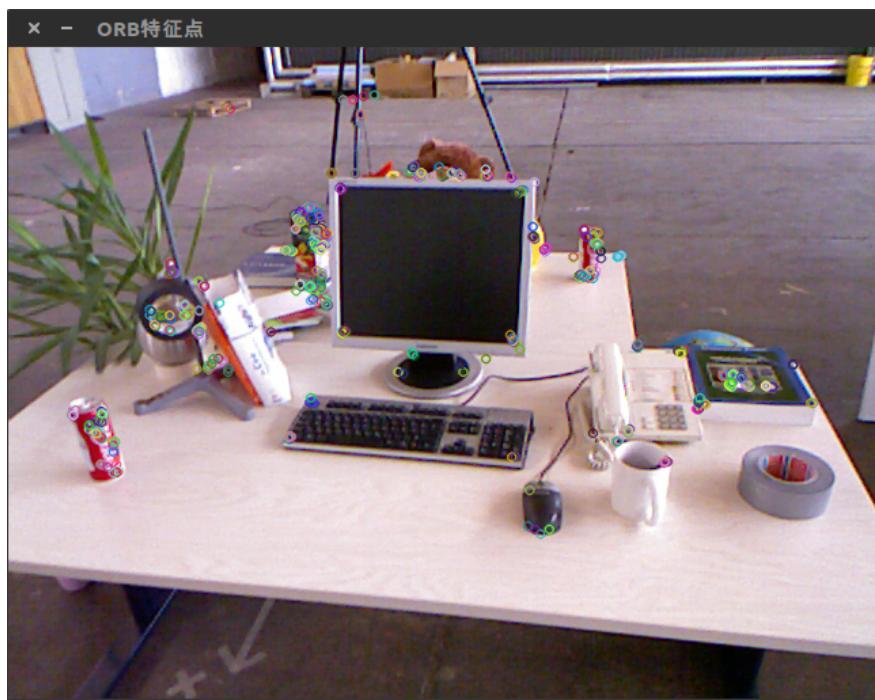


Figure 7-4: The result of the ORB feature point detection provided by OpenCV.



Figure 7-5: Feature matching between two frames of images.

If the feature point x_t^m is extracted in the image I_t , $m = 1, 2, \dots, M$ is extracted in the image I_{t+1} . Feature points x_{t+1}^n , $n = 1, 2, \dots, N$, how do you find the correspondence between these two collection elements? The simplest feature matching method is **Brute-Force Matcher**. That is, for each feature point x_t^m and all x_{t+1}^n measure the distance of the descriptor, and then sort, taking the nearest one as the matching point. The descriptor distance represents the **similarity degree** between the two features, but in practice, different distance metric norms can be taken. For the description of the floating point type, measure with Euclidean distance. For binary descriptors (such as the BRIEF), we often use the Hamming distance as a measure - the Hamming distance between two binary strings, which refers to its **different digits number**.

However, when the number of feature points is large, the amount of computation of the brute force matching method will become very large, especially when it is desired to match a certain frame and a map. This does not meet our real-time needs in SLAM. At this point, the **Fast Approximate Nearest Neighbor (FLANN)** algorithm is more suitable for the case of a large number of matching points. Since the theory of these matching algorithms has matured and the implementation has been integrated into OpenCV, its technical details are not described here. Interested readers can refer to the reading [?].

7.2 Practice: Feature Extraction and Matching



Figure 7-6: Two frames of images used by the experiment.

OpenCV has integrated most of the mainstream image features that we can easily call. Let's complete two experiments: In the first experiment, we demonstrate the feature matching of ORB using OpenCV. In the second experiment, we demonstrate how to handwrite a simple ORB feature according to the principle described above. Through the process of handwriting, the reader can understand the calculation process of the ORB more clearly and analogize it to other features.

7.2.1 OpenCV ORB feature

First we call OpenCV to extract and match the ORB. I have prepared two images for this experiment, located at 1.png and 2.png under slambook2/ch7/, as shown by ?? . They are two images from the public dataset [?] and we see a tiny movement in the camera. This section demonstrates how to extract and match ORB features. In the next section, we will demonstrate how to use the matching results to estimate camera motion.

The following program demonstrates how to use ORB:

Listing 7.1: slambook2/ch7/orb_cv.cpp

```

1 #include <iostream>
2 #include <opencv2/core/core.hpp>
3 #include <opencv2/features2d/features2d.hpp>
4 #include <opencv2/highgui/highgui.hpp>
5 #include <chrono>
6
7 using namespace std;
8 using namespace cv;
9
10 int main(int argc, char **argv) {
11     if (argc != 3) {
12         cout << "usage: feature_extraction img1 img2" << endl;
13         return 1;
14     }
15     //— □ □ □ □
16     Mat img_1 = imread(argv[1], CV_LOAD_IMAGE_COLOR);
17     Mat img_2 = imread(argv[2], CV_LOAD_IMAGE_COLOR);
18     assert(img_1.data != nullptr && img_2.data != nullptr);
19
20     //— □ □ □
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-
26         Hamming");
27
28     //— □ □ □ □ : Oriented FAST □ □ □ □
29     chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
30     detector->detect(img_1, keypoints_1);
31     detector->detect(img_2, keypoints_2);
32
33     //— □ □ □ □ □ □ □ □ □ : BRIEF □ □ □
34     descriptor->compute(img_1, keypoints_1, descriptors_1);
35     descriptor->compute(img_2, keypoints_2, descriptors_2);
36     chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
37     chrono::duration<double> time_used = chrono::duration_cast<chrono::
38         duration<double>>(t2 - t1);
39     cout << "extract ORB cost = " << time_used.count() << " seconds. " <<
40         endl;
41
42     Mat outimg1;
43     drawKeypoints(img_1, keypoints_1, outimg1, Scalar::all(-1),
44         DrawMatchesFlags::DEFAULT);
45     imshow("ORB features", outimg1);
46
47     //— □ □ □ □ □ □ □ □ □ : □ □ □ □ □ □ □ □ □ □ BRIEF Hamming □ □
48     vector<DMatch> matches;
49     t1 = chrono::steady_clock::now();
50     matcher->match(descriptors_1, descriptors_2, matches);
51     t2 = chrono::steady_clock::now();
52     time_used = chrono::duration_cast<chrono::duration<double>>(t2 - t1);
53     cout << "match ORB cost = " << time_used.count() << " seconds. " <<
54         endl;
55
56     //— □ □ □ □ □ □ □ □ □ :
57     // □ □ □ □ □ □ □ □ □ □
58     auto min_max = minmax_element(matches.begin(), matches.end(),
59         [] (const DMatch &m1, const DMatch &m2) { return m1.distance < m2.
60             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 //—————
62 std::vector<DMatch> good_matches;
63 for (int i = 0; i < descriptors_1.rows; i++) {
64     if (matches[i].distance <= max(2 * min_dist, 30.0)) {
65         good_matches.push_back(matches[i]);
66     }
67 }
68
69 //—————
70 Mat img_match;
71 Mat img_goodmatch;
72 drawMatches(img_1, keypoints_1, img_2, keypoints_2, matches, img_match)
73     ;
74 drawMatches(img_1, keypoints_1, img_2, keypoints_2, good_matches,
75     img_goodmatch);
76 imshow("all matches", img_match);
77 imshow("good matches", img_goodmatch);
78 waitKey(0);
79 }
```

Run this program (you need to enter two image locations) and the output will be output:

Listing 7.2: terminal input:

```

1 % build/orb_cv 1.png 2.png
2 Extract ORB cost = 0.0229183 seconds.
3 Match ORB cost = 0.000751868 seconds.
4 — Max dist : 95.000000
5 — Min dist : 4.000000
```

?? shows the results of the routine. We see a large number of mismatches in unfiltered matches. After a single screening, the number of matches has been reduced a lot, but most matches are correct. Here, the basis of the screening is **Hamming distance is less than twice the minimum distance**, which is an empirical method of engineering, not necessarily theoretical basis. However, although the correct match can be filtered out in the sample image, we still cannot guarantee that the match obtained in all other images is correct. Therefore, in the latter motion estimation, it is also necessary to use an algorithm for removing mismatch. On my machine, the ORB extraction took 22.9 milliseconds (two images) and the matching took 0.75 milliseconds, showing that most of the computation was spent on feature extraction.

7.2.2 Handwritten ORB feature

Below we demonstrate the method of handwritten ORB features. This part of the code is more, the book only shows the core part of the code, the rest of the surrounding code, please readers from the code base.

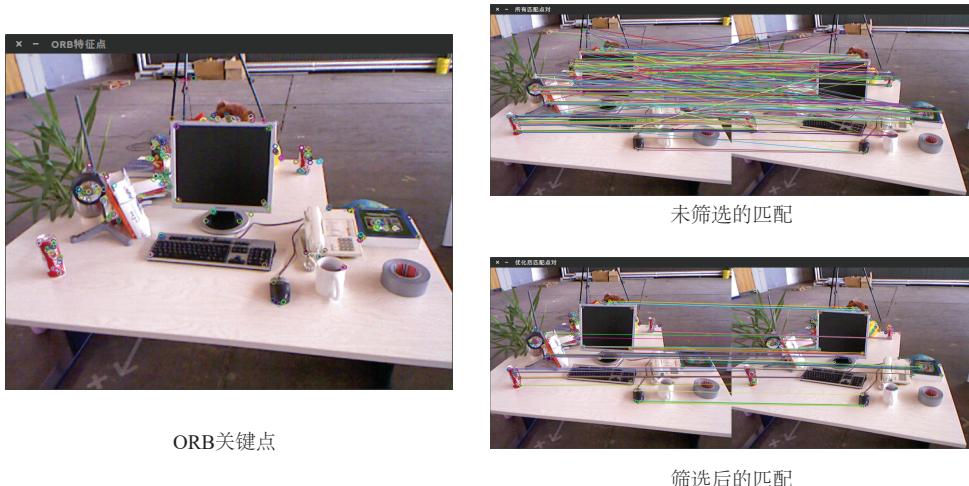


Figure 7-7: Feature extraction and matching results.

Listing 7.3: slambook2/ch7/orb_self.cpp

```

37         cv::Point2f p(ORB_pattern[idx_pq * 4], ORB_pattern[idx_pq *
38             4 + 1]);
39         cv::Point2f q(ORB_pattern[idx_pq * 4 + 2], ORB_pattern[
40             idx_pq * 4 + 3]);
41
42         // rotate with theta
43         cv::Point2f pp = cv::Point2f(cos_theta * p.x - sin_theta *
44             p.y, sin_theta * p.x + cos_theta * p.y) + kp.pt;
45         cv::Point2f qq = cv::Point2f(cos_theta * q.x - sin_theta *
46             q.y, sin_theta * q.x + cos_theta * q.y) + kp.pt;
47         if (img.at<uchar>(pp.y, pp.x) < img.at<uchar>(qq.y, qq.x))
48             {
49                 d |= 1 << k;
50             }
51         desc[i] = d;
52     }
53     descriptors.push_back(desc);
54 }
55
56 // brute-force matching
57 void BfMatch(
58     const vector<DescType> &desc1, const vector<DescType> &desc2, vector<cv
59         ::DMatch> &matches) {
60     const int d_max = 40;
61
62     for (size_t i1 = 0; i1 < desc1.size(); ++i1) {
63         if (desc1[i1].empty()) continue;
64         cv::DMatch m{i1, 0, 256};
65         for (size_t i2 = 0; i2 < desc2.size(); ++i2) {
66             if (desc2[i2].empty()) continue;
67             int distance = 0;
68             for (int k = 0; k < 8; k++) {
69                 distance += _mm_popcnt_u32(desc1[i1][k] ^desc2[i2][k]);
70             }
71             if (distance < d_max && distance < m.distance) {
72                 m.distance = distance;
73                 m.trainIdx = i2;
74             }
75         }
76         if (m.distance < d_max) {
77             matches.push_back(m);
78         }
79     }
80 }
```

In this demo we only show the calculation code and matching code of the ORB. In the calculation, we use a 256-bit binary description, which corresponds to eight 32-bit unsigned int data, and express it as DescType with typedef. Then, we calculate the angle of the FAST feature point according to the principle described above, and then use the angle to calculate the descriptor. In this code, the principle of trigonometric functions avoids the complicated arctan and sin, cos calculations, thus achieving an accelerated effect. In the BfMatch function, we also use the `_mm_popcnt_u32` function in the SSE instruction set to calculate the number of 1 in an unsigned int variable to achieve the effect of calculating the Hamming distance. The running result of this program is as follows, and the matching result is as shown in ??:

Listing 7.4: Terminal output:

```

1 Bad/total: 43/638
2 Bad/total: 8/595
3 Extract ORB cost = 0.00390721 seconds.
4 Match ORB cost = 0.000862984 seconds.
5 Matches: 51

```

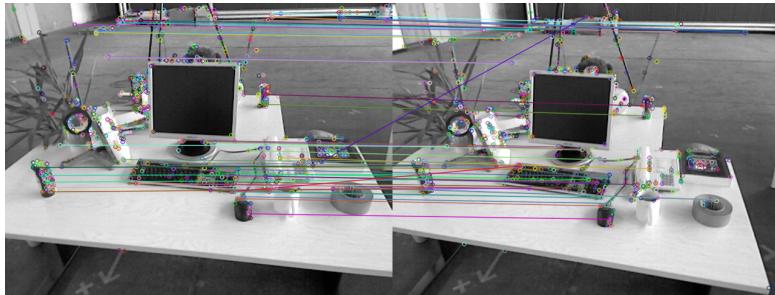


Figure 7-8: match result

It can be seen that in this program, the extraction of the ORB takes only 3.9 milliseconds, and the matching takes only 0.86 milliseconds. We accelerated the extraction of ORB by 5.8 times with some simple algorithm modifications. Please note that compiling this program requires your CPU to support the SSE instruction set, which should be supported on most modern home CPUs. If we can further parallelize the extracted features, the algorithm can also have room for acceleration.

7.2.3 Calculate camera motion

We already have matching pairs of points. Next, we need to estimate the camera's motion based on the point pairs. Here the situation has changed due to the different principles of the camera:

1. When the camera is monocular, we only know the 2D pixel coordinates, so the problem is to estimate the motion based on **two sets of 2D points**. This problem is solved with **polar geometry**.
2. When the camera is binocular, RGB-D, or by some way to get the distance information, then the problem is to estimate the motion according to **two sets of 3D points**. This problem is usually solved by ICP.
3. If a group is 3D and a group is 2D, ie we get some 3D points and their projection position in the camera, we can also estimate the camera's motion. This problem is solved by **PnP**.

Therefore, the following sections describe camera motion estimation in these three scenarios. We will start with the 2D–2D case with the least information, to see how it solves, and what are the troublesome problems in the solution process.

7.3 2D–2D: opposite geometry

7.3.1 polar constraint

Now, let's say we get a pair of paired feature points from the two images, as shown by ?? . If there are a number of such matching points, the motion of the camera between the two frames can be recovered by the correspondence of these two-dimensional image points. How many pairs of "several pairs" are there? We will introduce it below. Let's first look at the geometric relationship between the matching points in the two images.

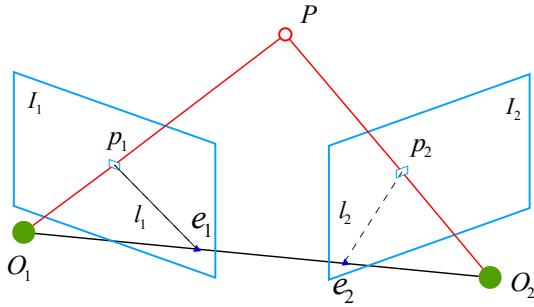


Figure 7-9:

Taking ?? as an example, we want to find the motion between two frames of images I_1, I_2 , and set the motion from the first frame to the second frame as \mathbf{R}, \mathbf{t} . The two camera centers are O_1, O_2 . Now, consider that there is a feature point p_1 in I_1 , which corresponds to the feature point p_2 in I_2 . We know that the two are obtained by feature matching. If the matches are correct, they are indeed **projections of the same spatial point on the two imaging planes**. Here some terms are needed to describe the geometric relationship between them. First, connect $\overrightarrow{O_1p_1}$ and connect $\overrightarrow{O_2p_2}$ to intersect at P in 3D space. At this time, point O_1, O_2, P three points can determine a plane, called **Epipolar plane**. The intersection of the O_1O_2 line and the image plane I_1, I_2 is e_1, e_2 . e_1, e_2 is called **Epipoles**, and O_1O_2 is called **Baseline**. We call the intersection line between the polar plane and the two image planes I_1, I_2 , and l_1 is **Epipolar line**.

Intuitively, from the perspective of the first frame, the ray $\overrightarrow{O_1p_1}$ is **the spatial position at which a pixel may appear**—because all points on the ray are projected to the same pixel. At the same time, if you don't know the position of P , then when we look at the second image, connect $\overrightarrow{e_2p_2}$ (that is, the polar line in the second image) is P possible. The position of the projected projection, which is the projection of the ray $\overrightarrow{O_1p_1}$ in the second camera. Now, since we determine the pixel position of p_2 by feature point matching, we can infer the spatial position of P and the motion of the camera. To remind the reader, **this is thanks to the correct feature matching**. Without feature matching, we can't determine where p_2 is in the polar line. At that time, you must search on the polar line to get the correct match, which will be mentioned in the 12th lecture.

Now let's look at the geometric relationship here from an algebraic perspective.

In the coordinate system of the first frame, the spatial position of P is set to

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

According to the pinhole camera model introduced in the fifth lecture, we know that the pixel position of two pixels $\mathbf{p}_1, \mathbf{p}_2$ is

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

Here \mathbf{K} is the camera internal reference matrix, \mathbf{R} , and \mathbf{t} is the camera motion of the two coordinate systems. Specifically, the \mathbf{R}_{21} and \mathbf{t}_{21} are calculated here because they convert the coordinates in the first coordinate system to the second coordinate system. If we want, we can also write them in Lie algebra.

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 are projection relationships, which are equal in the sense of homogeneous coordinates. We call this equality relationship **equal up to a scale**, which is recorded as:

$$s\mathbf{p} \simeq \mathbf{p}. \quad (7.2)$$

Then, the above two projection relationships can be written as:

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

Now, take:

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

Here $\mathbf{x}_1, \mathbf{x}_2$ is the coordinate on the normalized plane of two pixels. Substituting the above formula, you get:

$$\mathbf{x}_2 \simeq \mathbf{R}\mathbf{x}_1 + \mathbf{t}. \quad (7.5)$$

Both sides are left by \mathbf{t}^\wedge . Recall the definition of \wedge , which is equivalent to doing the outer product with \mathbf{t} on both sides:

$$\mathbf{t}^\wedge \mathbf{x}_2 \simeq \mathbf{t}^\wedge \mathbf{R}\mathbf{x}_1. \quad (7.6)$$

Then, both sides are simultaneously multiplied by \mathbf{x}_2^T :

$$\mathbf{x}_2^T \mathbf{t}^\wedge \mathbf{x}_2 \simeq \mathbf{x}_2^T \mathbf{t}^\wedge \mathbf{R}\mathbf{x}_1. \quad (7.7)$$

Looking at the left side of the equation, $\mathbf{t}^\wedge \mathbf{x}_2$ is a vector that is perpendicular to both \mathbf{t} and \mathbf{x}_2 . When you make it inner with \mathbf{x}_2 , you get 0. Since the left side of the equation is strictly zero, multiplying by any non-zero constant is also zero, so we can write \simeq as the usual equal sign. So we got a succinct expression:

$$\mathbf{x}_2^T \mathbf{t}^\wedge \mathbf{R}\mathbf{x}_1 = 0. \quad (7.8)$$

Re-substitute $\mathbf{p}_1, \mathbf{p}_2$, with:

$$\mathbf{p}_2^T \mathbf{K}^{-T} \mathbf{t}^\wedge \mathbf{R} \mathbf{K}^{-1} B \mathbf{m} \mathbf{p}_1 = 0. \quad (7.9)$$

These two expressions are called **polar constraint**, which is known for its simplicity. Its geometric meaning is O_1, P, O_2 . Both the translation and the rotation are included in the polar constraint. We refer to the middle part as two matrices:

the Fundamental Matrix \mathbf{F} and the Essential Matrix \mathbf{E} , which further simplifies the pole constraint:

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

The polar constraint gives a succinct representation of the spatial positional relationship of two matching points. Therefore, the camera pose estimation problem becomes the following two steps:

1. finds \mathbf{E} or \mathbf{F} based on the pixel location of the pairing point.
2. finds \mathbf{R}, \mathbf{t} from \mathbf{E} or \mathbf{F} .

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

7.3.2 essence matrix

By definition, the essence matrix $\mathbf{E} = \mathbf{t}^\wedge \mathbf{R}$. It is a matrix of 3×3 with 9 unknowns. So, isn't any matrix of 3×3 available as a cost matrix? From the construction of \mathbf{E} , there are some notable points:

- The essence matrix is defined by the pole constraint. Since the polar constraint is a constraint of **the equation is zero**, after multiplying \mathbf{E} by any non-zero constant, the **polar constraint remains satisfied**. We call this thing \mathbf{E} which is equivalent at different scales.
- According to $\mathbf{E} = \mathbf{t}^\wedge \mathbf{R}$, you can prove that [?], the singular value of the essence matrix \mathbf{E} must be in the form of $[\sigma, \sigma, 0]^T$. This is called the intrinsic property of **the essence matrix**.
- On the other hand, since there are 3 degrees of freedom for translation and rotation, $\mathbf{t}^\wedge \mathbf{R}$ has 6 degrees of freedom. But because of the scale equivalence, \mathbf{E} actually has 5 degrees of freedom.

The fact that \mathbf{E} has 5 degrees of freedom indicates that we can solve $b\mathbf{m}\mathbf{E}$ with at least 5 pairs of points. However, the intrinsic property of \mathbf{E} is a non-linear property that can cause trouble in estimation. Therefore, it can also be considered only by considering its **scale equivalence**, using 8 pairs of points to estimate \mathbf{E} - This is the classic **Eight-point-algorithm**[? ?]. The eight-point method only takes advantage of the linear nature of \mathbf{E} , so it can be solved in a linear algebra framework. Let's look at how the eight-point method works.

Consider a pair of matching points whose normalized coordinates are $\mathbf{x}_1 = [u_1, v_1, 1]^T$, $\mathbf{x}_2 = [u_2, v_2, 1]^T$. According to the pole constraint, there are:

$$(U_2, v_2, 1) \begin{pmatrix} E_1 & e_2 & e_3 \\ E_4 & e_5 & e_6 \\ E_7 & e_8 & e_9 \end{pmatrix} \begin{pmatrix} U_1 \\ v_1 \\ 1 \end{pmatrix} = 0. \quad (7.11)$$

We expand the matrix \mathbf{E} and write it as a vector:

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

* in the SfM study it may be unknown and to be estimated.

Then the polar constraint can be written as a linear form related to \mathbf{e} :

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

For the same reason, the same representation is given for other point pairs. We put all the points into an equation and become linear equations (u^i, v^i for the i feature points, and so on):

$$\begin{pmatrix} U_2^1 u_1^1 & u_2^1 v_1^1 & u_2^1 & v_2^1 u_1^1 & v_2^1 v_1^1 & v_2^1 & u_1^1 & v_1^1 & 1 \\ U_2^2 u_1^2 & u_2^2 v_1^2 & u_2^2 & v_2^2 u_1^2 & v_2^2 v_1^2 & v_2^2 & u_1^2 & v_1^2 & 1 \\ \vdots & \vdots \\ U_2^8 u_1^8 & u_2^8 v_1^8 & u_2^8 & v_2^8 u_1^8 & v_2^8 v_1^8 & v_2^8 & u_1^8 & v_1^8 & 1 \end{pmatrix} \begin{pmatrix} E_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \\ e_9 \end{pmatrix} = 0. \quad (7.13)$$

These eight equations form a linear system of equations. Its coefficient matrix consists of feature point locations and is 8×9 . \mathbf{e} is in the zero space of the matrix. If the coefficient matrix is full rank (ie rank is 8), then its zero space dimension is 1, ie \mathbf{e} constitutes a line. This is consistent with the scale equivalence of \mathbf{e} . If the matrix consisting of 8 pairs of matching points satisfies the condition of rank 8, then each element of \mathbf{E} can be solved by the above equation.

The next question is how to recover the camera's motion \mathbf{R}, \mathbf{t} based on the estimated essence matrix \mathbf{E} . This process is derived from singular value decomposition (SVD). Set the SVD of \mathbf{E} to

$$\mathbf{E} = \mathbf{U}\Sigma\mathbf{V}^T, \quad (7.14)$$

Where \mathbf{U}, \mathbf{V} is an orthogonal matrix, and Σ is a singular value matrix. According to the intrinsic nature of \mathbf{E} , we know $\Sigma = \text{diag}(\sigma, \sigma, 0)$. In the SVD decomposition, for any \mathbf{E} , there are two possible \mathbf{t}, \mathbf{R} corresponds to it:

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

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

?? visually shows the four solutions obtained by decomposing the essential matrix. We know the projection of the spatial point on the camera (blue line) (red dot) and want to solve the camera's motion. In the case of keeping the red point unchanged, four possible situations can be drawn. Fortunately, only the first solution, P , has a positive depth in both cameras. Therefore, as long as you substitute any point into the four solutions and detect the depth of the point under the two cameras, you can determine which solution is correct.

If you take advantage of the intrinsic nature of \mathbf{E} , then it has only 5 degrees of freedom. So at least 5 pairs of points can be used to solve camera motion [? ?]. However, this kind of practice is complicated. From the perspective of engineering implementation, since there are usually dozens of pairs or even hundreds of pairs of matching points, the significance of reducing from 8 pairs to 5 pairs is not obvious. To keep things simple, we will only introduce the basic eight-point method here.

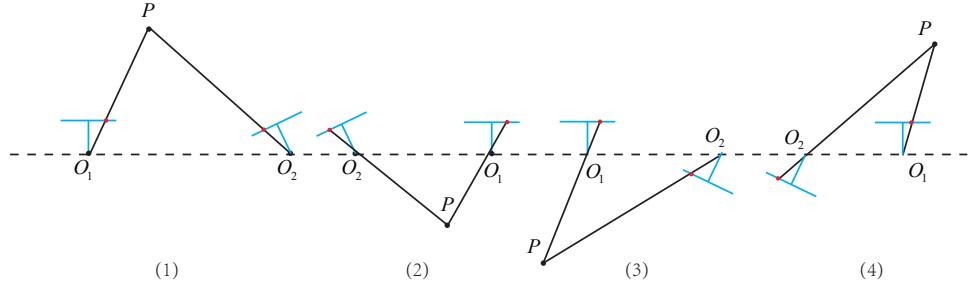


Figure 7-10: Decompose the four solutions obtained by the essence matrix. There are four possible cases for the two cameras and space points while keeping the projection point (red point) unchanged.

There is one more problem left: \mathbf{E} , which is solved according to the linear equation, may not satisfy the intrinsic property of \mathbf{E} - its singular value is not necessarily $\sigma, \sigma, 0$. At this time, we will deliberately adjust the Σ matrix to the above. The usual practice is to get the singular value matrix $\Sigma = \text{diag}(\sigma_1, \sigma_2, \sigma_3)$ after SVD decomposition of the \mathbf{E} obtained by the eight-point method. You can set $\sigma_1 \geq \sigma_2 \geq \sigma_3$. take:

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

This is equivalent to projecting the found matrix onto the manifold where \mathbf{E} is located. Of course, the simpler approach is to take the singular value matrix as $\text{diag}(1, 1, 0)$, because \mathbf{E} has scale equivalence, so it is reasonable to do so.

7.3.3

In addition to the basic and essential matrices, there is another common matrix in the two-view geometry: Homography \mathbf{H} , which describes the mapping between two planes. If the feature points in the scene fall on the same plane (such as wall, ground, etc.), motion estimation can be performed by homography. This is more common in overhead cameras that are carried by overhead drones or Sweepers. Since there was no mention of the singles before, I will introduce them here.

A homography matrix typically describes the transformation of some points on a common plane between two images. Consider having a pair of matching feature points p_1 and p_2 in the images I_1 and I_2 . These feature points fall on the plane P , and this plane satisfies the equation:

$$\mathbf{n}^T \mathbf{P} + d = 0. \quad (7.17)$$

Slightly sorted out, got:

$$-\frac{\mathbf{n}^T \mathbf{P}}{d} = 1. \quad (7.18)$$

Then, reviewing equation (7.1) at the beginning of this section, you get:

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

So, we get a transformation that directly describes the image coordinates \mathbf{p}_1 and \mathbf{p}_2 , and writes the middle part as \mathbf{H} , so:

$$\mathbf{p}_2 \simeq \mathbf{H}\mathbf{p}_1. \quad (7.19)$$

Its definition is related to the parameters of rotation, translation and plane. Similar to the base matrix \mathbf{F} , the homography matrix \mathbf{H} is also a matrix of 3×3 , and the idea of solving is similar to \mathbf{F} , as well as Calculate \mathbf{H} based on the matching point and then decompose it to calculate rotation and translation. Expand the above formula and get:

$$\begin{pmatrix} U_2 \\ v_2 \\ 1 \end{pmatrix} \simeq \begin{pmatrix} H_1 & h_2 & h_3 \\ H_4 & h_5 & h_6 \\ H_7 & h_8 & h_9 \end{pmatrix} \begin{pmatrix} U_1 \\ v_1 \\ 1 \end{pmatrix}. \quad (7.20)$$

Note that the equal sign here is still \simeq instead of the normal equal sign, so the \mathbf{H} matrix can also be multiplied by any non-zero constant. We can make $h_9 = 1$ in actual processing (when it takes a non-zero value). Then according to the third line, remove this non-zero factor, so there are:

$$\begin{aligned} U_2 &= \frac{h_1 u_1 + h_2 v_1 + h_3}{h_7 u_1 + h_8 v_1 + h_9} \\ V_2 &= \frac{h_4 u_1 + h_5 v_1 + h_6}{h_7 u_1 + h_8 v_1 + h_9}. \end{aligned}$$

Finished up:

$$\begin{aligned} H_1 u_1 + h_2 v_1 + h_3 - h_7 u_1 u_2 - h_8 v_1 u_2 &= u_2 \\ H_4 u_1 + h_5 v_1 + h_6 - h_7 u_1 v_2 - h_8 v_1 v_2 &= v_2. \end{aligned}$$

Such a set of matching point pairs can construct two constraints (in fact, there are three constraints, but because of the linear correlation, only the first two), so the homography matrix with a degree of freedom of 8 can be calculated by four pairs of matching feature points (In the case of non-degenerate, that is, the case where these feature points cannot have three points collinear), the following linear equations are solved (when $h_9 = 0$, the right side is zero):

$$\begin{pmatrix} U_1^1 & v_1^1 & 1 & 0 & 0 & 0 & -u_1^1 u_2^1 & -v_1^1 u_2^1 \\ 0 & 0 & 0 & u_1^1 & v_1^1 & 1 & -u_1^1 v_2^1 & -v_1^1 v_2^1 \\ U_1^2 & v_1^2 & 1 & 0 & 0 & 0 & -u_1^2 u_2^2 & -v_1^2 u_2^2 \\ 0 & 0 & 0 & u_1^2 & v_1^2 & 1 & -u_1^2 v_2^2 & -v_1^2 v_2^2 \\ U_1^3 & v_1^3 & 1 & 0 & 0 & 0 & -u_1^3 u_2^3 & -v_1^3 u_2^3 \\ 0 & 0 & 0 & u_1^3 & v_1^3 & 1 & -u_1^3 v_2^3 & -v_1^3 v_2^3 \\ U_1^4 & v_1^4 & 1 & 0 & 0 & 0 & -u_1^4 u_2^4 & -v_1^4 u_2^4 \\ 0 & 0 & 0 & u_1^4 & v_1^4 & 1 & -u_1^4 v_2^4 & -v_1^4 v_2^4 \end{pmatrix} \begin{pmatrix} H_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \end{pmatrix} = \begin{pmatrix} U_2^1 \\ v_2^1 \\ U_2^2 \\ v_2^2 \\ U_2^3 \\ v_2^3 \\ U_2^4 \\ v_2^4 \end{pmatrix}. \quad (7.21)$$

This approach treats the \mathbf{H} matrix as a vector and restores \mathbf{H} by solving the linear equation of the vector, also known as Direct Linear Transform. Similar to the essence matrix, the homography matrix needs to be decomposed in order to obtain the corresponding rotation matrix \mathbf{R} and the translation vector \mathbf{t} . The methods of decomposition include the numerical method [? ?] and the parsing method [?]. Similar to the decomposition of the essential matrix, the decomposition of the homography matrix also returns four sets of rotation matrices and translation vectors, and at the same time, the normal vectors of the planes of the corresponding scene points are calculated. If the depth of the imaged map points is known to be all positive (ie in front of the camera), then the two sets of solutions can be excluded. In the end, there are only two sets of solutions, and more need to be judged by more a priori information. Usually we can solve by assuming the normal vector of the known scene plane, such as the scene plane parallel to the camera plane, then the theoretical value of the normal vector \mathbf{n} is $\mathbf{1}^T$.

Homology is of great importance in SLAM. When the feature points are coplanar or the camera rotates purely, the degree of freedom of the base matrix decreases, and so-called degenerate occurs. The actual data always contains some noise. If you continue to use the eight-point method to solve the basic matrix, the excess degree of the basic matrix will be mainly determined by noise. In order to avoid the effects of degradation, we usually estimate the base matrix \mathbf{F} and the homography matrix \mathbf{H} , and choose the one with the smaller reprojection error as the final motion estimation matrix.

7.4 Practice: Solving camera motion with polar constraints

Below, let's practice how to solve camera motion through the essential matrix. The program in the previous section provides feature matching, and this time we use the matching feature points to calculate \mathbf{E}, \mathbf{F} and \mathbf{H} , which in turn breaks \mathbf{E} Get \mathbf{R}, \mathbf{t} . The entire program is solved using the algorithm provided by OpenCV. We encapsulate the feature extraction from the previous section into a function for later use. This section only shows the code for the pose estimation section.

Listing 7.5: slambook2/ch7/pose_estimation_2d2d.cpp

```

1 void pose_estimation_2d2d(std::vector<KeyPoint> keypoints_1,
2     std::vector<KeyPoint> keypoints_2,
3     std::vector<DMatch> matches,
4     Mat &R, Mat &t) {
5     // ... , TUM Freiburg2
6     Mat K = (Mat<double>(3, 3) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0,
7     1);
8
8     //— ... vector<Point2f> points1;
9     vector<Point2f> points1;
10    vector<Point2f> points2;
11
12    for (int i = 0; i < (int) matches.size(); i++) {
13        points1.push_back(keypoints_1[matches[i].queryIdx].pt);
14        points2.push_back(keypoints_2[matches[i].trainIdx].pt);
15    }
16
17    //— ...
18    Mat fundamental_matrix;
```

```

19 fundamental_matrix = findFundamentalMat(points1, points2, CV_FM_8POINT)
20 ;
21 cout << "fundamental_matrix is " << endl << fundamental_matrix << endl;
22 //— □ □ □ □ □ □
23 Point2d principal_point(325.1, 249.7); //□ □ □ , TUM □ □ □ dataset
24 double focal_length = 521; //□ □ □ , TUM □ □ □ dataset
25 Mat essential_matrix;
26 essential_matrix = findEssentialMat(points1, points2, focal_length,
27     principal_point);
28 cout << "essential_matrix is " << endl << essential_matrix << endl;
29
30 //— □ □ □ □ □ □
31 //— □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
32 Mat homography_matrix;
33 homography_matrix = findHomography(points1, points2, RANSAC, 3);
34 cout << "homography_matrix is " << endl << homography_matrix << endl;
35
36 //— □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ .
37 recoverPose(essential_matrix, points1, points2, R, t, focal_length,
38     principal_point);
39 cout << "R is " << endl << R << endl;
40 cout << "t is " << endl << t << endl;
41 }
```

This function provides the part of the camera that solves the motion of the camera from the feature point. Then we call it in the main function to get the camera's motion:

Listing 7.6: slambook2/ch7/pose_estimation_2d2d.cpp

```

1 int main( int argc, char** argv ){
2     if (argc != 3) {
3         cout << "usage: pose_estimation_2d2d img1 img2" << endl;
4         return 1;
5     }
6     //— □ □ □ □
7     Mat img_1 = imread(argv[1], CV_LOAD_IMAGE_COLOR);
8     Mat img_2 = imread(argv[2], CV_LOAD_IMAGE_COLOR);
9     assert(img_1.data && img_2.data && "Can not load images!");
10
11    vector<KeyPoint> keypoints_1, keypoints_2;
12    vector<DMatch> matches;
13    find_feature_matches(img_1, img_2, keypoints_1, keypoints_2, matches);
14    cout << "□ □ □ □ " << matches.size() << "□ □ □ □ " << endl;
15
16    //— □ □ □ □ □ □ □
17    Mat R, t;
18    pose_estimation_2d2d(keypoints_1, keypoints_2, matches, R, t);
19
20    //— □ □ E=t^R*scale
21    Mat t_x =
22        (Mat_<double>(3, 3) << 0, -t.at<double>(2, 0), t.at<double>(1, 0),
23        t.at<double>(2, 0), 0, -t.at<double>(0, 0),
24        -t.at<double>(1, 0), t.at<double>(0, 0), 0);
25    cout << "t^R=" << endl << t_x * R << endl;
26
27    //— □ □ □ □ □ □
28    Mat K = (Mat_<double>(3, 3) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0,
29        1);
30    for (DMatch m: matches) {
31        Point2d pt1 = pixel2cam(keypoints_1[m.queryIdx].pt, K);
32        Mat y1 = (Mat_<double>(3, 1) << pt1.x, pt1.y, 1);
```

```

32     Point2d pt2 = pixel2cam(keypoints_2[m.trainIdx].pt, K);
33     Mat y2 = (Mat<double>(3, 1) << pt2.x, pt2.y, 1);
34     Mat d = y2.t() * t_x * R * y1;
35     cout << "epipolar constraint = " << d << endl;
36   }
37   return 0;
38 }
```

$$\mathbf{E}, \mathbf{F} \mathbf{H} \quad \mathbf{t}^T \mathbf{R} \mathbf{E}$$

Listing 7.7:

```

% build/pose_estimation_2d2d 1.png 2.png
1 — Max dist : 95.000000
2 — Min dist : 4.000000000000000
3 79 0 0 0 0
4 fundamental_matrix is
5 [4.844484382466111e-06, 0.0001222601840188731, -0.01786737827487386;
6 -0.0001174326832719333, 2.12288800459598e-05, -0.01775877156212593;
7 0.01799658210895528, 0.008143605989020664, 1]
8 essential_matrix is
9 [ -0.0203618550523477, -0.4007110038118445, -0.03324074249824097;
10 0.3939270778216369, -0.03506401846698079, 0.5857110303721015;
11 -0.006788487241438284, -0.5815434272915686, -0.01438258684486258]
12 homography_matrix is
13 [0.9497129583105288, -0.143556453147626, 31.20121878625771;
14 0.04154536627445031, 0.9715568969832015, 5.306887618807696;
15 -2.81813676978796e-05, 4.353702039810921e-05, 1]
16 R is
17 [0.9985961798781875, -0.05169917220143662, 0.01152671359827873;
18 0.05139607508976055, 0.9983603445075083, 0.02520051547522442;
19 -0.01281065954813571, -0.02457271064688495, 0.9996159607036126]
20 t is
21 [-0.8220841067933337;
22 -0.03269742706405412;
23 0.5684264241053522]
24
25 t^T R =
26 [0.02879601157010516, 0.5666909361828478, 0.04700950886436416;
27 -0.5570970160413605, 0.0495880104673049, -0.8283204827837456;
28 0.009600370724838804, 0.8224266019846683, 0.02034004937801349]
29 epipolar constraint = [0.002528128704106625]
30 epipolar constraint = [-0.001663727901710724]
31 epipolar constraint = [-0.0008009088410884102]
32 .....
```

As can be seen from the output of the program, the accuracy of the polarity constraint is about 10^{-3} . According to the previous discussion, the decomposition of \mathbf{R}, \mathbf{t} has a total of 4 possibilities. However, OpenCV will use the triangulation to detect if the depth of the corner is positive, so as to choose the correct solution.

discussion

As you can see from the demo, the output of \mathbf{E} and \mathbf{F} differs from the camera's internal reference matrix. Although they are not intuitive in value, they can be verified for their mathematical relationship. From \mathbf{E}, \mathbf{F} and \mathbf{H} , motion can be decomposed, but \mathbf{H} needs to assume that the feature points are on the plane. For the data in this experiment, this assumption is not good, so we mainly use \mathbf{E} to decompose the movement here.

It is worth mentioning that since \mathbf{E} itself has scale equivalence, it decomposes \mathbf{t} , and \mathbf{R} also has a scale equivalence. And $\mathbf{R} \in \text{SO}(3)$ itself has constraints, so we think \mathbf{t} has a **scale**. In other words, in the decomposition process, multiplying \mathbf{t} by any non-zero constant, the decomposition is true. Therefore, we usually make **normalized** to \mathbf{t} and let it be equal to 1.

Scale uncertainty

The normalization of the length of \mathbf{t} directly leads to **Scale Ambiguity**. For example, the first dimension of \mathbf{t} output in the program is about 0.822. Whether this 0.822 refers to 0.822 meters or 0.822 cm, we can't be sure. Since multiplying \mathbf{t} by an arbitrary proportional constant, the polar constraint is still true. In other words, in the monocular SLAM, the trajectory and the map are simultaneously scaled by any multiple, and the image we get is still the same. This has already been introduced to the reader in the second lecture.

In monocular vision, we normalize the \mathbf{t} of the two images to **fixed scale**. Although we don't know what the actual length is, we calculate the 3D position of the camera motion and feature points in units of \mathbf{t} . This is called **initialization** for monocular SLAM. After initialization, you can use 3D–2D to calculate camera motion. The trajectory after initialization and the unit of the map are the fixed scales at initialization. Therefore, the monocular SLAM has one step inevitable **initialization**. The two images that are initialized must have a certain degree of translation, and the subsequent tracks and maps will be in units of the translation of this step.

In addition to normalizing \mathbf{t} , another method is to make all feature points have an average depth of 1 at initialization or a fixed scale. Compared to the method of making \mathbf{t} length 1, the feature point depth can be normalized to control the size of the scene, making the calculation more stable. However, there is no theoretical difference.

Initialized pure rotation problem

In the process of decomposing from \mathbf{E} to \mathbf{R}, \mathbf{t} , if the camera is a pure rotation, causing \mathbf{t} to be zero, then the result is \mathbf{E} will also be zero, which will cause us to solve \mathbf{R} . However, at this point we can rely on \mathbf{H} to get the rotation, but only when we rotate, we can't estimate the spatial position of the feature points with triangulation (this will be mentioned below), so another conclusion is, **Mono initialization can not only have a pure rotation, there must be a certain degree of translation**. If there is no panning, the monocular will not be initialized. In practice, if the translation is too small during initialization, the pose solution and triangulation results will be unstable, resulting in failure. In contrast, if the camera is moved left and right instead of rotating in place, it is easy to initialize the monocular SLAM. Thus, experienced SLAM researchers often choose to have the camera pan left and right for smooth initialization in the case of a monocular SLAM.

More than 8 pairs of points

When the given number of points is more than 8 pairs (for example, the routine finds 79 pairs of matches), we can calculate a least squares solution. To recall the linearized pole constraint in (??), we write the coefficient matrix on the left as \mathbf{A} :

$$\mathbf{A}\mathbf{e} = \mathbf{0}. \quad (7.22)$$

For the eight-point method, the size of \mathbf{A} is 8×9 . If the given matching point is more than 8, the equation constitutes an overdetermined equation, ie, there is not necessarily \mathbf{e} to make the above formula hold. Therefore, you can find by minimizing a quadratic form:

$$\min_{\mathbf{e}} \|\mathbf{A}\mathbf{e}\|_2^2 = \min_{\mathbf{e}} \mathbf{e}^T \mathbf{A}^T \mathbf{A} \mathbf{e}. \quad (7.23)$$

Then we find the \mathbf{E} matrix in the sense of least squares. However, when there may be mismatches, we prefer to use **Random Sample Consensus (RANSAC)** instead of least squares. RANSAC is a common practice for many cases with erroneous data and can handle data with mismatched matches.

7.5 Triangulation

In the previous two sections we estimated camera motion using polar geometry constraints and discussed the limitations of this approach. After getting the motion, the next step is to estimate the spatial position of the feature points using the motion of the camera. In monocular SLAM, the depth information of a pixel cannot be obtained by a single image. We need to estimate the depth of the map point by **Triangulation (or triangulation)**, such as ?? is shown.

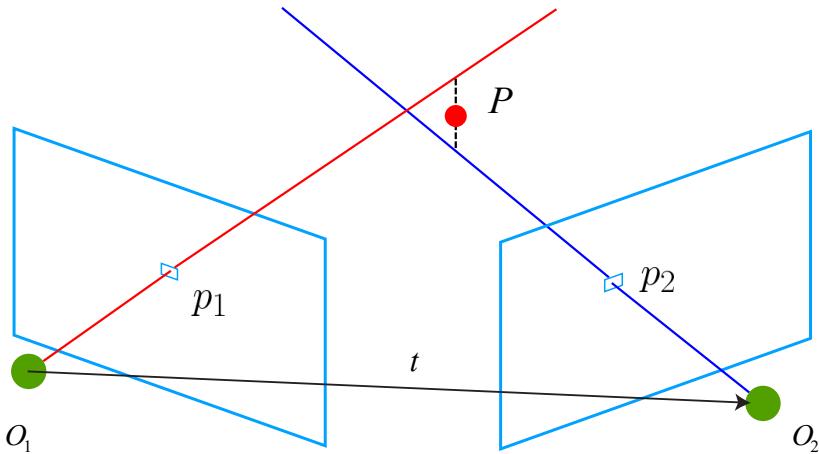


Figure 7-11: Triangular to get the depth of the map point.

Triangulation refers to inferring the distance of a landmark point from the observed position by observing at the same landmark point at different locations. Triangulation was first proposed by Gauss and applied to surveying. It is used in the measurement of astronomy and geography. For example, we can estimate the distance from the stars by the angles of the stars 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 right transformation matrix as \mathbf{T} . The camera's optical center is O_1 and O_2 . There is a feature point p_1 in I_1 , and there is a feature point p_2 in I_2 . In theory, the straight line O_1p_1 and O_2p_2 will intersect in the scene at a point of P , which is the map point corresponding to the two feature points in the 3D scene. The location in . However, due to the influence of noise, these two lines often cannot intersect. Therefore, it can be solved by the second best multiplication method.

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

$$S_2 \mathbf{x}_2 = s_1 \mathbf{R} \mathbf{x}_1 + \mathbf{t}. \quad (7.24)$$

Now that we know \mathbf{R}, \mathbf{t} , we want to solve the depth of two feature points s_1, s_2 . Geometrically, you can find 3D points on the ray $O_1 p_1$ so that its projection position is close to p_2 . Similarly, you can find it on $O_2 p_2$, or in the middle of two lines. . Different strategies correspond to different calculation methods, of course, they are similar. For example, if we want to calculate s_1 , first multiply the left side of the top by a \mathbf{x}_2^\wedge to get:

$$S_2 \mathbf{x}_2^\wedge \mathbf{x}_2 = 0 = s_1 \mathbf{x}_2^\wedge \mathbf{R} \mathbf{x}_1 + \mathbf{x}_2^\wedge \mathbf{t}. \quad (7.25)$$

The left side of the equation is zero, and the right side can be seen as an equation of s_2 , which can be directly obtained from s_2 . With s_2 , s_1 is also very easy to find. So we get the depth of the points under the two frames and determine their spatial coordinates. Of course, due to the existence of noise, we estimate that \mathbf{R}, \mathbf{t} does not necessarily make the formula (??) zero, so it is more common to find the least squares solution. Not a direct solution.

7.6 Practice: Triangulation

7.6.1 Triangulation code

Below, we demonstrate how to determine the spatial position of the feature points of the previous section by triangulation based on the camera pose previously solved with the polar geometry. We call the triangulation function provided by OpenCV for triangulation.

Listing 7.8: slambook2/ch7/triangulation.cpp

```

1 void triangulation(
2     const vector<KeyPoint> &keypoint_1,
3     const vector<KeyPoint> &keypoint_2,
4     const std::vector<DMatch> &matches,
5     const Mat &R, const Mat &t,
6     vector<Point3d> &points) {
7     Mat T1 = (Mat_<float>(3, 4) <<
8         1, 0, 0, 0,
9         0, 1, 0, 0,
10        0, 0, 1, 0);
11    Mat T2 = (Mat_<float>(3, 4) <<
12        R.at<double>(0, 0), R.at<double>(0, 1), R.at<double>(0, 2), t.at<double>(0, 0),
13        R.at<double>(1, 0), R.at<double>(1, 1), R.at<double>(1, 2), t.at<double>(1, 0),
14        R.at<double>(2, 0), R.at<double>(2, 1), R.at<double>(2, 2), t.at<double>(2, 0)
15    );
16
17    Mat K = (Mat_<double>(3, 3) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1)
18        ;
19    vector<Point2f> pts_1, pts_2;
20    for (DMatch m:matches) {
21        // ...
22        pts_1.push_back(pixel2cam(keypoint_1[m.queryIdx].pt, K));
23        pts_2.push_back(pixel2cam(keypoint_2[m.trainIdx].pt, K));

```

```
23 }
24
25 Mat pts_4d;
26 cv::triangulatePoints(T1, T2, pts_1, pts_2, pts_4d);
27
28 // 0 0 0 0 0 0 0 0
29 for (int i = 0; i < pts_4d.cols; i++) {
30     Mat x = pts_4d.col(i);
31     x /= x.at<float>(3, 0); // 0 0 0
32     Point3d p(
33         x.at<float>(0, 0),
34         x.at<float>(1, 0),
35         x.at<float>(2, 0)
36     );
37     points.push_back(p);
38 }
39 }
```

At the same time, add a triangulation part to the main function, and then draw a depth map of each point. Readers can run this program themselves to view the triangulation results.

7.6.2 discussion

There is one more point to note about triangulation.

Triangulation is obtained by **translation**, and there is a translation to have a triangle in the opposite geometry, so that triangulation can be said. Therefore, pure rotation is impossible to use triangulation because the polar constraint will always be satisfied. Of course, the actual data is often not exactly equal to zero. In the case of translation, we also care about the uncertainty of triangulation, which leads to a **triangulation contradiction**.

As shown in ?? , when the translation is small, the uncertainty on the pixel will result in a large depth uncertainty. That is, if the feature point moves by one pixel δx such that the line of sight angle changes by an angle of $\delta\theta$, then the depth value has a change of δd . As you can see from the geometric relationship, when t is large, δd will be significantly smaller, which means that when the translation is large, the triangulation measurement will be more accurate at the same camera resolution. . Quantitative analysis of the process can be obtained using the sine theorem.

Therefore, to improve the accuracy of triangulation, one is to improve the extraction precision of feature points, that is, to improve the image resolution - but this will lead to larger images and increase computational cost. Another way is to increase the amount of translation. However, this can cause significant changes to the **appearance** of the image, such as the side of the box that was originally blocked, or the illumination of the object, and so on. Changes in appearance can make feature extraction and matching difficult. In summary, increasing the translation may result in a match failure; while the translation is too small, the triangulation is not accurate enough - this is the contradiction of triangulation. We call this problem paraparx.

In monocular vision, since the monocular image has no depth information, we wait for the feature point to be tracked for a few frames, then generate enough angle of view, and then use triangulation to determine the depth value of the new feature point. This time is also called delay triangulation [?]. However, if the camera is rotated in situ, resulting in a small parallax, it is not easy to estimate the depth of the newly observed feature points. This situation is more common in robotic

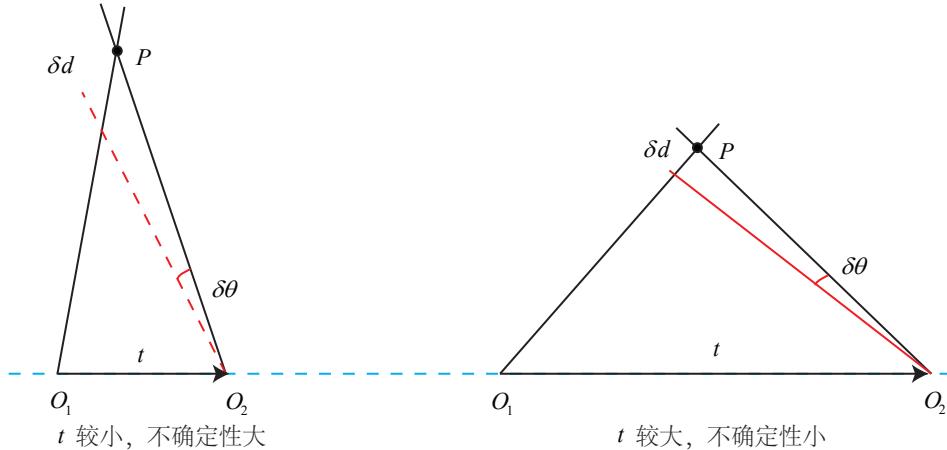


Figure 7-12: The contradiction of triangulation.

situations because in-situ rotation is often a common command for robots. In this case, monocular vision may be subject to tracking failures, incorrect scales, and so on.

Although this section only introduces the depth estimation of triangulation, we can quantitatively calculate **location** and **uncertainty** for each feature point as long as we are willing. Therefore, if we assume that the feature point obeys the Gaussian distribution and continuously observes it, we can expect **its variance will decrease or even converge when the information is correct**. This gives a filter called **Depth Filter**. However, because its principle is more complicated, we will stay behind to discuss it in detail. Below, we discuss the estimation of camera motion from the 3D–2D matching points, and the 3D–3D estimation method.

7.7 3D–2D:PnP

PnP (Perspective-n-Point) is a method for solving 3D to 2D point pair motion. It describes how to estimate the pose of the camera when knowing n 3D spatial points and their projected positions. As mentioned earlier, the 2D–2D parapolar geometry method requires 8 or more point pairs (taking the eight-point method as an example), and there are problems with initialization, pure rotation, and scaling. However, if the 3D position of one of the two feature points is known, then at least 3 point pairs (and at least one additional point verification result) are needed to estimate camera motion. The 3D position of the feature point can be determined by a triangulation or a depth map of the RGB-D camera. Therefore, in binocular or RGB-D visual odometers, we can directly estimate camera motion using PnP. In a monocular visual odometer, initialization must be performed before PnP can be used. The 3D–2D method is the most important attitude estimation method without using the polar constraint and obtaining better motion estimation in few matching points.

There are many ways to solve PnP problems, for example, P3P^[?], direct linear transformation (DLT), EPnP (Efficient PnP)^[?], UPnP^[?], and so on. In addition, you can use the **non-linear optimization** method to construct the least squares

problem and iteratively solve it, which is the Bundle Adjustment. Let's look at the DLT first, then the Bundle Adjustment.

7.7.1 Direct linear transformation

We consider the problem of knowing the position of a set of 3D points and their projected position in a camera to find the pose of the camera. This problem can also be used to solve camera state problems for a given map and image. If the 3D point is viewed as a point in another camera coordinate system, it can also be used to solve the relative motion problems of the two cameras. We start with simple questions.

Consider a space point P whose equality coordinate is $\mathbf{P} = (X, Y, Z, 1)^T$. In the image I_1 , projected to the feature point $\mathbf{x}_1 = (u_1, v_1, 1)^T$ (Expressed in normalized plane homogeneous coordinates). At this point the camera's pose \mathbf{R}, \mathbf{t} is unknown. Similar to the solution of the homography matrix, we define the augmented matrix $[\mathbf{R}|\mathbf{t}]$ as a matrix of 3×4 , containing the rotation and translation information *. We will write the expanded form as follows:

$$s \begin{pmatrix} U_1 \\ V_1 \\ 1 \end{pmatrix} = \begin{pmatrix} T_1 & t_2 & t_3 & t_4 \\ T_5 & t_6 & t_7 & t_8 \\ T_9 & t_{10} & t_{11} & t_{12} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}. \quad (7.26)$$

Use the last line to eliminate s and get two constraints:

$$U_1 = \frac{t_1 X + t_2 Y + t_3 Z + t_4}{t_9 X + t_{10} Y + t_{11} Z + T_{12}}, \quad V_1 = \frac{t_5 X + t_6 Y + t_7 Z + t_8}{t_9 X + t_{10} Y + t_{11} Z + T_{12}}.$$

To simplify the representation, define the row vector for \mathbf{T} :

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

Then there are:

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

with

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

Note that \mathbf{t} is the variable to be sought. As you can see, each feature point provides two linear constraints on \mathbf{t} . Assuming a total of N feature points, the following linear equations can be listed:

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

Since \mathbf{t} has a total of 12 dimensions, a linear solution of the matrix \mathbf{T} can be achieved with a minimum of 6 pairs of matching points. This method is called Direct Linear Transform (DLT). When the matching point is greater than 6 pairs, the

* Please note that This is different from the transformation matrix \mathbf{T} in SE(3).

least squares solution can be obtained for the overdetermined equation using SVD or the like.

In the DLT solution, we directly consider the \mathbf{T} matrix as 12 unknowns, ignoring the connection between them. Because the rotation matrix $\mathbf{R} \in \text{SO}(3)$, 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 find a best rotation matrix for the matrix block of the \mathbf{T} left 3×3 estimated by the DLT. This can be done by QR decomposition [? ?], or you can calculate [? ?] like this:

$$\mathbf{R} \leftarrow (\mathbf{R}\mathbf{R}^T)^{-\frac{1}{2}} \mathbf{R}. \quad (7.28)$$

This is equivalent to re-projecting the result from the matrix space onto the SE(3) manifold and converting it into two parts, rotation and translation.

It should be explained that our \mathbf{x}_1 uses the normalized plane coordinates and removes the influence of the internal parameter matrix \mathbf{K} - this is because the internal parameter \mathbf{K} It is usually assumed to be known in SLAM. Even if the internal parameters are unknown, PnP can be used to estimate $\mathbf{K}, \mathbf{R}, \mathbf{t}$ three quantities. However, due to the increased amount of unknowns, the effect will be worse.

7.7.2 P3P

The P3P described below is another way to solve PnP. It uses only 3 pairs of matching points and requires less data, so it's also briefly introduced here (this part of the derivation draws on the literature [?]).

P3P needs to utilize the geometric relationship of a given 3 points. Its input data is 3 pairs of 3D–2D matching points. The 3D points are A, B, C , and the 2D points are a, b, c , where the lowercase letters represent the points corresponding to the uppercase letters on the camera's imaging plane, such as ?? is shown. In addition, P3P also needs to use a pair of verification points to select the correct one from the possible solutions (similar to the polar geometry case). The verification point pair is $D - d$ and the camera's optical center is O . Note that we know that $A, B, \text{and} C$ are in **coordinates in the world coordinate system**, not **coordinates in the camera coordinate system**. Once the coordinates of the 3D point in the camera coordinate system can be calculated, we get the corresponding point of 3D–3D and convert the PnP problem into an ICP problem.

First of all, it is obvious that there is a correspondence between the triangles:

$$\Delta Oab = \Delta OAB, \quad \Delta Obc = \Delta OBC, \quad \Delta Oac = \Delta OAC. \quad (7.29)$$

Consider the relationship between Oab and OAB . Using the cosine theorem, there are:

$$OA^2 + OB^2 - 2OA \cdot OB \cdot \cos \langle a, b \rangle = AB^2. \quad (7.30)$$

For the other two triangles, there are similar properties, so there are:

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

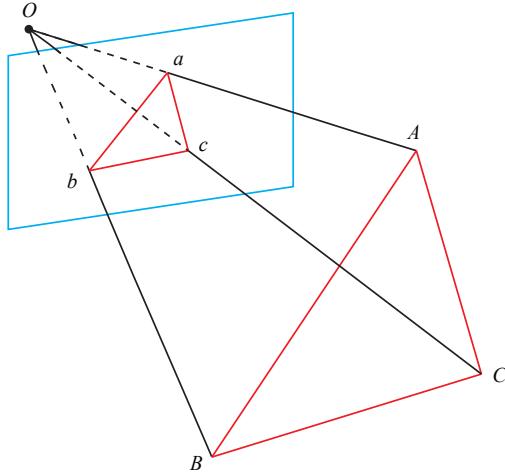


Figure 7-13: P3P

Divide all of the above three formulas by OC^2 , and record $x = OA/OC, y = OB/OC$, and get:

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

Remember $v = AB^2/OC^2, uv = BC^2/OC^2, wv = AC^2/OC^2$, with:

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

We can put v in the first expression on the side of the equation and substitute the following two equations to get:

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

Note the known and unknown quantities in these equations. Since we know the image position of the 2D point, the 3 cosine angles $\cos \langle a, b \rangle, \cos \langle b, c \rangle, \cos \langle a, c \rangle$ is known. At the same time, $u = BC^2/AB^2, w = AC^2/AB^2$ can be calculated by the coordinates of A, B, C in the world coordinate system, after changing to the camera coordinate system, this The ratio does not change. The x, y in this formula is unknown and will change as the camera moves. Therefore, the system of equations is a binary quadratic equation (polynomial equation) for x, y . Solving the equations analytically is a complex process that requires the elimination of the Wu method. The introduction to the solution to this equation is not extended here. For interested readers, please refer to the literature [?]. Similar to the case of breaking E , the equation may get up to 4 solutions, but we can use the verification point to calculate the most probable solution, and get 3D of A, B, C in the camera coordinate system. coordinate. Then, based on the point pair of 3D–3D, calculate the camera's motion R, t . This section will be introduced in Section 7.9.

As can be seen from the principle of P3P, in order to solve PnP, we use the similarity of triangles to solve the 3D coordinates of projection points a, b, c in the camera coordinate system, and finally convert the problem into a 3D to 3D pose. Estimate the problem. As will be seen later, the 3D–3D pose solution with matching information is very easy to solve, so this idea is very effective. Other methods, such as EPnP, also use this approach. However, P3P also has some problems:

1. P3P only uses 3 points of information. When there are more than 3 groups of given points, it is difficult to use more information.
2. If the 3D point or 2D point is affected by noise, or if there is a mismatch, the algorithm fails.

So follow-up people have proposed many other methods, such as EPnP, UPnP and so on. They use more information and iteratively optimizes camera poses to eliminate the effects of noise as much as possible. However, the principle is more complicated than P3P, so we recommend that readers read the original paper or understand the PnP process through practice. In SLAM, the usual practice is to estimate the camera pose using methods such as P3P/EPnP, and then construct a least squares optimization problem to adjust the estimate (Bundle Adjustment). When the camera motion is continuous enough, it can also be assumed that the camera does not move or move at a constant speed, and the estimated value is used as an initial value for optimization. Next we look at the PnP problem from the perspective of nonlinear optimization.

7.7.3 Minimize reprojection error solver PnP

In addition to using linear methods, we can also construct the PnP problem as a nonlinear least squares problem with reprojection errors. This will use the knowledge in the book ?? and ???. The linear method mentioned above is often **first seek camera pose, then find the position of the space point**, while the nonlinear optimization is to regard them as optimization variables and put them together for optimization. This is a very versatile solution that we can use to optimize the results given by PnP or ICP. This type of **to minimize the camera and 3D points together** is collectively referred to as Bundle Adjustment, referred to as BA*.

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

Considering n 3D space point P and its projection p , we want to calculate the camera's pose \mathbf{R}, \mathbf{t} , whose Lie group is represented as \mathbf{T} . Suppose the coordinates of a space point are $\mathbf{P}_i = [X_i, Y_i, Z_i]^T$, and the pixel coordinates of the projection are $\mathbf{u}_i = [u_i, v_i]^T$. According to the content of ??, the relationship between pixel

* It should be noted that BA is in different documents and contexts. The meaning is not exactly the same. Some scholars only refer to the problem of minimizing the reprojection error as BA, while others have a broader BA concept. Even if the BA has only one camera or other similar sensors, it can be called BA. I personally prefer a broader BA concept, so the method of calculating PnP here is also called BA.

position and spatial point position is as follows:

$$S_i \begin{bmatrix} U_i \\ v_i \\ 1 \end{bmatrix} = \mathbf{K} \mathbf{T} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix}. \quad (7.35)$$

Written in matrix form is:

$$s_i \mathbf{u}_i = \mathbf{K} \mathbf{T} \mathbf{P}_i.$$

This expression implies a conversion from homogeneous coordinates to non-homogeneous, otherwise the dimensions are wrong by the multiplication of the matrix. *. Now, there is an error in the equation due to the unknown camera pose and the noise at the observation point. Therefore, we sum the errors, build a least squares problem, and then find the best camera pose to minimize it:

$$\mathbf{T}^* = \arg \min_{\mathbf{T}} \frac{1}{2} \sum_{i=1}^n \left\| \mathbf{u}_i - \frac{1}{s_i} \mathbf{K} \mathbf{T} \mathbf{P}_i \right\|_2^2. \quad (7.36)$$

The error term of this problem is that the projection position of the 3D point is different from the observation position, 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 we use non-homogeneous coordinates more often, so the error is only 2 dimensions. As shown in ?? , we know through feature matching that p_1 and p_2 are projections of the same space point P , but I don't know the pose of the camera. In the initial value, there is a certain distance between the P projection \hat{p}_2 and the actual p_2 . So we adjust the pose of the camera to make this distance smaller. However, since this adjustment requires consideration of many points, the final effect is a reduction in the overall error, and the error at each point is usually not exactly zero.

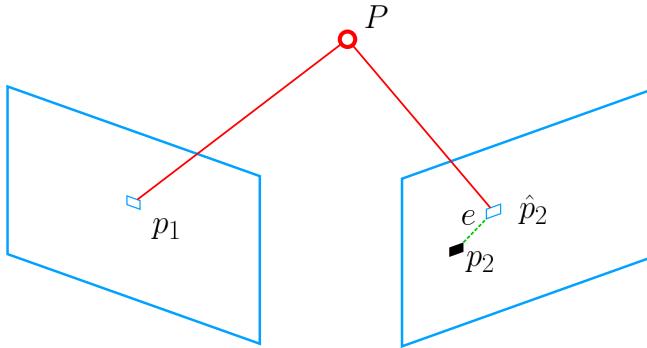


Figure 7-14: Reprojection error diagram.

The least squares optimization problem has been introduced in the ?. Using Lie algebra, it is possible to construct an unconstrained optimization problem, which

* $\mathbf{T} \mathbf{P}_i$ 4×1 , and the \mathbf{K} on the left is 3×3 , so you must take the first 3D of $\mathbf{T} \mathbf{P}_i$ In three-dimensional non-homogeneous coordinates. Or, use $\mathbf{R} \mathbf{P} + \mathbf{t}$.

is conveniently solved by an optimization algorithm such as Gauss-Newton method and Levenberg-Marquart method. However, before using the Gauss-Newton method and the Levenberg-Marquart method, we need to know the derivative of each error term with respect to the optimization variable, ie **linearization**:

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

The form of \mathbf{J}^T is worth discussing, and it can even be said to be the key. We can certainly use numerical derivatives, but if we can derive the analytical form, we will give priority to the analytical derivative. Now, when \mathbf{e} is the pixel coordinate error (2 dimensions) and \mathbf{x} is the camera pose (6 dimensions), \mathbf{J}^T will be a matrix of 2×6 . Let us derive the form of \mathbf{J}^T .

Recalling the contents of Lie algebra, we introduced how to use the perturbation model to find the derivative of Lie algebra. First, the coordinates of the space point transformed into the camera coordinate system are \mathbf{P}' , and the first 3 dimensions are taken out:

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

Then, the camera projection model is relative to \mathbf{P}'

$$s\mathbf{u} = \mathbf{KP}'. \quad (7.39)$$

Expand:

$$\begin{bmatrix} Su \\ Sv \\ s \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix}. \quad (7.40)$$

Using line 3 to eliminate s (actually the distance of \mathbf{P}'), you get:

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

This is consistent with the camera model in ???. When we ask for the error, we can compare the u, v here with the actual measured value and find the difference. After defining the intermediate variable, we multiply the \mathbf{T} left perturbation $\delta\xi$ and then consider the variation of \mathbf{e} on the derivative of the disturbance. Using the chain rule, you can write the following:

$$\frac{\partial \mathbf{e}}{\partial \delta\xi} = \lim_{\delta\xi \rightarrow 0} \text{Frace}(\delta\xi \oplus \xi) - \mathbf{e}(\xi)\delta\xi = \frac{\partial \mathbf{e}}{\partial \mathbf{P}'} \frac{\partial \mathbf{P}'}{\partial \delta\xi}. \quad (7.42)$$

Here \oplus refers to the left-handed perturbation on the Lie algebra. The first term is the derivative of the error about the projection point. The relationship between the variables is already listed in the formula ???, which is easy to get:

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

The second term is the derivative of the transformed point on the Lie algebra. According to the derivation in the ?? section, we get:

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

In the definition of \mathbf{P}' , we took out the first 3 dimensions and got:

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

2×6

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

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

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

$$\frac{\partial \mathbf{e}}{\partial \mathbf{P}} = \frac{\partial \mathbf{e}}{\partial \mathbf{P}'} \frac{\partial \mathbf{P}'}{\partial \mathbf{P}}. \quad (7.47)$$

The first item has been derived before, with regard to the second item, by definition

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

We found that \mathbf{P}' will only leave \mathbf{R} after \mathbf{P} . then:

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

Thus, we derive the two derivative matrices of the camera equation for camera pose and feature points. They are **important**, which provides important gradient directions during the optimization process and guides the iteration of optimization.

7.8 Practice: Solving PnP

7.8.1 Using EPnP to solve poses

Below, we understand the process of PnP through experiments. First, we demonstrate how to solve the PnP problem using OpenCV's EPnP and then solve it again through nonlinear optimization. In the second edition, we will add a handwritten optimization experiment. Since PnP needs to use 3D points, in order to avoid the trouble caused by initialization, we used the depth map (1_depth.png) in the RGB-D camera as the 3D position of the feature point. First look at the PnP functions provided by OpenCV:

Listing 7.9: slambook2/ch7/pose_estimation_3d2d.cpp(fragment)

```
1 Int main( int argc, char** argv ) {
```

```

2   Mat r, t;
3   solvePnP(pts_3d, pts_2d, K, Mat(), r, t, false); // Call OpenCV's PnP
   solution, select EPnP, DLS, etc.
4   Mat R;
5   Cv::Rodrigues(r, R); // r is a rotation vector form, converted to a
   matrix using the Rodrigues formula
6   Cout << "R=" << endl << R << endl;
7   Cout << "t=" << endl << t << endl;
8 }
```

In the routine, after getting the paired feature points, we look for their depth in the depth map of the first graph and find the spatial position. The spatial position is 3D point, and then the pixel position of the second image is 2D point, and EPnP is called to solve the PnP problem. The program output is as follows:

Listing 7.10: terminal input:

```

1 % build/pose_estimation_3d2d 1.png 2.png d1.png d2.png
2 — Max dist : 95.000000
3 — Min dist : 4.000000
4 A total of 79 matching points were found.
5 3d-2d pairs: 76
6 R=
7 [0.9978662025826269, -0.05167241613316376, 0.03991244360207524;
8 0.0505958915956335, 0.998339762771668, 0.02752769192381471;
9 -0.04126860182960625, -0.025449547736074, 0.998823919929363]
10 t=
11 [-0.1272259656955879;
12 -0.007507297652615337;
13 0.06138584177157709]
```

The reader can compare the \mathbf{R} , \mathbf{t} solved in the previous 2D–2D case to see what is different. It can be seen that when there is 3D information, the estimated \mathbf{R} is almost the same, and \mathbf{t} is much different. This is due to the introduction of new depth information. However, since the depth map acquired by Kinect has some errors, the 3D points here are not accurate. In larger BAs, we would like to optimize both pose and all 3D feature points.

7.8.2 Handwritten pose estimation

The following shows how to calculate the camera pose using nonlinear optimization. We first write a PnP of Gaussian Newton method, and then demonstrate how to call g2o to solve.

Listing 7.11: slambook2/ch7/pose_estimation_3d2d.cpp

```

1 void bundleAdjustmentGaussNewton(
2 const VecVector3d &points_3d,
3 const VecVector2d &points_2d,
4 const Mat &K,
5 Sophus::SE3d &pose) {
6     typedef Eigen::Matrix<double, 6, 1> Vector6d;
7     const int iterations = 10;
8     double cost = 0, lastCost = 0;
9     double fx = K.at<double>(0, 0);
10    double fy = K.at<double>(1, 1);
11    double cx = K.at<double>(0, 2);
12    double cy = K.at<double>(1, 2);
13
14    for (int iter = 0; iter < iterations; iter++) {
```

```

15 Eigen::Matrix<double, 6, 6> H = Eigen::Matrix<double, 6, 6>::Zero();
16 Vector6d b = Vector6d::Zero();
17
18 cost = 0;
19 // compute cost
20 for (int i = 0; i < points_3d.size(); i++) {
21     Eigen::Vector3d pc = pose * points_3d[i];
22     double inv_z = 1.0 / pc[2];
23     double inv_z2 = inv_z * inv_z;
24     Eigen::Vector2d proj(fx * pc[0] / pc[2] + cx, fy * pc[1] / pc[2] + cy
25 );
26     Eigen::Vector2d e = points_2d[i] - proj;
27     cost += e.squaredNorm();
28     Eigen::Matrix<double, 2, 6> J;
29     J << -fx * inv_z,
30     0,
31     fx * pc[0] * inv_z2,
32     fx * pc[0] * pc[1] * inv_z2,
33     -fx - fx * pc[0] * pc[0] * inv_z2,
34     fx * pc[1] * inv_z,
35     0,
36     -fy * inv_z,
37     fy * pc[1] * inv_z,
38     fy + fy * pc[1] * pc[1] * inv_z2,
39     -fy * pc[0] * pc[1] * inv_z2,
40     -fy * pc[0] * inv_z;
41
42     H += J.transpose() * J;
43     b += -J.transpose() * e;
44 }
45
46 Vector6d dx;
47 dx = H.ldlt().solve(b);
48
49 if (isnan(dx[0])) {
50     cout << "result is nan!" << endl;
51     break;
52 }
53
54 if (iter > 0 && cost >= lastCost) {
55     // cost increase, update is not good
56     cout << "cost: " << cost << ", last cost: " << lastCost << endl;
57     break;
58 }
59
60 // update your estimation
61 pose = Sophus::SE3d::exp(dx) * pose;
62 lastCost = cost;
63
64 cout << "iteration " << iter << " cost=" << cout.precision(12) << cost
65     << endl;
66 if (dx.norm() < 1e-6) {
67     // converge
68     break;
69 }
70
71 cout << "pose by g-n: \n" << pose.matrix() << endl;
72 }
```

In this small function, we implement a simple Gauss-Newton iterative optimization based on the previous theoretical derivation. We will then compare the efficiency

differences between OpenCV, handwriting implementations, and g2o implementations.

7.8.3 BA optimization with g2o

After handwriting the optimization process, let's look at how to implement the same operation with g2o (in fact, using Ceres is completely similar). The basics of g2o have already been introduced in the ?? talk. Before using g2o, we need to model the problem as a graph optimization problem, as shown by ?? .

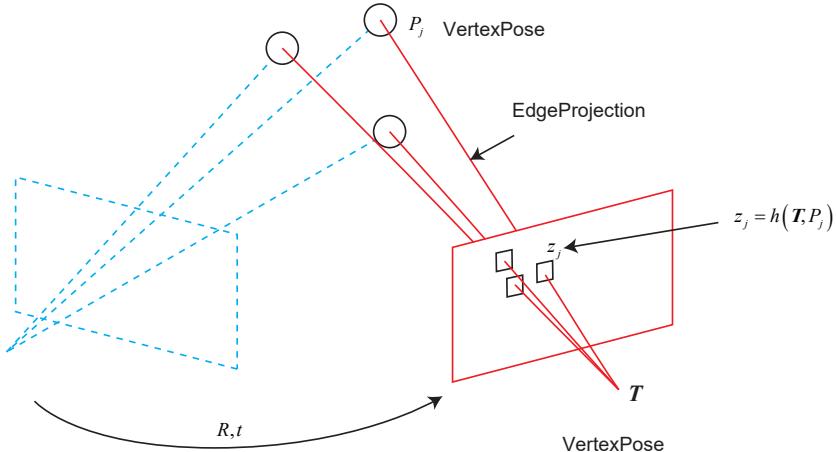


Figure 7-15: PnP's Bundle Adjustment's graph optimization representation.

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

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

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

Since the first camera pose is fixed at zero, we didn't write it into the optimization variable, but on more occasions, we will consider more camera estimates. Now we estimate the pose of the second camera based on a set of 3D points and a 2D projection in the second image. So we draw the first camera as a dotted line, indicating that we don't want to consider it.

G2o provides a number of nodes and edges for BA, such as g2o/types/sba/types_six_dof_expmmap.h which provides nodes and edges for Lie algebra expressions. In the second edition, we implement a VertexPose vertex and an EdgeProjection edge ourselves, as follows:

Listing 7.12: slambook2/ch7/pose_estimation_3d2d.cpp

```

1 // vertex and edges used in g2o ba
2 class VertexPose : public g2o::BaseVertex<6, Sophus::SE3d> {
3     public:
4         EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
5         virtual void setToOriginImpl() override {

```

```

7     _estimate = Sophus::SE3d();
8 }
9
10 /// left multiplication on SE3
11 virtual void oplusImpl(const double *update) override {
12     Eigen::Matrix<double, 6, 1> update_eigen;
13     update_eigen << update[0], update[1], update[2], update[3], update[4],
14         update[5];
15     _estimate = Sophus::SE3d::exp(update_eigen) * _estimate;
16 }
17
18 virtual bool read(istream &in) override {}
19
20 virtual bool write(ostream &out) const override {}
21 };
22
23 class EdgeProjection : public g2o::BaseUnaryEdge<2, Eigen::Vector2d,
24     VertexPose> {
25 public:
26     EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
27
28     EdgeProjection(const Eigen::Vector3d &pos, const Eigen::Matrix3d &K) :
29         _pos3d(pos), _K(K) {}
30
31     virtual void computeError() override {
32         const VertexPose *v = static_cast<VertexPose *>(_vertices[0]);
33         Sophus::SE3d T = v->estimate();
34         Eigen::Vector3d pos_pixel = _K * (T * _pos3d);
35         pos_pixel /= pos_pixel[2];
36         _error = _measurement - pos_pixel.head<2>();
37     }
38
39     virtual void linearizeOplus() override {
40         const VertexPose *v = static_cast<VertexPose *>(_vertices[0]);
41         Sophus::SE3d T = v->estimate();
42         Eigen::Vector3d pos_cam = T * _pos3d;
43         double fx = _K(0, 0);
44         double fy = _K(1, 1);
45         double cx = _K(0, 2);
46         double cy = _K(1, 2);
47         double X = pos_cam[0];
48         double Y = pos_cam[1];
49         double Z = pos_cam[2];
50         double Z2 = Z * Z;
51         _jacobianOplusXi
52             << -fx / Z, 0, fx * X / Z2, fx * X * Y / Z2, -fx - fx * X * X / Z2, fx
53                 * Y / Z,
54             0, -fy / Z, fy * Y / (Z * Z), fy + fy * Y * Y / Z2, -fy * X * Y / Z2, -
55                 fy * X / Z;
56     }
57
58     virtual bool read(istream &in) override {}
59
60     virtual bool write(ostream &out) const override {}
61
62     private:
63     Eigen::Vector3d _pos3d;
64     Eigen::Matrix3d _K;
65 };

```

Here, the update of the vertex and the error calculation of the edge are implemented. Here's how to group them into a graph optimization problem:

Listing 7.13: slambook2/ch7/pose_estimation_3d2d.cpp

```

1 void bundleAdjustmentG2O(
2     const VecVector3d &points_3d,
3     const VecVector2d &points_2d,
4     const Mat &K,
5     Sophus::SE3d &pose) {
6     // g2o::BlockSolver<g2o::BlockSolverTraits<6, 3>> BlockSolverType;
7     // pose is 6, landmark is 3
8     typedef g2o::LinearSolverDense<BlockSolverType::PoseMatrixType>
9         LinearSolverType; // g2o::BlockSolver<g2o::LinearSolverType>;
10    // g2o::OptimizationAlgorithmGaussNewton<
11        g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>());
12    );
13    g2o::SparseOptimizer optimizer; // g2o::SparseOptimizer
14    optimizer.setAlgorithm(solver); // g2o::OptimizationAlgorithmGaussNewton<
15    optimizer.setVerbose(true); // g2o::SparseOptimizer
16
17    // vertex
18    VertexPose *vertex_pose = new VertexPose(); // camera vertex_pose
19    vertex_pose->setId(0);
20    vertex_pose->setEstimate(Sophus::SE3d());
21    optimizer.addVertex(vertex_pose);
22
23    // K
24    Eigen::Matrix3d K_eigen;
25    K_eigen <<
26        K.at<double>(0, 0), K.at<double>(0, 1), K.at<double>(0, 2),
27        K.at<double>(1, 0), K.at<double>(1, 1), K.at<double>(1, 2),
28        K.at<double>(2, 0), K.at<double>(2, 1), K.at<double>(2, 2);
29
30    // edges
31    int index = 1;
32    for (size_t i = 0; i < points_2d.size(); ++i) {
33        auto p2d = points_2d[i];
34        auto p3d = points_3d[i];
35        EdgeProjection *edge = new EdgeProjection(p3d, K_eigen);
36        edge->setId(index);
37        edge->setVertex(0, vertex_pose);
38        edge->setMeasurement(p2d);
39        edge->setInformation(Eigen::Matrix2d::Identity());
40        optimizer.addEdge(edge);
41        index++;
42    }
43
44    chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
45    optimizer.setVerbose(true);
46    optimizer.initializeOptimization();
47    optimizer.optimize(10);
48    chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
49    chrono::duration<double> time_used = chrono::duration_cast<chrono::
50        duration<double>>(t2 - t1);
51    cout << "optimization costs time: " << time_used.count() << " seconds."
52        << endl;
53    cout << "pose estimated by g2o =\n" << vertex_pose->estimate().matrix()
54        << endl;
55    pose = vertex_pose->estimate();
56 }
```

The program is roughly similar to g2o in Lecture 6. We first declare the g2o graph optimizer and configure the optimization solver and gradient descent method.

Then place the pose and the spatial point into the graph based on the estimated feature points. Finally, the optimization function is called to solve. Finally, the partial output of the run is as follows:

Listing 7.14:

```

1 ./build/pose_estimation_3d2d 1.png 2.png 1_depth.png 2_depth.png
2 — Max dist : 95.000000
3 — Min dist : 4.000000 0 0 0 0 0 0 0 0
4 79
5 3d-2d pairs: 76
6 solve pnp in opencv cost time: 0.000332991 seconds.
7 R=
8 [0.9978662025826269, -0.05167241613316376, 0.03991244360207524;
9 0.0505958915956335, 0.998339762771668, 0.02752769192381471;
10 -0.04126860182960625, -0.025449547736074, 0.998823919929363]
11 t=
12 [-0.1272259656955879;
13 -0.007507297652615337;
14 0.06138584177157709]
15 calling bundle adjustment by gauss newton
16 iteration 0 cost=645538.1857253
17 iteration 1 cost=12750.239874896
18 iteration 2 cost=12301.774589343
19 iteration 3 cost=12301.427574651
20 iteration 4 cost=12301.426806652
21 pose by g-n:
22 0.99786618832 -0.0516873580423 0.039893448423 -0.127218696289
23 0.0506143671126 0.998340854865 0.0274540224544 -0.00738695798083
24 -0.0412462852904 -0.0253762590968 0.998826706403 0.0617019263823
25 0 0 0 1
26 solve pnp by gauss newton cost time: 0.000159492 seconds.
27 calling bundle adjustment by g2o
28 iteration= 0 chi2= 413.390599 time= 2.7291e-05 cumTime= 2.7291e-05
29 edges= 76 schur= 0 lambda= 79.000412 levenbergIter= 1
30 iteration= 1 chi2= 301.367030 time= 1.47e-05 cumTime= 4.1991e-05
31 edges= 76 schur= 0 lambda= 26.333471 levenbergIter= 1
32 iteration= 2 chi2= 301.365779 time= 1.7794e-05 cumTime= 5.9785e-05
33 edges= 76 schur= 0 lambda= 17.555647 levenbergIter= 1
34 iteration= 3 chi2= 301.365779 time= 1.4875e-05 cumTime= 7.466e-05
35 edges= 76 schur= 0 lambda= 11.703765 levenbergIter= 1
36 iteration= 4 chi2= 301.365779 time= 1.3132e-05 cumTime= 8.7792e-05
37 edges= 76 schur= 0 lambda= 7.802510 levenbergIter= 1
38 iteration= 5 chi2= 301.365779 time= 2.0379e-05 cumTime= 0.000108171
39 edges= 76 schur= 0 lambda= 41.613386 levenbergIter= 3
40 iteration= 6 chi2= 301.365779 time= 3.4186e-05 cumTime= 0.000142357
41 edges= 76 schur= 0 lambda= 2859650082279.672363 levenbergIter= 8
42 optimization costs time: 0.000763649 seconds.
43 pose estimated by g2o =
44 0.997866202583 -0.0516724161336 0.0399124436024 -0.127225965696
45 0.050595891596 0.998339762772 0.0275276919261 -0.00750729765631
46 -0.04126860183 -0.0254495477384 0.998823919929 0.0613858417711
47 0 0 0 1
48 solve pnp by g2o cost time: 0.000923095 seconds.

```

From the estimation results, the three are basically the same. From the optimization time point of view, our own implementation of the Gauss Newton method ranked first with 0.15 milliseconds, followed by OpenCV's PnP, and finally g2o implementation. Despite this, the three uses less than 1 millisecond, which means that the pose estimation algorithm does not consume computation.

Bundle Adjustment is a common practice. It can be not limited to two images.

We can iteratively optimize the poses and spatial points matched by multiple images, and even put the entire SLAM process into it. That kind of approach is large, mainly used in the back end, we will encounter this problem again in the 10th lecture. At the front end, we usually consider the small Bundle Adjustment problem of the local camera pose and feature points, and hope to solve and optimize it in real time.

7.9 3D–3D:ICP

Finally, let's introduce the pose estimation problem of 3D–3D. Suppose we have a set of paired 3D points (for example, we matched two RGB-D images):

$$\mathbf{P} = \{\mathbf{p}_1, \dots, \mathbf{p}_n\}, \quad \mathbf{P}' = \{\mathbf{p}'_1, \dots, \mathbf{B}\mathbf{m}\mathbf{p}'_n\},$$

Now, I want to find an Euclidean transformation of \mathbf{R}, \mathbf{t} , which makes the *:

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

This problem can be solved with the Iterative Closest Point (ICP). The reader should note that the camera model does not appear in the 3D–3D pose estimation problem, that is, it only has no relationship with the camera when considering the transition between the two sets of 3D points. Therefore, ICP is also encountered in the laser SLAM, but because the laser data features are not rich enough, we have no way to know the **matching relationship** between the two point sets, we can only think that the two closest points are the same. So this method is called the iterative closest point. In the visual, the feature points provide us with a good matching relationship, so the whole problem becomes simpler. In RGB-D SLAM, the camera pose can be estimated in this way. Below we use ICP to refer to the motion estimation problem between the two sets of **matched** points.

Similar to PnP, ICP is solved in two ways: by linear algebra (mainly SVD) and by nonlinear optimization (similar to Bundle Adjustment). The following is introduced separately.

7.9.1 SVD method

First look at the algebraic method represented by SVD. Based on the ICP problem described earlier, we first define the error term for the i pair:

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

Then, build a least squares problem and find the sum of the squared errors to a minimum of \mathbf{R}, \mathbf{t} :

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

Let's deduce its solution. First, define the centroids of the two sets of points:

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

* example slightly different from the symbols in the first two chapters. If you associate them, then look at \mathbf{p}_i as the data in the second image and \mathbf{p}'_i as the data in the first image. \mathbf{R}, \mathbf{t} is consistent.

Please note that the centroid is not subscripted. Then, do the following processing in the error function:

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

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

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

Looking closely at the left and right, we find that the left side is only related to the rotation matrix \mathbf{R} , while the right side has both \mathbf{R} and \mathbf{t} , but only with centroid. As long as we get \mathbf{R} , we can get \mathbf{t} if the second item is zero. Therefore, ICP can be solved in the following three steps:

1. calculates the centroid positions \mathbf{p}, \mathbf{p}' for the two sets of points, and then calculates the **decent centroid coordinates** for each point:

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

2. calculates the rotation matrix based on the following optimization problems:

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

3. calculates \mathbf{t} from \mathbf{R} in step 2:

$$\mathbf{t}^* = \mathbf{p} - \mathbf{R}\mathbf{p}'. \quad (7.54)$$

We see that as long as the rotation between the two sets of points is found, the amount of translation is very easy to obtain. So we focus on the calculation of \mathbf{R} . Expand the error term for \mathbf{R} and get:

$$\frac{1}{2} \sum_{i=1}^n \| \mathbf{q}_i - \mathbf{R}\mathbf{q}'_i \|^2 = \frac{1}{2} \sum_{i=1}^n \mathbf{q}_i^T \mathbf{q}_i + \mathbf{q}_i^{PrimeT} \mathbf{R}^T \mathbf{R} \mathbf{q}'_i - 2\mathbf{q}_i^T \mathbf{R} \mathbf{q}'_i. \quad (7.55)$$

Notice that the first item is not related to \mathbf{R} , and the second item is due to $\mathbf{R}^T \mathbf{R} = \mathbf{I}$, also with \mathbf{R} has nothing to do. Therefore, in fact, the optimization objective function becomes

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

Next, we describe how to solve the best \mathbf{R} of the above problem through SVD. Proof of optimality is more complicated, and interested readers can refer to the

literature [? ?]. To understand \mathbf{R} , first define the matrix:

$$\mathbf{W} = \sum_{i=1}^n \mathbf{q}_i \mathbf{q}_i'^T. \quad (7.57)$$

\mathbf{W} is a matrix of 3×3 , SVD decomposition of \mathbf{W} , which gives:

$$\mathbf{W} = \mathbf{U} \Sigma \mathbf{V}^T. \quad (7.58)$$

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

$$\mathbf{R} = \mathbf{U} \mathbf{V}^T. \quad (7.59)$$

After \mathbf{R} is solved, t can be solved by (??). If the determinant of \mathbf{R} is negative at this time, then $-\mathbf{R}$ is taken as the optimal value.

7.9.2 Nonlinear optimization method

Another way to solve ICP is to use nonlinear optimization to find the optimal value in an iterative manner. This method is very similar to the PnP we described earlier. When expressing a pose with Lie algebra, the objective function can be written as

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

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

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

Thus, in nonlinear optimization, it is only necessary to continuously iterate to find a minimum value. Moreover, it can be proved that [?] has a unique solution or an infinite number of solutions to the ICP problem. In the case of a unique solution, as long as a minimal solution is found, then **this minimum value is the global optimal value** - so there is no case of local minima rather than global minima. This also means that the ICP solution can arbitrarily select the initial value. This is a big benefit of solving ICP when matching points.

It should be noted that the ICP we are talking about here refers to the problem of pose estimation under the condition that the image features have been matched. In the case where the matching is known, this least squares problem actually has an analytical solution [? ? ?], so there is no need for iterative optimization. ICP researchers are often more concerned with matching unknown situations. So why do we want to introduce optimization-based ICP? This is because, in some cases, such as in RGB-D SLAM, the depth data of one pixel may or may not be measured, so we can mix and match PnP and ICP optimization: for feature points with known depth, Model their 3D–3D errors; for feature points with unknown depths, model the 3D–2D reprojection error. Thus, all errors can be considered in the same problem, making the solution more convenient.

7.10 Practice: Solving ICP

7.10.1 SVD method

The following demonstrates how to solve ICP using SVD and nonlinear optimization. In this section, we use two RGB-D images to obtain two sets of 3D points by feature matching, and finally calculate their pose transformation by ICP. Since OpenCV does not currently calculate two sets of ICP methods with matching points, and its principle is not complicated, we implement an ICP ourselves.

Listing 7.15: slambook2/ch7/pose_estimation_3d3d.cpp

```

1 void pose_estimation_3d3d(
2     const vector<Point3f> &pts1,
3     const vector<Point3f> &pts2,
4     Mat &R, Mat &t) {
5     Point3f p1, p2; // center of mass
6     int N = pts1.size();
7     for (int i = 0; i < N; i++) {
8         p1 += pts1[i];
9         p2 += pts2[i];
10    }
11    p1 = Point3f(Vec3f(p1) / N);
12    p2 = Point3f(Vec3f(p2) / N);
13    vector<Point3f> q1(N), q2(N); // remove the center
14    for (int i = 0; i < N; i++) {
15        q1[i] = pts1[i] - p1;
16        q2[i] = pts2[i] - p2;
17    }
18
19    // compute q1*q2^T
20    Eigen::Matrix3d W = Eigen::Matrix3d::Zero();
21    for (int i = 0; i < N; i++) {
22        W += Eigen::Vector3d(q1[i].x, q1[i].y, q1[i].z) * Eigen::Vector3d(q2[i]
23            .x, q2[i].y, q2[i].z).transpose();
24    }
25    cout << "W=" << W << endl;
26
27    // SVD on W
28    Eigen::JacobiSVD<Eigen::Matrix3d> svd(W, Eigen::ComputeFullU | Eigen:::
29        ComputeFullV);
30    Eigen::Matrix3d U = svd.matrixU();
31    Eigen::Matrix3d V = svd.matrixV();
32
33    cout << "U=" << U << endl;
34    cout << "V=" << V << endl;
35
36    Eigen::Matrix3d R_ = U * (V.transpose());
37    if (R_.determinant() < 0) {
38        R_ = -R_;
39    }
40    Eigen::Vector3d t_ = Eigen::Vector3d(p1.x, p1.y, p1.z) - R_ * Eigen:::
41        Vector3d(p2.x, p2.y, p2.z);
42
43    // convert to cv::Mat
44    R = (Mat<double>(3, 3) <<
45        R_(0, 0), R_(0, 1), R_(0, 2),
46        R_(1, 0), R_(1, 1), R_(1, 2),
47        R_(2, 0), R_(2, 1), R_(2, 2)
48    );
49    t = (Mat<double>(3, 1) << t_(0, 0), t_(1, 0), t_(2, 0));
50}

```

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

Listing 7.16: terminal input:

```

1 ./build/pose_estimation_3d3d 1.png 2.png 1_depth.png 2_depth.png
2 — Max dist : 95.000000
3 — Min dist : 4.000000
4 A total of 79 matching points were found.
5 3d-3d pairs: 74
6 W= 11.9404 -0.567258 1.64182
7 -1.79283 4.31299 -6.57615
8 3.12791 -6.55815 10.8576
9 U= 0.474144 -0.880373 -0.0114952
10 -0.460275 -0.258979 0.849163
11 0.750556 0.397334 0.528006
12 V= 0.535211 -0.844064 -0.0332488
13 -0.434767 -0.309001 0.84587
14 0.724242 0.438263 0.532352
15 ICP via SVD results:
16 R = [0.9972395977366739, 0.05617039856770099, -0.04855997354553433;
17 -0.05598345194682017, 0.9984181427731508, 0.005202431117423125;
18 0.0487753812298326, -0.002469515369266572, 0.9988067198811421]
19 t = [0.1417248739257469;
20 -0.05551033302525193;
21 -0.03119093188273858]
22 R_inv = [0.9972395977366739, -0.05598345194682017, 0.0487753812298326;
23 0.05617039856770099, 0.9984181427731508, -0.002469515369266572;
24 -0.04855997354553433, 0.005202431117423125, 0.9988067198811421]
25 T_inv = [-0.1429199667309695;
26 0.04738475446275858;
27 0.03832465717628181]
```

The reader can compare the difference between the ICP and PnP, the motion estimation results of the polar geometry. It can be argued that in this process we are using more and more information (no depth - there is a depth of a graph - there are depths of two graphs), so in the case of depth accuracy, the resulting estimates will also come The more accurate. However, due to the noise of Kinect's depth map and the possibility of data loss, we have to discard some feature points without depth data. This may result in an inaccurate estimation of the ICP, and if the feature points are discarded too much, it may cause a situation in which motion estimation cannot be performed due to too few feature points.

7.10.2 Nonlinear optimization method

Let's consider nonlinear optimization to calculate ICP. We still use Lie algebra to optimize the camera pose. For us, the RGB-D camera can observe the three-dimensional position of the landmark points each time, resulting in a 3D observation. We use VertexPose from the previous experiment and then define the unary side of 3D-3D:

Listing 7.17: slambook2/ch7/pose_estimation_3d3d.cpp

```

1  /// g2o edge
2  class EdgeProjectXYZRGBDPoseOnly : public g2o::BaseUnaryEdge<3, Eigen::
3      Vector3d, VertexPose> {
4  public:
5      EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
6
7      EdgeProjectXYZRGBDPoseOnly(const Eigen::Vector3d &point) : _point(point)
8          {}
9
10     virtual void computeError() override {
11         const VertexPose *pose = static_cast<const VertexPose *>(_vertices[0]);
12         _error = _measurement - pose->estimate() * _point;
13     }
14
15     virtual void linearizeOplus() override {
16         VertexPose *pose = static_cast<VertexPose *>(_vertices[0]);
17         Sophus::SE3d T = pose->estimate();
18         Eigen::Vector3d xyz_trans = T * _point;
19         _jacobianOplusXi.block<3, 3>(0, 0) = -Eigen::Matrix3d::Identity();
20         _jacobianOplusXi.block<3, 3>(0, 3) = Sophus::SO3d::hat(xyz_trans);
21     }
22
23     bool read(istream &in) {}
24
25     bool write(ostream &out) const {}
26
27 protected:
28     Eigen::Vector3d _point;
29 };

```

g2o::EdgeSE3ProjectXYZ 2 3
6
g2o

3×

Listing 7.18:

```

1 iteration= 0    chi2= 1.811539   time= 1.7046e-05   cumTime= 1.7046e-05
2       edges= 74    schur= 0
3 iteration= 1    chi2= 1.811051   time= 1.0422e-05   cumTime= 2.7468e-05
4       edges= 74    schur= 0
5 iteration= 2    chi2= 1.811050   time= 9.589e-06    cumTime= 3.7057e-05
6       edges= 74    schur= 0[] []
7 ...
8 iteration= 9    chi2= 1.811050   time= 9.113e-06    cumTime= 0.000100604
9       edges= 74    schur= 0
10 optimization costs time: 0.000559208 seconds.
11
12 after optimization:
13 T=
14 0.99724  0.0561704  -0.04856  0.141725
15 -0.0559834  0.998418  0.00520242 -0.0555103
16 0.0487754 -0.0024695  0.998807 -0.0311913
17 0           0           0           1

```

We find that the overall error is stable after only one iteration, indicating that the algorithm converges only after one iteration. It can be seen from the result of the pose solution that it is almost identical to the pose result given by the previous SVD, which indicates that the SVD has given an analytical solution to the optimization problem. Therefore, in this experiment, 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 ICP of this example, we used feature points with depth readings in both figures. In reality, however, as long as one of the graph depths is determined, we can add them to the optimization using an error approach similar to PnP. At the same time, in addition to the camera pose, considering the space point as an optimization variable is also a way to solve the problem. We should be clear that the actual solution is very flexible and does not have to be stuck in a fixed form. If you consider points and cameras at the same time, the whole problem becomes **more free**, and you may get other solutions. For example, you can make the camera turn a little less and move the point a bit more. This reflects on the other hand, in the Bundle Adjustment, we would like to have as many constraints as possible, because multiple observations will bring more information, allowing us to estimate each variable more accurately.

7.11

This lecture introduces several important issues in feature-based visual odometers. 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 lecture is rich in content, and combined with the basic knowledge of the previous few lectures. If readers find it difficult to understand, they can review the previous knowledge. It is best to do the experiment yourself to understand the content of the entire motion estimation.

It should be explained that in order to ensure smooth writing, we have omitted a lot of discussions about certain special situations. For example, what happens if a given feature point is coplanar during the solution to the polar geometry (this is mentioned in the homography matrix \mathbf{H})? What happens to the collinear line? What would happen if such a solution was given in PnP and ICP? Can the algorithm identify these particular situations and report that the resulting solution may be unreliable? Can you give an estimate of the uncertainty of \mathbf{T} ? Although they are worthy of research and exploration, their discussion is more suitable for staying in specific papers. The goal of this book is extensive knowledge coverage and basic knowledge. We are not working on these issues, and they are rarely seen in engineering implementation. If you are concerned about these rare situations, you can read papers such as [?].

Exercises

1. In addition to the ORB feature points introduced in this book, which feature points can you find? Please talk about the principles of SIFT or SURF and compare the advantages and disadvantages between them and ORB.

2. The design program calls other kinds of feature points in OpenCV. Counts the 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 throughout the image. Can you find or propose a way to make the distribution of feature points more uniform?
4. investigates why FLANN can handle matching problems quickly. In addition to FLANN, what other means of speeding up the match?
5. changes the EPnP used by the demo to other PnP methods and studies how they work.
6. In PnP optimization, the observation of the first camera is also taken into account, how should the program be written? What will happen to the final result?
7. In the ICP program, the spatial point is also taken into account as an optimization variable. How should the program be written? What will happen to the final result?
- 8.*In the process of feature point matching, it is inevitable that a mismatch will be encountered. What happens if we enter the wrong match into PnP or ICP? What methods can you think of to avoid mismatches?
- 9.*Use Sophus's SE3 class to design g2o nodes and edges to optimize PnP and ICP.
- 10.*Optimize PnP and ICP in Ceres.

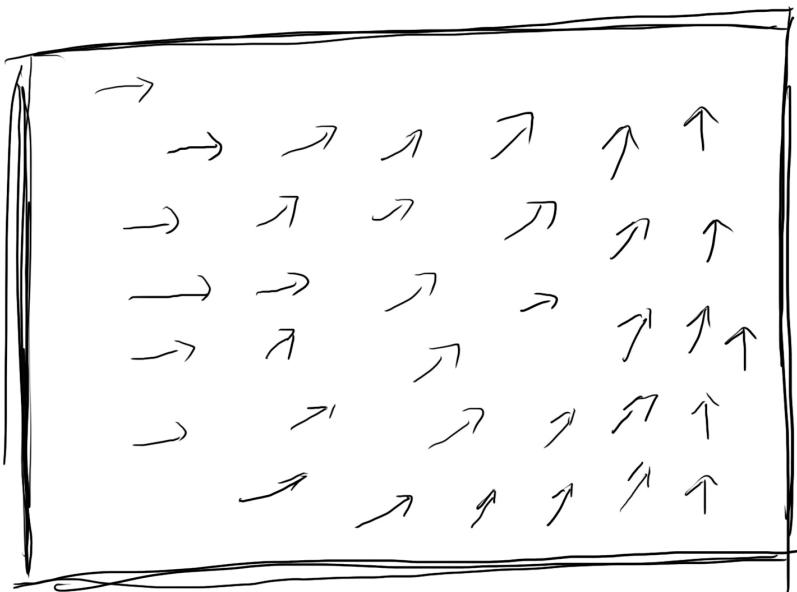
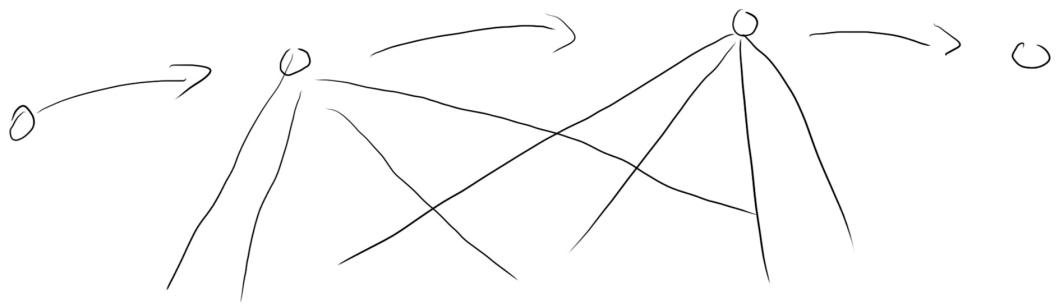
Chapter 8

visual odometer 2

main target

1. Understand the principle of optical flow tracking feature points.
2. Understand how the direct method estimates the pose of the camera.
3. uses g2o for direct method calculations.

The direct method is another major branch of the visual odometer, which is quite different from the feature point method. Although it has not become the mainstream of the current VO, but after several years of development, the direct method has been able to equal the feature point method to a certain extent. In this lecture we will introduce the principles of direct law and implement the core part of the direct method.



$$\min \| I_1(p) - I_2(K(Rp+t)) \|_2$$

$$J = \frac{\partial J}{\partial n} \quad \frac{\partial u}{\partial \ell} \quad \frac{\partial v}{\partial \{}}$$

8.1 Direct method extraction

In the last lecture, we introduced the method of estimating camera motion using feature points. Although the feature point method dominates the visual odometer, researchers have realized that it has at least the following shortcomings:

1. The extraction of key points and the calculation of descriptors are time consuming. In practice, SIFT is currently not calculated in real time on the CPU, and ORB requires nearly 20ms of calculation. If the entire SLAM is running at 30ms/frame, then most of the time will be spent calculating feature points.
2. When uses feature points, all information except feature points is ignored. An image has hundreds of thousands of pixels, and the feature points are only a few hundred. Most of the **maybe useful** image information is discarded using only feature points.
3. The camera sometimes moves to the **feature missing** place, which often has no obvious texture information. For example, sometimes we 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.

We see that there are some problems with the use of feature points. Is there any way to overcome these shortcomings? We have the following ideas:

- retains feature points, but only computes key points and does not calculate descriptors. At the same time, use **Optical Flow** to track the motion of feature points. This avoids the time required to calculate and match the descriptor, and the calculation time of the optical flow itself is smaller than the calculation and matching of the descriptor.
- only computes key points and does not calculate descriptors. At the same time, use **Direct Method** to calculate the position of the feature point in the image at the next moment. This also skips the calculation process of the descriptor and also saves the calculation time of the optical flow.

The first method still uses feature points, but replaces the matching descriptor with optical flow tracking. It is estimated that the camera geometry is still using the inverse geometry, PnP or ICP algorithm. This will still require the key points extracted to be distinguishable, that is, we need to mention the corner points. In the direct method, we estimate the motion of the camera and the projection of the point based on the **pixel grayscale information** of the image. The point that is not required to be extracted must be a corner point. As will be seen later, they can even be random selection points.

When estimating the camera motion using the feature point method, we regard the feature point as a fixed point fixed in the three-dimensional space. Camera motion is optimized by **Reprojection error** based on their projected position in the camera. In this process, we need to know exactly where the pixel points are projected in the two cameras - this is why we want to match or track the features. At the same time, we also know that computing and matching features require a lot of computation. In contrast, in the direct method, we do not need to know

the correspondence between points, but to find them by minimizing **Photometric error**.

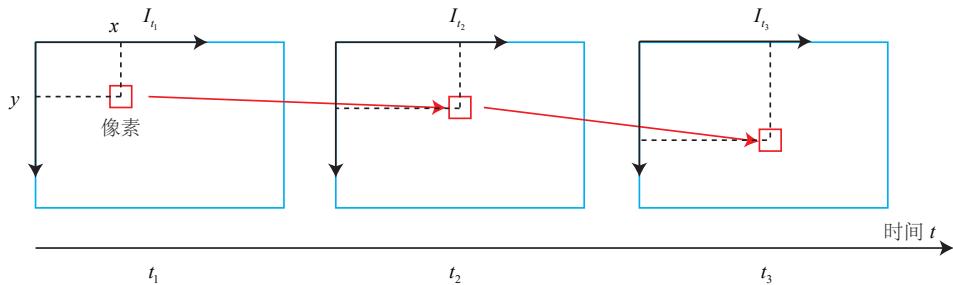
The direct method is the focus of this lecture. It exists to overcome the above disadvantages of the feature point method. The direct method estimates the motion of the camera based on the luminance information of the pixel, and can completely eliminate the calculation of key points and descriptors, thus avoiding the calculation time of the feature and avoiding the missing feature. As long as there are light and dark changes in the scene (which can be gradients without local image gradients), the direct method works. According to the number of pixels used, the direct method is divided into three types: sparse, dense, and semi-dense. Compared with the feature point method, only the sparse feature points (sparse maps) can be reconstructed, and the direct method also has the ability to restore dense or semi-dense structures.

Historically, there was also the use of direct law in the early days [?]. With the emergence of open source projects using direct methods (such as SVO[?], LSD-SLAM[?], DSO[?], etc. They are gradually taking the mainstream stage and becoming an important part of the visual mileage calculation.

8.2 2D Optical Flow

The direct method evolved from the optical flow. They are very similar and have the same assumptions. The optical flow describes the motion of the pixel in the image, while the direct law is accompanied by a camera motion model. In order to illustrate the direct method, we may wish to introduce the optical flow first.

Optical flow is a method of describing the movement of pixels between images over time, as shown by ?? . As time goes by, the same pixel will move in the image, and we want to track its motion. Among them, the calculation of partial pixel motion is called **sparse optical flow**, and the calculation of all pixels is called **dense optical flow**. The sparse optical flow is represented by Lucas-Kanade optical flow [?] and can be used to track feature point locations in SLAM. The dense optical flow is represented by Horn-Schunck optical flow [?]. Therefore, this section mainly introduces Lucas-Kanade optical flow, also known as LK optical flow.



$$\text{灰度不变假设: } I(x_1, y_1, t_1) = I(x_2, y_2, t_2) = I(x_3, y_3, t_3)$$

Figure 8-1: LK optical flow diagram.

Lucas-Kanade Optical Stream

In the LK optical flow, we think that the image from the camera changes over time. The image can be thought of as a function of time: $\mathbf{I}(t)$. Then, at the time of t , the

pixel at (x, y) , its gray scale can be written as

$$\mathbf{I}(x, y, t).$$

This way the image is seen as a function of position and time, and its range is the gray level of the pixels in the image. Now consider a fixed spatial point whose pixel coordinates at t is x, y . Due to the motion of the camera, its image coordinates will change. We want to estimate the position of this spatial point in the image at other times. How to estimate it? Here we introduce the basic assumptions of the optical flow method.

Gray invariant hypothesis: The pixel gray value of the same spatial point is fixed in each image.

For the pixel where t is at (x, y) , we set $t + dt$ to move it to $(x + dx, y + dy)$. Since the gray level is unchanged, we have:

$$\mathbf{I}(x + dx, y + dy, t + dt) = \mathbf{I}(x, y, t). \quad (8.1)$$

Note that the gray-scale invariant assumption is a strong assumption, and it may not be true in practice. In fact, due to the different materials of the object, the pixels will appear highlights and shadows; sometimes, the camera will automatically adjust the exposure parameters to make the image overall brighter or darker. At these times, the gray-scale invariant assumptions are not true, so the result of the optical flow is not necessarily reliable. However, on the other hand, all algorithms work under certain assumptions. If we don't make any assumptions, we can't design a practical algorithm. So let's assume that the assumption is true and see how to calculate the motion of the pixel.

Taylor expansion on the left, retaining the first-order item, you get:

$$\mathbf{I}(x + dx, y + dy, t + dt) \approx \mathbf{I}(x, y, t) + \frac{\partial \mathbf{I}}{\partial x} dx + \frac{\partial \mathbf{I}}{\partial y} dy + \frac{\partial \mathbf{I}}{\partial t} dt. \quad (8.2)$$

Because we assume that the gray level is constant, then the gray level at the next moment is equal to the previous gray level, thus:

$$\frac{\partial \mathbf{I}}{\partial x} dx + \frac{\partial \mathbf{I}}{\partial y} dy + \frac{\partial \mathbf{I}}{\partial t} dt = 0. \quad (8.3)$$

Dividing both sides by dt gives:

$$\frac{\partial \mathbf{I}}{\partial x} \frac{dx}{dt} + \frac{\partial \mathbf{I}}{\partial y} \frac{dy}{dt} = -\frac{\partial \mathbf{I}}{\partial t}. \quad (8.4)$$

Where dx/dt is the speed of the pixel on the x axis, and dy/dt is on the y axis. The speed, remember 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 $\mathbf{I}_x, \mathbf{I}_y$. Record the amount of change in grayscale of the image as \mathbf{I}_t , written in matrix form, with:

$$\begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = -\mathbf{I}_t. \quad (8.5)$$

We want to calculate the motion of the pixel u, v , but this is a one-time equation with two variables, which can't calculate u, v . Therefore, additional constraints

must be introduced to calculate u, v . In the LK optical flow, we assume that **pixels within a window have the same motion**.

Consider a window of size $w \times w$ that contains w^2 of pixels. Since the pixels in the window have the same motion, we have a total of w^2 equations:

$$\begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix}_k \begin{bmatrix} u \\ v \end{bmatrix} = -\mathbf{I}_{tk}, \quad k = 1, \dots, w^2. \quad (8.6)$$

Remember:

$$\mathbf{A} = \begin{bmatrix} [\mathbf{I}_x, \mathbf{I}_y]_1 \\ \vdots \\ [\mathbf{I}_x, \mathbf{I}_y]_k \end{bmatrix}, \mathbf{b} = \begin{bmatrix} \mathbf{I}_{t1} \\ \vdots \\ \mathbf{I}_{tk} \end{bmatrix}. \quad (8.7)$$

So the whole equation is

$$\mathbf{A} \begin{bmatrix} u \\ v \end{bmatrix} = -\mathbf{b}. \quad (8.8)$$

This is an overdetermined linear equation for u, v , and the traditional solution is to find the least squares solution. Least squares have been used many times:

$$\begin{bmatrix} u \\ v \end{bmatrix}^* = -(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}. \quad (8.9)$$

This gives the speed of movement of the pixels between the images u, v . When t takes discrete moments instead of continuous time, we can estimate where a block of pixels appears in several images. Since the pixel gradient is only valid locally, if one iteration is not good enough, we will iterate several times more. In SLAM, LK optical flow is often used to track the movement of corner points, we may wish to experience it through the program.

8.3 Practice: LK optical flow

8.3.1 Using LK optical flow

In the practice section, we will use several sample images to track the feature points above using OpenCV's optical flow. At the same time, we will also manually implement an LK optical flow to achieve a deeper understanding. We use two sample images from the Euroc dataset to extract the corners in the first image and then track their position in the second sheet with the optical flow. First let's use the LK optical flow in OpenCV:

Listing 8.1: slambook2/ch8/optical_flow.cpp(fragment)

```

1 // use opencv's flow for validation
2 Vector<Point2f> pt1, pt2;
3 For (auto &kp: kp1) pt1.push_back(kp.pt);
4 Vector<uchar> status;
5 Vector<float> error;
6 Cv::calcOpticalFlowPyrLK(img1, img2, pt1, pt2, status, error);

```

OpenCV's optical flow is very simple to use, just call the `cv::calcOpticalFlowPyrLK` function, provide two images before and after and the corresponding feature points, you can get the tracked points, as well as the status and error of each point. We can determine if the corresponding point is correctly tracked based on whether the

status variable is 1. This function has some optional parameters, but in the demo we only use the default parameters. We omit other code that mentions features and draws results here, which have been shown in previous programs.

8.3.2 Using Gauss-Newton method to achieve optical flow

single layer optical flow

Optical flow can also be viewed as an optimization problem: the optimal pixel offset is estimated by minimizing grayscale errors. So, similar to the various Gaussian Newton normalizations that were implemented before, we now also implement an optical flow based on the Gauss-Newton method.

Listing 8.2: slambook2/ch8/optical_flow.cpp(fragment)

```

44     auto kp = kp1[i];
45     double dx = 0, dy = 0; // dx,dy need to be estimated
46     if (has_initial) {
47         dx = kp2[i].pt.x - kp.pt.x;
48         dy = kp2[i].pt.y - kp.pt.y;
49     }
50
51     double cost = 0, lastCost = 0;
52     bool succ = true; // indicate if this point succeeded
53
54     // Gauss-Newton iterations
55     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 +
77                             dy + y) -
78                             GetPixelValue(img2, kp.pt.x + dx + x - 1, kp.pt.y + dy +
79                             y)),
80                         0.5 * (GetPixelValue(img2, kp.pt.x + dx + x, kp.pt.y + dy +
81                             y + 1) -
82                             GetPixelValue(img2, kp.pt.x + dx + x, kp.pt.y + dy + y - 1))
83                     );
84                 } else if (iter == 0) {
85                     // in inverse mode, J keeps same for all iterations
86                     // NOTE this J does not change when dx, dy is updated, so we
87                     // can store it and only compute error
88                     J = -1.0 * Eigen::Vector2d(
89                         0.5 * (GetPixelValue(img1, kp.pt.x + x + 1, kp.pt.y + y) -
90                             GetPixelValue(img1, kp.pt.x + x - 1, kp.pt.y + y)),
91                         0.5 * (GetPixelValue(img1, kp.pt.x + x, kp.pt.y + y + 1) -
92                             GetPixelValue(img1, kp.pt.x + x, kp.pt.y + y - 1))
93                     );
94                 }
95                 // compute H, b and set cost;
96                 b += -error * J;
97                 cost += error * error;
98                 if (inverse == false || iter == 0) {
99                     // also update H
100                     H += J * J.transpose();
101                 }
102             }
103
104             // 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
105     // is irreversible
106     cout << "update is nan" << endl;
107     succ = false;
108     break;
109 }
110
111 if (iter > 0 && cost > lastCost) {
112     break;
113 }
114
115 // update dx, dy
116 dx += update[0];
117 dy += update[1];
118 lastCost = cost;
119 succ = true;
120
121 if (update.norm() < 1e-2) {
122     // converge
123     break;
124 }
125
126 success[i] = succ;
127
128 // set kp2
129 kp2[i].pt = kp.pt + Point2f(dx, dy);
130
131 }
```

We implemented a single-layer optical flow function in the OpticalFlowSingleLevel function, which calls cv::parallel_for_ to call OpticalFlowTracker::calculateOpticalFlow in parallel, which calculates the optical flow of the feature points in the specified range. This parallel for loop is internally implemented by the Intel tbb library. We only need to define the function ontology according to its interface, and then pass the function as a std::function object to it.

In the concrete function implementation (ie calculateOpticalFlow), we solve such a problem:

$$\min_{\Delta x, \Delta y} \| \mathbf{I}_1(x, y) - \mathbf{I}_2(x + \Delta x, y + \Delta y) \|_2^2. \quad (8.10)$$

Therefore, the residual is the part inside the parentheses, and the corresponding Jacobi is the gradient of the second image at $x + \Delta x, y + \Delta y$. In addition, according to the literature [?], the gradient here can also be replaced by the gradient of the first image $\mathbf{I}_1(x, y)$. This alternative method is called the **inverse** optical flow method. In the reverse optical flow, the gradient of $\mathbf{I}_1(x, y)$ remains the same, so we can retain the result of the calculation on the first iteration and use it in subsequent iterations. When the Jacobian is unchanged, the \mathbf{H} matrix is unchanged, and only one residual is calculated for each iteration, which saves a portion of the calculation.

Multilayer optical flow

Since we write the optical flow as an optimization problem, we must assume that the initial value of the optimization is close to the optimal value, in order to guarantee the convergence of the algorithm to a certain extent. Therefore, if the camera moves

faster and the difference between the two images is more obvious, the single-layer image optical flow method easily reaches a local minimum value. This situation can be improved by introducing an image pyramid.

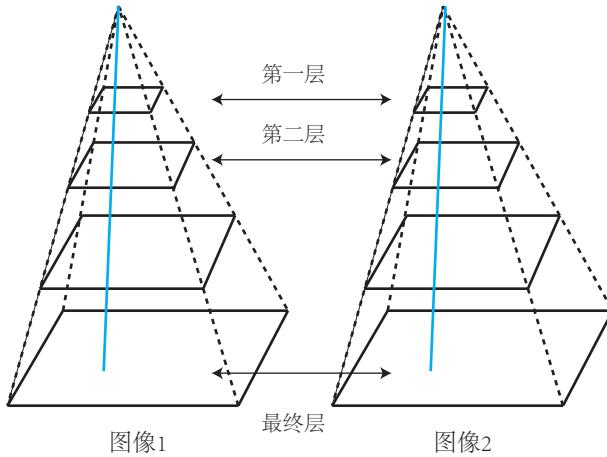


Figure 8-2: Image pyramid and Coarse-to-fine process.

Image pyramid refers to scaling the same image to get images at different resolutions, as shown by ???. Taking the original image as the bottom layer of the pyramid, each time the layer is up, the lower layer image is scaled at a certain magnification to obtain a pyramid. Then, when calculating the optical flow, the calculation is started from the top image, and then the tracking result of the upper layer is taken as the initial value of the next optical flow. Since the image of the upper layer is relatively rough, this process is also called **Coarse-to-fine** optical flow, which is also the usual flow of practical optical flow method.

The advantage from coarse to fine is that when the pixel motion of the original image is large, the motion is still in a small range in the view of the image at the top of the pyramid. For example, the feature points of the original image are moved by 20 pixels, and it is easy to be trapped in the minimum value due to the non-convexity of the image. But now suppose that there is a pyramid with a magnification of 0.5 times, then in the upper two images, the pixel motion is only 5 pixels, and the result is obviously better than optimizing directly on the original image.

We implemented a multi-layer optical flow in the program, the code is as follows:

Listing 8.3: slambook2/ch8/optical_flow.cpp(fragment)

```

1 Void OpticalFlowMultiLevel(
2 Const Mat &img1,
3 Const Mat &img2,
4 Const vector<KeyPoint> &kp1,
5 Vector<KeyPoint> &kp2,
6 Vector<bool> &success,
7 Bool inverse) {
8
9 // parameters
10 Int pyramids = 4;
11 Double pyramid_scale = 0.5;
12 Double scales[] = {1.0, 0.5, 0.25, 0.125};
13

```

```

14 // create pyramids
15 Vector<Mat> pyr1, pyr2; // image pyramids
16 For (int i = 0; i < pyramids; i++) {
17 If (i == 0) {
18 Pyr1.push_back(img1);
19 Pyr2.push_back(img2);
20 } else {
21 Mat img1_pyr, img2_pyr;
22 Cv::resize(pyr1[i - 1], img1_pyr,
23 Cv::Size(pyr1[i - 1].cols * pyramid_scale, pyr1[i - 1].rows * pyramid_scale
24 )););
25 Cv::resize(pyr2[i - 1], img2_pyr,
26 Cv::Size(pyr2[i - 1].cols * pyramid_scale, pyr2[i - 1].rows * pyramid_scale
27 )););
28 Pyr1.push_back(img1_pyr);
29 Pyr2.push_back(img2_pyr);
30 }
31 }
32 // coarse-to-fine LK tracking in pyramids
33 Vector<KeyPoint> kp1_pyr, kp2_pyr;
34 For (auto &kp:kp1) {
35 Auto kp_top = kp;
36 Kp_top.pt *= scales[pyramids - 1];
37 Kp1_pyr.push_back(kp_top);
38 Kp2_pyr.push_back(kp_top);
39 }
40 For (int level = pyramids - 1; level >= 0; level--) {
41 // from coarse to fine
42 Success.clear();
43 OpticalFlowSingleLevel(pyr1[level], pyr2[level], kp1_pyr, kp2_pyr, success,
44 inverse, true);
45 If (level > 0) {
46 For (auto &kp: kp1_pyr)
47 Kp.pt /= pyramid_scale;
48 For (auto &kp: kp2_pyr)
49 Kp.pt /= pyramid_scale;
50 }
51 }
52 For (auto &kp: kp2_pyr)
53 Kp2.push_back(kp);
54 }
55

```

This code constructs a four-layer pyramid with a magnification of 0.5 and calls a single-layer optical flow function to implement a multi-layer optical flow. In the main function, we tested the performance of OpenCV optical flow, single-layer optical flow and multi-layer optical flow for two images, and calculated their running time:

Listing 8.4: terminal input:

```

1 ./build/optical_flow
2 Build pyramid time: 0.000150349
3 Track pyr 3 cost time: 0.000304633
4 Track pyr 2 cost time: 0.000392889
5 Track pyr 1 cost time: 0.000382347
6 Track pyr 0 cost time: 0.000375099
7 Optical flow by gauss-newton: 0.00189268
8 Optical flow by opencv: 0.00220134

```

In terms of runtime, the time-consuming of the multi-layer optical flow method is roughly equivalent to that of OpenCV. Since the parallelizers behave differently on each run, these numbers are not exactly the same on the reader's machine. See the ?? for a comparison of the optical flows. From the results graph, the multi-layer optical flow is equivalent to the effect of OpenCV, and the single-layer optical flow is significantly weaker than the multi-layer optical flow.

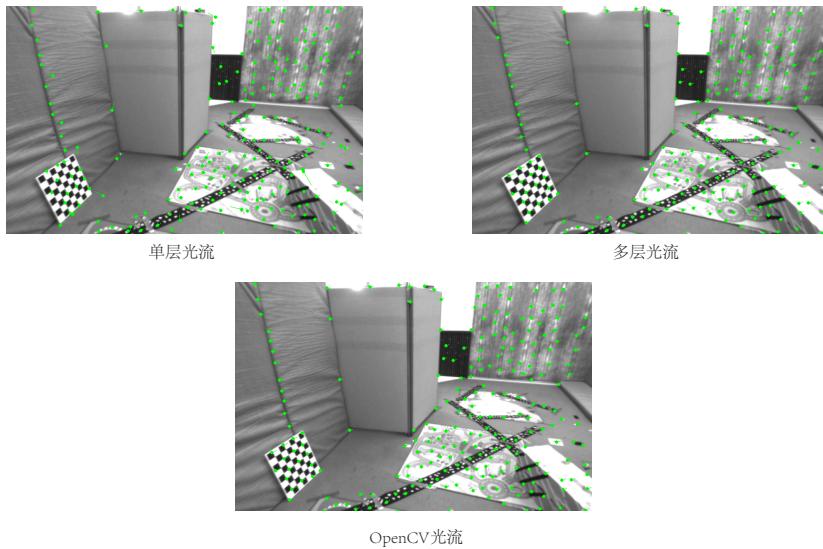


Figure 8-3: Comparison of results of various optical flows

8.3.3 Optical flow practice summary

We can see that LK optical flow tracking can directly obtain the correspondence of feature points. This correspondence is like a match of descriptors, but the optical flow requires more continuity of images and illumination stability. We can estimate camera motion using PnP, ICP, or counter-geometry by feature points of optical flow tracking. These methods were introduced in the previous lecture and are not discussed here.

In terms of runtime, the demo experiment is about 230 feature points. OpenCV and multi-layer optical flow take about 2 milliseconds to complete tracking (the CPU I use is Intel I7-8550U), which is actually quite fast. If we use a key point like FAST before, then the entire optical flow calculation can be done for about 5 milliseconds, which is very fast compared to feature matching. However, if the position of the corner is not good, the optical flow is easy to lose or give a wrong result. This requires the subsequent algorithm to have a certain abnormal value removal mechanism. We will leave it to the engineering chapter.

In summary, the optical flow method can accelerate the visual mileage calculation based on feature points, avoiding the process of calculating and matching the descriptors, but requires the camera to be smoother (or higher acquisition frequency).

8.4 Direct Method

Next, let's discuss the direct method that has some similarity to the optical flow. Similar to the previous content, we first introduce the principle of the direct method, and then use the direct method to achieve one pass.

8.4.1 Derivation of direct method

In the optical flow, we first track the position of the feature points and then determine the motion of the camera based on these positions. Then, such a two-step approach, it is difficult to guarantee global optimality. We can ask, can you adjust the results of the previous step in the next step? For example, if I think that the camera turns right 15 degrees, can the optical flow be based on the assumption that the 15 degree motion is the initial value, and adjust the calculation result of the optical flow? The direct method is the result of following this idea.

As shown in ?? , consider a space point P and a camera at two moments. The world coordinate of P is $[X, Y, Z]$, which is imaged on two cameras with pixel coordinates of $\mathbf{p}_1, \mathbf{p}_2$.

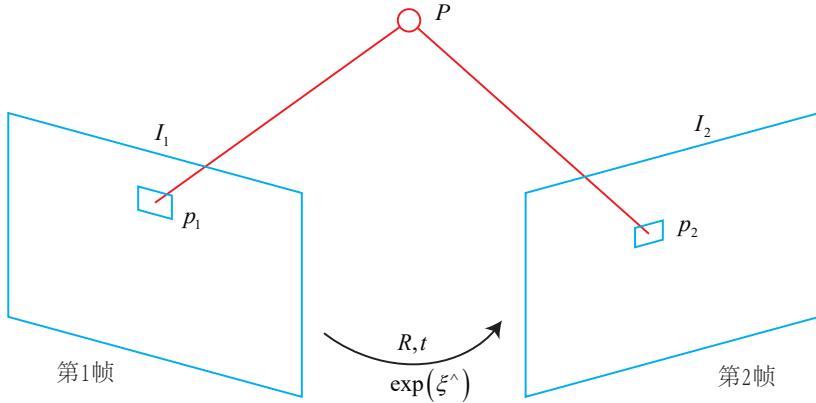


Figure 8-4: Direct method diagram.

Our goal is to find the relative pose change of the first camera to the second camera. We use the first camera as the frame of reference, and set the rotation and translation of the second camera to \mathbf{R}, \mathbf{t} (corresponding to Lie group as \mathbf{T}). At the same time, the internal parameters of the two cameras are the same, recorded as \mathbf{K} . For the sake of clarity, we write the complete projection equation:

$$\begin{aligned}\mathbf{p}_1 &= \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}_1 = \frac{1}{Z_1} \mathbf{K} \mathbf{P}, \\ \mathbf{p}_2 &= \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}_2 = \frac{1}{Z_2} \mathbf{K} (\mathbf{R} \mathbf{P} + \mathbf{t}) = \frac{1}{Z_2} \mathbf{K} (\mathbf{T} \mathbf{P})_{1:3}.\end{aligned}$$

Where Z_1 is the depth of P , Z_2 is the depth of P in the second camera coordinate system, which is the third coordinate value of $\mathbf{R} \mathbf{P} + \mathbf{t}$. Since \mathbf{T} can only be multiplied

by the homogeneous coordinates, we need to take the first 3 elements after we have finished. This is consistent with the content of ??.

In the recall feature point method, since we know the pixel position of $\mathbf{p}_1, \mathbf{p}_2$ by matching the descriptor, we can calculate the position of the reprojection. But in the direct method, since there is no feature matching, we have no way of knowing which \mathbf{p}_2 corresponds to the same point as \mathbf{p}_1 . The idea of the direct method is to find the location of \mathbf{p}_2 based on the current camera pose estimate. But if the camera is not well positioned, the appearance of \mathbf{p}_2 will be significantly different from \mathbf{p}_1 . So, to reduce this difference, we optimized the camera's pose to look for \mathbf{p}_2 more similar to \mathbf{p}_1 . This can also be done by solving an optimization problem, but at this point the minimum is not the reprojection error, but the **Photometric Error**, which is the brightness error of two pixels of P :

$$e = \mathbf{I}_1(\mathbf{p}_1) - \mathbf{I}_2(\mathbf{p}_2). \quad (8.11)$$

Note that e is a scalar here. Similarly, the optimization goal is the two norm of the error, temporarily taking the form of unweighted, as:

$$\min_{\mathbf{T}} J(\mathbf{T}) = \|e\|^2. \quad (8.12)$$

The reason for this optimization is still based on the **gray invariant assumption**. We assume that the gradation of a spatial point imaged at various angles is constant. We have a lot of (such as N) space points P_i , then the whole camera pose estimation problem becomes

$$\min_{\mathbf{T}} J(\mathbf{T}) = \sum_{i=1}^N e_i^T e_i, \quad e_i = \mathbf{I}_1(\mathbf{p}_{1,i}) - \mathbf{I}_2(\mathbf{p}_{2,i}). \quad (8.13)$$

Note that the optimization variable here is the camera pose \mathbf{T} , instead of optimizing the motion of each feature point like the optical flow. To solve this optimization problem, we are concerned with how the error e changes with the camera pose \mathbf{T} , and we need to analyze their derivative relationships. Therefore, define two intermediate variables:

$$\begin{aligned} \mathbf{q} &= \mathbf{T}\mathbf{P}, \\ \mathbf{u} &= \frac{1}{Z_2} \mathbf{K}\mathbf{q}. \end{aligned}$$

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} , \mathbf{u} is a function of \mathbf{q} , which is also a function of \mathbf{T} . Consider the left perturbation model of Lie algebra, using a first-order Taylor expansion, because:

$$e(\mathbf{T}) = \mathbf{I}_1(\mathbf{p}_1) - \mathbf{I}_2(\mathbf{u}), \quad (8.14)$$

and so:

$$\frac{\partial e}{\partial \mathbf{T}} = \frac{\partial \mathbf{I}_2}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \delta \xi} \delta \xi, \quad (8.15)$$

Where $\delta \xi$ is the left perturbation of \mathbf{T} . We see that the first derivative is divided into three terms by the chain rule, and these three items are easy to calculate:

1. $\partial \mathbf{I}_2 / \partial \mathbf{u}$ is the pixel gradient at \mathbf{u} .

2. $\partial \mathbf{u} / \partial \mathbf{q}$ is the derivative of the projection equation for the 3D point in the camera coordinate system. Remember $\mathbf{q} = [X, Y, Z]^T$, according to the deduction of ??, the derivative is

$$\frac{\partial \mathbf{u}}{\partial \mathbf{q}} = \begin{bmatrix} \frac{\partial u}{\partial X} & \frac{\partial u}{\partial Y} & \frac{\partial u}{\partial Z} \\ \frac{\partial v}{\partial X} & \frac{\partial v}{\partial Y} & \frac{\partial v}{\partial Z} \end{bmatrix} = \begin{bmatrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} \end{bmatrix}. \quad (8.16)$$

3. $\partial \mathbf{q} / \partial \delta \xi$ is the derivative of the transformed 3D point-to-point transformation, which was introduced in Lie algebra:

$$\frac{\partial \mathbf{q}}{\partial \delta \xi} = [\mathbf{I}, -\mathbf{q}^\wedge]. \quad (8.17)$$

In practice, since the last two items are only related to the 3D point \mathbf{q} and are not related to the image, we often combine them together:

$$\frac{\partial \mathbf{u}}{\partial \delta \xi} = \begin{bmatrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} & -\frac{f_x XY}{Z^2} & f_x + \frac{f_x X^2}{Z^2} & -\frac{f_x Y}{Z} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} & -f_y - \frac{f_y Y^2}{Z^2} & \frac{f_y XY}{Z^2} & \frac{f_y X}{Z} \end{bmatrix}. \quad (8.18)$$

2×6

$$\mathbf{J} = -\frac{\partial \mathbf{I}_2}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \delta \xi}. \quad (8.19)$$

For the problem of N points, we can use this method to calculate the Jacobian matrix of the optimization problem, and then use the Gauss-Newton method or the Levinberg-Marquart method to calculate the increment and iterative solution. So far, we have derived the direct process of estimating the pose of the camera. Here is a program to demonstrate how the direct method is used.

8.4.2 direct law discussion

In the above derivation, P is a spatial point of a known location. How did it come about? Under the RGB-D camera, we can backproject any pixel back into 3D space and then project it into the next image. If it is binocular, the depth of the pixel can also be calculated from the parallax. If you are in a monocular camera, this is even more difficult because we have to consider the uncertainty caused by the depth of P . Detailed depth estimates are discussed in Lecture 13. Now let's consider the simple case where the P depth is known.

Based on the source of P , we can classify the direct method:

1. P comes from sparse key points, which we call sparse direct methods. Usually we use hundreds to thousands of key points, and like the L-K optical flow, assume that the pixels around it are also invariant. This sparse direct method does not have to calculate descriptors and uses only a few hundred pixels, so it is the fastest, but only sparse reconstruction can be computed.
2. P comes from a partial pixel. We see that in the ??, if the pixel gradient is zero, the entire Jacobian matrix is zero and does not contribute to the calculated motion increment. Therefore, consider using only pixels with gradients and discarding where the pixel gradient is not obvious. This is called the semi-dense direct method and can reconstruct a semi-dense structure.

3. P is all pixels, called the dense direct method. Dense reconciliation requires the calculation of all pixels (typically hundreds of thousands to millions), so most of them cannot be calculated in real time on existing CPUs and require GPU acceleration. However, as discussed earlier, points where the pixel gradient is not significant will not contribute much to motion estimation, and it will be difficult to estimate the position during reconstruction.

It can be seen that from sparse to dense reconstruction, it can be calculated by the direct method. Their calculations are gradually increasing. The sparse method can quickly solve the camera pose, while the dense method can create a complete map. Which method is used depends on the application environment of the robot. In particular, on the low-end computing platform, the sparse direct method can achieve very fast results, suitable for occasions with high real-time and limited computing resources^[?].

8.5 Practice: Direct Law

8.5.1 single layer direct method

Now let's demonstrate how to use the sparse direct method. Since the book does not involve GPU programming, the dense direct method is omitted. At the same time, in order to keep the program simple, we use data with depth instead of unary data, which can omit the single-purpose deep recovery part. Depth recovery based on feature points (ie, triangulation) has been introduced in the previous lecture, and depth recovery based on block matching will be described later. So in this section we consider the sparse direct method of dual purpose.

Since solving the direct method is equivalent to solving an optimization problem, you can use the optimization library g2o or Ceres to help solve the problem, or you can implement the Gauss-Newton method yourself. Similar to the optical flow, the direct method can also be divided into a single-layer direct method and a pyramid-based multi-layer direct method. We also first implement the single-layer direct method, and then expand 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 we define a class that is Jacobian:

Listing 8.5: slambook2/ch8/direct_method.cpp

```

1 // class for accumulator jacobians in parallel
2 class JacobianAccumulator {
3 public:
4     JacobianAccumulator(
5         const cv::Mat &img1_,
6         const cv::Mat &img2_,
7         const VecVector2d &px_ref_,
8         const vector<double> depth_ref_,
9         Sophus::SE3d &T21_) :
10     img1(img1_), img2(img2_), px_ref(px_ref_), depth_ref(depth_ref_), T21(
11         T21_) {
12         projection = VecVector2d(px_ref.size(), Eigen::Vector2d(0, 0));
13     }
14
15     /// accumulate jacobians in a range
16     void accumulate_jacobian(const cv::Range &range);

```

```

16
17     /// get hessian matrix
18     Matrix6d hessian() const { return H; }
19
20     /// get bias
21     Vector6d bias() const { return b; }
22
23     /// get total cost
24     double cost_func() const { return cost; }
25
26     /// get projected points
27     VecVector2d projected_points() const { return projection; }
28
29     /// reset h, b, cost to zero
30     void reset() {
31         H = Matrix6d::Zero();
32         b = Vector6d::Zero();
33         cost = 0;
34     }
35
36 private:
37     const cv::Mat &img1;
38     const cv::Mat &img2;
39     const VecVector2d &px_ref;
40     const vector<double> depth_ref;
41     Sophus::SE3d &T21;
42     VecVector2d projection; // projected points
43
44     std::mutex hessian_mutex;
45     Matrix6d H = Matrix6d::Zero();
46     Vector6d b = Vector6d::Zero();
47     double cost = 0;
48 };
49
50 void JacobianAccumulator::accumulate_jacobian(const cv::Range &range) {
51
52     // parameters
53     const int half_patch_size = 1;
54     int cnt_good = 0;
55     Matrix6d hessian = Matrix6d::Zero();
56     Vector6d bias = Vector6d::Zero();
57     double cost_tmp = 0;
58
59     for (size_t i = range.start; i < range.end; i++) {
60         // compute the projection in the second image
61         Eigen::Vector3d point_ref =
62             depth_ref[i] * Eigen::Vector3d((px_ref[i][0] - cx) / fx, (px_ref[i][1]
63             - cy) / fy, 1);
64         Eigen::Vector3d point_cur = T21 * point_ref;
65         if (point_cur[2] < 0) // depth invalid
66             continue;
67
68         float u = fx * point_cur[0] / point_cur[2] + cx, v = fy * point_cur[1]
69             / point_cur[2] + cy;
70         if (u < half_patch_size || u > img2.cols - half_patch_size || v <
71             half_patch_size ||
72             v > img2.rows - half_patch_size)
73             continue;
74
75         projection[i] = Eigen::Vector2d(u, v);
76         double X = point_cur[0], Y = point_cur[1], Z = point_cur[2],
77         Z2 = Z * Z, Z_inv = 1.0 / Z, Z2_inv = Z_inv * Z_inv;
78         cnt_good++;
79     }
80 }
```

```

76 // and compute error and jacobian
77 for (int x = -half_patch_size; x <= half_patch_size; x++) {
78   for (int y = -half_patch_size; y <= half_patch_size; y++) {
79     double error = GetPixelValue(img1, px_ref[i][0] + x, px_ref[i][1] + y
80       ) -
81       GetPixelValue(img2, u + x, v + y);
82     Matrix26d J_pixel_xi;
83     Eigen::Vector2d J_img_pixel;
84
85     J_pixel_xi(0, 0) = fx * Z_inv;
86     J_pixel_xi(0, 1) = 0;
87     J_pixel_xi(0, 2) = -fx * X * Z2_inv;
88     J_pixel_xi(0, 3) = -fx * X * Y * Z2_inv;
89     J_pixel_xi(0, 4) = fx + fx * X * X * Z2_inv;
90     J_pixel_xi(0, 5) = -fx * Y * Z_inv;
91
92     J_pixel_xi(1, 0) = 0;
93     J_pixel_xi(1, 1) = fy * Z_inv;
94     J_pixel_xi(1, 2) = -fy * Y * Z2_inv;
95     J_pixel_xi(1, 3) = -fy - fy * Y * Y * Z2_inv;
96     J_pixel_xi(1, 4) = fy * X * Y * Z2_inv;
97     J_pixel_xi(1, 5) = fy * X * Z_inv;
98
99     J_img_pixel = Eigen::Vector2d(
100       0.5 * (GetPixelValue(img2, u + 1 + x, v + y) - GetPixelValue(img2,
101         u - 1 + x, v + y)),
102       0.5 * (GetPixelValue(img2, u + x, v + 1 + y) - GetPixelValue(img2,
103         u + x, v - 1 + y))
104     );
105
106     // total jacobian
107     Vector6d J = -1.0 * (J_img_pixel.transpose() * J_pixel_xi).transpose
108     ();
109     hessian += J * J.transpose();
110     bias += -error * J;
111     cost_tmp += error * error;
112   }
113
114   if (cnt_good) {
115     // set hessian, bias and cost
116     unique_lock<mutex> lck(hessian_mutex);
117     H += hessian;
118     b += bias;
119     cost += cost_tmp / cnt_good;
120   }
121 }
```

In the accumulate_jacobian function of this class, we calculate the pixel error and the Jacobian matrix for the pixels in the specified range according to the previous derivation, and finally add it to the overall \mathbf{H} matrix. Then, define a function to iterate through the process:

Listing 8.6: slambook2/ch8/direct_method.cpp

```

1 void DirectPoseEstimationSingleLayer(
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   const int iterations = 10;
```

```

8   double cost = 0, lastCost = 0;
9   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
15          ::placeholders::_1));
16      Matrix6d H = jaco_accu.hessian();
17      Vector6d b = jaco_accu.bias();
18
19      // solve update and put it into estimation
20      Vector6d update = H.ldlt().solve(b);
21      T21 = Sophus::SE3d::exp(update) * T21;
22      cost = jaco_accu.cost_func();
23
24      if (std::isnan(update[0])) {
25          // sometimes occurred when we have a black or white patch and H is
26          // irreversible
27          cout << "update is nan" << endl;
28          break;
29      }
30      if (iter > 0 && cost > lastCost) {
31          cout << "cost increased: " << cost << ", " << lastCost << endl;
32          break;
33      }
34      if (update.norm() < 1e-3) {
35          // converge
36          break;
37      }
38
39      lastCost = cost;
40      cout << "iteration: " << iter << ", cost: " << cost << endl;
}

```

The function finds the corresponding pose update based on the calculated \mathbf{H} and \mathbf{b} , and then updates to the current estimate. Because we have already introduced the details in the theoretical part, this part of the code does not seem to be particularly difficult.

8.5.2 Multilayer Direct Method

Then, similar to the optical flow, we extend the direct method to the pyramid and use the Coarse-to-fine process to calculate the relative motion. This part of the code and the optical flow are also very similar:

Listing 8.7: slambook2/ch8/direct_method.cpp

```

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 *
22                                   pyramid_scale));
23             cv::resize(pyr2[i - 1], img2_pyr,
24                         cv::Size(pyr2[i - 1].cols * pyramid_scale, pyr2[i - 1].rows *
25                                   pyramid_scale));
26             pyr1.push_back(img1_pyr);
27             pyr2.push_back(img2_pyr);
28         }
29     }
30
31     double fxG = fx, fyG = fy, cxG = cx, cyG = cy; // backup the old values
32     for (int level = pyramids - 1; level >= 0; level--) {
33         VecVector2d px_ref_pyr; // set the keypoints in this pyramid level
34         for (auto &px: px_ref) {
35             px_ref_pyr.push_back(scales[level] * px);
36
37         // scale fx, fy, cx, cy in different pyramid levels
38         fx = fxG * scales[level];
39         fy = fyG * scales[level];
40         cx = cxG * scales[level];
41         cy = cyG * scales[level];
42         DirectPoseEstimationSingleLayer(pyr1[level], pyr2[level], px_ref_pyr,
43                                         depth_ref, T21);
44     }
45 }
```

It should be noted that because the direct method is used to get the internal parameters of the camera, when the pyramid scales the image, the corresponding internal parameters also need to be multiplied by the corresponding magnification.

8.5.3 Result Discussion

Finally, we use some sample images to test the results of the direct method. We use several images of the Kitti[?] autopilot dataset. First, we read the first image left.png, calculate the depth corresponding to each pixel in the corresponding disparity map disparity.png, and then calculate the camera using the direct method for the five images of 000001.png-000005.png. Pose. In order to demonstrate the insensitivity of the direct method to feature points, we randomly select some points in the first image, and do not use any corner points or feature point extraction algorithms to see its results.

Listing 8.8: slambook2/ch8/direct_method.cpp

```

1 int main(int argc, char **argv) {
2
3     cv::Mat left_img = cv::imread(left_file, 0);
4     cv::Mat disparity_img = cv::imread(disparity_file, 0);
5
6     // let's randomly pick pixels in the first image and generate some 3d
7     // points in the first image's frame
8     cv::RNG rng;
```

```

8   int nPoints = 2000;
9   int boarder = 20;
10  VecVector2d pixels_ref;
11  vector<double> depth_ref;
12
13 // generate pixels in ref and load depth data
14 for (int i = 0; i < nPoints; i++) {
15     int x = rng.uniform(boarder, left_img.cols - boarder); // don't pick
16         pixels close to boarder
17     int y = rng.uniform(boarder, left_img.rows - boarder); // don't pick
18         pixels close to boarder
19     int disparity = disparity_img.at<uchar>(y, x);
20     double depth = fx * baseline / disparity; // you know this is disparity
21         to depth
22     depth_ref.push_back(depth);
23     pixels_ref.push_back(Eigen::Vector2d(x, y));
24 }
25
26 // estimates 01~05.png's pose using this information
27 Sophus::SE3d T_cur_ref;
28
29 for (int i = 1; i < 6; i++) { // 1~10
30     cv::Mat img = cv::imread((fmt_others % i).str(), 0);
31     DirectPoseEstimationMultiLayer(left_img, img, pixels_ref, depth_ref,
32                                     T_cur_ref);
33 }
34 return 0;
35 }
```

The reader can try to run this program on your machine, it will output the tracking points on each layer of the pyramid of each image, and output the running time. The result of the multi-layer direct method is shown in ???. According to the program output, you can see that the fifth image is about 3.8 meters when the camera moves 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 motion of the camera. There is no process of feature extraction, matching or optical flow in between. From the perspective of running time, at 2000 points, the direct method takes 1-2 milliseconds per iteration, so the four-layer gold layer takes about 8 milliseconds. In contrast, the optical flow of 2000 points is about ten milliseconds, and does not include subsequent pose estimation. Therefore, the direct method is usually faster than the traditional feature points and optical flow.

Below we briefly explain the iterative process of the direct method. Compared to the feature point method, the direct method relies entirely on optimization to solve the camera pose. As can be seen from the formula (??), the pixel gradient guides the direction of optimization. If you want to get the right optimization results, you must ensure that **most pixel gradients can direct optimization to the right direction**.

What does it mean? Let's take a look at the optimization algorithm. 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 (?? the gray level measured in I_1).

At this point we get a new image and we need to estimate its camera pose. This pose is obtained by continuously optimizing the iteration of an initial value. Suppose our initial value is relatively poor. Under this initial value, the pixel gray value after the spatial point P projection is 126. Thus, the error of this pixel is $229 - 126 = 103$. In order to reduce this error, we want **fine-tune the camera's pose to make the**



Figure 8-5: The experimental result of the direct method. Top left: original image; top right: disparity map corresponding to the original image; bottom left: fifth tracking image; bottom right: tracking result

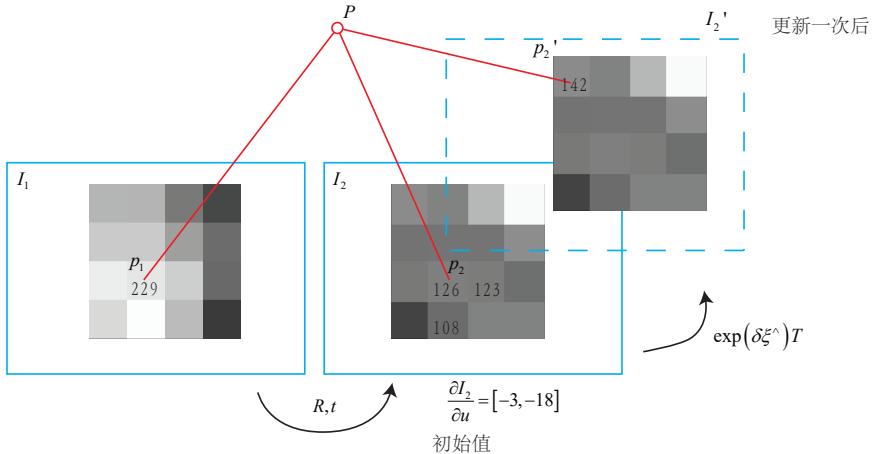


Figure 8-6: A graphical display of one iteration.

pixels brighter.

How do you know where to fine tune the pixels to be brighter? This requires a partial pixel gradient. We found in the image that we took a step forward along the u axis, where the gray value became 123, which is minus 3. Similarly, a step forward along the v axis reduces the gray value by 18 to 108. Around this pixel, we see that the gradient is $[-3, -18]$. To improve the brightness, we recommend an optimization algorithm to fine tune the camera and move the P image toward **top left**. In this process, we approximate the grayscale distribution near it with the local gradient of the pixel, but note that the real image is not smooth, so this gradient is not true in the distance.

However, the optimization algorithm can't just listen to the side of this pixel, but also need to listen to other pixels' suggestions*. After listening to the opinions of many pixels, the optimization algorithm chooses a place that is not far from the direction we suggest, and calculates an update amount $\exp(\xi^\wedge)$. With the update, the image moved from I_2 to I'_2 , and the pixel's projected position also changed to a brighter place. We see that with this update, **error has become smaller**. In the ideal case, we expect the error to continue to drop and finally converge.

But is this actually the case? Are we really going to go along the gradient direction and get to an optimal value? Note that the gradient of the direct method is determined directly by the image gradient, so we must ensure that the **following the image gradient, the grayscale error will continue to drop**. However, the image is usually a very strong **non-convex function**, as shown by ?? . In practice, if we proceed along the image gradient, it is easy to continue optimization because the non-convexity (or noise) of the image itself falls into a local minimum. The direct method can only be established when the camera motion is small and the gradient in the image does not have a strong non-convexity.

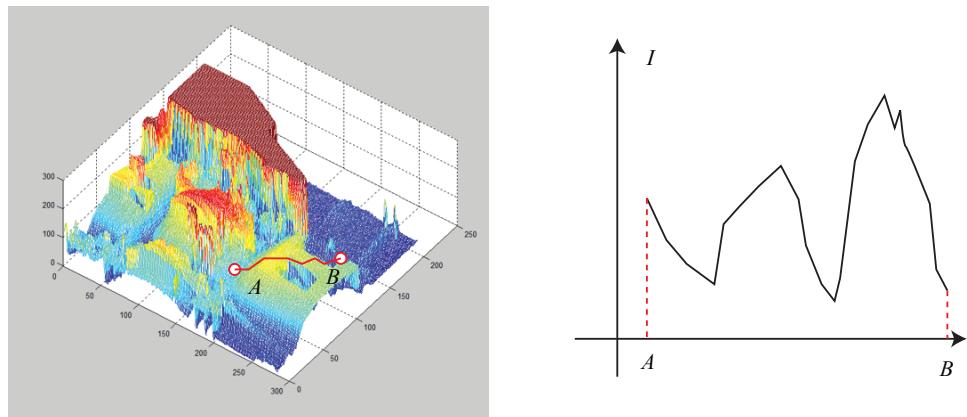


Figure 8-7: 3D display of an image. The path from one point in the image to another is not necessarily a "straight downhill", but it is often necessary to "over the mountains." This embodies the non-convexity of the image itself.

In the routine, we only calculated the difference of a single pixel, and this difference is directly subtracted from the grayscale. However, a single pixel is not discriminating, and there are likely to be many pixels around it and its brightness is similar. Therefore, we sometimes use small image patches and use more com-

* This may be an imprecise anthropomorphic statement, but it helps to understand.

plex measure of difference, such as Normalized Cross Correlation (NCC). For the sake of simplicity, the routine uses the sum of the squares of the errors to maintain consistency with the derivation.

8.5.4 Direct method advantages and disadvantages summary

Finally, we summarize the advantages and disadvantages of the direct method. In general, its advantages are as follows:

- can save the time to calculate feature points and descriptors.
- only requires a pixel gradient, no feature points are required. Therefore, the direct method can be used in the case where the feature is missing. An extreme example is an image with only a gradient. It may not be able to extract the corner feature, but it can be estimated directly using the direct method. In the demonstration experiment, we saw that the direct method works well for randomly selected points. This is critical in practice, as it is likely that there are not many corner points available for use in a practical scenario.
- can build semi-dense or even dense maps, which is not possible with feature points.

On the other hand, its shortcomings are also obvious:

- **non-convex.** The direct method relies entirely on gradient search to reduce 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 enter very small, and the direct method can only succeed when the movement is very small. In response to this, the introduction of the pyramid can reduce the influence of non-convexity to some extent.
- **single pixel has no discrimination.** It's too much like it! So we either calculate the image block or calculate the complex correlation. Since each pixel is inconsistent with the "opinion" of changing camera motion, only a few can obey the majority, replacing the quality by quantity. Therefore, the direct method has a significant decline in performance when the number of points is small. We usually recommend more than 500 points.
- **The gray value is constant is a strong assumption.** If the camera is auto-exposure, it will make the image as bright or dark as it adjusts the exposure parameters. This can also happen when the light changes. The feature point method has a certain tolerance to illumination, while the direct method calculates the gray level difference, and the overall gray level change will destroy the gray level invariant hypothesis, which makes the algorithm fail. For this, a practical direct method will simultaneously estimate the camera's exposure parameter [?] to work even when the exposure time changes.

Exercises

1. In addition to LK optical flow, what other optical flow methods? What are their characteristics?

2. In the image gradient process of this section of the program, we simply find the difference between the grayscales of $u + 1$ and $u - 1$ divided by 2 as the gradient value in the u direction. What are the disadvantages of this approach? Hint: For features with close distances, the change should be faster; while the farther features change slowly in the image. Can you use this information when seeking gradients? Can the
3. direct method be the same as the optical flow, and propose the concept of "reverse method"? That is, use the gradient of the original image instead of the gradient of the target image?
- 4.*Uses Ceres or g2o to implement sparse direct and semi-dense direct methods.
5. Compared to the direct method of RGB-D, the monocular direct method is often more complicated. In addition to the unknown match, the distance of the pixel is also to be estimated, we need to use the pixel depth as an optimization variable in the optimization. Read the article [? ?], can you understand its principles?

Chapter 9

backend 1

main target

1. understands the concept of the backend.
2. understands the working principle of the filter backend represented by EKF.
3. Understand the back end of nonlinear optimization and understand how sparsity is utilized.
4. uses g2o and Ceres to actually manipulate backend optimization.

At the beginning of this lecture, we transferred to another important module of the SLAM system: back-end optimization.

We see that the front-end visual odometer can give a short time trajectory and map, but due to the inevitable accumulation of errors, this map is inaccurate for a long time. Therefore, based on the visual odometer, we also hope to construct a scale and scale optimization problem to consider the optimal trajectory and map over a long period of time. However, considering the balance between accuracy and performance, there are many different practices in practice.



$$z_{11} = h(x_1, y_1)$$



$$J = \begin{bmatrix} \square & \square \\ \square & \square \end{bmatrix}$$

$$z_{21} = h(x_2, y_1)$$

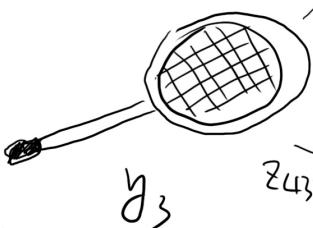


$$H = \begin{bmatrix} \square & \square \\ \square & \square \end{bmatrix}$$

$$z_{22} = h(x_2, y_2)$$



$$z_{23} = h(x_2, y_3)$$



$$z_{43} = h(x_4, y_3)$$

$$z_{33} = h(x_3, y_3)$$



$$z_{34} = h(x_3, y_4)$$



y_4

9.1 overview

9.1.1 Probability interpretation of state estimation

As mentioned in the second lecture, the visual odometer has only a short memory, and we hope that the entire motion trajectory will remain optimal for a long time. We may use the latest knowledge to update the more distant state - from the perspective of "a long-term state", as if the future information tells it "Where should you be?" So, in back-end optimization, we usually consider a state estimation problem for a longer period of time (or all time), and not only use past information to update its state, but also update itself with future information. The processing method may be called "batch". Otherwise, if the current state is determined only by past moments, or even by the previous moment, it may be called "incremental".

We already know that the SLAM process can be described by equations of motion and observation equations. So, suppose that during the time from $t = 0$ to $t = N$, there is a pose \mathbf{x}_0 to \mathbf{x}_N , and there is a road sign $\mathbf{y}_1, \dots, \mathbf{y}_M$. According to the previous writing, the motion and observation equations are

$$\begin{cases} \mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k & k = 1, \dots, N, \\ \mathbf{z}_{k,j} = h(\mathbf{y}_j, \mathbf{x}_k) + \mathbf{v}_{k,j} & j = 1, \dots, M. \end{cases} \quad (9.1)$$

Note the following:

1. In the observation equation, observation data is only generated when \mathbf{x}_k sees \mathbf{y}_j , otherwise it is not. In fact, only a small number of road signs can usually be seen in one location. Moreover, due to the large number of visual SLAM feature points, the number of observation equations in practice will be much larger than the motion equation.
2. We may not have a device to measure motion, so there may be no equations of motion. In this case, there are several ways to deal with it: think that there is really no equation of motion, or that the camera is not moving, or that the camera is moving at a constant speed. These methods are all feasible. In the absence of an equation of motion, the entire optimization problem consists of only a few observation equations. This is very similar to the SfM (Structure from Motion) problem, which is equivalent to restoring motion and structure through a set of images. Unlike SfM, images in SLAM have a temporal order, while SfM allows the use of completely unrelated images.

We know that each equation is affected by noise, so we should treat the pose \mathbf{x} and the signpost \mathbf{y} as **random variables with a certain probability distribution** instead of A single number. So, the question we care about becomes: How do I determine the state quantity \mathbf{x} , when I have some motion data \mathbf{u} and observation data \mathbf{z} ? Furthermore, if the data of the new moments are obtained, how will their distribution change again? In the more common and reasonable case, we assume that the state quantities and noise terms obey the Gaussian distribution - this means that only the mean and covariance matrices need to be stored in the program. The mean can be seen as an estimate of the optimal value of the variable, while the covariance matrix measures its uncertainty. Then, the question turns into: How do we estimate the Gaussian distribution of state quantities when there are some motion data and observation data?

We still play the role of a small radish. Only the equation of motion is equivalent to walking blindly in an unknown place. Although we know how far we go every step of the way, as time goes by, we will become more and more uncertain about our position – the more uneasy the heart. This shows that when the input data is affected by noise, **error is gradually accumulated**, we will estimate the position variance will be larger and larger. However, when we open our eyes, we are more and more confident because we can continuously observe external scenes and make the uncertainty of position estimation smaller. If you visually express the covariance matrix with an ellipse or ellipsoid, then the process is a bit like the feeling of walking in mobile map software. Taking ?? as an example, the reader can imagine that when there is no observation data, the circle will grow larger and larger; if there is correct observation, the circle will shrink to a certain size and keep stable.

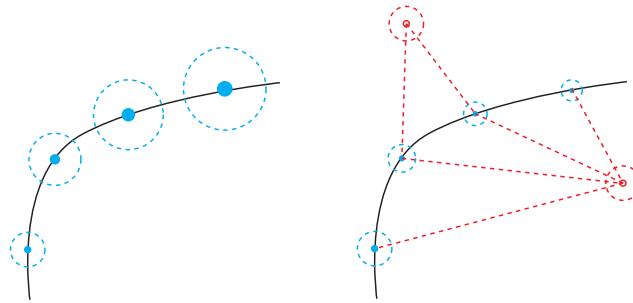


Figure 9-1: Visual description of uncertainty. Left side: When there is only the motion equation, since the pose at the next moment adds noise on the basis of the previous moment, the uncertainty becomes larger and larger. Right side: When there are landmark points, the uncertainty will be significantly reduced. However, please note that this is just an intuitive diagram, not actual data.

The above process explains the problem in state estimation in the form of a metaphor. Let us look at it quantitatively. In the ?? lecture, we introduce the maximum likelihood estimation, mentioning that the **batch state estimation problem can be transformed into the maximum likelihood estimation problem and solved by the least squares method**. In this section, we will explore how to apply this conclusion to progressive problems and get some classic conclusions. At the same time, in the visual SLAM, the least squares method has a special structure.

First, since both the pose and the landmark are variables to be estimated, we change the token so that \mathbf{x}_k is all unknowns at time k . It contains the current camera pose and m landmark points. In the sense of this mark (although slightly different from the previous one, but the meaning is clear), write:

$$\mathbf{x}_k \triangleq \{\mathbf{x}_k, \mathbf{y}_1, \dots, \mathbf{y}_m\}. \quad (9.2)$$

At the same time, all observations at time k are recorded as \mathbf{z}_k . Thus, the form of the equation of motion and the equation of observation can be written more concisely. \mathbf{y} won't appear here, but we have to understand that \mathbf{y} is already included in \mathbf{x} :

$$\begin{cases} \mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k & k = 1, \dots, N. \\ \mathbf{z}_k = h(\mathbf{x}_k) + \mathbf{v}_k \end{cases} \quad (9.3)$$

Now consider the situation at the time of the k . We want to use the data from

the past 0 to k to estimate the current state distribution:

$$P(\mathbf{x}_k | \mathbf{x}_0, \mathbf{u}_{1:k}, \mathbf{z}_{1:k}). \quad (9.4)$$

The subscript $0 : k$ represents all data from time 0 to time k . Note that \mathbf{z}_k means all observations at k , it may be more than one, but this is more convenient. At the same time, \mathbf{x}_k is actually related to $\mathbf{x}_{k-1}, \mathbf{x}_{k-2}$, but this is not explicitly Write them out.

Let's look at how to estimate the state. According to Bayes' rule, swap \mathbf{z}_k with \mathbf{x}_k , with:

$$P(\mathbf{x}_k | \mathbf{x}_0, \mathbf{u}_{1:k}, \mathbf{z}_{1:k}) \propto P(\mathbf{z}_k | \mathbf{x}_k) P(\mathbf{x}_k | \mathbf{x}_0, \mathbf{u}_{1:k}, \mathbf{z}_{1:k-1}). \quad (9.5)$$

Readers should not be unfamiliar. The first item here is called **likelihood** and the second item is called **priority**. The likelihood is given by the observation equation, and in the a priori part, we have to understand that the current state \mathbf{x}_k is estimated based on all past states. At the very least, it will be affected by \mathbf{x}_{k-1} , so the conditional probability is expanded according to \mathbf{x}_{k-1} :

$$P(\mathbf{x}_k | \mathbf{x}_0, \mathbf{u}_{1:k}, \mathbf{z}_{1:k-1}) = \int P(\mathbf{x}_k | \mathbf{x}_{k-1}, \mathbf{x}_0, \mathbf{u}_{1:k}, \mathbf{z}_{1:k-1}) P(\mathbf{x}_{k-1} | \mathbf{x}_0, \mathbf{u}_{1:k}, \mathbf{z}_{1:k-1}) d\mathbf{x}_{k-1}. \quad (9.6)$$

If you consider the state before the longer, you can continue to expand this style, but now we only care about the time of k and $k - 1$. So far, we have given Bayesian estimates, although the above formula does not have a specific form of probability distribution, so it is not practical to operate it. There are some differences in the method for the subsequent processing of this step. In general, there are several choices: one is the assumption of **Markovity**, and the simple first-order Markov property considers that the state of k is only related to the state of $k - 1$, and Nothing before. If we make this assumption, we will get a filter method represented by **Extended Kalman Filter** (EKF). In the filtering method, we will estimate the state from a certain moment and derive it to the next moment. Another way is to still consider the relationship between the state of k and the previous **all** state, and then get the **non-linear optimization** as the main optimization framework. The basics of nonlinear optimization have been introduced in the previous section. At present, the mainstream of visual SLAM is a nonlinear optimization method. However, in order to make this book more comprehensive, we must first introduce the principle of Kalman filter and EKF.

9.1.2 Linear System and KF

Let's first look at the filter model. When we assume Markovity, what changes will happen from a mathematical perspective? First, the current time state is only related to the previous time. The first part of the right side of the formula (??) can be further simplified:

$$P(\mathbf{x}_k | \mathbf{x}_{k-1}, \mathbf{x}_0, \mathbf{u}_{1:k}, \mathbf{z}_{1:k-1}) = P(\mathbf{x}_k | \mathbf{x}_{k-1}, \mathbf{u}_k). \quad (9.7)$$

Here, since the state of k is not related to before $k - 1$, it is reduced to a form related only to \mathbf{x}_{k-1} and \mathbf{u}_k . Corresponds to the equation of motion at the time of k . The second part can be simplified to

$$P(\mathbf{x}_{k-1} | \mathbf{x}_0, \mathbf{u}_{1:k}, \mathbf{z}_{1:k-1}) = P(\mathbf{x}_{k-1} | \mathbf{x}_0, \mathbf{u}_{1:k-1}, \mathbf{z}_{1:k-1}). \quad (9.8)$$

This is because the input bmu_k at the time of k is not related to the state of $k - 1$, so we take \mathbf{u}_k off. As you can see, this item is actually a state distribution at the time of $k - 1$. Thus, this series of equations shows that what we are actually doing is "how to derive the state distribution at the time of $k - 1$ to the time of k ". That is to say, during the running of the program, we only need to maintain a state quantity, and iterate and update it continuously. Further, if we assume that the state quantity obeys the Gaussian distribution, then we only need to consider the mean and covariance of the maintenance state quantity. You can imagine that the positioning system on the radish has been outputting two positioning information outwards: one is his own posture and the other is his own uncertainty. This is often the case in practice.

We start with the simplest linear Gaussian system, and finally we will get the Kalman filter. After clarifying the starting point and the end point, let's consider the middle route. Linear Gaussian systems say that equations of motion and observation equations can be described by linear equations:

$$\begin{cases} \mathbf{x}_k = \mathbf{A}_k \mathbf{x}_{k-1} + \mathbf{u}_k + \mathbf{w}_k & k = 1, \dots, N. \\ \mathbf{z}_k = \mathbf{C}_k \mathbf{x}_k + \mathbf{v}_k \end{cases} \quad (9.9)$$

It is also assumed that all states and noises satisfy the Gaussian distribution. Note that the noise here follows a zero-mean Gaussian distribution:

$$\mathbf{w}_k \sim N(\mathbf{0}, \mathbf{R}). \quad \mathbf{v}_k \sim N(\mathbf{0}, \mathbf{Q}). \quad (9.10)$$

For the sake of brevity, I omitted the subscripts of \mathbf{R} and \mathbf{Q} . Now, using Markov property, suppose we know the posterior of $k-1$ (in the case of $k-1$) state estimation $\hat{\mathbf{x}}_{k-1}$ and its covariance $\hat{\mathbf{P}}_{k-1}$, now the posterior distribution of \mathbf{x}_k is determined based on the input and observation data at time k . To distinguish between a priori and posterior in the derivation, we make a difference in the notation: the above hat $\hat{\mathbf{x}}_k$ indicates a posteriori, the following hat $\check{\mathbf{x}}_k$ indicates a prior distribution, please do not confuse the reader.

The first step of the Kalman filter is to determine the prior distribution of \mathbf{x}_k by the equation of motion. This step is linear, while the linear transformation of the Gaussian distribution is still a Gaussian distribution. So obviously there are:

$$P(\mathbf{x}_k | \mathbf{x}_0, \mathbf{u}_{1:k}, \mathbf{z}_{1:k-1}) = N\left(\mathbf{A}_k \hat{\mathbf{x}}_{k-1} + \mathbf{u}_k, \mathbf{A}_k \hat{\mathbf{P}}_{k-1} \mathbf{A}_k^T + \mathbf{R}\right). \quad (9.11)$$

This step is called **prediction**, and the principle is in the appendix ???. It shows how the state distribution of the current time is inferred from the state of the previous moment based on the input information (but with noise). This distribution is also a priori. Remember:

$$\check{\mathbf{x}}_k = \mathbf{A}_k \hat{\mathbf{x}}_{k-1} + \mathbf{u}_k, \quad \check{\mathbf{P}}_k = \mathbf{A}_k \hat{\mathbf{P}}_{k-1} \mathbf{A}_k^T + \mathbf{R}. \quad (9.12)$$

This is very natural. Obviously, the uncertainty of this step state will become larger because noise is added to the system. On the other hand, from the observation equation, we can calculate what kind of observation data should be produced under a certain state:

$$P(\mathbf{z}_k | \mathbf{x}_k) = N(\mathbf{C}_k \mathbf{x}_k, \mathbf{Q}). \quad (9.13)$$

In order to get the posterior probability, we want to calculate their product, which is the Bayesian formula given by the formula (??). However, although we

know that we will get a Gaussian distribution about \mathbf{x}_k in the end, it is a bit of a computational problem. Let's first set the result to $\mathbf{x}_k \sim N(Bm\hat{\mathbf{x}}_k, \hat{\mathbf{P}}_k)$, then:

$$N(\hat{\mathbf{x}}_k, \hat{\mathbf{P}}_k) = \eta N(\mathbf{C}_k \mathbf{x}_k, \mathbf{Q}) \cdot N(\check{\mathbf{x}}_k, \check{\mathbf{P}}_k). \quad (9.14)$$

Here we use a little bit of a clever way. Now that we know that the equations are Gaussian on both sides, we only need to compare the exponential parts without regard to the factor part before the Gaussian distribution. The index part is very similar to a quadratic formula, let's deduce it. First expand the exponent section, there is *:

$$(\mathbf{x}_k - \hat{\mathbf{x}}_k)^T \hat{\mathbf{P}}_k^{-1} (\mathbf{x}_k - \hat{\mathbf{x}}_k) = (\mathbf{z}_k - \mathbf{C}_k \mathbf{x}_k)^T \mathbf{Q}^{-1} (\mathbf{z}_k - \mathbf{C}_k \mathbf{x}_k) + (\mathbf{x}_k - \check{\mathbf{x}}_k)^T \check{\mathbf{P}}_k^{-1} (\mathbf{x}_k - \check{\mathbf{x}}_k). \quad (9.15)$$

To find $\hat{\mathbf{x}}_k$ and $\hat{\mathbf{P}}_k$ on the left, we expand the two sides and compare the second sum of \mathbf{x}_k Once coefficient. For quadratic coefficients, there are:

$$\hat{\mathbf{P}}_k^{-1} = \mathbf{C}_k^T \mathbf{Q}^{-1} \mathbf{C}_k + \check{\mathbf{P}}_k^{-1}. \quad (9.16)$$

This formula gives the calculation process of covariance. To facilitate the following formula, define an intermediate variable:

$$\mathbf{K} = \hat{\mathbf{P}}_k \mathbf{C}_k^T \mathbf{Q}^{-1}. \quad (9.17)$$

According to this definition, the $\hat{\mathbf{P}}_k$ is multiplied by the left and right of the formula (??), with:

$$\mathbf{I} = \hat{\mathbf{P}}_k \mathbf{C}_k^T \mathbf{Q}^{-1} \mathbf{C}_k + \hat{\mathbf{P}}_k \check{\mathbf{P}}_k^{-1} = \mathbf{K} \mathbf{C}_k + \hat{\mathbf{P}}_k \check{\mathbf{P}}_k^{-1}. \quad (9.18)$$

So there is *:

$$\hat{\mathbf{P}}_k = (\mathbf{I} - \mathbf{K} \mathbf{C}_k) \check{\mathbf{P}}_k. \quad (9.19)$$

Then compare the coefficients of the item once, with:

$$-2\hat{\mathbf{x}}_k^T \hat{\mathbf{P}}_k^{-1} \mathbf{x}_k = -2\mathbf{z}_k^T \mathbf{Q}^{-1} \mathbf{C}_k \mathbf{x}_k - 2\check{\mathbf{x}}_k^T \check{\mathbf{P}}_k^{-1} \mathbf{x}_k. \quad (9.20)$$

Finishing (take factor and transpose) to get:

$$\hat{\mathbf{P}}_k^{-1} \hat{\mathbf{x}}_k = \mathbf{C}_k^T \mathbf{Q}^{-1} \mathbf{z}_k + \check{\mathbf{P}}_k^{-1} \check{\mathbf{x}}_k. \quad (9.21)$$

Multiply both sides by $\hat{\mathbf{P}}_k$ and substitute (??) to get:

$$\hat{\mathbf{x}}_k = \hat{\mathbf{P}}_k \mathbf{C}_k^T \mathbf{Q}^{-1} \mathbf{z}_k + \hat{\mathbf{P}}_k \check{\mathbf{P}}_k^{-1} \check{\mathbf{x}}_k \quad (9.22)$$

$$= \mathbf{K} \mathbf{z}_k + (\mathbf{I} - \mathbf{K} \mathbf{C}_k) \check{\mathbf{x}}_k = \check{\mathbf{x}}_k + \mathbf{K} (\mathbf{z}_k - \mathbf{C}_k \check{\mathbf{x}}_k). \quad (9.23)$$

So we got the expression of the posterior mean. In summary, the above two steps can be summarized into two steps: "Predict" and "Update":

* The equal sign here is not strict, and the constants that are not related to \mathbf{x}_k are actually allowed.

* here seems to have a little loop definition. We defined \mathbf{K} by $\hat{\mathbf{P}}_k$ and wrote $\hat{\mathbf{P}}_k$ as an expression for \mathbf{K} . However, in practice \mathbf{K} can be calculated without relying on $\hat{\mathbf{P}}_k$, but this requires the introduction of Sherman-Morrison-Woodbury identity[?], see the exercises in this lecture.

1. forecast:

$$\check{\mathbf{x}}_k = \mathbf{A}_k \hat{\mathbf{x}}_{k-1} + \mathbf{u}_k, \quad \check{\mathbf{P}}_k = \mathbf{A}_k \hat{\mathbf{P}}_{k-1} \mathbf{A}_k^T + \mathbf{R}. \quad (9.24)$$

2. update: First calculate \mathbf{K} , which is also known as the Kalman gain.

$$\mathbf{K} = \check{\mathbf{P}}_k \mathbf{C}_k^T (\mathbf{C}_k \check{\mathbf{P}}_k \mathbf{C}_k^T + \mathbf{Q}_k)^{-1}. \quad (9.25)$$

Then calculate the distribution of posterior probabilities.

$$\begin{aligned} \hat{\mathbf{x}}_k &= \check{\mathbf{x}}_k + \mathbf{K} (\mathbf{z}_k - \mathbf{C}_k \check{\mathbf{x}}_k) \\ \hat{\mathbf{P}}_k &= (\mathbf{I} - \mathbf{K} \mathbf{C}_k) \check{\mathbf{P}}_k. \end{aligned} \quad (9.26)$$

So far, we have derived the entire process of the classic Kalman filter. In fact, the Kalman filter has several derivations, and we use the form of the maximum a posteriori probability estimate from a probabilistic perspective. We see that in linear Gaussian systems, the Kalman filter constitutes the largest a posteriori probability estimate in the system. Moreover, since the Gaussian distribution still obeys the Gaussian distribution after linear transformation, we have not made any approximation throughout the process. It can be said that the Kalman filter constitutes the optimal unbiased estimation of the linear system.

9.1.3 Nonlinear System and EKF

After understanding the Kalman filter, we must clarify that the motion equations and observation equations in SLAM are usually nonlinear functions, especially the camera model in the visual SLAM, which requires the use of the camera internal parameter model and the pose expressed by Lie algebra. It can't be a linear system. A Gaussian distribution, after nonlinear transformation, is often no longer a Gaussian distribution. Therefore, in a nonlinear system, we must take a certain approximation and approximate a non-Gaussian distribution to a Gaussian distribution.

We hope to extend the results of the Kalman filter into a nonlinear system called the Extended Kalman Filter (EKF). It is common practice to consider the first-order Taylor expansion of the equation of motion and the observation equation near a certain point, leaving only the first-order term, the linear part, and then deriving it according to the linear system. Let the mean and covariance matrix of $k-1$ moment be $\hat{\mathbf{x}}_{k-1}, \hat{\mathbf{P}}_{k-1}$. At k , we move the equations of motion and observations at $\check{\mathbf{x}}_{k-1}, \hat{\mathbf{P}}_{k-1}$ Textbf{linearization} (equivalent to first-order Taylor expansion), with:

$$\mathbf{x}_k \approx f(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k) + \left. \frac{\partial f}{\partial \mathbf{x}_{k-1}} \right|_{\hat{\mathbf{x}}_{k-1}} (\mathbf{x}_{k-1} - \hat{\mathbf{x}}_{k-1}) + \mathbf{w}_k. \quad (9.27)$$

Note that the partial derivative here is

$$\mathbf{F} = \left. \frac{\partial f}{\partial \mathbf{x}_{k-1}} \right|_{\hat{\mathbf{x}}_{k-1}}. \quad (9.28)$$

Similarly, for the observation equations, there are also:

$$\mathbf{z}_k \approx h(\check{\mathbf{x}}_k) + \left. \frac{\partial h}{\partial \mathbf{x}_k} \right|_{\check{\mathbf{x}}_k} (\mathbf{x}_k - \check{\mathbf{x}}_k) + \mathbf{n}_k. \quad (9.29)$$

Note that the partial derivative here is

$$\mathbf{H} = \left. \frac{\partial h}{\partial \mathbf{x}_k} \right|_{\check{\mathbf{x}}_k}. \quad (9.30)$$

Then, in the **prediction** step, according to the equation of motion:

$$P(\mathbf{x}_k | \mathbf{x}_0, \mathbf{u}_{1:k}, \mathbf{z}_{0:k-1}) = N(f(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k), \mathbf{F}\hat{\mathbf{P}}_{k-1}\mathbf{F}^T + \mathbf{R}_k). \quad (9.31)$$

These derivations and Kalman filters are very similar. For convenience, the mean of the prior and covariance here is recorded.

$$\check{\mathbf{x}}_k = f(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k), \quad \check{\mathbf{P}}_k = \mathbf{F}\hat{\mathbf{P}}_{k-1}\mathbf{F}^T + \mathbf{R}_k. \quad (9.32)$$

Then, consider that in the observation we have:

$$P(\mathbf{z}_k | \mathbf{x}_k) = N(h(\check{\mathbf{x}}_k) + \mathbf{H}(\mathbf{x}_k - \check{\mathbf{x}}_k), \mathbf{Q}_k). \quad (9.33)$$

Finally, based on the initial Bayesian expansion, the posterior probability form of \mathbf{x}_k can be derived. We skip the middle of the derivation process and only introduce the results. The reader can derive the prediction and update equations of the EKF in the same way as the Kalman filter. In short, we will first define a **Kalman gain** \mathbf{K}_k :

$$\mathbf{K}_k = \check{\mathbf{P}}_k \mathbf{H}^T (\mathbf{H} \check{\mathbf{P}}_k \mathbf{H}^T + \mathbf{Q}_k)^{-1}. \quad (9.34)$$

Based on the Kalman gain, the posterior probability is in the form of

$$\hat{\mathbf{x}}_k = \check{\mathbf{x}}_k + \mathbf{K}_k (\mathbf{z}_k - h(\check{\mathbf{x}}_k)), \quad \hat{\mathbf{P}}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{H}) \check{\mathbf{P}}_k. \quad (9.35)$$

The Kalman filter gives the evolution of the distribution of state variables after linearization. Under linear systems and Gaussian noise, the Kalman filter gives an unbiased optimal estimate. In the case of nonlinearity of SLAM, it gives the maximum a posteriori estimate (MAP) for a single linear approximation.

9.1.4 EKF Discussion

EKF is known for its compact form and wide application. When we want to estimate an uncertainty in a certain period of time, the first thing we think of is EKF. In the early SLAM, EKF dominated for a long time, and researchers discussed the use of various filters in SLAM, such as IF (Information Filter)^[?], IKF^[?](Iterated KF), UKF^[?](Unscented KF) and particle filter^[? ?], SWF (Sliding Window Filter)^[?], etc. ^{[?]*}, or use the ideas of divide and conquer to improve the efficiency of EKF^[? ?]. To this day, although we recognize that nonlinear optimization has a clear advantage over filters, EKF is still an effective way when computing resources are limited or when the amount to be estimated is relatively simple.

What are the limitations of EKF?

1. First, the filter method assumes **Markoviness** to some extent, that is, the state of k is only related to the time of $k-1$, and the state before $k-1$ It has nothing to do with observations (or related to the state of the first few finite moments). This is a bit like thinking about the relationship between two

* The principle of particle filter is quite different from Kalman filter.

adjacent frames in a visual odometer. If the current frame is indeed related to data from a long time ago (for example, loopback), the filter will be difficult to process.

Non-linear optimization methods tend to use all historical data. It not only considers the relationship between feature points and trajectories in the vicinity, but also considers the state long before, called SLAM (Full-SLAM) in all time. In this sense, the nonlinear optimization method uses more information and of course requires more calculations.

2. Compared with the optimization method introduced in the sixth lecture, the EKF filter only linearizes **once** at \hat{x}_{k-1} , and then directly. This linearization result calculates the posterior probability. This is equivalent to saying that we think that **the linearization approximation at this point is still valid at the posterior probability**. In fact, when we are away from the working point, the first-order Taylor expansion does not necessarily approximate the entire function, depending on the nonlinearity of the motion model and the observation model. If they have strong nonlinearities, then the linear approximation is only found in a small range, and it cannot be considered that it can be approximated by linearity at a great distance. This is the EKF's **non-linearity error** and its main problem.

In the optimization problem, although we also do the first-order (slowest descent) or second-order (Gaussian Newton method or Levin Berg-Marquartt method) approximation, after each iteration, after the state estimation changes, we will Re-expanding Taylor to the new estimate, instead of doing a Taylor expansion at a fixed point like EKF. This makes the optimization method more applicable and can be applied when the state changes greatly. So in general, you can roughly think that **EKF is just an iteration in optimization***.

3. From a program implementation point of view, the EKF needs to store the mean and variance of the state quantities and maintain and update them. If the road sign is also put into the state, due to the large number of road signs in the visual SLAM, this amount of storage is considerable and squared with the state quantity (because the covariance matrix is to be stored). Therefore, it is generally accepted that EKF SLAM is not suitable for large scenes.
4. Finally, filter methods such as EKF have no anomaly detection mechanism, which causes the system to diverge easily when there are outliers. In visual SLAM, outliers are common: both feature matching and optical flow methods are easy to track or match to the wrong point. No outlier detection mechanism can make the system very unstable in practice.

Due to these obvious shortcomings of EKF, we generally believe that nonlinear optimization can achieve better results under the same amount of computation [?]. "Better" here means precision and robustness at the same time to achieve better meaning. Let's discuss the backend based on nonlinear optimization. We will mainly introduce graph optimization and demonstrate backend optimization with g2o and Ceres.

* More carefully, it is better than an iteration because the linearization of the update step is based on prediction. If you linearize the motion and observation models simultaneously at the time of prediction, it is exactly the same as an iteration.

9.2 BA and graph optimization

If you have done a visual 3D reconstruction, you should be familiar with this concept. The so-called Bundle Adjustment[†], refers to the extraction of the optimal 3D model and camera parameters (internal and external parameters) from the visual image. Consider the bundles of light rays emitted from any feature point, which become pixels or detected feature points on the imaging plane of several cameras. If we adjust the camera pose and the spatial position of each feature point so that the light eventually reaches the camera's optical center [?], it is called BA.

We have briefly introduced the principle of BA in the ?? and ???. The focus of this section is to introduce the characteristics of its corresponding graph model structure, and then introduce some common fast Solution.

9.2.1 Projection model and BA cost function

First we review the entire projection process. Starting from the point \mathbf{p} in a world coordinate system, taking into account the internal and external parameters and distortion of the camera, and finally projecting into pixel coordinates, the following steps are required.

1. First, convert the world coordinates to camera coordinates. Here you will use the extra camera parameter (\mathbf{R}, \mathbf{t}):

$$\mathbf{P}' = \mathbf{R}\mathbf{p} + \mathbf{t} = [X', Y', Z']^T. \quad (9.36)$$

2. Then, cast \mathbf{P}' to the normalized plane to get the normalized coordinates:

$$\mathbf{P}_c = [u_c, v_c, 1]^T = [X'/Z', Y'/Z', 1]^T. \quad (9.37)$$

3. Considers the distortion of the normalized coordinates and obtains the original pixel coordinates before dedistortion. For the time being, only radial distortion is considered:

$$\begin{cases} U'_c = u_c (1 + k_1 r_c^2 + k_2 r_c^4) \\ V'_c = v_c (1 + k_1 r_c^2 + k_2 r_c^4) \end{cases}. \quad (9.38)$$

4. Finally, calculate the pixel coordinates based on the internal parameter model:

$$\begin{cases} u_s = f_x u'_c + c_x \\ v_s = f_y v'_c + c_y \end{cases} \quad (9.39)$$

This series of calculation processes may seem complicated. We use the process ?? to visually represent the whole process to help the reader understand. The reader should be able to understand that this process is also the **observation equation** mentioned earlier. We previously wrote it abstractly as:

$$\mathbf{z} = h(\mathbf{x}, \mathbf{y}). \quad (9.40)$$

Now we give its detailed parameterization process. Specifically, \mathbf{x} here refers to the pose of the camera at this time, that is, the external parameter \mathbf{R}, \mathbf{t} , which

[†] is also translated into beam adjustment, bundle adjustment, etc., but I feel that there is no Bundle Adjustment in English, so the English name is reserved here.

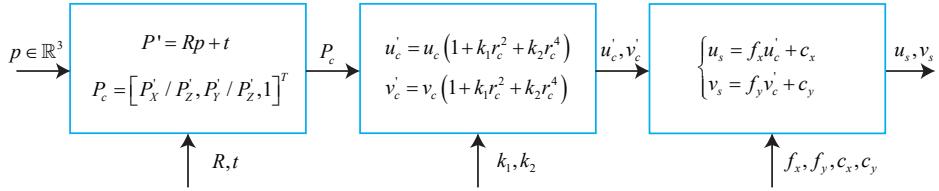


Figure 9-2: Calculation process diagram. The p on the left is the 3D coordinate point in the global coordinate system, and the u_s on the right, v_s is the final pixel coordinate of the point on the image plane. $r_c^2 = u_c^2 + v_c^2$ in the middle distortion module.

corresponds to the Li group as T , Lie algebra is ξ . The road sign y is the three-dimensional point p , and the observation data is the pixel coordinates $z \triangleq [u_s, v_s]^T$. Considering the least squares angle, you can write the error about this observation:

$$e = z - h(T, p). \quad (9.41)$$

Then, taking into account the observations at other times, we can add a subscript to the error. Let \mathbf{z}_{ij} be the data generated by observing the road sign \mathbf{p}_j in the pose \mathbf{T}_i , then the overall **cost function** (Cost) Function is

$$\frac{1}{2} \sum_{i=1}^m \sum_{j=1}^n \|e_{ij}\|^2 = \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^n \|z_{ij} - h(T_i, p_j)\|^2. \quad (9.42)$$

Solving this least squares is equivalent to adjusting the pose and the road sign at the same time, which is called BA. Next, we will gradually explore the solution of the model based on the objective function and the nonlinear optimization content introduced in ??.

9.2.2 BA's solution

Observing the observation model $h(\mathbf{T}, \mathbf{p})$ in the previous section, it is easy to judge that the function is not a linear function. So we want to optimize it using some of the nonlinear optimizations introduced in the ?? section. According to the idea of nonlinear optimization, we should start from a certain initial value and constantly search for the descending direction $\Delta\mathbf{x}$ to find the optimal solution of the objective function, that is, continuously solve the incremental equation (??) The increment $\Delta\mathbf{x}$ in . Although the error term is for a single pose and landmark, on the overall BA objective function, we should define the argument as all variables to be optimized:

$$x = [T_1, \dots, T_m, p_1, \dots, p_n]^T. \quad (9.43)$$

Accordingly, Δx in the incremental equation is an increment to the overall argument. In this sense, when we give an increment to the argument, the objective function becomes

$$\frac{1}{2} \|f(\mathbf{x} + \Delta\mathbf{x})\|^2 \approx \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^n \|e_{ij} + \mathbf{F}_{ij}\Delta\xi_i + \mathbf{E}_{ij}\Delta\mathbf{p}_j\|^2. \quad (9.44)$$

Where \mathbf{F}_{ij} represents the partial derivative of the entire cost function to the camera pose in the current state, and \mathbf{E}_{ij} represents the partial derivative of the function to the location of the landmark. We have introduced their specific forms in the ?? section, so we will not start deriving here. Now put the camera pose variables together:

$$\mathbf{x}_c = [\xi_1, \xi_2, \dots, \xi_m]^T \in \mathbb{R}^{6m}, \quad (9.45)$$

And put the variables of the space point together:

$$\mathbf{x}_p = [p_1, p_2, \dots, p_n]^T \in \mathbb{R}^{3n}, \quad (9.46)$$

Then, the formula (??) can be simplified as follows:

$$\frac{1}{2} \|f(\mathbf{x} + \Delta\mathbf{x})\|^2 = \frac{1}{2} \|\mathbf{e} + \mathbf{F}\Delta\mathbf{x}_c + \mathbf{E}\Delta\mathbf{x}_p\|^2. \quad (9.47)$$

It should be noted that this formula changes from a sum of many small quadratic terms to a more holistic one. Here the Jacobian matrix \mathbf{E} and \mathbf{F} must be the derivative of the overall objective function to the global variable, it will be a large block matrix, and each small block inside, need The " \mathbf{F}_{ij} and \mathbf{E}_{ij} " spliced together" for each error term. Then, whether we use the Gauss Newton method or the Levenberg-Marquart method, we will eventually face the incremental linear equation:

$$\mathbf{H}\Delta\mathbf{x} = \mathbf{g}. \quad (9.48)$$

According to the knowledge of ??, we know that the main difference between the Gauss Newton method and the Levenberg-Marquart method is that \mathbf{H} is $\mathbf{J}^T\mathbf{J}$ is still in the form of $\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I}$. Since we classify variables into poses and spatial points, the Jacobian matrix can be divided into blocks.

$$\mathbf{J} = [\mathbf{F} \ \mathbf{E}]. \quad (9.49)$$

Then, taking the Gauss-Newton method as an example, the \mathbf{H} matrix is

$$\mathbf{H} = \mathbf{J}^T\mathbf{J} = \begin{bmatrix} \mathbf{F}^T\mathbf{F} & \mathbf{F}^T\mathbf{E} \\ \mathbf{E}^T\mathbf{F} & \mathbf{E}^T\mathbf{E} \end{bmatrix}. \quad (9.50)$$

Of course, we also need to calculate this matrix in the Levenberg-Marquart method. It's not hard to find that because all the optimization variables are considered, the dimensions of this linear equation will be very large, including all camera poses and landmark points. Especially in visual SLAM, an image will present hundreds of feature points, greatly increasing the scale of this linear equation. If you calculate the incremental equation directly by inverting \mathbf{H} , since the matrix inversion is an operation of $O(n^3)$, [?], this is very expensive. Computing resources. Fortunately, the \mathbf{H} matrix here has a certain special structure. With this special structure, we can speed up the solution process.

9.2.3 Sparseness and marginalization

An important development of the 21st century visual SLAM is the recognition of the sparse structure of the matrix \mathbf{H} , and found that the structure can naturally and explicitly use graph optimization to represent [? ?]. This section will discuss the matrix sparse structure in detail.

The sparsity of the \mathbf{H} matrix is caused by the Jacobian matrix $\mathbf{J}(\mathbf{x})$. Consider one of these cost functions, e_{ij} . Note that this error term only describes the \mathbf{p}_j in \mathbf{T}_i , which only involves the i camera pose and the j landmark point. The derivative of the remaining variables is 0. Therefore, the Jacobian matrix corresponding to the error term has the following form:

$$\mathbf{J}_{ij}(\mathbf{x}) = \left(\mathbf{0}_{2 \times 6}, \dots, \mathbf{0}_{2 \times 6}, \frac{\partial e_{ij}}{\partial T_i}, \mathbf{0}_{2 \times 6}, \dots, \mathbf{0}_{2 \times 3}, \dots, \mathbf{0}_{2 \times 3}, \frac{\partial e_{ij}}{\partial p_j}, \mathbf{0}_{2 \times 3}, \dots, \mathbf{0}_{2 \times 3} \right). \quad (9.51)$$

Where $\mathbf{0}_{2 \times 6}$ represents the $\mathbf{0}$ matrix with dimensions 2×6 , and the same $\mathbf{0}_{2 \times 3}$ same. The bias term for the camera pose is $\partial e_{ij} / \partial \xi_i$ dimension is 2×6 , partial guide to the landmark point The $\partial e_{ij} / \partial p_j$ dimension is 2×3 . The Jacobian matrix of this error term is zero except that the two are non-zero blocks. This embodies the fact that the error term is independent of other road signs and tracks. From the perspective of graph optimization, this observation edge is only related to two vertices. So, what effect does it have on the incremental equation? Why does the \mathbf{H} matrix produce sparsity?

Taking ?? as an example, we set \mathbf{J}_{ij} to have only non-zero blocks at i, j , then its contribution to \mathbf{H} For $\mathbf{J}_{ij}^T \mathbf{J}_{ij}$, it has the sparse form drawn on the graph. The $\mathbf{J}_{ij}^T \mathbf{J}_{ij}$ matrix also has only 4 non-zero blocks, located at $(i, i), (i, j), (j, i), (j, j)$. For the overall \mathbf{H} , there are:

$$\mathbf{H} = \sum_{i,j} \mathbf{J}_{ij}^T \mathbf{J}_{ij}, \quad (9.52)$$

Note that i takes values in all camera poses, and j takes values in all landmark points. We divide \mathbf{H} into blocks:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_{11} & \mathbf{H}_{12} \\ \mathbf{H}_{21} & \mathbf{H}_{22} \end{bmatrix}. \quad (9.53)$$

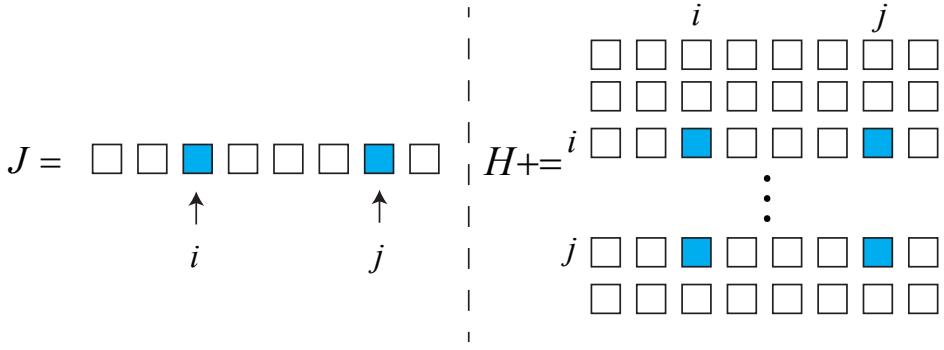


Figure 9-3: When an error term \mathbf{J} is sparse, its contribution to \mathbf{H} is also sparse.

Here \mathbf{H}_{11} is only related to the camera pose, and \mathbf{H}_{22} is only related to the landmark points. When we traverse i, j , the following facts are always true:

1. No matter how i, j changes, \mathbf{H}_{11} is a diagonal matrix, with only non-zero blocks at $\mathbf{H}_{i,i}$.
2. Similarly, \mathbf{H}_{22} is also a diagonal array with only non-zero blocks at $\mathbf{H}_{j,j}$.

3. For \mathbf{H}_{12} and \mathbf{H}_{21} , they may be sparse or dense, depending on the specific observations.

This shows the sparse structure of \mathbf{H} . Later, in solving the linear equation, it is also necessary to use its sparse structure. Perhaps the reader has not yet grasped the meaning of this, and we give an example to illustrate its situation. Suppose there are 2 camera poses (C_1, C_2) and 6 road signs ($P_1, P_2, P_3, P_4, P_5, P_6$) in a scene. The variables for these cameras and point clouds are $\mathbf{T}_i, i = 1, 2$ and $\mathbf{p}_j, j = 1, \dots, 6$. The camera C_1 observed the road signs P_1, P_2, P_3, P_4 , and the camera C_2 observed the road signs P_3, P_4, P_5, P_6 . We draw this process as a schematic ?? . Cameras and road signs are represented by circular nodes. If the i camera is able to observe the j point, we will connect an edge to their corresponding node.

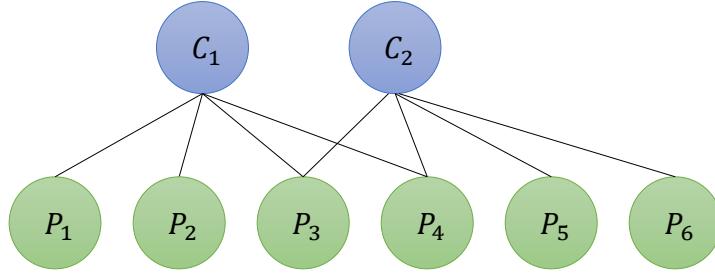


Figure 9-4: A diagram of the composition of points and edges. The figure shows that the camera C_1 has observed the signposts P_1, P_2, P_3, P_4 , and the camera C_2 has seen P_3 to P_6 .

Can be launched, the BA target function under the scene should be

$$\frac{1}{2} \left(\|e_{11}\|^2 + \|e_{12}\|^2 + \|e_{13}\|^2 + \|e_{14}\|^2 + \|e_{23}\|^2 + \|e_{24}\|^2 + \|e_{25}\|^2 + \|e_{26}\|^2 \right). \quad (9.54)$$

Here e_{ij} uses the previously defined cost function, ie, (??). Take e_{11} as an example. It describes the P_1 seen in C_1 , regardless of other camera poses and road signs. Let \mathbf{J}_{11} be the Jacobian matrix for e_{11} , it's not hard to see that e_{11} is for the camera variable $\mathbf{\xi}_1$ and the landmarks $\mathbf{p}_2, \dots, \mathbf{p}_6$ are all zero. We put all variables as $\mathbf{x} = (\mathbf{\xi}_1, \mathbf{\xi}_2, \mathbf{p}_1, \dots, \mathbf{p}_2)$ The order of \mathbf{x} is placed, then:

$$\mathbf{J}_{11} = \frac{\partial e_{11}}{\partial \mathbf{x}} = \left(\frac{\partial e_{11}}{\partial \mathbf{\xi}_1}, \mathbf{0}_{2 \times 6}, \frac{\partial e_{11}}{\partial \mathbf{p}_1}, \mathbf{0}_{2 \times 3}, \mathbf{0}_{2 \times 3}, \mathbf{0}_{2 \times 3}, \mathbf{0}_{2 \times 3}, \mathbf{0}_{2 \times 3} \right). \quad (9.55)$$

In order to facilitate the representation of sparsity, we use a square with a color to indicate that the matrix has a value in the square, and the remaining areas without color indicate that the matrix has a value of zero at that point. Then the above \mathbf{J}_{11} can be expressed as ?? . Similarly, other Jacobian matrices will have similar sparse patterns.

In order to get the Jacobian matrix corresponding to the objective function, we can classify these \mathbf{J}_{ij} as vectors in a certain order, then the overall Jacobian matrix and the corresponding \mathbf{H} matrix The sparse situation is as shown in ?? .

$$J_{11} = \begin{bmatrix} C_1 & C_2 & P_1 & P_2 & P_3 & P_4 & P_5 & P_6 \end{bmatrix}$$

The non-zero
block distribution of the

Figure 9-5: J_{11} matrix. The upper mark indicates the variable corresponding to the column of the matrix. Since the camera parameter dimension is larger than the point cloud parameter dimension, the matrix block corresponding to C_1 is wider than the matrix block corresponding to P_1 .

$$J = \begin{bmatrix} J_{11} \\ J_{12} \\ J_{13} \\ J_{14} \\ J_{23} \\ J_{24} \\ J_{25} \\ J_{26} \end{bmatrix} = \begin{bmatrix} C_1 & C_2 & P_1 & P_2 & P_3 & P_4 & P_5 & P_6 \end{bmatrix}$$

$$H = J^T J = \begin{bmatrix} \text{filled} & \text{empty} \\ \text{empty} & \text{filled} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} \\ \text{empty} & \text{empty} & \text{filled} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} \\ \text{empty} & \text{empty} & \text{empty} & \text{filled} & \text{empty} & \text{empty} & \text{empty} & \text{empty} \\ \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{filled} & \text{empty} & \text{empty} & \text{empty} \\ \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{filled} & \text{empty} & \text{empty} \\ \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{filled} & \text{empty} \\ \text{empty} & \text{filled} \end{bmatrix}$$

Figure 9-6: Sparseness of the Jacobian matrix (left) and the sparsity of the H matrix (right). The filled squares indicate that the matrix has values at the corresponding matrix block, and the remaining parts with no color represent the matrix. The value there is always 0.

Perhaps you have noticed that ?? corresponds to **adjacency matrix*** and the H matrix in the above figure, except for the diagonal elements, have the same structure. In fact, it is. The above H matrix has a total of 8×8 matrix blocks. For matrix blocks in the H matrix that are not diagonal, if the matrix block is non-zero, then there will be an edge between the variables corresponding to their position in the figure, we can clearly see this from ?? . Therefore, the non-zero matrix block of the non-diagonal portion of the H matrix can be understood as a relationship between its corresponding two variables, or can be called a constraint. Therefore, we find that there is a clear connection between the graph optimization structure and the sparseness of the incremental equation.

Now consider the more general case, if we have m camera pose, n landmark points. Since there are usually more road signs than cameras, there is $n \gg m$. From the above reasoning, the actual H matrix will be as shown in ?? . Its upper left corner block is very small, while the diagonal corner block in the lower right corner occupies a lot of places. In addition to this, the non-diagonal parts are distributed with scattered observation data. Because it is shaped like an arrow, it is also called an arrow-like matrix [?]. At the same time, it is also very similar to a scorpion, so I also call it the matrix matrix *.

For H with this sparse structure, what is the difference between the linear equa-

* the so-called adjacency matrix is a matrix whose i, j elements describe whether there is an edge between nodes i and j . If this edge exists, set this element to 1, otherwise set to 0.

* This is a joke, please do not write in the formal academic paper.

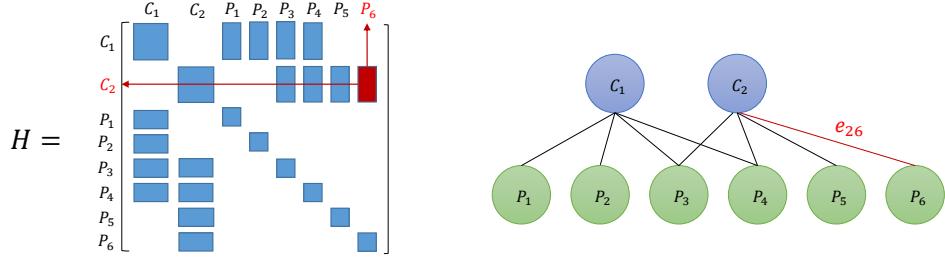


Figure 9-7: \mathbf{H} The correspondence between non-zero matrix blocks in the matrix and the edges in the graph. For example, the red matrix block on the right side of the \mathbf{H} matrix on the left shows that there is an edge between the corresponding variables C_2 and P_6 in the right image. e_{26} .

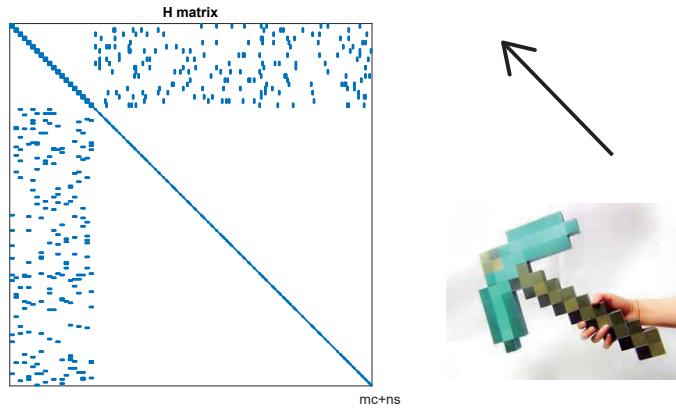


Figure 9-8: Generally \mathbf{H} matrix.

tion $\mathbf{H}\Delta\mathbf{x} = \mathbf{g}$? There are several ways to use the sparse acceleration calculation of \mathbf{H} in reality. This section introduces one of the most commonly used methods in visual SLAM: Schur elimination. Also known as Marginalization in SLAM research.

Looking closely at ??, we can easily see that this matrix can be divided into 4 blocks, which is consistent with the formula (??). The upper left corner is a diagonal block matrix, and the dimensions of each diagonal block element are the same as the dimensions of the camera pose, and are a diagonal block matrix. The lower right corner is also a diagonal block matrix, and the dimension of each diagonal block is the dimension of the road sign. The structure of the non-diagonal block is related to the specific observation data. We first divide the matrix into regions according to the way shown in ???. It is not difficult for the reader to find that the four regions correspond to the four matrix blocks in the formula (??). For the convenience of subsequent analysis, we remember that these 4 blocks are $\mathbf{B}, \mathbf{E}, \mathbf{E}^T, \mathbf{C}$.

Thus, the corresponding linear system of equations can also be changed from $\mathbf{H}\Delta\mathbf{x} = \mathbf{g}$ to the following form:

$$\begin{bmatrix} \mathbf{B} & \mathbf{E} \\ \mathbf{E}^T & \mathbf{C} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_c \\ \Delta\mathbf{x}_p \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ \mathbf{w} \end{bmatrix}. \quad (9.56)$$

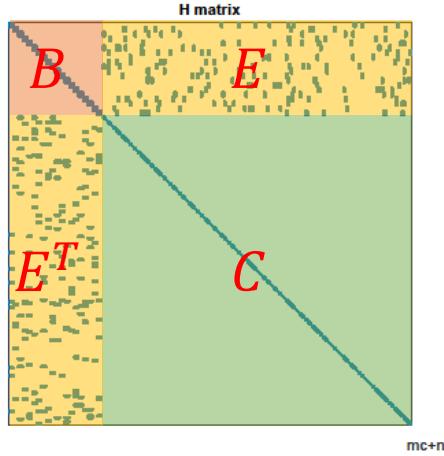


Figure 9-9: \mathbf{H} The division of the matrix.

Where \mathbf{B} is a diagonal block matrix, the dimensions of each diagonal block are the same as the dimensions of the camera parameters, and the number of diagonal blocks is the number of camera variables. Since the number of road signs is much larger than the number of camera variables, \mathbf{C} is often much larger than \mathbf{B} . Each landmark in 3D space is three-dimensional, so the \mathbf{C} matrix is a diagonal block matrix, and each block is a 3×3 matrix. The difficulty of inverting the diagonal block matrix is much less difficult than the inverse of the general matrix, because we only need to invert the diagonal matrix blocks separately. Considering this property, we perform Gaussian elimination on the linear equations. The goal is to eliminate the non-diagonal part of the upper right corner, \mathbf{E} , to get:

$$\begin{bmatrix} \mathbf{I} & -\mathbf{E}\mathbf{C}^{-1} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{B} & \mathbf{E} \\ \mathbf{E}^T & \mathbf{C} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_c \\ \Delta\mathbf{x}_p \end{bmatrix} = \begin{bmatrix} \mathbf{I} & -\mathbf{E}\mathbf{C}^{-1} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \mathbf{w} \end{bmatrix}. \quad (9.57)$$

Finishing, get:

$$\begin{bmatrix} \mathbf{B} - \mathbf{E}\mathbf{C}^{-1}\mathbf{E}^T & \mathbf{0} \\ \mathbf{E}^T & \mathbf{C} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_c \\ \Delta\mathbf{x}_p \end{bmatrix} = \begin{bmatrix} \mathbf{v} - \mathbf{E}\mathbf{C}^{-1}\mathbf{w} \\ \mathbf{w} \end{bmatrix}. \quad (9.58)$$

After the elimination, the first line of the equation becomes an item unrelated to $\Delta\mathbf{x}_p$. Take it out alone and get the incremental equation for the pose part:

$$[\mathbf{B} - \mathbf{E}\mathbf{C}^{-1}\mathbf{E}^T] \Delta\mathbf{x}_c = \mathbf{v} - \mathbf{E}\mathbf{C}^{-1}\mathbf{w}. \quad (9.59)$$

The dimensions of this linear equation are the same as the \mathbf{B} matrix. Our approach is to solve this equation first, then substitute the solved $\Delta\mathbf{x}_c$ into the original equation, and then solve $\Delta\mathbf{x}_p$. This process is called **Marginalization**^[?], or **Schur Elimination**. Compared to the direct solution of linear equations, its advantages are:

1. In the elimination process, since \mathbf{C} is a diagonal block, \mathbf{C}^{-1} is easy to solve.
After
2. solves $\Delta\mathbf{x}_c$, the incremental equation for the road sign portion is $\Delta\mathbf{x}_p = \mathbf{C}^{-1}(\mathbf{w} - \mathbf{E}^T\Delta\mathbf{x}_c)$ is given. This still uses the features of \mathbf{C}^{-1} that are easy to solve.

Thus, the main computational amount of marginalization lies in the solution (??). There is not much we can say about this equation. It is just an ordinary linear equation and no special structure can be utilized. Let's write the coefficient of this equation as \mathbf{S} . How is it sparse? ?? shows a \mathbf{S} instance after the Schur elimination, and it can be seen that its sparsity is irregular.

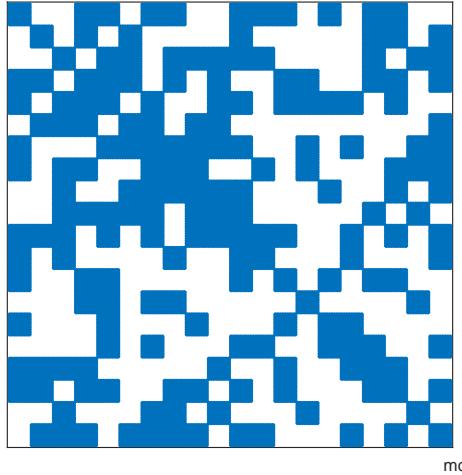


Figure 9-10: The sparse state of the \mathbf{S} matrix after the Schur elimination of the \mathbf{H} matrix.

As mentioned earlier, the non-zero elements at the non-diagonal blocks of the \mathbf{H} matrix correspond to the association of the camera and the landmark. So, does the sparsity of \mathbf{S} after Schur elimination have physical meaning? The answer is yes. Here we do not prove that the non-zero matrix block on the off-diagonal line of the \mathbf{S} matrix indicates that there are common observations between the two camera variables. It is called Co-visibility. Conversely, if the block is zero, it means that the two cameras are not observed together. For example, the sparse matrix shown by ?? , the top left 4×4 matrix blocks can indicate that the corresponding camera variables C_1, C_2, C_3, C_4 have common observations.

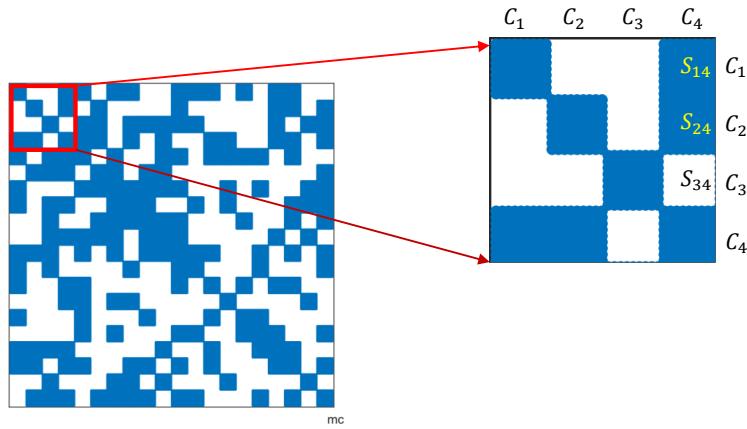


Figure 9-11: Take the 4×4 matrix block in the \mathbf{S} matrix as an example. The matrix block in this area is S_{14} , S_{24} is not zero, indicating that there is a common observation point between the camera C_4 and the camera C_1 , C_2 ; and S_{34} is zero, indicating that there is no common observation between C_3 and C_4 . Road sign.

Thus, the sparse structure of the \mathbf{S} matrix depends on the actual observations, and we cannot predict in advance. In practice, for example, the Local Mapping link in ORB-SLAM^[?] deliberately selects frames with common observations as key frames when doing BA, in which case Schur eliminates The resulting \mathbf{S} is a dense matrix. However, since this module is not executed in real time, this is acceptable. But in other methods, such as DSO^[?], OKVIS^[?], etc., they use the sliding window method (Sliding Window). This type of method requires a BA for each frame to prevent the accumulation of errors, so they must also use some techniques to maintain the sparsity of the \mathbf{S} matrix. Readers who want to be able to go deeper into this piece can refer to their papers. I won't talk about these details in detail here.

From a probabilistic point of view, we call this step marginal because we actually turn the problem of $(\Delta\mathbf{x}_c, \Delta\mathbf{x}_p)$ into a fixed one. $\Delta\mathbf{x}_p$, find $\Delta\mathbf{x}_c$, and then ask for $\Delta\mathbf{x}_p$. This step is equivalent to doing a conditional probability expansion:

$$P(\mathbf{x}_c, \mathbf{x}_p) = P(\mathbf{x}_c|\mathbf{x}_p)P(\mathbf{x}_p), \quad (9.60)$$

The result is the edge distribution for \mathbf{x}_p , so it is called marginalization. In the marginalization process we talked about earlier, we actually marginalized all the landmark points. According to the actual situation, we can also choose a part to be marginalized. At the same time, Schur elimination is only one way to achieve marginalization, and can also be marginalized using Cholesky decomposition.

The reader may continue to ask, after the Schur elimination, we also need to solve the linear equations (??). Is there any trick to solve it? Unfortunately, this part is a traditional matrix numerical solution, usually calculated by decomposition. Regardless of which solution is used, we recommend Schur elimination using the sparsity of \mathbf{H} . Not only does this increase speed, but also because the conditional cost of the \mathbf{S} matrix after the elimination is often smaller than the previous \mathbf{H} matrix. Schur elimination does not mean that all road signs are eliminated. The elimination of camera variables is also the means used in SLAM.

The solution to The

9.2.4 Robust kernel function

In the previous BA problem, we minimize the sum of the squares of the two norms of the error term as the objective function. Although this method is very intuitive, there is a serious problem: If the data given by an error item is wrong for reasons such as mismatching, what will happen? We add an edge that should not be added to the graph. However, the optimization algorithm does not recognize that this is an erroneous data. It treats all data as errors. This data, in the opinion of the algorithm, is observed with a small probability of seeing the data. At this point, there will be a very large margin in the graph optimization, and its gradient is also large, meaning that adjusting the variables associated with it will cause the objective function to drop more. Therefore, the algorithm will attempt to prioritize the estimates of the nodes to which this edge is connected, so that they conform to the unreasonable requirements of this edge. Since the error on this side is really large, it tends to smooth out the effects of other correct edges, allowing the optimization algorithm to focus on adjusting an incorrect value. This is obviously not what we want to see.

The reason for this problem is that the two norms grow too fast when the error is large. Then there is the existence of a nuclear function. The kernel function guarantees that the error on each side will not be too large to cover up the other edges. The specific way is to replace the two norm metric of the original error

with a function that does not grow so fast, while at the same time guaranteeing its own smooth nature (otherwise it can't be derived!). Because they make the whole optimization result more robust, they are also called Robust Kernel.

There are many types of robust kernel functions, such as the most commonly used Huber kernel:

$$H(e) = \begin{cases} \frac{1}{2}e^2 & |e| \leq \delta, \\ \delta(|e| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \quad (9.61)$$

We see that when the error e is greater than a certain threshold of δ , the function growth changes from a quadratic form to a form, which is equivalent to limiting the maximum value of the gradient. At the same time, the Huber kernel function is smooth and can be easily derived. ?? shows the comparison between the Huber kernel function and the quadratic function. It can be seen that the Huber kernel function grows significantly lower than the quadratic function when the error is large.

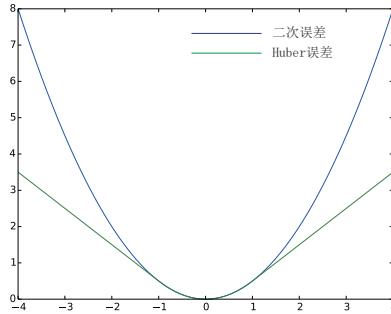


Figure 9-12: Huber kernel function.

In addition to the Huber core, there are Cauchy kernels, Tukey cores, etc. Readers can see which kernel functions are provided by both g2o and Ceres.

9.2.5

In this section we focus on sparsity issues in BA. However, in practice, most software libraries have implemented detailed operations for us, and what we need to do is mainly to construct the Bundle Adjustment problem, set the Schur elimination, and then call the dense or sparse matrix solver to optimize the variables. If you want to know more about BA, you can read the [?] reference on the basis of reading this section.

In the next two sections, we will use the two libraries of Ceres and g2o to do the Bundle Adjustment. In order to reflect their differences, we will use a public data set BAL[?] and use the shared read and write code.

9.3 Practice: Ceres BA

9.3.1 BALDataset

We used the BAL data set for the demonstration of BA. The BAL data set provides several scenes, and the camera and landmark information in each scene is given by a text file. In this example, use the file problem-16-22106-pre.txt as an example. The

file stores the information of the BA problem in a row. See the detailed format. We read the contents of the file using the BALProblem class defined in common.h and then solve it with Ceres and g2o respectively.

It should be noted that the BAL data set has some special features:

1. The camera internal parameter model of BAL is given by the focal length f and the distortion parameters k_1, k_2 . f is similar to the f_x and f_y we mentioned. Since the photo pixels are basically square, in many practical situations f_x is very close to f_y , and it is not a bad idea to use the same value. In addition, there is no c_x, c_y in this model, because the stored data has already removed these two values.
2. The BAL data is projected with the projection plane behind the camera's optical center, so we need to multiply the coefficient by -1 after projection, as calculated by our previous model. However, most datasets still use the projection plane in front of the optical center, and we should read the format specification carefully before using the dataset.

After reading the data with the BALProblem class, we can call the Normalize function to normalize the original data, or add noise to the data through the Perturb function. Normalization refers to zeroing the center of all landmark points and then scaling them to a suitable scale. This will make the values more stable during the optimization process, preventing BA problems that are large or have large offsets in extreme cases.

Readers are advised to read the other interfaces of the BALProblem class. Since these codes are only responsible for reading and writing peripheral functions such as data, we don't give them in the text to save space. After solving the BA, we can also use the function of this class to write the result to a ply file (a point cloud file format), and then use the meshlab software to view. Meshlab can be installed by apt-get, and the installation method will not be described here.

9.3.2 Ceres BA Writing

In the bundle_adjustment_ceres.cpp file, we implemented the process of Ceres solving BA. The key to using Ceres is to define the projection error model, which is given in SnavelyReprojectionError.h:

Listing 9.1: slambook2/ch9/SnavelyReprojectionError.cpp

```

1 class SnavelyReprojectionError {
2 public:
3     SnavelyReprojectionError(double observation_x, double observation_y) :
4         observed_x(observation_x),
5         observed_y(observation_y) {}
6
7     template<typename T>
8     bool operator()(const T *const camera,
9                      const T *const point,
10                     T *residuals) const {
11             // camera[0,1,2] are the angle-axis rotation
12             T predictions[2];
13             CamProjectionWithDistortion(camera, point, predictions);
14             residuals[0] = predictions[0] - T(observed_x);
15             residuals[1] = predictions[1] - T(observed_y);
16
17             return true;

```

```

17 }
18 // camera : 9 dims array
19 // [0–2] : angle-axis rotation
20 // [3–5] : translation
21 // [6–8] : camera parameter, [6] focal length, [7–8] second and forth
22 // order radial distortion
23 // point : 3D location.
24 // predictions : 2D predictions with center of the image plane.
25 template<typename T>
26 static inline bool CamProjectionWithDistortion(const T *camera, const T *
27     point, T *predictions) {
28     // Rodrigues' formula
29     T p[3];
30     AngleAxisRotatePoint(camera, point, p);
31     // camera[3,4,5] are the translation
32     p[0] += camera[3];
33     p[1] += camera[4];
34     p[2] += camera[5];
35
36     // Compute the center fo distortion
37     T xp = -p[0] / p[2];
38     T yp = -p[1] / p[2];
39
40     // Apply second and fourth order radial distortion
41     const T &l1 = camera[7];
42     const T &l2 = camera[8];
43
44     T r2 = xp * xp + yp * yp;
45     T distortion = T(1.0) + r2 * (l1 + l2 * r2);
46
47     const T &focal = camera[6];
48     predictions[0] = focal * distortion * xp;
49     predictions[1] = focal * distortion * yp;
50
51     return true;
52 }
53
54 static ceres::CostFunction *Create(const double observed_x, const double
55     observed_y) {
56     return (new ceres::AutoDiffCostFunction<SnavelyReprojectionError, 2, 9,
57         3>(
58         new SnavelyReprojectionError(observed_x, observed_y)));
59 }
60
61 private:
62     double observed_x;
63     double observed_y;
64 };

```

The parentheses operator of this class implements the interface for Ceres calculation error, and the actual calculation is in the `CamProjectionWithDistortion` function. Note that in Ceres, we must store the optimization variables as a double array. Now each camera has a total of 6-dimensional pose, 1D focal length and 2D distortion parameters, which are described by a total of 9-dimensional parameters. We must also store them in this order in actual storage. The static function `Create` of this class acts as an external call interface and directly returns a Ceres cost function that can be automatically derived. We just call the `Create` function and put the cost function into `ceres::Problem`.

Next we implement the part of BA building and solving:

Listing 9.2: slambook2/ch9/SnavelyReprojectionError.cpp

```

1 void SolveBA(BALProblem &bal_problem) {
2     const int point_block_size = bal_problem.point_block_size();
3     const int camera_block_size = bal_problem.camera_block_size();
4     double *points = bal_problem.mutable_points();
5     double *cameras = bal_problem.mutable_cameras();
6
7     // Observations is 2 * num_observations long array observations
8     // [u_1, u_2, ... u_n], where each u_i is two dimensional, the x
9     // and y position of the observation.
10    const double *observations = bal_problem.observations();
11    ceres::Problem problem;
12
13    for (int i = 0; i < bal_problem.num_observations(); ++i) {
14        ceres::CostFunction *cost_function;
15
16        // Each Residual block takes a point and a camera as input
17        // and outputs a 2 dimensional Residual
18        cost_function =
19            SnavelyReprojectionError::Create(observations[2 * i + 0],
20                                            observations[2 * i + 1]);
21
22        // If enabled use Huber's loss function.
23        ceres::LossFunction *loss_function = new ceres::HuberLoss(1.0);
24
25        // Each observation corresponds to a pair of a camera and a point
26        // which are identified by camera_index()[i] and point_index()[i]
27        // respectively.
28        double *camera = cameras + camera_block_size * bal_problem.camera_index()
29                      [i];
30        double *point = points + point_block_size * bal_problem.point_index()[i];
31
32        problem.AddResidualBlock(cost_function, loss_function, camera, point);
33    }
34
35    // Observations is 2 * num_observations long array observations
36    // [u_1, u_2, ... u_n], where each u_i is two dimensional, the x
37    // and y position of the observation.
38    Const double *observations = bal_problem.observations();
39    Ceres::Problem problem;
40
41    std::cout << "Solving ceres BA ... " << endl;
42    ceres::Solver::Options options;
43    options.linear_solver_type = ceres::LinearSolverType::SPARSE_SCHUR;
44    options.minimizer_progress_to_stdout = true;
45    ceres::Solver::Summary summary;
46    ceres::Solve(options, &problem, &summary);
47    std::cout << summary.FullReport() << "\n";
48 }
```

Visible problem building part is quite simple. If you want to add another cost function, the whole process will not change much. Finally, in `ceres::Solver::Options`, we can set the solution method. The use of `SPARSE_SCHUR` will allow Ceres to actually solve the same process as we described earlier, that is, Schur marginalization of the road sign portion to solve this problem in an accelerated manner. However, in Ceres we can't control which parts of the variable are marginalized, which is automatically found and calculated by the Ceres solver.

Ceres' BA solution output is as follows:

Listing 9.3: terminal output

```

1 ./build/bundle_adjustment_ceres problem_16_22106_pre.txt
2 Header: 16 22106 83718bal problem file loaded...
3 bal problem have 16 cameras and 22106 points.
4 Forming 83718 observations.
5 Solving ceres BA ...
6 iter      cost      cost_change lgradient  lstep1    tr_ratio  tr_radius
7     ls_iter   iter_time total_time
8 0  1.842900e+07  0.00e+00  2.04e+06  0.00e+00  0.00e+00  1.00e+04
9     0       6.10e-02  2.24e-01
10 1  1.449093e+06  1.70e+07  1.75e+06  2.16e+03  1.84e+00  3.00e+04
11    1       1.79e-01  4.03e-01
12 2  5.848543e+04  1.39e+06  1.30e+06  1.55e+03  1.87e+00  9.00e+04
13    1       1.56e-01  5.59e-01
14 3  1.581483e+04  4.27e+04  4.98e+05  4.98e+02  1.29e+00  2.70e+05
15    1       1.51e-01  7.10e-01
16 .....
```

The overall error should continue to decrease as the number of iterations increases. Finally, we will optimize the pre-optimized and optimized point cloud output as initial.ply and final.ply, and use meshlab to open these two point clouds directly. The resulting graph is shown in ??.

9.4 Practice: g2o solves BA

Let's consider how to solve this BA problem using g2o. As before, g2o uses a graph model to describe the structure of the problem, so we use nodes to represent the camera and road signs, and then use edges to represent the observations between them. We still use custom points and edges, just cover some key functions. For cameras and road signs, we can define the following structure and use the override keyword to represent the coverage of the base class virtual function:

Listing 9.4: slambook2/ch9/bundle_adjustment_g2o.cpp(fragment)

```

1 //////////////////////////////////////////////////////////////////
2 struct PoseAndIntrinsics {
3     PoseAndIntrinsics() {}
4
5     // set from given data address
6     explicit PoseAndIntrinsics(double *data_addr) {
7         rotation = SO3d::exp(Vector3d(data_addr[0], data_addr[1], data_addr[2]));
8         );
9         translation = Vector3d(data_addr[3], data_addr[4], data_addr[5]);
10        focal = data_addr[6];
11        k1 = data_addr[7];
12        k2 = data_addr[8];
13    }
14
15    //////////////////////////////////////////////////////////////////
16    void set_to(double *data_addr) {
17        auto r = rotation.log();
18        for (int i = 0; i < 3; ++i) data_addr[i] = r[i];
19        for (int i = 0; i < 3; ++i) data_addr[i + 3] = translation[i];
20        data_addr[6] = focal;
21        data_addr[7] = k1;
22        data_addr[8] = k2;
23    }
24
25    SO3d rotation;
26    Vector3d translation = Vector3d::Zero();
```

```

26     double focal = 0;
27     double k1 = 0, k2 = 0;
28 };
29
30 //////////////////////////////////////////////////////////////////
31 class VertexPoseAndIntrinsics : public g2o::BaseVertex<9, PoseAndIntrinsics>
32 {
33 public:
34     EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
35
36     VertexPoseAndIntrinsics() {}
37
38     virtual void setToOriginImpl() override {
39         _estimate = PoseAndIntrinsics();
40     }
41
42     virtual void oplusImpl(const double *update) override {
43         _estimate.rotation = S03d::exp(Vector3d(update[0], update[1], update[2]));
44         _estimate.translation += Vector3d(update[3], update[4], update[5]);
45         _estimate.focal += update[6];
46         _estimate.k1 += update[7];
47         _estimate.k2 += update[8];
48     }
49
50     Vector2d project(const Vector3d &point) {
51         Vector3d pc = _estimate.rotation * point + _estimate.translation;
52         pc = -pc / pc[2];
53         double r2 = pc.squaredNorm();
54         double distortion = 1.0 + r2 * (_estimate.k1 + _estimate.k2 * r2);
55         return Vector2d(_estimate.focal * distortion * pc[0],
56                         _estimate.focal * distortion * pc[1]);
57     }
58
59     virtual bool read(istream &in) {}
60
61     virtual bool write(ostream &out) const {}
62 };
63
64 class VertexPoint : public g2o::BaseVertex<3, Vector3d> {
65 public:
66     EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
67
68     VertexPoint() {}
69
70     virtual void setToOriginImpl() override {
71         _estimate = Vector3d(0, 0, 0);
72     }
73
74     virtual void oplusImpl(const double *update) override {
75         _estimate += Vector3d(update[0], update[1], update[2]);
76     }
77
78     virtual bool read(istream &in) {}
79
80     virtual bool write(ostream &out) const {}
81 };
82
83 class EdgeProjection :
84 public g2o::BaseBinaryEdge<2, Vector2d, VertexPoseAndIntrinsics,
85 VertexPoint> {
86 public:

```

```
86 EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
87
88 virtual void computeError() override {
89     auto v0 = (VertexPoseAndIntrinsics *) _vertices[0];
90     auto v1 = (VertexPoint *) _vertices[1];
91     auto proj = v0->project(v1->estimate());
92     _error = proj - _measurement;
93 }
94
95 // use numeric derivatives
96 virtual bool read(istream &in) {}
97
98 virtual bool write(ostream &out) const {}
99 };
```

We define the rotation, translation, focus, and distortion parameters in the same camera vertex, and then define the observation edge between the camera and the landmark. Here we do not implement the edge Jacobian calculation function, so g2o will automatically provide a numerical calculation of Jacobian. Finally, according to the data in the BAL, the g2o optimization problem can be built up:

Listing 9.5: slambook2/ch9/bundle_adjustment_g2o.cpp(fragment)

```

39 }
40
41 // edge
42 for (int i = 0; i < bal_problem.num_observations(); ++i) {
43     EdgeProjection *edge = new EdgeProjection;
44     edge->setVertex(0, vertex_pose_intrinsics[bal_problem.camera_index()[i]
45         ]);
46     edge->setVertex(1, vertex_points[bal_problem.point_index()[i]]);
47     edge->setMeasurement(Vector2d(observations[2 * i + 0], observations[2 *
48         i + 1]));
49     edge->setInformation(Matrix2d::Identity());
50     edge->setRobustKernel(new g2o::RobustKernelHuber());
51     optimizer.addEdge(edge);
52 }
53
54 optimizer.initializeOptimization();
55 optimizer.optimize(40);
56
57 // set to bal problem
58 for (int i = 0; i < bal_problem.num_cameras(); ++i) {
59     double *camera = cameras + camera_block_size * i;
60     auto vertex = vertex_pose_intrinsics[i];
61     auto estimate = vertex->estimate();
62     estimate.set_to(camera);
63 }
64 for (int i = 0; i < bal_problem.num_points(); ++i) {
65     double *point = points + point_block_size * i;
66     auto vertex = vertex_points[i];
67     for (int k = 0; k < 3; ++k) point[k] = vertex->estimate()[k];
68 }
69 }

```

The above defines the nodes and edges used in this question. Next, we need to generate some nodes and edges based on the actual data in the BALProblem class, and give them to g2o for optimization. It's worth noting that in order to take full advantage of the sparsity in BA, you need to set the setMarginalized property in the roadmap to true here. The main fragments of the code are as follows:

Listing 9.6: slambook2/ch9/bundle_adjustment_g2o(fragment)

```

1 void SolveBA(BALProblem &bal_problem) {
2     const int point_block_size = bal_problem.point_block_size();
3     const int camera_block_size = bal_problem.camera_block_size();
4     double *points = bal_problem.mutable_points();
5     double *cameras = bal_problem.mutable_cameras();
6
7     // pose dimension 9, landmark is 3
8     typedef g2o::BlockSolver<g2o::BlockSolverTraits<9, 3>> BlockSolverType;
9     typedef g2o::LinearSolverCSparse<BlockSolverType::PoseMatrixType>
10    LinearSolverType;
11    // use LM
12    auto solver = new g2o::OptimizationAlgorithmLevenberg(
13        g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>()));
14    g2o::SparseOptimizer optimizer;
15    optimizer.setAlgorithm(solver);
16    optimizer.setVerbose(true);
17
18    /// build g2o problem
19    const double *observations = bal_problem.observations();
20    // vertex
21    vector<VertexPoseAndIntrinsics *> vertex_pose_intrinsics;

```

```

22 for (int i = 0; i < bal_problem.num_cameras(); ++i) {
23     VertexPoseAndIntrinsics *v = new VertexPoseAndIntrinsics();
24     double *camera = cameras + camera_block_size * i;
25     v->setId(i);
26     v->setEstimate(PoseAndIntrinsics(camera));
27     optimizer.addVertex(v);
28     vertex_pose_intrinsics.push_back(v);
29 }
30 for (int i = 0; i < bal_problem.num_points(); ++i) {
31     VertexPoint *v = new VertexPoint();
32     double *point = points + point_block_size * i;
33     v->setId(i + bal_problem.num_cameras());
34     v->setEstimate(Vector3d(point[0], point[1], point[2]));
35     // ベクトルをg2o::Vector3dに変換する
36     v->setMarginalized(true);
37     optimizer.addVertex(v);
38     vertex_points.push_back(v);
39 }
40
41 // edge
42 for (int i = 0; i < bal_problem.num_observations(); ++i) {
43     EdgeProjection *edge = new EdgeProjection();
44     edge->setVertex(0, vertex_pose_intrinsics[bal_problem.camera_index()[i]]);
45     edge->setVertex(1, vertex_points[bal_problem.point_index()[i]]);
46     edge->setMeasurement(Vector2d(observations[2 * i + 0], observations[2 * i + 1]));
47     edge->setInformation(Matrix2d::Identity());
48     edge->setRobustKernel(new g2o::RobustKernelHuber());
49     optimizer.addEdge(edge);
50 }
51
52 optimizer.initializeOptimization();
53 optimizer.optimize(40);
54
55 // set to bal problem
56 for (int i = 0; i < bal_problem.num_cameras(); ++i) {
57     double *camera = cameras + camera_block_size * i;
58     auto vertex = vertex_pose_intrinsics[i];
59     auto estimate = vertex->estimate();
60     estimate.set_to(camera);
61 }
62 for (int i = 0; i < bal_problem.num_points(); ++i) {
63     double *point = points + point_block_size * i;
64     auto vertex = vertex_points[i];
65     for (int k = 0; k < 3; ++k) point[k] = vertex->estimate()[k];
66 }
67 }
```

One big difference between g2o and Ceres is that when using sparse optimization, g2o must manually set which vertices are marginalized vertices, otherwise a runtime error will be reported (the reader can try to comment out the line `v->setMarginalized(true)`). The rest of the place and the Ceres experiment are similar, we will not introduce more. The g2o experiment also outputs a point cloud before and after optimization for comparison.

9.5

This lecture explores the problem of state estimation and graph optimization in depth. We see that SLAM can be seen as a state estimation problem in the classical

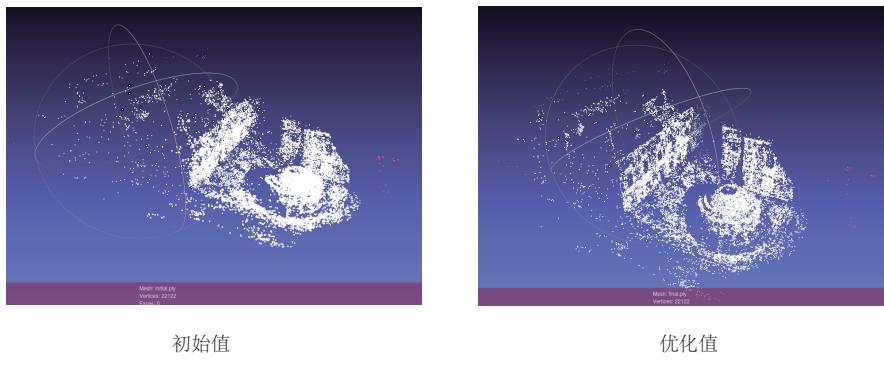


Figure 9-13: Visualization point cloud before and after optimization. The left side is the initial value before optimization, and the right side is optimized.

model. If we assume Markov properties and only consider the current state, we get a filter model represented by EKF. If not, we can also choose to consider all motions and observations, which constitute a least squares problem. In the case of only observation equations, this problem is called BA and can be solved using nonlinear optimization methods. We have carefully discussed the sparsity problem in the solution process and pointed out the connection between the problem and the graph optimization. Finally, we demonstrate how to use the g2o and Ceres libraries to solve the same optimization problem, giving the reader an intuitive understanding of BA.

Exercises

1. The proof (??) is established. Hint: You might use the SMW (Sherman-Morrison-Woodbury) formula, reference [? ?].
2. Compare the values of the optimized objective function of g2o and Ceres. Point out why the two have the same effect in Meshlab but the values are different.
3. Perform a Schur elimination on some of the point clouds in Ceres to see what the difference is.
4. proves that the S matrix is a semi-positive matrix.
5. Read the documentation [?] and see how g2o handles kernel functions. How does it relate to the Loss function in Ceres?
- 6.*In both examples, we optimized the camera pose, camera coordinates and landmarks with f, k_1, k_2 as parameters. Consider optimizing with the full camera model described in Lecture 5, ie, consider at least $f_x, f_y, p_1, p_2, k_1, k_2$. Modify the current Ceres and g2o programs to complete the experiment.

Chapter 10

backend 2

main target

1. Understand Sliding Window optimization.
2. Understand Pose Graph optimization.
3. Understands the optimization with tight coupling of the IMU.
4. Master the g2o Pose Graph and the IMU tightly coupled through experiments.

In the last lecture, we focused on the optimization of BA-based graphs. BA accurately optimizes the position and feature point of each camera. However, in larger scenes, the existence of a large number of feature points will seriously reduce the computational efficiency, resulting in an increasing amount of computation so that it cannot be real-time. The first part of this lecture introduces a simplified BA: pose diagram.

When the IMU sensor is carried on the robot, we can also calculate the relative motion between the image frames through the IMU. Similarly, the measurement data of the IMU can also be used as an optimization observation and placed in the optimization of the graph. The second part of this talk will introduce the mainstream optimization method in VIO: IMU tight coupling. This requires the reader to be proficient in the previously proposed graph optimization method.

10.1 Sliding window filtering and optimization

10.1.1 BA structure in the actual environment

The map optimization with camera pose and spatial points is called BA, which can effectively solve large-scale positioning and mapping problems. This is very useful in the SfM problem, but in the SLAM process, we often need to control the size of the BA and keep the calculations real-time. If the computing power is infinite, then it's a good idea to calculate the entire BA all the time - but that doesn't fit the reality. The reality is that we must limit the computation time of the backend, such as no more than 20 iterations, no more than 0.5 seconds, and so on. An algorithm like SfM that uses a week to rebuild a city map is not necessarily effective in SLAM.

There are many ways to control the size of the calculation, such as extracting a part of the continuous video as **keyframe**^[? 1], constructing only the BA between the key frame and the landmark point, so the non-key frame only Used for positioning and does not contribute to the construction. Even so, as time goes by, the number of key frames will increase and the map size will continue to grow. With batch optimization methods like BA, the computational efficiency will (worryly) continue to drop. In order to avoid this situation, we need to use some means to control the size of the back-end BA. These means can be theoretical or engineering.

For example, the easiest way to control the size of a BA is to keep only the nearest N keyframes from the current moment and remove the earlier keyframes. Thus, our BA will be fixed in a time window, and the one leaving the window will be discarded. This method is called the sliding window method ^[?]. Of course, there are some changes to the specific method of taking this N keyframe. For example, it may not be necessary to take the most recent time, but according to certain principles, the keyframes that are close in time and spatially expandable can be taken. To ensure that the structure of the BA does not shrink into a group even when the camera is stopped (this easily leads to some bad degradation). If we consider the structure of the frame and frame deeper, we can also define a structure called "Covisibility graph" like ORB-SLAM2^[?] (See ??). The so-called common view refers to those "keyframes that are observed together with the current camera." Therefore, in BA optimization, we optimize some key frames and landmarks in the common view according to certain principles, for example. The key frame with more than 20 common road signs with the current frame remains unchanged for the other. When the common view relationship can be correctly constructed, the optimization based on the common view will remain optimal for a longer period of time.

Sliding windows, common views, and, in general, are some of the engineering compromises we have for real-time computing. But in theory, they also introduce a new problem: we just talked about "discarding" the sliding window, or "fixing" the variables outside the common view. What is this "discard" and "fixed"? ? "Fixed" seems easy to understand, we only need to keep the keyframe estimates outside the common view unchanged. But "discarding" means completely discarding, that is, the variables outside the window have no effect on the variables in the window at all, or the data outside the window **should** have some influence, but actually we ignore it? If there is an impact, what should this impact look like? It is not obvious enough, can you ignore it?

Next we will talk about these issues. How should they be handled theoretically, and whether they can be simplified in engineering.

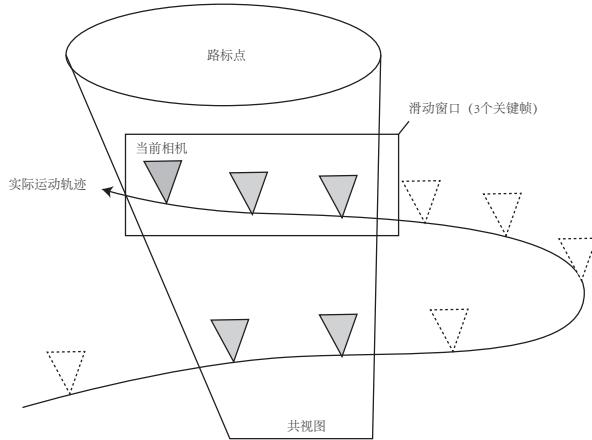


Figure 10-1: Straight view of sliding window and common view

10.1.2 Sliding window method

Now consider a sliding window. Suppose there are N keyframes in this window. Their poses are expressed as:

$$\boldsymbol{x}_1, \dots, \boldsymbol{x}_N,$$

We assume that they are in vector space, ie use Li Algebraic expression, then, what can we talk about about these keyframes?

Obviously we care about where these keyframes are located and how they are uncertain, which corresponds to their mean covariance under Gaussian distribution assumptions. If these keyframes also correspond to a partial map, we can also ask what the mean and variance of the entire local system should be. There are also M landmarks in this sliding window: $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$, which form a partial map with N keyframes. Obviously we can use the Bundle Adjustment method introduced in the previous section to deal with this sliding window, including building a graph optimization model, constructing a whole Hessian matrix, and then marginalizing all landmark points to speed up the solution. When marginalizing, we consider the pose of the keyframe, namely

$$[\boldsymbol{x}_1, \dots, \boldsymbol{x}_N]^T \sim N([\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_N]^T, \boldsymbol{\Sigma}).$$

where $\boldsymbol{\mu}_k$ is the k keyframe The pose average, $\boldsymbol{\Sigma}$ is the covariance matrix of all keyframes, then obviously, the mean part is the result after the BA iteration, and $\boldsymbol{\Sigma}$ is the *for the entire BA. The result of the marginalization of the bmmH* matrix, the matrix \boldsymbol{S} mentioned in the previous lecture. We believe that readers are already familiar with this process.

In the sliding window, our other question is, how do these state variables change when the window structure changes? This matter can be divided into two parts:

1. We need to add a keyframe to the window and the landmark points it observes.
2. We need to delete an old keyframe in the window, and we may also delete the landmark points it observes.

At this time, the difference between the sliding window method and the traditional BA is revealed. Obviously, if it is processed according to the traditional BA, then this only corresponds to two BAs of different structures, and there is no difference in the solution. But if you are sliding the window, we will discuss these specific details.

Add a keyframe and landmarks

Considering that at the last moment, the sliding window has created N keyframes, we also know that they obey a Gaussian distribution whose mean and variance are as described above. At this point, a new keyframe \mathbf{x}_{N+1} is added, and the variable in the whole question becomes a collection of $N + 1$ keyframes and more landmark points. This is actually still trivial, we just need to follow the normal BA process. When all points are marginalized, the Gaussian distribution parameters of the $N + 1$ keyframes are obtained.

Delete an old keyframe

A theoretical issue will emerge when considering the removal of old keyframes. Let's say we want to delete the old keyframe \mathbf{x}_1 , but \mathbf{x}_1 is not isolated, it will observe the same road signs as other frames. Edged \mathbf{x}_1 will cause the entire issue to be no longer sparse. As in the previous lecture, let's take a diagram, as shown by ??.

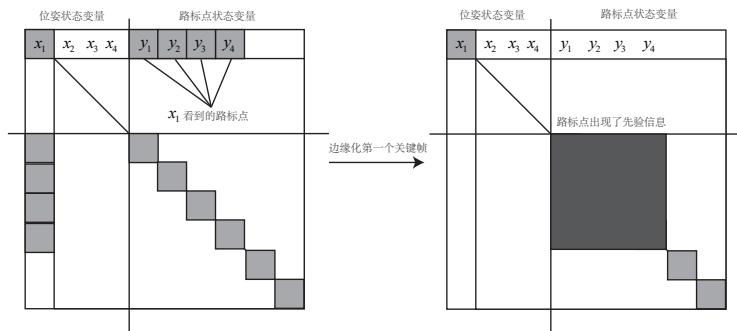


Figure 10-2: Sliding window to delete keyframes will destroy the diagonal block structure of the signpost section

In this example, we assume that \mathbf{x}_1 sees the landmarks \mathbf{y}_1 to \mathbf{y}_4 , so before processing, the Hessian matrix of the BA problem should look like Like the left side of the diagram, the \mathbf{y}_1 to \mathbf{y}_4 column in the \mathbf{x}_1 line has a non-zero block representing \mathbf{x}_1 saw them. Considering the marginalization of \mathbf{x}_1 , then the Schur elimination process is equivalent to eliminating several non-zero matrix blocks at the non-diagonal line by matrix row and column operations, which obviously leads to the landmark point matrix in the lower right corner. Blocks are no longer non-diagonal matrices. This process is called **Fill-in**[?] in marginalization.

Looking back at the marginalization of the previous lecture, when we point the way to the point, Fill-in will appear in the pose block in the upper left corner. However, because BA does not require the pose block to be a diagonal block, the sparse BA solution is still feasible. However, when the keyframe is marginalized, the diagonal block structure between the landmarks in the lower right corner will

be destroyed, and the BA will not be iteratively solved according to the previous sparse mode. This is obviously a very bad problem. In fact, in the early EKF filter backend, people did maintain a dense Hessian matrix, which made the EKF backend unable to handle larger sliding windows.

However, if we make some modifications to the marginalization process, we can also maintain the sparseness of the sliding window BA. For example, when edged an old keyframe, it simultaneously marginalizes the landmark points it observes. In this way, the information of the landmark points is converted into the common view information between the remaining key frames, thereby maintaining the diagonal block structure of the lower right corner portion. In some SLAM frameworks [? ?], the marginalization strategy is more complicated. For example, in OKVIS, we will determine which keyframe to be marginalized, and whether the landmark points it sees can still be seen in the latest keyframes. If not, the marginal point is directly marginalized; if it is, then the observation of the landmark point by the marginal keyframe is discarded, thereby maintaining the sparsity of the BA.

Intuitive interpretation of marginalization in SWF

We know that the meaning of marginalization in probability refers to conditional probability. So intuitively, when we marginalize a keyframe, that is, "keep the current estimate of this keyframe and ask for the conditional probability that other state variables are conditional on this keyframe." Therefore, when a keyframe is marginalized, the landmark points it observes will produce a priori information of "**Where should these roadmaps be**", affecting the rest of the estimates. If these landmarks are marginalized, their observers will get a priori information about "**observe where their keyframes should be**".

Mathematically, when we marginalize a keyframe, the way the state variables are described in the entire window changes from a joint distribution to a conditional probability distribution. Looking at the above example, it means:

$$p(\mathbf{x}_1, \dots, \mathbf{x}_4, \mathbf{y}_1, \dots, \mathbf{y}_6) = p(\mathbf{x}_2, \dots, \mathbf{x}_4, \mathbf{y}_1, \dots, \mathbf{y}_6 | \underbrace{\mathbf{x}_1}_{}) p(\mathbf{x}_1). \quad (10.1)$$

Then the information of the marginalized part is discarded. After the variable is marginalized, we should not use it again in the project. Therefore, the sliding window method is more suitable for VO systems, and is not suitable for systems with large-scale mapping.

Since g2o and Ceres do not directly support the marginalization operation in the sliding window method. *, we skip the corresponding experimental part of this section. It is hoped that the theoretical part will help the reader understand some SLAM systems based on sliding windows.

10.2 Pose Graph

10.2.1 The meaning of Pose Graph

Based on the previous discussion, we found that feature points account for the vast majority of optimization problems. In fact, after several observations, those

* We can bypass the g2o and Ceres framework restrictions by some clever means, but this is often very cumbersome and not suitable for the book. Demonstration.

convergent feature points, the spatial position estimate will converge to a value that remains unchanged, while the divergent outer point is usually invisible. Optimizing the convergence point seems to be somewhat unrewarding. Therefore, we prefer to fix the feature points after several optimizations, and only regard them as constraints of pose estimation, and no longer actually optimize their position estimates.

Going down this line of thought, we will think: Can you completely ignore the road signs and just follow the track? We can construct a trajectory-only graph optimization, and the edges between the pose nodes can be given initial values by motion estimation obtained after feature matching between two key frames. The difference is that once the initial estimate is complete, we no longer optimize the location of those landmarks, but only the connections between all camera poses. In this way, we save a lot of feature point optimization calculations, only retain the trajectory of key frames, thus constructing a so-called Pose Graph, such as ?? Show.

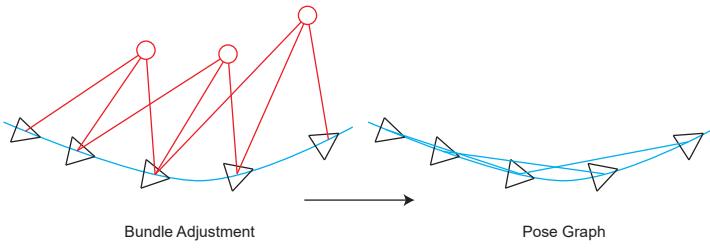


Figure 10-3: Pose Graph diagram. When we no longer optimize the landmark points in the Bundle Adjustment and only regard them as constraints on the pose nodes, we get a Pose Graph with a much smaller computational scale.

We know that the number of feature points in the BA is much larger than the pose node. A keyframe is often associated with hundreds of key points, and the maximum computational scale of real-time BA, even with sparsity, is typically around tens of thousands of points on current mainstream CPUs. This limits the SLAM application scenario. Therefore, when the robot moves in a wider range of time and space, some solutions must be considered: either discard some historical data [?] like the sliding window method; or like Pose Graph , discard the optimization of the landmark points, only keep the edges between Pose, use Pose Graph[? ? ?]. In addition, if we have additional sensors that measure Pose, then Pose Graph is also a common method of fused Pose measurements.

10.2.2 Pose Graph Optimization

So, what do the nodes and edges in the Pose Graph optimization mean? The node here represents the camera pose and is expressed as $\mathbf{T}_1, \dots, \mathbf{T}_n$. The edge is an estimate of the relative motion between the two pose nodes. The estimate may come from the feature point method or the direct method, or it may come from GPS or IMU points, but in any case, we estimate, say \mathbf{A} movement between \mathbf{T}_i and \mathbf{T}_j $\Delta\mathbf{T}_{ij}$. There are several ways to express this movement, and we take a more natural one:

$$\Delta\xi_{ij} = \xi_i^{-1} \circ \xi_j = \ln(\mathbf{T}_i^{-1}\mathbf{T}_j)^\vee, \quad (10.2)$$

Or by Li Qun's writing:

$$\mathbf{T}_{ij} = \mathbf{T}_i^{-1}\mathbf{T}_j. \quad (10.3)$$

According to the idea of graph optimization, the equation is not exactly established in practice, so we set the least squares error and then discuss the error about the derivative of the optimization variable as before. Here, we move the \mathbf{T}_{ij} of the above formula to the right side of the equation, constructing the error e_{ij} :

$$e_{ij} = \Delta\xi_{ij} \ln (\mathbf{T}_{ij}^{-1} \mathbf{T}_i^{-1} \mathbf{T}_j)^\vee \quad (10.4)$$

Note that there are two optimization variables: ξ_i and ξ_j , so we ask e_{ij} for the derivatives of these two variables. According to the Lie algebra's method of derivation, give ξ_i and ξ_j one left perturbation: $\delta\xi_i$ and $\Delta\xi_j$. Then the error becomes

$$\hat{e}_{ij} = \ln (\mathbf{T}_{ij}^{-1} \mathbf{T}_i^{-1} \exp((-\delta\xi_i)^\wedge) \exp(\delta\xi_j^\wedge) \mathbf{T}_j)^\vee. \quad (10.5)$$

In this equation, two disturbance terms are sandwiched. In order to take advantage of the BCH approximation, we want to move the perturbation term to the left or right side of the expression. Recall the adjoint nature of the fourth exercise, namely the formula (??). If you have not done this exercise, then use it as a correct conclusion for the time being:

$$\exp((\text{Ad}(\mathbf{T})\xi)^\wedge) = \mathbf{T} \exp(\xi^\wedge) \mathbf{T}^{-1}. \quad (10.6)$$

A little change, there are:

$$\exp(\xi^\wedge) \mathbf{T} = \mathbf{T} \exp\left((\text{Ad}(\mathbf{T}^{-1})\xi)^\wedge\right). \quad (10.7)$$

This formula shows that by introducing a companion item, we can "swap" \mathbf{T} on the left and right sides of the perturbation item. With it, you can move the disturbance to the far right (of course, the leftmost is also possible), and derive the Jacobian matrix of the right multiplication form (left multiplication when moving to the left):

$$\begin{aligned} \hat{e}_{ij} &= \ln (\mathbf{T}_{ij}^{-1} \mathbf{T}_i^{-1} \exp((-\delta\xi_i)^\wedge) \exp(\delta\xi_j^\wedge) \mathbf{T}_j)^\vee \\ &= \ln \left(\mathbf{T}_{ij}^{-1} \mathbf{T}_i^{-1} \mathbf{T}_j \exp\left((- \text{Ad}(\mathbf{T}_j^{-1})\delta\xi_i)^\wedge\right) \exp\left((\text{Ad}(\mathbf{T}_j^{-1})\delta\xi_j)^\wedge\right) \right)^\vee \\ &\approx \ln \left(\mathbf{T}_{ij}^{-1} \mathbf{T}_i^{-1} \mathbf{T}_j [\mathbf{I} - (\text{Ad}(\mathbf{T}_j^{-1})\delta\xi_i)^\wedge + (\text{Ad}(\mathbf{T}_j^{-1})\delta\xi_j)^\wedge] \right)^\vee. \end{aligned} \quad (10.8)$$

Therefore, according to the derivation rule on Lie algebra, we find the Jacobian matrix of the error about the two poses. About \mathbf{T}_i :

$$\frac{\partial e_{ij}}{\partial \delta\xi_i} = -\mathcal{J}_r^{-1}(e_{ij}) \text{Ad}(\mathbf{T}_j^{-1}). \quad (10.9)$$

And about \mathbf{T}_j :

$$\frac{\partial e_{ij}}{\partial \delta\xi_j} = \mathcal{J}_r^{-1}(e_{ij}) \text{Ad}(\mathbf{T}_j^{-1}). \quad (10.10)$$

If the reader feels that this part of the guide is difficult to understand, you can go back to the fourth part to review the contents of the Lie algebra. However, as I said before, since the left and right Jacobs on $\mathfrak{se}(3)$ are too complicated for \mathcal{J}_r , we

usually take their approximation. If the error is close to zero, we can set them to approximate \mathbf{I} or

$$\mathcal{J}_r^{-1}(\mathbf{e}_{ij}) \approx \mathbf{I} + \frac{1}{2} \begin{bmatrix} \phi_e^\wedge & \rho_e^\wedge \\ \mathbf{0} & \phi_e^\wedge \end{bmatrix}. \quad (10.11)$$

In theory, even after optimization, because the observation data given by each edge is not consistent, the error is usually not similar to zero, so simply set \mathcal{J}_r here to \mathbf{I} will have a certain loss. Later we will see through practice whether the theoretical difference is obvious.

After learning about Jacobi's derivation, the rest is the same as the normal graph optimization. In short, all pose vertices and poses - poses constitute a graph optimization, essentially a least squares problem, the optimization variable is the pose of each vertex, and the edges are derived from pose observation constraints. Remember that \mathcal{E} is a collection of all edges, then the overall objective function is

$$\min \frac{1}{2} \sum_{i,j \in \mathcal{E}} \mathbf{e}_{ij}^T \boldsymbol{\Sigma}_{ij}^{-1} \mathbf{e}_{ij}. \quad (10.12)$$

We can still solve this problem by Gauss-Newton method, Levinburg-Marquart method, etc., except that Lie algebra is used to represent the optimized pose, everything else is similar. Based on previous experience, this can naturally be solved with Ceres or g2o. We are no longer discussing the detailed process of optimization. The last lecture has already been said enough.

10.3 Practice: Pose Optimization

10.3.1 g2o native pose

The following demonstrates the use of g2o for pose optimization. First, the reader is asked to open our pre-generated simulation pose with `g2o_viewer`, located in `slambook2/ch10/sphere.g2o`, as shown in ?? .

The pose is generated by g2o's own create sphere program simulation. Its true trajectory is a ball, consisting of multiple layers from bottom to top. Each layer is a perfect circle, and many circular layers of different sizes form a complete sphere, which contains 2500 pose nodes (?? top left), which can be seen as a circle. The process of ascending. The emulator then generates the edge of $t - 1$ to t , called the odometry edge (odometer). In addition, the edge between the layer and the layer is generated, which is called loop closure (loopback, loopback detection algorithm will be described in detail in the next lecture). Then, add observation noise on each side and reset the initial value of the node based on the noise of the odometer side. In this way, the pose data with cumulative error (?? bottom right) is obtained. It looks like a part of the sphere, but the overall shape is far from the sphere. Now we start from these noisy edges and node initial values, try to optimize the entire pose map to get the data of approximate truth value.

Of course, the actual robot will certainly not have such a spherical motion trajectory, and such complete odometer and loop observation data. The advantage of simulating a positive sphere is that we can visually see if the optimization result is correct (just look at it and the angles are not round). The reader can click on the optimize function in `g2o_viewer` to see the optimization results and the convergence process for each step. On the other hand, `sphere.g2o` is also a text file that can be

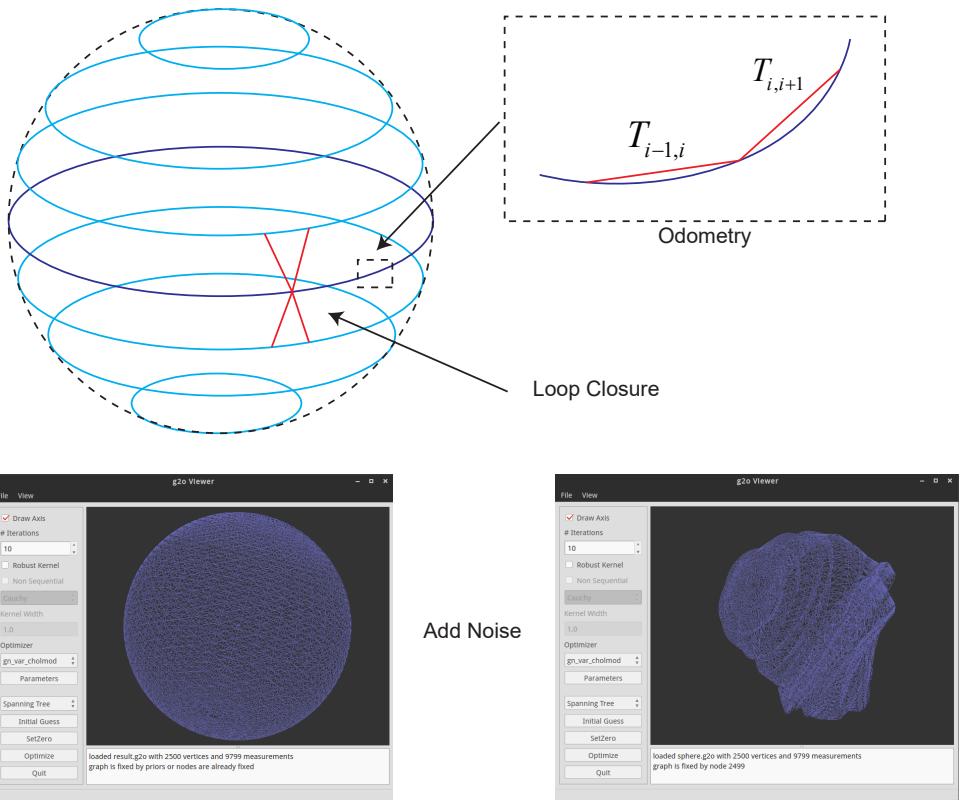


Figure 10-4: g2o The pose map generated by the simulation. The true value is a complete sphere, and the simulation data with cumulative error is obtained by adding noise to the true value.

opened with a text editor to see what is inside it. The first half of the file consists of nodes, and the second half is edges:

```

1 VERTEX_SE3:QUAT 0 -0.125664 -1.53894e-17 99.9999 0.706662 4.32706e-17
  0.707551 -4.3325e-17
2 .....
3 EDGE_SE3:QUAT 1524 1574 -0.210399 -0.0101193 -6.28806 -0.00122939 0.0375067
  -2.85291e-05 0.999296 10000 0 0 0 0 10000 0 0 0 10000 0 0 0 40000
  0 0 40000 0 40000

```

As you can see, the node type is VERTEX_SE3, which expresses a camera pose. By default, g2o uses quaternions and translation vectors to express poses, so the following fields mean: ID, $t_x, t_y, t_z, q_x, q_y, q_z, q_w$. The first three are translation vector elements, and the last four are unit quaternions representing rotation. Similarly, the information of the edge is the ID of two nodes, $t_x, t_y, t_z, q_x, q_y, q_z, q_w$, the upper right corner of the information matrix (since the information matrix is a symmetric matrix, only half of it can be saved). It can be seen that the information matrix is set to a diagonal matrix.

To optimize this pose, we can use g2o's default vertices and edges, which are represented by quaternions. Since the simulation data is also generated by g2o, optimization with g2o itself does not require us to do more work, just configure the optimization parameters. The program slambook2/ch10/pose_graph_g2o_SE3.cpp demonstrates how to optimize the pose using the Levenberg-Marquart method and store the result in the result.g2o file.

Listing 10.1: slambook2/ch10/pose_graph_g2o_SE3.cpp

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4
5 #include <g2o/types/slam3d/types_slam3d.h>
6 #include <g2o/core/block_solver.h>
7 #include <g2o/core/optimization_algorithm_levenberg.h>
8 #include <g2o/solvers/eigen/linear_solver_eigen.h>
9
10 using namespace std;
11
12 //*****
13 * This program demonstrates how to optimize poses with g2o solver
14 * sphere.g2o is a manually generated Pose graph, let's optimize it.
15 * Although we can read the entire graph directly through the load function,
16 * we still implement the read code ourselves in order to gain a deeper
17 * understanding.
18 * ****
19 int main(int argc, char **argv) {
20     if (argc != 2) {
21         cout << "Usage: pose_graph_g2o_SE3 sphere.g2o" << endl;
22         return 1;
23     }
24     ifstream fin(argv[1]);
25     if (!fin) {
26         cout << "file " << argv[1] << " does not exist." << endl;
27         return 1;
28     }
29
30     // [] g2o

```

```

31     typedef g2o::BlockSolver<g2o::BlockSolverTraits<6, 6>> BlockSolverType;
32     typedef g2o::LinearSolverEigen<BlockSolverType::PoseMatrixType>
33         LinearSolverType;
34     auto solver = new g2o::OptimizationAlgorithmLevenberg(
35         g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>())
36             );
37     g2o::SparseOptimizer optimizer;      // 0 0 0
38     optimizer.setAlgorithm(solver);    // 0 0 0 0 0
39     optimizer.setVerbose(true);        // 0 0 0 0 0 0
40
41     int vertexCnt = 0, edgeCnt = 0; // 0 0 0 0 0 0 0
42     while (!fin.eof()) {
43         string name;
44         fin >> name;
45         if (name == "VERTEX_SE3:QUAT") {
46             // SE3 0 0
47             g2o::VertexSE3 *v = new g2o::VertexSE3();
48             int index = 0;
49             fin >> index;
50             v->setId(index);
51             v->read(fin);
52             optimizer.addVertex(v);
53             vertexCnt++;
54             if (index == 0)
55                 v->setFixed(true);
56         } else if (name == "EDGE_SE3:QUAT") {
57             // SE3-SE3 0
58             g2o::EdgeSE3 *e = new g2o::EdgeSE3();
59             int idx1, idx2; // 0 0 0 0 0 0
60             fin >> idx1 >> idx2;
61             e->setId(edgeCnt++);
62             e->setVertex(0, optimizer.vertices()[idx1]);
63             e->setVertex(1, optimizer.vertices()[idx2]);
64             e->read(fin);
65             optimizer.addEdge(e);
66         }
67         if (!fin.good()) break;
68     }
69
70     cout << "read total " << vertexCnt << " vertices, " << edgeCnt << "
71         edges." << endl;
72
73     cout << "optimizing ..." << endl;
74     optimizer.initializeOptimization();
75     optimizer.optimize(30);
76
77     cout << "saving optimization results ..." << endl;
78     optimizer.save("result.g2o");
79
80     return 0;
81 }
```

We chose the block solver for 6×6 , using the Levinberg-Marquart drop method, and selecting the number of iterations 30 times. Call this program to optimize the pose map:

Listing 10.2: terminal input:

```

1 $ build/pose_graph_g2o_SE3 sphere.g2o
2 Read total 2500 vertices, 9799 edges.
3 Optimizing ...
4 Iteration= 0 chi2= 1023011093.851879 edges= 9799 schur= 0 lambda=
805.622433 levenbergIter= 1
```

```

5 Iteration= 1 chi2= 385118688.233188 time= 0.863567 cumTime= 1.71545 edges=
  9799 schur= 0 lambda= 537.081622 levenbergIter= 1
6 Iteration= 2 chi2= 166223726.693659 time= 0.861235 cumTime= 2.57668 edges=
  9799 schur= 0 lambda= 358.054415 levenbergIter= 1
7 Iteration= 3 chi2= 86610874.269316 time= 0.844105 cumTime= 3.42079 edges=
  9799 schur= 0 lambda= 238.702943 levenbergIter= 1
8 Iteration= 4 chi2= 40582782.710190 time= 0.862221 cumTime= 4.28301 edges=
  9799 schur= 0 lambda= 159.135295 levenbergIter= 1
9 .....
10 Iteration= 28 chi2= 45095.174398 time= 0.869451 cumTime= 30.0809 edges=
  9799 schur= 0 lambda= 0.003127 levenbergIter= 1
11 Iteration= 29 chi2= 44811.248504 time= 1.76326 cumTime= 31.8442 edges= 9799
  schur= 0 lambda= 0.003785 levenbergIter= 2
12 Saving optimization results ...

```

Then, open the result.g2o with g2o_viewer to see the results, as shown by ?? .

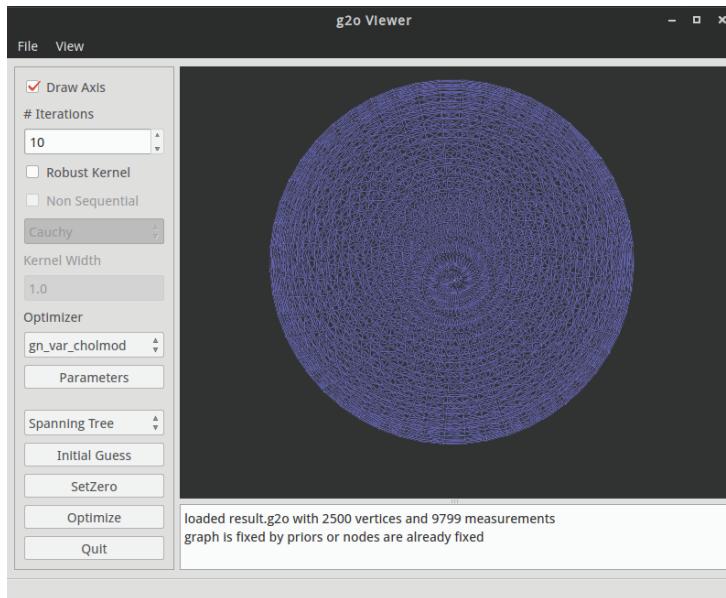


Figure 10-5: Use the result of the vertex and edge solution that comes with g2o.

The result is optimized from an irregular shape into a seemingly complete ball. This process is essentially the same as the one we clicked on the Optimize button on g2o_viewer. Below, we use the previous Lie algebra to derive the optimization on the Lie algebra.

10.3.2 Pose map optimization on Lie algebra

Remember when we used Sophus to express Lie algebra? Let's try to use Sophus to define its own vertices and edges in g2o.

Listing 10.3: slambook2/ch10/pose_graph_g2o_lie_algebra.cpp

```

1  typedef Matrix<double, 6, 6> Matrix6d;
2
3 // Approximate error for J_R^{-1}
4 Matrix6d JRIInv(const SE3d &e) {
5     Matrix6d J;
6     J.block(0, 0, 3, 3) = S03d::hat(e.so3().log());
7     J.block(0, 3, 3, 3) = S03d::hat(e.translation());
8     J.block(3, 0, 3, 3) = Matrix3d::Zero(3, 3);
9     J.block(3, 3, 3, 3) = S03d::hat(e.so3().log());
10    J = J * 0.5 + Matrix6d::Identity();
11    return J;
12 }
13
14 // Lie algebra vertices
15 Typedef Matrix<double, 6, 1> Vector6d;
16
17 class VertexSE3LieAlgebra : public g2o::BaseVertex<6, SE3d> {
18     public:
19     EIGEN_MAKE_ALIGNED_OPERATOR_NEW
20
21     virtual bool read(istream &is) override {
22         double data[7];
23         for (int i = 0; i < 7; i++)
24             is >> data[i];
25         setEstimate(SE3d(
26             Quaternionnd(data[6], data[3], data[4], data[5]),
27             Vector3d(data[0], data[1], data[2])
28         ));
29     }
30
31     virtual bool write(ostream &os) const override {
32         os << id() << " ";
33         Quaternionnd q = _estimate.unit_quaternion();
34         os << _estimate.translation().transpose() << " ";
35         os << q.coeffs()[0] << " " << q.coeffs()[1] << " " << q.coeffs()[2]
36             << " " << q.coeffs()[3] << endl;
37         return true;
38     }
39
40     virtual void setToOriginImpl() override {
41         _estimate = SE3d();
42     }
43
44     // left multiply update
45     Virtual void oplusImpl(const double *update) override {
46         Vector6d upd;
47         upd << update[0], update[1], update[2], update[3], update[4],
48             update[5];
49         _estimate = SE3d::exp(upd) * _estimate;
50     };
51
52 // sides of two Lie algebra nodes
53 Class EdgeSE3LieAlgebra : public g2o::BaseBinaryEdge<6, SE3d,
54     VertexSE3LieAlgebra, VertexSE3LieAlgebra> {
55     Public:
56     EIGEN_MAKE_ALIGNED_OPERATOR_NEW
57
58     virtual bool read(istream &is) override {
59         double data[7];
60         for (int i = 0; i < 7; i++)
61     }

```

```

59     is >> data[i];
60     Quaternionnd q(data[6], data[3], data[4], data[5]);
61     q.normalize();
62     setMeasurement(SE3d(q, Vector3d(data[0], data[1], data[2])));
63     for (int i = 0; i < information().rows() && is.good(); i++)
64     for (int j = i; j < information().cols() && is.good(); j++) {
65         is >> information()(i, j);
66         if (i != j)
67             information()(j, i) = information()(i, j);
68     }
69     return true;
70 }
71
72 virtual bool write(ostream &os) const override {
73     VertexSE3LieAlgebra *v1 = static_cast<VertexSE3LieAlgebra *>(
74         _vertices[0]);
75     VertexSE3LieAlgebra *v2 = static_cast<VertexSE3LieAlgebra *>(
76         _vertices[1]);
77     os << v1->id() << " " << v2->id() << " ";
78     SE3d m = _measurement;
79     Eigen::Quaternionnd q = m.unit_quaternion();
80     os << m.translation().transpose() << " ";
81     os << q.coeffs()[0] << " " << q.coeffs()[1] << " " << q.coeffs()[2]
82         << " " << q.coeffs()[3] << " ";
83
84     // information matrix
85     for (int i = 0; i < information().rows(); i++)
86     for (int j = i; j < information().cols(); j++) {
87         os << information()(i, j) << " ";
88     }
89     os << endl;
90     return true;
91 }
92
93 // The error calculation is consistent with the derivation in the book
94 Virtual void computeError() override {
95     SE3d v1 = (static_cast<VertexSE3LieAlgebra *>(_vertices[0]))->
96         estimate();
97     SE3d v2 = (static_cast<VertexSE3LieAlgebra *>(_vertices[1]))->
98         estimate();
99     _error = (_measurement.inverse() * v1.inverse() * v2).log();
100 }
101
102 // Jacobi calculation
103 Virtual void linearizeOplus() override {
104     SE3d v1 = (static_cast<VertexSE3LieAlgebra *>(_vertices[0]))->
105         estimate();
106     SE3d v2 = (static_cast<VertexSE3LieAlgebra *>(_vertices[1]))->
107         estimate();
108     Matrix6d J = JRIInv(SE3d::exp(_error));
109     // Try to approximate J to I?
110     -jacobianOplusXi = -J * v2.inverse().Adj();
111     -jacobianOplusXj = J * v2.inverse().Adj();
112 }
113
114 };

```

In order to store and read g2o files, this section implements the read and write functions and "disguises" into g2o's built-in SE3 vertices, enabling g2o_viewer to recognize and render it. In fact, apart from the internal representation of Sophus's Lie algebra, there is no difference from the outside.

It is worth noting the calculation process of Jacobi here. We have several options: First, do not provide the Jacobian calculation function, let g2o automatically

calculate the value of Jacobi. The second is to provide a complete or approximate Jacobian calculation process. Here we use the JRIInv() function to provide an approximate \mathcal{J}_r^{-1} . The reader can try to approximate it as I , or simply comment out the oplusImpl function and see what the difference is.

Then call g2o to optimize the problem:

Listing 10.4: terminal input:

```

1 $ build/pose_graph_g2o_lie sphere.g2o
2 Read total 2500 vertices, 9799 edges.
3 Optimizing ...
4 Iteration= 0 chi2= 626657736.014949 time= 0.549125 cumTime= 0.549125 edges=
   9799 schur= 0 lambda= 6706.585223 levenbergIter= 1
5 Iteration= 1 chi2= 233236853.521434 time= 0.510685 cumTime= 1.05981 edges=
   9799 schur= 0 lambda= 2235.528408 levenbergIter= 1
6 Iteration= 2 chi2= 142629876.750105 time= 0.557893 cumTime= 1.6177 edges=
   9799 schur= 0 lambda= 745.176136 levenbergIter= 1
7 Iteration= 3 chi2= 84218288.615592 time= 0.525079 cumTime= 2.14278 edges=
   9799 schur= 0 lambda= 248.392045 levenbergIter= 1
8 .....

```

We found that after 23 iterations, the overall error remains the same, in fact the optimization algorithm can be stopped. In the previous experiment, the error was still declining after using 30 iterations. Please note that although the numerical error here is larger, we have redefine the calculation of the error because we customize the edge. So the size of the value here is not directly used for comparison. After calling the optimization, look at result_lie.g2o to see its results, as shown by ?? . No difference can be seen from the naked eye.

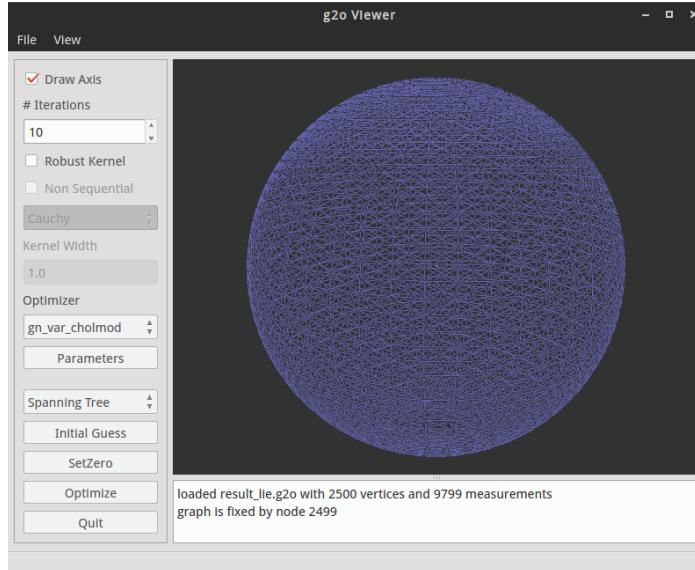


Figure 10-6: Use Lie algebra to customize the result of node and edge optimization.

If you press the Optimize button on this g2o_viewer interface, g2o will use its own SE3 vertices for optimization, you can see in the text box below:

```

1 Loaded result_lie.g2o with 2500 vertices and 9799 measurements
2 Graph is fixed by node 2499

```

```

3 # Using CHOLMOD poseDim -1 landMarkDim -1 blockOrdering 0
4 Preparing (no marginalization of Landmarks)
5 iteration= 0 chi2= 44360.509723 time= 0.567504 cumTime= 0.567504 edges=
   9799 schur= 0
6 iteration= 1 chi2= 44360.471110 time= 0.595993 cumTime= 1.1635   edges=
   9799 schur= 0
7 iteration= 2 chi2= 44360.471110 time= 0.582909 cumTime= 1.74641   edges=
   9799 schur= 0

```

The overall error is 44360 at the SE3 side, slightly less than the 44811 at the previous 30 iterations. This shows that after optimizing with Lie algebra, we get better results with fewer iterations*. In fact, even if we use the unit matrix to approximate \mathcal{J}_r^{-1} , you will converge to a similar value. This is mainly because when the error is close to zero, Jacobi is very close to the identity matrix.

10.3.3

The example of the ball is a more representative case. It has an Odometry and a Loop Closure similar to the actual one, which is what is possible in a pose in the actual SLAM. At the same time, the "ball" also has a certain scale of calculation: it has a total of 2,500 pose nodes and nearly 10,000 edges, and we found that it took a lot of time to optimize it (relative to the front end with strong real-time requirements). On the other hand, the pose map is generally considered to be one of the simplest structures. Under the premise of not assuming how the robot moves, it is difficult to further discuss its sparsity - because the robot may move straight forward, forming a band-like pose, which is sparse; it may also be "left-handed and right-handed A slow motion", the formation of a large number of small loops requires optimization (Loopy motion), which becomes a more dense pose of the "ball". In any case, before we have any further information, we can no longer use the solution structure of the pose map.

Since the introduction of PTAM^[?], people have realized that the optimization of the backend does not need to respond to the image data of the front end in real time. People tend to separate the front end and the back end, running in two separate threads, historically called Tracking and Mapping - although so, the mapping part mainly refers to the back-end optimization content. In layman's terms, the front end needs to respond to the video in real time, for example 30 frames per second; and optimization can be run slowly, as long as the results are returned to the front end when optimization is complete. So we usually don't put high speed requirements on backend optimization.

Exercises

1. If the error in the pose is defined as $\Delta\xi_{ij} = \xi_i \circ \xi_j^{-1}$, derivation The left-multiplying Jacobian matrix according to this definition.
2. If you use a right multiply update, derive the Jacobian matrix in this case.
3. Refer to the g2o program to optimize the "ball" pose in Ceres.
4. sorts the information in the "ball" by time, feeds them to g2o and gtsam, and compares their performance differences.

* Since there are no more experiments, this conclusion is only valid in the example of "ball".

5.* Read the iSAM related paper and understand how it implements incremental optimization.

Chapter 11

loopback detection

main target

1. Understand the need for loopback detection.
2. Master the appearance-based loopback detection based on the bag of words.
3. Through the experiments of DBoW3, learn the practical use of the bag of words model.

In this lecture, we introduce another major module in SLAM: loopback detection. We know that the main purpose of the SLAM main body (front-end, back-end) is to estimate camera movement, and the loop detection module, whether it is on the target or the method, is significantly different from the content described above, so it is generally considered as an independent module . We will introduce the way of detecting loops in mainstream visual SLAM: bag-of-words model, and through the program experiments on the DBoW library, make readers understand more intuitively.



11.1 Loop Detection Overview

11.1.1 Meaning of loopback detection

We have already introduced the front-end and back-end: the front-end provides feature point extraction and trajectory, the initial value of the map, and the back-end is responsible for optimizing all of this data. However, if only key frames in adjacent time are considered like VO, then the errors generated before will inevitably accumulate to the next moment, so that the whole SLAM will appear **cumulative error**, and the result of long-term estimation will not Reliably, or rather, we can't build **globally consistent** tracks and maps.

To give a simple example: In the mapping phase of automatic driving, we usually specify the collection vehicle to make a number of turns in a given area to cover all the collection range. Suppose we extract features on the front end, then ignore the feature points, and use Pose Graph on the back end to optimize the entire trajectory, as shown in ?? (a) . Because the front end only gives local constraints between poses, for example, it may be $x_1 - x_2, x_2 - x_3$, and so on. However, due to errors in the estimation of x_1 , and x_2 is determined based on x_1 , x_3 is again determined by x_2 . By analogy, the errors will be accumulated, so that the results of the back-end optimization are slowly inaccurate as shown in ?? (b). In this application scenario, we should use loopback information to determine that the same data should be collected each time it passes the same place.

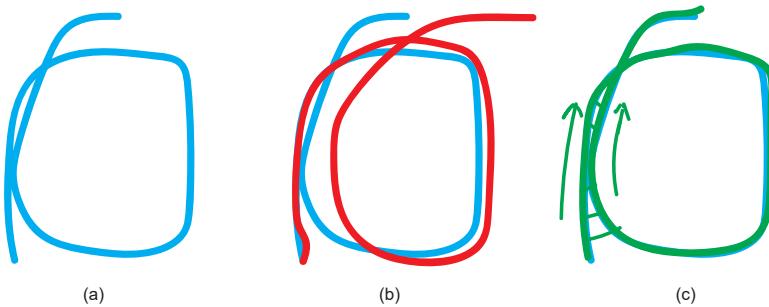


Figure 11-1: Drift illustration. (A) True trajectory; (b) Because the front end only gives estimates between adjacent frames, the optimized Pose Graph drifts; (c) Adding loop detection Pose Graph can eliminate the cumulative error.

Although the back end can estimate the maximum a posteriori error, the so-called "good model cannot hold bad data". When there is only adjacent key frame data, we can do a lot of things and we cannot eliminate the accumulated error. However, the loopback detection module can give some constraints on **more time apart** besides adjacent frames: for example, the bits between $x_1 - x_{100}$ Posture change. Why are there constraints between them? This is because we noticed that the camera **passed the same place** and **collected similar data**. The key to loop detection is how to effectively detect **the camera passes through the same place**. If we can successfully detect this, we can provide more valid data for the back-end Pose Graph, so that it can get a better estimate, especially a **globally consistent** estimate. Since Pose Graph can be regarded as a mass-spring system, loop detection is equivalent to adding an additional spring to the image, which improves the stability of the system. The reader can also intuitively imagine that

the loopback edge "pulls" the edge with accumulated errors to the correct position-if the loopback itself is correct.

Loopback detection is significant for SLAM systems. It is related to the accuracy of our estimated trajectory and map at **under long time**. On the other hand, since loopback detection provides the association of current data with all historical data, we can also use loopback detection for **relocation**. Relocation is more useful. For example, if we recorded a trajectory and built a map in advance for a scene, then we can always follow this trajectory to navigate in that scene, and relocation can help us determine our position on this trajectory. Therefore, the improvement of the accuracy and robustness of the entire SLAM system by loopback detection is very obvious. Even sometimes, we call a system with only front-end and local back-end as VO, and a system with loopback detection and global back-end as SLAM.

11.1.2 method

Let us consider how to implement loopback detection. In fact, there are several different ways to look at this problem, including theoretical and engineering.

The easiest way is to do a feature match on any two images and determine which two images are related according to the number of correct matches-this is indeed a simple and effective idea. The disadvantage is that we blindly assume that "any two images may have loops", so that the number to be detected is too large: for N possible loops, we want to detect C_N^2 as many times, this The complexity is $O(N^2)$, which grows too fast as the trajectory becomes longer, which is not practical in most real-time systems. Another simple way is to randomly extract historical data and perform loopback detection. For example, randomly select 5 frames among the n frames to compare with the current frame. This method can maintain a constant amount of computing time, but when this blind heuristic method increases the number of frames N , the probability of drawing loops decreases significantly, making the detection efficiency not high.

The simple ideas mentioned above are too rough. Although random detection is indeed useful in some implementations [?], we at least hope that there is a "**where loopbacks can occur**" prediction, so that it can be detected less blindly. This approach is broadly divided into two ideas: Odometry based, or Appearance based. Based on the geometric relationship, it means that when we find that the current camera moves near a certain position before, check whether they have a loopback relationship [?]-this is naturally an intuitive idea, but due to accumulation The existence of the error, we often can not correctly find the fact that "moved to a certain location before", the loop detection is also impossible to talk about. Therefore, this approach is logically problematic because the goal of loop detection is to discover the fact that the camera is back to its previous position, thereby eliminating the cumulative error. The geometry-based approach assumes that the "camera returns to near its previous position" so that loops can be detected. This is suspected to have a negative effect, so it cannot work when the cumulative error is large [?].

Another method is based on appearance. It has nothing to do with the estimation of the front end and the back end, and only determines the loop detection relationship based on the similarity of the two images. This approach gets rid of the accumulated error and makes the loopback detection module a relatively independent module in the SLAM system (of course, the front end can provide it with characteristic points). Since it was proposed at the beginning of the 21st century, the appearance-based loop detection method can effectively work in different scenarios, has become the

mainstream method in visual SLAM, and has been applied to actual systems. , [?].

In addition, from the engineering perspective, we can also propose some solutions to loop detection. For example, outdoor unmanned vehicles are often equipped with GPS, which can provide global location information. GPS information can be used to easily determine whether a car has returned to a certain passing point, but such methods are not very useful indoors.

In the appearance-based loop detection algorithm, the core problem is **how to calculate the similarity between images**. For example, for image \mathbf{A} and image \mathbf{B} , we will design a method to calculate the similarity score between them: $s(\mathbf{A}, \mathbf{B})$. Of course, this score will take a value within a certain interval. When it is greater than a certain amount, we think that a loop occurs. Readers may have questions: Is it difficult to calculate the similarity between two images? For example, intuitively, images can be represented as matrices, so what if we just subtract two images and then take some norm?

$$s(\mathbf{A}, \mathbf{B}) = \|\mathbf{A} - \mathbf{B}\|. \quad (11.1)$$

Why don't we do this?

1. First of all, as mentioned earlier, pixel grayscale is an unstable measurement that is heavily affected by ambient light and camera exposure. Assuming the camera is not moving and we turn on a light, the image will be brighter overall. In this way, even for the same data, we will get a large difference.
2. On the other hand, when the camera's viewing angle changes slightly, even if the brightness of each object does not change, their pixels will shift in the image, causing a large difference.

Due to the existence of these two situations, in practice, even for very similar images, $\mathbf{A} - \mathbf{B}$ will often get a (not realistic) large value. So we say that this function **cannot reflect the similarity relationship between images well**. This involves a definition of "good" and "bad". We have to ask, what kind of function can better reflect the similarity relationship, and what kind of function is not good enough? From here, two concepts, **Perceptual Aliasing** and **Perceptual Variability** can be derived. Now let's discuss it in more detail.

11.1.3 Accuracy and recall

From a human perspective, (at least we think of it) we are able to perceive the fact that "the two images are similar" or "the two photos were taken from the same place" with high accuracy, but due to the working principle of the human brain is not yet known, and we cannot clearly describe how we accomplish this. From a procedural point of view, we hope that procedural algorithms can make judgments that are consistent with humans or with facts. When we think, or in fact, that two images are taken from the same place, then the loop detection algorithm should also give a "this is a loop" result. Conversely, if we feel, or in fact, that the two images were taken from different places, the program should also give a "this is not a loopback" judgment. * Of course, the judgment of the program is not always consistent with our human thoughts, so there may be 4 cases in ?? :

Table 11-1: Result classification of loopback detection

Algorithm \ fact	is loopback	is not loopback
Yes Loopback	True Positive	False Positive
Not loopback	False Negative	True Negative

The negative/positive argument here is borrowed from the medical term. False Positive is also called Perceptual Bias, and False Negative is called Perceptual Variation (see ??). For the convenience of writing, the abbreviation TP stands for True Positive, and the rest is analogized. Because we want the algorithm to be consistent with human judgments, we want TP and TN to be as high as possible, and FP and FN to be as low as possible. Therefore, for a certain algorithm, we can count

* Readers with a background in machine learning should appreciate how similar this passage is to machine learning. Are you already thinking about how to train the network?

the occurrences of TP, TN, FP, and FN on a certain data set, and calculate two statistics: **accuracy** and **recall rate** (Precision & Recall).

$$\text{Precision} = \text{TP}/(\text{TP} + \text{FP}), \quad \text{Recall} = \text{TP}/(\text{TP} + \text{FN}). \quad (11.2)$$



Figure 11-2: Examples of false positives and false negatives. The left side is a false positive, the two images look similar, but not the same corridor; the right side is a false negative. Due to changes in lighting, photos at different times in the same place look very different.

In the literal sense of the formula, the accuracy rate describes the probability that all loops extracted by the algorithm are indeed true loops. The recall rate is the probability of being correctly detected in all real loops. Why take these two statistics? Because they are somewhat representative and usually a pair of **contradictions**.

An algorithm often has many setting parameters. For example, when a certain threshold is raised, the algorithm may become more "strict"—it detects fewer loops and improves accuracy. But at the same time, because the number of detections has decreased, many places that were originally loopbacks may be missed, resulting in a decline in the recall rate. Conversely, if we choose a more relaxed configuration, the number of detected loops will increase and a higher recall rate will be obtained, but there may be some cases that are not loopbacks, and the accuracy rate will decrease.

In order to evaluate the quality of the algorithm, we will test its *P* and *R* values in various configurations, and then make a Precision-Recall curve (see ??). When the recall is used on the horizontal axis and the accuracy is used on the vertical axis, we will care about the degree of the entire curve deviating to the upper right, the recall rate at 100 % accuracy rate, or the accuracy rate at 50 % recall rate, as the evaluation indicator of the algorithm. Note, however, that apart from some "world-wide" algorithms, we cannot generally say that Algorithm A is better than Algorithm B. We may say that A has a good recall rate when the accuracy rate is high, and B can guarantee a better accuracy rate with a 70

It is worth mentioning that in SLAM, we have higher requirements for accuracy, and we are relatively tolerant of recall. Because false positive (the test result is but not actually) loops will add fundamentally wrong edges to the backend Pose Graph, sometimes it will cause the optimization algorithm to give completely wrong results. Imagine if the SLAM program mistakenly regarded all the desks as the same, what would happen to the created graph? You may see that the corridors are not straight, the walls are intertwined, and the entire map is broken. In contrast, the recall rate is lower, at most part of the loopbacks are not detected, and the map may be affected by some cumulative errors—but only one or two loopbacks can completely eliminate them. Therefore, when choosing a loop detection algorithm, we prefer to set the parameters more strictly, or add the **loop verification** step after detection.

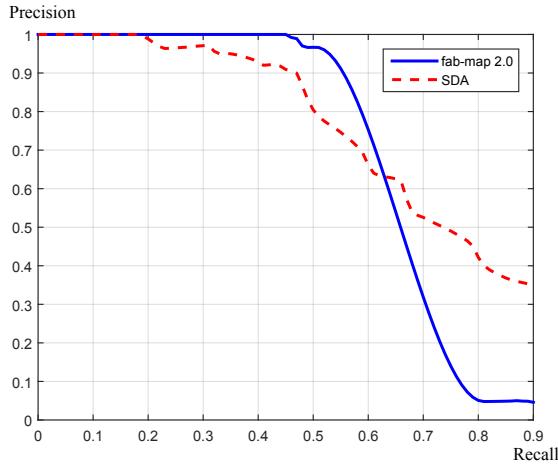


Figure 11-3: Example of accuracy-recall curve [?]. As the recall rate increases, the detection conditions become relaxed, and the accuracy rate decreases. Good algorithms can still guarantee better accuracy at higher recalls.

So, back to the previous question, why not use $\mathbf{A} - \mathbf{B}$ to calculate similarity? We will find that its accuracy and recall are very poor, and there may be a lot of False Positive or False Negative situations, so it is "bad" to do so. So, which method is better?

11.2 Word bag model

Since the method of subtracting two images directly is not good enough, we need a more reliable method. Combining the content of the previous lectures, an intuitive idea is: why not use feature points for loop detection like VO? As with VO, we match the feature points of the two images. As long as the number of matches is greater than a certain value, a loop is considered to have occurred. Further, based on the feature point matching, we can also calculate the motion relationship between the two images. Of course, there are some problems with this approach. For example, feature matching can be time-consuming, and feature descriptions may be unstable when the lighting changes. Let's talk about the bag of words first, and then discuss the implementation details such as data structures.

Bags of words, also known as Bag-of-Words (BoW), are designed to describe an image by "how many features are there in the image". For example, let's say there is one person, a car in one photo, and two people, a dog in another. According to this description, the similarity of the two images can be measured. To be more specific, we need to do the following:

1. Determines the concepts of "person", "car", and "dog" —corresponding to "**Word**" (Word) in BoW. Many words are put together to form "**Dictionary**".
2. Determines what dictionary-defined concepts appear in an image—we describe the entire image in terms of word occurrences (or histograms). This transforms an image into a vector description.

3. Compares the similarity described in the previous step.

In the example above, we first got a "dictionary" somehow. There are many words recorded in the dictionary, and each word has a certain meaning. For example, "person", "car" and "dog" are words recorded in the dictionary. We might as well write them as w_1, w_2, w_3 . Then, for any image A, based on the words they contain, it can be written as

$$A = 1 \cdot w_1 + 1 \cdot w_2 + 0 \cdot w_3. \quad (11.3)$$

The dictionary is fixed, so as long as the vector $[1, 1, 0]^T$ can express the meaning of A. With a dictionary and words, only one vector is needed to describe the entire image. This vector describes the information of "whether the image contains certain features", which is more stable than the simple gray value. And because the description vector says "**whether it appears**", regardless of their "**where it appears**", it has nothing to do with the spatial position and order of the objects. Therefore, when the camera undergoes a small amount of motion, as long as the Still appearing in the field of view, we still guarantee that the description vector will not change *. Based on this feature, we call it Bag-of-Words rather than List-of-Words, and emphasize the presence or absence of Words, regardless of their order. Therefore, it can be said that a dictionary is similar to a collection of words.

Going back to the example above, for the same reason, $[2, 0, 1]^T$ can describe the image B. If you only consider "whether it appears" without considering the quantity, it can also be $[1, 0, 1]^T$. At this time, this vector is binary. Therefore, based on these two vectors, a certain calculation method is designed to determine the similarity between the images. Of course, if there are still some different ways to calculate the difference between two vectors, such as $\mathbf{a}, \mathbf{b} \in \mathbb{R}^W$, you can calculate:

$$s(\mathbf{a}, \mathbf{b}) = 1 - \frac{1}{W} \|\mathbf{a} - \mathbf{b}\|_1. \quad (11.4)$$

The norm is L_1 norm, which is the sum of the absolute values of the elements. Note that when the two vectors are exactly the same, we will get 1; when they are exactly the opposite (where a is 0, b is 1), we get 0. This defines the similarity between the two description vectors, and thus the degree of similarity between the images.

What's the next question?

1. Although we know the definition of the dictionary, how does it come from?
2. If we can calculate the similarity score between two images, is it enough to judge the loopback?

So next, we first introduce how to generate a dictionary, and then how to use the dictionary to actually calculate the similarity between two images.

11.3 dictionary

11.3.1 Dictionary structure

According to the previous introduction, the dictionary is composed of many words, and each word represents a concept. A word is different from a single feature point.

* Although this property sometimes brings some problems, for example, is the face with eyes under the mouth still a human face?

It is not extracted from a single image, but a combination of certain types of features. So, the dictionary generation problem is similar to a **Clustering** problem.

The clustering problem is particularly common in Unsupervised ML, which is used to let the machine find the rules in the data by itself. BoW's dictionary generation problem is also one of them. First, suppose we have extracted feature points from a large number of images, such as N . Now, we want to find a dictionary with k words. Each word can be regarded as a set of local neighboring feature points. What should we do? This can be solved with the classic K-means (K-means) algorithm [?].

K-means is a very simple and effective method, so it is widely used in unsupervised learning. The principle is briefly introduced below. To put it simply, when there are N data and want to be classified into k categories, then using K-means to do it mainly includes the following steps:

1. Randomly selects k center points: c_1, \dots, c_k .
2. For each sample, calculate the distance between it and each center point, and take the smallest one as its classification.
3. Recalculates the center point of each class.
4. If each center point changes little, the algorithm converges and exits; otherwise returns to step 2.

K-means' approach is simple and simple and effective, but there are some problems, such as the need to specify the number of clusters, randomly select the center point so that the results of each cluster are different, and some efficiency problems. Later, researchers also developed algorithms such as hierarchical clustering and K-means ++ [?] to make up for its shortcomings, but this is all later, we will not discuss them in detail. In short, according to K-means, we can cluster a large number of feature points that have been extracted into a dictionary containing k words. The question now becomes how to find the corresponding word in the dictionary based on a feature point in the image.

There is still a simple idea: just compare with each word and take the one that is most similar-this is of course a simple and effective way. However, given the generality of a dictionary *, We usually use a large-scale dictionary to ensure that image features in the current use environment have appeared in the dictionary, or at least have similar expressions. If you don't think it is troublesome to compare ten words one by one, then what about ten thousand words? What about 100,000?

Perhaps the reader has learned the data structure, this $O(n)$ lookup algorithm is obviously not what we want. If the dictionary is sorted, then binary search can obviously improve search efficiency and achieve logarithmic level of complexity. In practice, we may use more complex data structures, such as Chou-Liu tree [?] in Fabmap [? ? ?]. But we don't want to write this book as a collection of complex details, so we introduce another simple and practical tree structure [?].

In the literature [?], a k fork tree is used to express the dictionary. The idea is simple (as shown in ??), similar to hierarchical clustering, and is a direct extension of k-means. Suppose we have N feature points, and we want to build a tree with a depth of d and a fork of k each time. The method is as follows And depth, this

* Would you call a page of paper with only ten words a dictionary? I believe that the dictionary in most people's minds is quite heavy.

might remind you of kd-tree [?]. The author thinks that although the approaches are different, the meanings they express are really the same.:

1. At the root node, k-means is used to aggregate all samples into the k class (in practice, k-means ++ is used to ensure cluster uniformity). This resulted in the first layer.
2. For each node in the first layer, the samples belonging to the node are regrouped into the k class to obtain the next layer.
3. and so on, and finally get the leaf layer. The leaf layer is called Words.

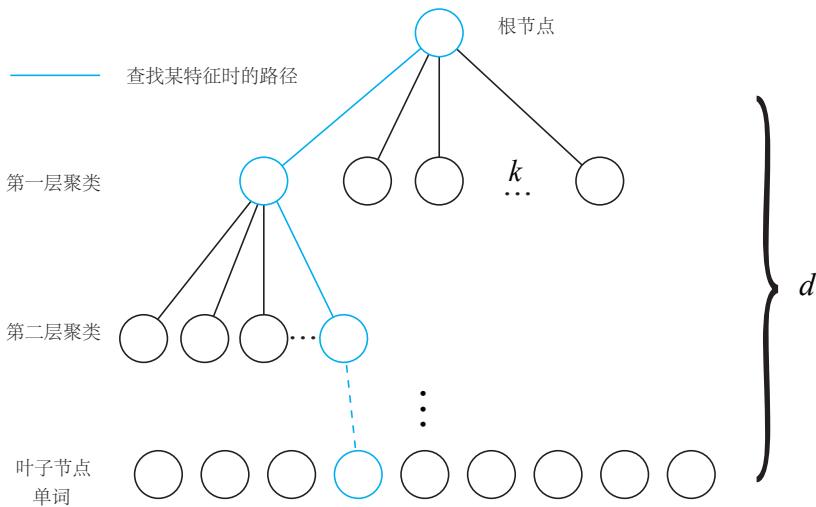


Figure 11-4: K fork tree dictionary diagram. When training the dictionary, K-means clustering is used layer by layer. When searching for words based on known features, you can also compare them layer by layer to find the corresponding words.

In fact, we ended up building words at the leaf level, and the intermediate nodes in the tree structure were only used for quick lookups. Such a k -branch with a depth of d can hold k^d words. On the other hand, when looking for a word corresponding to a given feature, you only need to compare it with the clustering center of each intermediate node (a total of d times) to find the last word, ensuring a logarithmic level search efficiency.

11.3.2 Practice: Creating a dictionary

Now that we've covered dictionary generation, let's actually demonstrate it. The previous VO part used a lot of ORB feature descriptions, so here we will demonstrate how to generate and use ORB dictionaries.

In this experiment, we select 10 images in the TUM dataset (located in slam-book2/ch11/data, as shown in ??), which are from a set of actual camera motion trajectories. It can be seen that the first image and the last image are obviously taken from the same place. Now we need to see if the program can detect this.

According to the bag of words model, we first generate a dictionary corresponding to these ten images.



Figure 11-5: Demonstrate the 10 images used in the experiment, collected from trajectories at different times.

It needs to be stated that the dictionary is often generated from a larger data set when the actual BoW is used, and it is best to come from a place similar to the target environment. We usually use larger-scale dictionaries—the larger the dictionary, the richer the number of words, the easier it is to find the words corresponding to the current image, but it cannot be larger than our computing power and memory. I don't plan to store a large dictionary file on GitHub, so we will temporarily train a small dictionary from ten images. If you want to pursue better results, you should download more data and train a larger dictionary so that the program will be practical. You can also use a dictionary trained by others, but please note that the dictionary uses the same type of features.

Let's start training the dictionary. First, please install the BoW library used by this program. We use DBoW3 *: <https://github.com/rmsalinas/DBow3>. Readers can also find it from the 3rdparty folder of the code in this book, it is a cmake project, please compile and install it according to the cmake process.

Next consider the training dictionary:

Listing 11.1: slambook2/ch11/feature_training.cpp

```

1 int main(int argc, char **argv) {
2     // read the image
3     cout << "reading images... " << endl;
4     vector<Mat> images;
5     for (int i = 0; i < 10; i++) {
6         string path = "./data/" + to_string(i + 1) + ".png";
7         images.push_back(imread(path));
8     }
9     // detect ORB features
10    cout << "detecting ORB features ... " << endl;
11    Ptr<Feature2D> detector = ORB::create();
12    vector<Mat> descriptors;
13    for (Mat &image:images) {
14        vector<KeyPoint> keypoints;
15        Mat descriptor;
16        detector->detectAndCompute(image, Mat(), keypoints, descriptor);
17        descriptors.push_back(descriptor);

```

* The main reason for choosing it is that it has good compatibility with OpenCV3, and it is easy to get started with compilation and use.

```

18 }
19
20 // create vocabulary
21 cout << "creating vocabulary ... " << endl;
22 DBoW3::Vocabulary vocab;
23 vocab.create(descriptors);
24 cout << "vocabulary info: " << vocab << endl;
25 vocab.save("vocabulary.yml.gz");
26 cout << "done" << endl;
27
28 return 0;
29 }
```

Using DBoW3 is very easy. We extract the ORB features of the 10 target images and store them in the vector container, and then call the dictionary generation interface of DBoW3. In the constructor of the DBoW3 :: Vocabulary object, we can specify the number of branches and the depth of the tree, but the default constructor is used here, which is $k = 10, d = 5$. This is a small dictionary with a maximum capacity of 100,000 words. For image features, we also use the default parameters, which are 500 feature points per image. Finally, we store the dictionary as a compressed file.

Run this program, you will see the following dictionary information output:

```

1 $build/feature_training
2 reading images...
3 detecting ORB features ...
4 creating vocabulary ...
5 vocabulary info: Vocabulary: k = 10, L = 5, Weighting = tf-idf, Scoring =
   L1-norm, Number of words = 4983
6 done
```

We see that the number of branches k is 10, and the depth L is 5 *. The number of words is 4983, and the maximum capacity is not reached. But what are the remaining Weighting and Scoring? Literally, Weighting is weighting, Scoring seems to be referring to ratings, but how are ratings calculated?

11.4 Similarity calculation

11.4.1 Theoretical section

Let us discuss the problem of similarity calculation. After having a dictionary, given any feature f_i , as long as it is searched layer by layer in the dictionary tree, the corresponding word w_j can be found at the end-when the dictionary is large enough, we can think of f_i and w_j comes from the same type of object (although there is no theoretical guarantee, it is only said in the sense of clustering). Then, assuming that N features are extracted from an image, and after finding the words corresponding to these N features, we are equivalent to having the distribution of the image in the word list, or a histogram. Intuitively (or ideally), it is equivalent to saying "there is a person and a car in this picture". According to Bag-of-Words, this might be considered a Bag.

Note that in this approach, we treat all words equally-there is, there is, there is no. Is this good? Considering that different words are not the same in terms of importance. For example, words such as "yes" and "yes" may appear in many sentences, and we cannot judge the type of sentence based on them; but if there

* here L is the mentioned earlier.

are words like "document" and "football", it will be of great help to judge sentences. Some, it can be said that they provide more information. So to sum up, we want to evaluate the distinctiveness or importance of words and give them different weights for better results.

TF-IDF (Term Frequency–Inverse Document Frequency) [? ?], or translation frequency–inverse document frequency *, A weighting method commonly used in text retrieval, is also used in BoW models. The idea of the TF part is that a word often appears in an image, and its discrimination is high. On the other hand, the idea of IDF is that the lower the frequency of a word in a dictionary, the higher the degree of discrimination when classifying images.

We can calculate the IDF when building a dictionary: Count the ratio of the number of features in a leaf node w_i to the number of all features as the IDF part. Suppose the number of all features is n and the number of w_i is n_i , then the IDF of the word is

$$\text{IDF}_i = \log \frac{n}{n_i}. \quad (11.5)$$

On the other hand, the TF part refers to how often a feature appears in a single image. Suppose the word w_i appears n_i times in the image A , and the total number of words appears n , then TF is

$$\text{TF}_i = \frac{n_i}{n}. \quad (11.6)$$

So, the weight of w_i is equal to the product of TF times IDF:

$$\eta_i = \text{TF}_i \times \text{IDF}_i. \quad (11.7)$$

After considering the weight, for an image A , its feature points can correspond to many words, which make up its Bag-of-Words:

$$A = \{(w_1, \eta_1), (w_2, \eta_2), \dots, (w_N, \eta_N)\} \triangleq \mathbf{v}_A. \quad (11.8)$$

Since similar features may fall into the same class, there will be a lot of zeros in the actual \mathbf{v}_A . Anyway, with the bag of words we describe an image A with a single vector \mathbf{v}_A . This vector \mathbf{v}_A is a sparse vector. Its non-zero parts indicate which words are contained in the image A , and the value of these parts is the value of TF-IDF.

The next question is: given \mathbf{v}_A and \mathbf{v}_B , how do you calculate their difference? This problem is the same as the norm definition. There are several solutions, such as the L_1 norm form mentioned in the document [?]:

$$s(\mathbf{v}_A - \mathbf{v}_B) = 2 \sum_{i=1}^N |\mathbf{v}_{Ai}| + |\mathbf{v}_{Bi}| - |\mathbf{v}_{Ai} - \mathbf{v}_{Bi}|. \quad (11.9)$$

Of course, there are many other ways waiting for you to explore, here we only give an example as a demonstration.

11.4.2

Bag-of-Words

* I personally think that TF-IDF is easier to call, so I will use English abbreviations in the following Not Chinese translation.

Listing 11.2: slambook/ch12/loop_closure.cpp

```

1 int main(int argc, char **argv) {
2     // read the images and database
3     cout << "reading database" << endl;
4     DBoW3::Vocabulary vocab("./vocabulary.yml.gz");
5     // DBoW3::Vocabulary vocab("./vocab_larger.yml.gz"); // use large vocab
6     if (vocab.empty()) {
7         cerr << "Vocabulary does not exist." << endl;
8         return 1;
9     }
10    cout << "reading images..." << endl;
11    vector<Mat> images;
12    for (int i = 0; i < 10; i++) {
13        string path = "./data/" + to_string(i + 1) + ".png";
14        images.push_back(imread(path));
15    }
16
17    // NOTE: in this case we are comparing images with a vocabulary generated
18    // by themselves, this may lead to overfit.
19    // detect ORB features
20    cout << "detecting ORB features ..." << endl;
21    Ptr<Feature2D> detector = ORB::create();
22    vector<Mat> descriptors;
23    for (Mat &image:images) {
24        vector<KeyPoint> keypoints;
25        Mat descriptor;
26        detector->detectAndCompute(image, Mat(), keypoints, descriptor);
27        descriptors.push_back(descriptor);
28    }
29
30    // we can compare the images directly or we can compare one image to a
31    // database
32    // images :
33    cout << "comparing images with images " << endl;
34    for (int i = 0; i < images.size(); i++) {
35        DBoW3::BowVector v1;
36        vocab.transform(descriptors[i], v1);
37        for (int j = i; j < images.size(); j++) {
38            DBoW3::BowVector v2;
39            vocab.transform(descriptors[j], v2);
40            double score = vocab.score(v1, v2);
41            cout << "image " << i << " vs image " << j << " : " << score << endl;
42        }
43        cout << endl;
44    }
45
46    // or with database
47    cout << "comparing images with database " << endl;
48    DBoW3::Database db(vocab, false, 0);
49    for (int i = 0; i < descriptors.size(); i++)
50        db.add(descriptors[i]);
51    cout << "database info: " << db << endl;
52    for (int i = 0; i < descriptors.size(); i++) {
53        DBoW3::QueryResults ret;
54        db.query(descriptors[i], ret, 4); // max result=4
55        cout << "searching for image " << i << " returns " << ret << endl <<
56        endl;
57    }
58    cout << "done." << endl;
59 }
```

Bag-of-Words

Listing 11.3:

```

1 $build/feature_training
2 reading database
3 reading images...
4 detecting ORB features ...
5 comparing images with images
6 desp 0 size: 500
7 transform image 0 into BoW vector: size = 455
8 key value pair = <1, 0.00155622>, <3, 0.00222645>, <12, 0.00222645>, <13,
   0.00222645>, <14, 0.00222645>, <22, 0.00222645>, <33, 0.00222645>, <37,
   0.00155622>, <38, 0.00222645>, <39, 0.00222645>, <43, 0.00222645>,
   <57, 0.00155622> .....

```

It can be seen that the BoW description vector contains the ID and weight of each word, and they form the entire sparse vector. When we compare two vectors, DBoW3 will calculate a score for us. The calculation method is defined by the previous dictionary construction:

Listing 11.4: Terminal output:

```

1 image 0 vs image 0 : 1
2 image 0 vs image 1 : 0.0234552
3 image 0 vs image 2 : 0.0225237
4 image 0 vs image 3 : 0.0254611
5 image 0 vs image 4 : 0.0253451
6 image 0 vs image 5 : 0.0272257
7 image 0 vs image 6 : 0.0217745
8 image 0 vs image 7 : 0.0231948
9 image 0 vs image 8 : 0.0311284
10 image 0 vs image 9 : 0.0525447

```

When querying the database, DBoW sorted the above scores and gave the most similar results:

Listing 11.5:

```

1 searching for image 0 returns 4 results:
2 <EntryId: 0, Score: 1>
3 <EntryId: 9, Score: 0.0525447>
4 <EntryId: 8, Score: 0.0311284>
5 <EntryId: 5, Score: 0.0272257>
6
7 searching for image 1 returns 4 results:
8 <EntryId: 1, Score: 1>
9 <EntryId: 2, Score: 0.0339641>
10 <EntryId: 8, Score: 0.0299387>
11 <EntryId: 3, Score: 0.0256668>
12
13 searching for image 2 returns 4 results:
14 <EntryId: 2, Score: 1>
15 <EntryId: 7, Score: 0.036092>
16 <EntryId: 9, Score: 0.0348702>
17 <EntryId: 1, Score: 0.0339641>
18
19 searching for image 3 returns 4 results:
20 <EntryId: 3, Score: 1>
21 <EntryId: 9, Score: 0.0357317>
22 <EntryId: 8, Score: 0.0278496>
23 <EntryId: 5, Score: 0.0270168>
24

```

```

25 searching for image 4 returns 4 results:
26 <EntryId: 4, Score: 1>
27 <EntryId: 5, Score: 0.0493492>
28 <EntryId: 0, Score: 0.0253451>
29 <EntryId: 6, Score: 0.0253017>
30
31 searching for image 5 returns 4 results:
32 <EntryId: 5, Score: 1>
33 <EntryId: 4, Score: 0.0493492>
34 <EntryId: 9, Score: 0.028996>
35 <EntryId: 6, Score: 0.0277584>
36
37 searching for image 6 returns 4 results:
38 <EntryId: 6, Score: 1>
39 <EntryId: 8, Score: 0.0306241>
40 <EntryId: 5, Score: 0.0277584>
41 <EntryId: 3, Score: 0.0267135>
42
43 searching for image 7 returns 4 results:
44 <EntryId: 7, Score: 1>
45 <EntryId: 2, Score: 0.036092>
46 <EntryId: 1, Score: 0.0239091>
47 <EntryId: 0, Score: 0.0231948>
48
49 searching for image 8 returns 4 results:
50 <EntryId: 8, Score: 1>
51 <EntryId: 9, Score: 0.0329149>
52 <EntryId: 0, Score: 0.0311284>
53 <EntryId: 6, Score: 0.0306241>
54
55 searching for image 9 returns 4 results:
56 <EntryId: 9, Score: 1>
57 <EntryId: 0, Score: 0.0525447>
58 <EntryId: 3, Score: 0.0357317>
59 <EntryId: 2, Score: 0.0348702>

```

The reader can look at all the output and see how much different images score from similar images. We see that Figures 1 and 10, which are clearly similar (subscripted as 0 and 9 in C++, respectively), have a similarity score of about 0.0525; the other images are about 0.02.

In the demonstration experiments in this section, we saw that the similar images 1 and 10 scored significantly higher than other image pairs, but numerically it was not as obvious as we thought. It is said that if the similarity between ourselves and itself is 100

11.5 Experimental Analysis and Review

11.5.1 Increase dictionary size

In the field of machine learning, if the code is not wrong and the results are unsatisfactory, we first suspect "whether the network structure is large enough, the number of layers is deep enough, whether there are enough data samples", etc. This is still due to "good models can't beat bad data" (Partly because of the lack of deeper theoretical analysis). Although we are now studying SLAM, when this happens, we will first wonder: Is the dictionary too small? After all, we are generating a dictionary from ten images, and then calculate the image similarity based on this dictionary.

slambook2/ch11/vocab _larger.yml.gz is a slightly larger dictionary that we

generate-in fact, it is generated for all images of the same data sequence, about 2,900 images. The size of the dictionary is still $k = 10, d = 5$, which is a maximum of 10,000 words. Readers can use the gen_vocab_large.cpp file in the same directory to train the dictionary on their own. Please note that to train a large dictionary, you may need a machine with large memory, and wait patiently for a while. We slightly modified the program in the previous section to use a larger dictionary to detect image similarity:

Listing 11.6:

```

1 comparing images with database
2 database info: Database: Entries = 10, Using direct index = no. Vocabulary:
   k = 10, L = 5, Weighting = tf-idf, Scoring = L1-norm, Number of words
   = 99566
3 searching for image 0 returns 4 results:
4 <EntryId: 0, Score: 1>
5 <EntryId: 9, Score: 0.0320906>
6 <EntryId: 8, Score: 0.0103268>
7 <EntryId: 4, Score: 0.0066729>
8
9 searching for image 1 returns 4 results:
10 <EntryId: 1, Score: 1>
11 <EntryId: 2, Score: 0.0238409>
12 <EntryId: 8, Score: 0.00814409>
13 <EntryId: 3, Score: 0.00697527>
14
15 searching for image 2 returns 4 results:
16 <EntryId: 2, Score: 1>
17 <EntryId: 1, Score: 0.0238409>
18 <EntryId: 5, Score: 0.00897928>
19 <EntryId: 8, Score: 0.00893477>
20
21 searching for image 3 returns 4 results:
22 <EntryId: 3, Score: 1>
23 <EntryId: 5, Score: 0.0107005>
24 <EntryId: 8, Score: 0.00870392>
25 <EntryId: 6, Score: 0.00720695>
26
27 searching for image 4 returns 4 results:
28 <EntryId: 4, Score: 1>
29 <EntryId: 6, Score: 0.0069998>
30 <EntryId: 0, Score: 0.0066729>
31 <EntryId: 5, Score: 0.0062834>
32
33 searching for image 5 returns 4 results:
34 <EntryId: 5, Score: 1>
35 <EntryId: 3, Score: 0.0107005>
36 <EntryId: 2, Score: 0.00897928>
37 <EntryId: 4, Score: 0.0062834>
38
39 searching for image 6 returns 4 results:
40 <EntryId: 6, Score: 1>
41 <EntryId: 7, Score: 0.00915307>
42 <EntryId: 3, Score: 0.00720695>
43 <EntryId: 4, Score: 0.0069998>
44
45 searching for image 7 returns 4 results:
46 <EntryId: 7, Score: 1>
47 <EntryId: 6, Score: 0.00915307>
48 <EntryId: 8, Score: 0.00814517>
49 <EntryId: 1, Score: 0.00538609>
50

```

```

51 searching for image 8 returns 4 results:
52 <EntryId: 8, Score: 1>
53 <EntryId: 0, Score: 0.0103268>
54 <EntryId: 2, Score: 0.00893477>
55 <EntryId: 3, Score: 0.00870392>
56
57 searching for image 9 returns 4 results:
58 <EntryId: 9, Score: 1>
59 <EntryId: 0, Score: 0.0320906>
60 <EntryId: 8, Score: 0.00636511>
61 <EntryId: 1, Score: 0.00587605>
```

It can be seen that when the dictionary size increases, the similarity of irrelevant images becomes significantly smaller. For similar images, for example, images 1 and 10, although the scores have dropped slightly, the scores of other images have become more significant. This shows that adding dictionary training samples is beneficial. In the same way, readers can try using a larger dictionary and see how the results change.

11.5.2 Handling of similarity scores

For any two images, we can give a similarity score, but using the absolute size of this score is not necessarily helpful. For example, some environments have inherently similar appearances. For example, offices often have many tables and chairs of the same style; other environments are very different in different places. Considering this situation, we will take a **transcendental similarity** $s(\mathbf{v}_t, \mathbf{v}_{t-\Delta t})$, which means The similarity between the keyframe image at a moment and the keyframe at the previous moment. Then, the other scores are normalized with reference to this value:

$$s(\mathbf{v}_t, \mathbf{v}_{t_j})' = s(\mathbf{v}_t, \mathbf{v}_{t_j}) / s(\mathbf{v}_t, \mathbf{v}_{t-\Delta t}). \quad (11.10)$$

From this perspective, we say: if the similarity between the current frame and a previous key frame exceeds three times the similarity between the current frame and the previous key frame, it is considered that there may be a loopback. This step avoids the introduction of absolute similarity thresholds, allowing the algorithm to adapt to more environments.

11.5.3 Processing of key frames

When detecting loops, we must consider the selection of key frames. If the key frames are selected too close, the similarity between the two key frames will be too high, and by comparison, it is not easy to detect loops in historical data. For example, the detection result is often the most similar between the n frame, the $n - 2$ frame, and the $n - 3$ frame. This kind of result seems too ordinary and has little meaning. So in practice, the frames used for loopback detection are preferably sparse, they are not the same as each other, and they can cover the entire environment.

On the other hand, if a loopback is successfully detected, for example, it appears in frames 1 and n . Then it is very likely that frames $n + 1$, $n + 2$ will all form a loop with frame 1. However, confirming that there is a loopback between the first frame and the n frame is helpful for trajectory optimization, and the next $n + 1$ frame, $n + 2$ frame will be the first frame The help generated by loopbacks is not so great, because we have used the previous information to eliminate the cumulative error, and more loopbacks will not bring more information. Therefore, we will group "close"

loops into one type, so that the algorithm does not repeatedly detect loops of the same type.

11.5.4 Validation after detection

The bag-of-words loop detection algorithm is completely dependent on appearance without using any geometric information, which results in images with similar appearances being easily treated as loops. In addition, since the bag of words does not care about the order of the words, and only cares about the expression of the words, it is more likely to cause perception bias. Therefore, after loopback detection, we usually have a verification step [? ?].

There are many ways to verify. One is to set up a cache mechanism for loopbacks. It is considered that a single loopback detection is not enough to constitute a good constraint, and a loopback that has been detected for a period of time is considered a correct loopback. This can be seen as a consistency check in time. Another method is spatial consistency detection, that is, feature matching is performed on the two frames detected by the loop, and the camera motion is estimated. Then, put the motion into the previous Pose Graph, and check whether there is a big difference from the previous estimation. In short, the verification part is usually necessary, but how to achieve it is a matter of opinion.

11.5.5 Relationship with Machine Learning

As can be seen from the previous discussion, loop detection is inextricably linked to machine learning. Loopback detection itself is very much like a classification problem. The difference from traditional pattern recognition is that the number of categories in the loop is large, and the samples of each category are small-in extreme cases, when the robot moves, the image changes, and new categories are generated. We can even put Classes are treated as continuous variables rather than discrete variables. Loopback detection, which is equivalent to two images falling into the same class, is rare. From another perspective, loop detection is also equivalent to a study of the "similarity between images" concept. Now that humans can determine whether images are similar, it is very possible for machines to learn such concepts.

From the bag-of-words model, it is an unsupervised machine learning process itself-building a dictionary is equivalent to clustering feature descriptors, and a tree is just a fast-looking data structure for clustered classes. Since it is clustering, combined with the knowledge in machine learning, we can at least ask:

1. Is it possible to cluster machine learning image features instead of clustering artificially designed features like SURF, ORB?
2. Is there a better way to do clustering than the simpler way of adding tree structure and K-means?

Combined with the current development of machine learning, the learning of binary descriptors and unsupervised clustering are all very promising problems that can be solved in deep learning frameworks. We have also seen the use of machine learning for loop detection. Although the current bag-of-words method is still mainstream, I believe that future deep learning methods are very promising to defeat these artificially designed, "traditional" machine learning methods [? ?]. After all, the bag-of-words method is obviously inferior to neural networks in object recognition, and loop detection is a very similar problem. For example, the improved

version of the BoW model, VLAD, has a CNN-based implementation [? ?], and there are also some meshes that can be returned from the image to collect the camera pose after training [?], which may become New loop detection algorithm.

EXERCISE

1. Please write a small program to calculate PR curve. It may be easier to use MATLAB or Python because they are good at drawing.
2. To verify the loopback detection algorithm, you need a dataset with manually labeled loopbacks, such as [?]. However, it is inconvenient to manually mark loops. We will consider calculating loops based on standard trajectories. That is, if two frames in the track have very similar poses, they are considered loops. Please calculate the loopback in a data set according to the standard trajectory given by the TUM data set. Are these loopback images really similar?
3. Learn the DBoW3 or DBoW2 library, find a few pictures yourself, and see if you can correctly detect loops from them.
4. What are the commonly used measures of similarity scoring?
5. What is the Chow-Liu tree? How is it used for dictionary construction and loop detection?
6. Read the literature [?]. In addition to the bag of words model, what other methods are used for loop detection?

Chapter 12

Build map

main target

1. Understand the principle of dense depth estimation in monocular SLAM.
2. Experimentally understand the process of monocular dense reconstruction.
3. Learn about several map formats in RGB-D reconstruction.

In this lecture we start to introduce the algorithm of the mapping part. In the front-end and back-end, we focus on the problem of simultaneously estimating the camera motion trajectory and the spatial position of feature points. However, when SLAM is actually used, in addition to positioning the camera body, there are many other requirements. For example, consider SLAM placed on a robot, then we would like the map to be used for positioning, navigation, obstacle avoidance, and interaction. The feature point map obviously cannot meet all these needs. Therefore, in this lecture, we will discuss various forms of maps in more detail, and point out the current flaws in visual SLAM maps.

12.1 Overview

Mapping should be one of the two goals of SLAM-because SLAM is called simultaneous positioning and mapping. But until now, we have discussed positioning issues, including positioning by feature points, direct method, and back-end optimization. So, does this suggest that mapping is not so important in SLAM, so we didn't discuss it until this lecture?

the answer is negative. In fact, in the classic SLAM model, what we call a map is a collection of all landmark points. Once the location of the landmarks is determined, it can be said that we have completed the mapping. Therefore, the visual odometer mentioned above is good, as is the Bundle Adjustment. In fact, the positions of the landmark points are modeled and optimized. From this perspective, we have discussed the problem of mapping. So why bother with a separate picture?

This is because people have different needs for building maps. As a low-level technology, SLAM is often used to provide information for upper-level applications. If the upper layer is a robot, then the developers at the application layer may want to use SLAM to do global positioning and let the robot navigate in the map-for example, the sweeper needs to complete the sweeping work and wants to calculate a path that covers the entire map. Or, if the upper layer is an augmented reality device, then the developer may wish to superimpose the virtual object on the real object, and in particular, may also need to handle the occlusion relationship between the virtual object and the real object.

We found that the requirements for "positioning" at the application level are similar, and they want SLAM to provide the spatial pose information of the camera or the subject with the camera. For maps, there are many different needs. From the perspective of visual SLAM, "mapping" serves "positioning"; but at the application level, "mapping" obviously carries many other requirements. Regarding the usefulness of the map, we can roughly summarize it as follows:

1. **targeting.** Positioning is a basic function of maps. In the previous visual odometer section, we discussed how to use local maps to achieve positioning. In the loop detection part, we also saw that as long as there is global descriptor information, we can also determine the position of the robot through loop detection. Furthermore, we also hope to save the map so that the robot can still locate in the map after the next boot, so that only the model of the map needs to be modeled once, instead of re-doing a complete SLAM every time the robot is started.
2. **navigation.** Navigation refers to the process by which a robot can plan a path in a map, find a path between any two map points, and then control its movement to a target point. In the process, we need to know at least **where in the map is not passable and which is passable**. This is beyond the capabilities of the sparse feature point map, we must have another map form. We will say later that this must be at least a **dense** map.
3. **Avoidance.** Obstacle avoidance is also a problem that robots often encounter. It is similar to navigation, but focuses more on the handling of local, dynamic obstacles. Similarly, with only feature points, we cannot determine whether a feature point is an obstacle, so we need a **dense** map.
4. **rebuild.** Sometimes, we want to use SLAM to obtain the reconstruction effect of the surrounding environment. This kind of map is mainly used to show

people, so we want it to look more comfortable and beautiful. Alternatively, we can also use the map for communication so that others can remotely view the 3D objects or scenes we have reconstructed-such as 3D video calls or online shopping. This map is also **dense**, and we also have some requirements for its appearance. We may not be satisfied with the reconstruction of dense point clouds, but prefer to be able to construct textured planes, just like 3D scenes in video games.

5. **interactive.** Interaction mainly refers to the interaction between people and the map. For example, in augmented reality, we would place virtual objects in the room and interact with these virtual objects-for example, I would click on a virtual web browser on the wall to watch the video, or Throw objects on the wall, hoping they have (virtual) physical collisions. On the other hand, there will be interactions with people and maps in robot applications. For example, the robot may receive the command "take the newspaper on the table". In addition to the environment map, the robot also needs to know which map is the "table", what is called "above" and what is called "above" newspaper". This requires robots to have a higher level of understanding of maps-also known as semantic maps.

?? vividly explains the relationship between the various map types and uses discussed above. Our previous discussion has basically focused on the "sparse roadmap map" section, and we have not discussed dense maps. The so-called dense map is relative to the sparse map. The sparse map only models the part of interest, that is, the feature points (signpost points) that have been said for a long time. A dense map is a model that **all** has seen. For the same table, a sparse map might model only the four corners of the table, while a dense map models the entire desktop. Although from a positioning perspective, a map with only four corners can also be used to locate the camera, but because we cannot infer the spatial structure between these points from the four corners, we cannot use only four corners to complete the navigation. , Obstacle avoidance, and other tasks that require dense maps to complete.

As can be seen from the discussion above, dense maps occupy a very important place. So the remaining question is: Can dense maps be created with visual SLAM? If so, how to build it?

12.2 Monocular dense reconstruction

12.2.1 Stereovision

The problem of dense reconstruction of visual SLAM is the first important topic of this lecture. Cameras have long been considered as angle-only sensors. The pixels in a single image can only provide the angle between the object and the imaging plane of the camera and the brightness collected by the object, but cannot provide the distance of the object. In dense reconstruction, we need to know the distance of each pixel (or most pixels). There are roughly the following solutions:

1. Use a monocular camera to measure the distance of pixels by triangulating after moving the camera.
2. Use a binocular camera to calculate the distance between pixels using the parallax of left and right (the principle of multi-eye is the same).

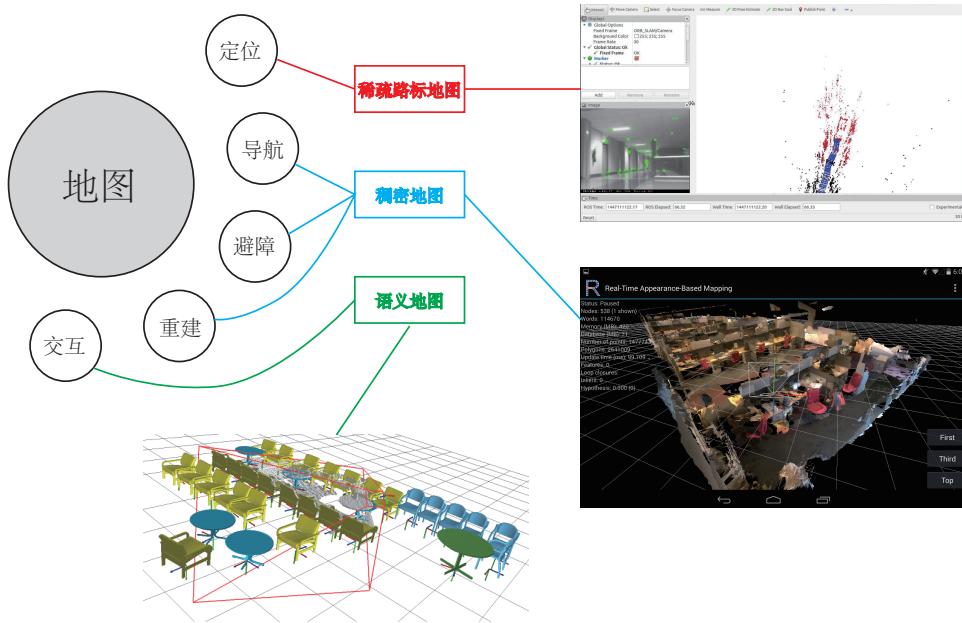


Figure 12-1: A schematic of various maps. The three example maps are from the literature [? ? ?].

3. Use RGB-D camera to get pixel distance directly.

The first two methods are called Stereo Vision, and the moving single purpose is also called moving view stereo (Moving View Stereo). Compared to the depth directly measured by RGB-D, the depth acquisition of monocular and binocular is often "laborious"—we need to spend a lot of calculations and finally get some unreliable * Depth estimation. Of course, RGB-D also has some range, application range, and lighting limitations, but compared to monocular and binocular results, dense reconstruction using RGB-D is often a more common choice. The advantage of monocular and binocular is that it can still estimate depth information through stereo vision in outdoor and large scene applications where RGB-D cannot be applied well.

Having said that, in this section we will lead the reader through a single-purpose dense estimation and experience why it is laborious and unpleasant. Let's start with the simplest case: how to estimate the depth of an image based on a given video sequence over a period of time. In other words, instead of considering SLAM, let's consider a slightly simpler mapping problem.

Suppose there is a video sequence, and we get the trajectory of each frame through some magic (of course it is also likely to be estimated by the visual odometry front end). Now we take the first image as the reference frame and calculate the depth (or distance) of each pixel in the reference frame. First, please remember how we completed the process in the feature points section:

1. First, we extract features from the image and calculate the matches between the features based on the descriptor. In other words, through the feature,

* officially called Fragile.

we tracked a certain spatial point and knew its position between the various images.

2. Then, because we can't use only one image to determine the location of a feature point, we must estimate its depth from observations at different perspectives, the principle is the triangulation described earlier.

In dense depth map estimation, the difference is that we cannot treat each pixel as a feature point calculation descriptor. Therefore, matching becomes a very important part of the dense depth estimation problem: how to determine where a certain pixel of the first image appears in other images? This requires **polar search** and **block matching technology** [?]. Then, when we know the position of a pixel in each picture, we can use triangulation to determine its depth like a feature point. The difference is that here we will use many triangulations to converge the depth estimation, not just one. We hope that the depth estimation will gradually converge to a stable value from a very uncertain quantity as the measurement increases. This is **depth filter technology**. So, the following content will mainly focus on this topic.

12.2.2 Epipolar search and block matching

Let's first explore the geometric relationship of observing the same point from different perspectives. This is very much like the epipolar geometry discussed in section ?? . See ?? . The camera on the left observes a pixel \mathbf{p}_1 . Since this is a monocular camera, we have no way of knowing its depth, so assuming that this depth may be within a certain area, let's say that it is between a certain minimum and infinity: $(d_{\min}, +\infty)$. Therefore, the spatial points corresponding to this pixel are distributed on a certain line segment (ray in this example). From another perspective (right camera), the projection of this line segment also forms a line on the image plane, which we know is called **polar line**. This epipolar line can also be determined when the motion between the two cameras is known. Conversely, if the motion is unknown, the epipolar line cannot be determined. So the question is: which point on the polar line is the \mathbf{p}_1 point we just saw?

Repeatedly, in the feature point method, we found the position of \mathbf{p}_2 through feature matching. However, now we have no descriptors, so we can only search for points that are similar to \mathbf{p}_1 . To be more specific, we may walk from one end to the other of the epipolar line in the second image and compare the similarity of each pixel with \mathbf{p}_1 one by one. From the perspective of directly comparing pixels, this approach is quite similar to the direct method.

In the discussion of the direct method, we know that comparing the brightness value of a single pixel is not necessarily stable and reliable. One very obvious thing is: in case there are many similar points on the pole line to \mathbf{p}_1 , how do we determine which one is true? This seems to return to the question we mentioned in loop detection: how to determine the similarity of two images (or two points)? Loop detection is solved by the bag of words, but because there are no features here, we have to find another way.

An intuitive idea is: Since the brightness of a single pixel is indistinguishable, is it possible to compare pixel blocks? We take a small block of $w \times w$ around \mathbf{p}_1 , and then take a lot of small blocks of the same size for comparison on the polar line, which can improve the discrimination to a certain extent . This is called **block matching**. Note that in this process, this comparison is only meaningful if it assumes that the

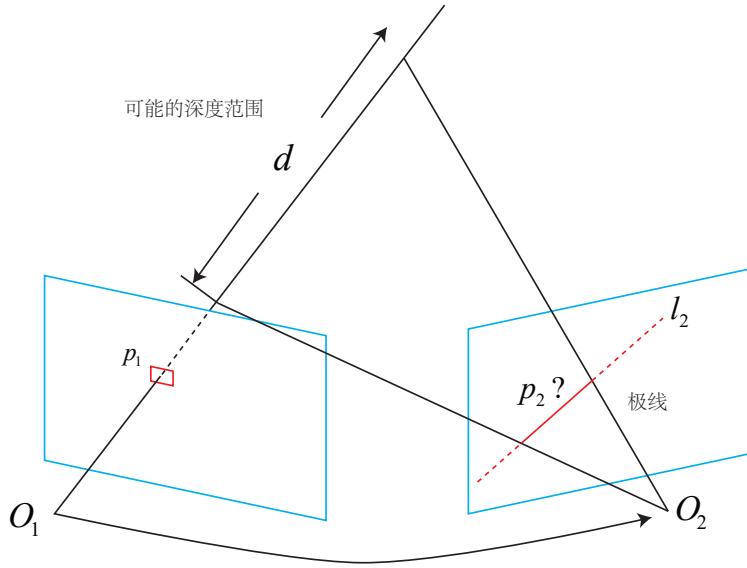


Figure 12-2: Epipolar search diagram.

gray value of the entire small block is constant between different images. So the assumption of the algorithm has changed from the invariance of the gray level of the pixel to the invariance of the gray level of the image block-to a certain extent, it becomes stronger.

OK, now we have taken the small pieces around p_1 , and we have taken many small pieces on the polar line. May wish to write the small blocks around p_1 as $\mathbf{A} \in \mathbb{R}^{w \times w}$, and write down the n small blocks on the pole Into $\mathbf{B}_i, i = 1, \dots, n$. So, how to calculate the difference between small blocks? There are several different calculation methods:

1. SAD (Sum of Absolute Difference). As the name suggests, the absolute value of the difference between two small blocks is summed:

$$S(\mathbf{A}, \mathbf{B})_{\text{SAD}} = \sum_{i,j} |\mathbf{A}(i,j) - \mathbf{B}(i,j)|. \quad (12.1)$$

2. SSD. The SSD here does not refer to the solid-state drive that everyone is familiar with, but the meaning of Sum of Squared Distance:

$$S(\mathbf{A}, \mathbf{B})_{\text{SSD}} = \sum_{i,j} ((\mathbf{A})(i,j) - \mathbf{B}(i,j))^2. \quad (12.2)$$

3. NCC (Normalized Cross Correlation). This method is more complicated than the first two, it calculates the correlation between two small blocks:

$$S(\mathbf{A}, \mathbf{B})_{\text{NCC}} = \frac{\sum_{i,j} \mathbf{A}(i,j)\mathbf{B}(i,j)}{\sqrt{\sum_{i,j} \mathbf{A}(i,j)^2 \sum_{i,j} \mathbf{B}(i,j)^2}}. \quad (12.3)$$

Note that because correlation is used here, a correlation close to 0 indicates that the two images are not similar, and a close to 1 indicates similarity. The first two distances are reversed. A value close to 0 indicates similarity, while a large value indicates dissimilarity.

As in many cases we have encountered, these calculations often have a contradiction between accuracy and efficiency. Good-precision methods often require complex calculations, while simple fast algorithms often do not work well. This requires us to make trade-offs in actual engineering. In addition, in addition to these simple versions, we can **remove the average of each small block first**, called SSD with average, NCC with average, and so on. After removing the mean value, we allow situations like "the small block B_i is brighter than A as a whole, but still very similar" *, So it's more reliable than before. If the reader is interested in more block matching measurement methods, it is recommended to read the article [? ?] as supplementary material.

Now, we have calculated the similarity measure between A and each B_i on the epipolar line. For the sake of description, suppose we use NCC, then we will get an NCC distribution along the epipolar line. The shape of this distribution depends heavily on how the image itself looks like ?? . In the case of a long search distance, we usually get a non-convex function: there are many peaks in this distribution, but there must be only one true corresponding point. In this case, we tend to use the probability distribution to describe the depth value rather than a single value to describe the depth. So, our question turns to how the estimated depth distribution will change when we continuously perform epipolar search on different images-this is the so-called **depth filter**.

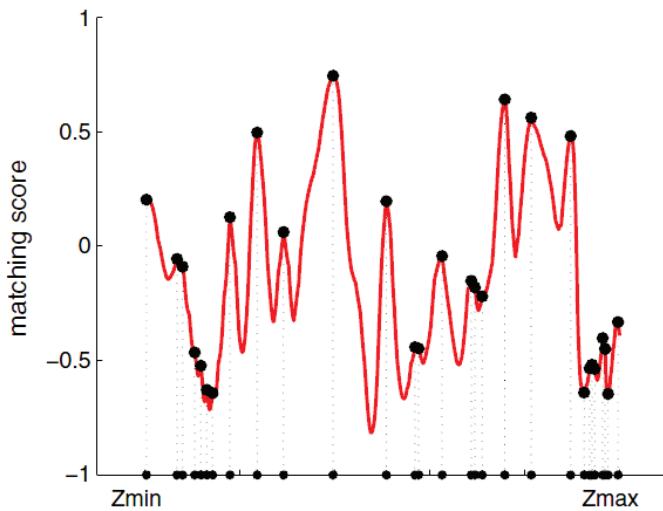


Figure 12-3: The distribution of matching scores along the distance, the image is from the literature [?].

* whole overall may be brightened by ambient light Or the camera exposure parameter is increased.

12.2.3 Gaussian depth filter

The estimation of the depth of a pixel can itself be modeled as a state estimation problem. Therefore, there are two solutions to the problem: filters and nonlinear optimization. Although the nonlinear optimization effect is better, in the case of SLAM, where real-time requirements are strong, considering that the front end has occupied a lot of calculations, the filter method is usually used in the construction of the map. This is also the purpose of the depth filter discussed in this section.

There are several different approaches to the distribution of depth assumptions. First, under relatively simple assumptions, we can assume that the depth value obeys the Gaussian distribution, and get a Kalman-like method (but it is actually just a normalized product, which we will see later). On the other hand, in the literature such as [? ?], the uniform-Gaussian mixed distribution hypothesis is also adopted, and another form of more complex depth filter is derived. Based on the simple and easy-to-use principle, we first introduce and demonstrate the depth filter under the assumption of Gaussian distribution, and then use the uniform-Gaussian mixed distribution filter as an exercise.

Let the depth d of a pixel obey:

$$P(d) = N(\mu, \sigma^2). \quad (12.4)$$

And whenever new data comes in, we will observe its depth. Similarly, suppose this observation is also a Gaussian distribution:

$$P(d_{\text{obs}}) = N(\mu_{\text{obs}}, \sigma_{\text{obs}}^2). \quad (12.5)$$

So, our question is how to use the observed information to update the original d distribution. This is exactly an information fusion problem. According to Appendix A, we understand that the product of two Gaussian distributions is still a Gaussian distribution. Let the distribution of d after fusion be $N(\mu_{\text{fuse}}, \sigma_{\text{fuse}}^2)$, then according to the product of the Gaussian distribution, there are:

$$\mu_{\text{fuse}} = \frac{\sigma_{\text{obs}}^2 \mu + \sigma^2 \mu_{\text{obs}}}{\sigma^2 + \sigma_{\text{obs}}^2}, \quad \sigma_{\text{fuse}}^2 = \frac{\sigma^2 \sigma_{\text{obs}}^2}{\sigma^2 + \sigma_{\text{obs}}^2}. \quad (12.6)$$

Since we only have observation equations and no equations of motion, we only use the information fusion part for the depth here, without the need to predict and update like the full Kalman. You can see that the fused equation is relatively easy to understand, but the question still remains: how to determine the distribution of the depth we observe? That is, how to calculate $\mu_{\text{obs}}, \sigma_{\text{obs}}$?

Regarding $\mu_{\text{obs}}, \sigma_{\text{obs}}$, there are some different processing methods. For example, the literature [?] considers the sum of geometric uncertainty and photometric uncertainty, while [?] only considers geometric uncertainty. For the time being, we only consider the uncertainty caused by geometric relations. Now, suppose we have determined the projection position of a pixel of the reference frame in the current frame through epipolar search and block matching. So, how big is this location's uncertainty about depth?

Take ?? as an example. Considering an epipolar search, we found the p_2 point corresponding to p_1 , and observed the depth value of p_1 . The corresponding three-dimensional point of p_1 is P . Thus, you can remember that $O_1 P$ is p , $O_1 O_2$ is the camera's translation t , $O_2 P$ is recorded as a . Also, note the lower two corners of this triangle as α, β . Now, consider that there is a pixel size error on the epipolar

line l_2 , so that the angle of β becomes β' , and p_2 also becomes p'_2 , and note that the corner above is γ . What we want to ask is, how much difference does this one pixel error cause p' and p ?

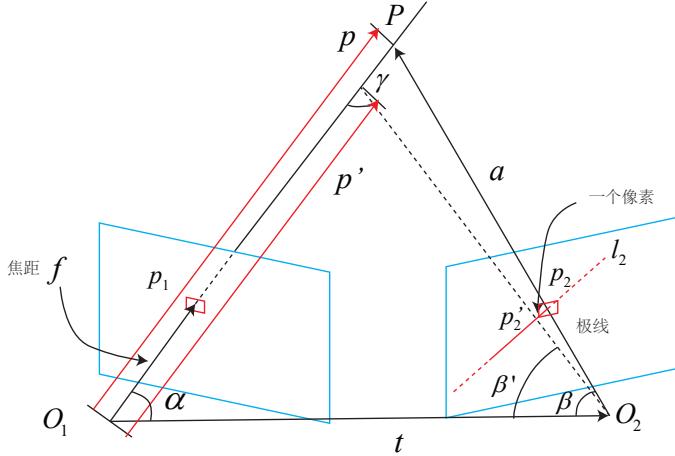


Figure 12-4: Uncertainty analysis.

This is a typical geometric problem. Let's list the geometric relationships between these quantities. Obviously:

$$\begin{aligned} \mathbf{a} &= \mathbf{p} - \mathbf{t} \\ \alpha &= \arccos \langle \mathbf{p}, \mathbf{t} \rangle \\ \beta &= \arccos \langle \mathbf{a}, -\mathbf{t} \rangle. \end{aligned} \quad (12.7)$$

Perturbing a pixel for p_2 will cause a change in β to become β' . According to the geometric relationship, there are:

$$\begin{aligned} \beta' &= \arccos \langle \mathbf{O}_2 \mathbf{p}'_2, -\mathbf{t} \rangle \\ \gamma &= \pi - \alpha - \beta'. \end{aligned} \quad (12.8)$$

Therefore, from the sine theorem, the size of p' can be obtained:

$$\|\mathbf{p}'\| = \|\mathbf{t}\| \frac{\sin \beta'}{\sin \gamma}. \quad (12.9)$$

From this, we determined the depth uncertainty caused by the uncertainty of a single pixel. If you think that the block search of the epipolar search has only one pixel error, then you can set:

$$\sigma_{\text{obs}} = \|\mathbf{p}\| - \|\mathbf{p}'\|. \quad (12.10)$$

Of course, if the uncertainty of the epipolar search is greater than one pixel, we can also amplify this uncertainty by following this derivation. The following deep data fusion has been introduced earlier. In actual engineering, when the uncertainty is less than a certain threshold, it can be considered that the depth data has converged.

In summary, we give a complete process for estimating the dense depth:

1. Assume that the depth of all pixels satisfies an initial Gaussian distribution.
2. When new data is generated, the position of the projection point is determined by epipolar search and block matching.
3. Calculates the depth and uncertainty after triangulation based on geometric relationships.
4. Fuse the current observation into the previous estimate. If it converges, stop the calculation, otherwise return to step 2.

These steps constitute a set of feasible depth estimation methods. Please note that the depth value mentioned here is the length of O_1P . It is slightly different from the "depth" we mentioned in the pinhole camera model—the depth in the pinhole camera refers to the z value of the pixel . We will demonstrate the results of the algorithm in the practice section.

12.3 Practice: Monocular Dense Reconstruction

The example program in this section will use the test data set of REMODE [? ?]. It provides a monocular overhead image collected by a drone, a total of 200, and provides the true pose of each image. Let us consider, on the basis of these data, estimate the depth value corresponding to each pixel of the first frame image, that is, perform monocular dense reconstruction.

First, the reader is requested to download the data used by the sample program from http://rpg.ifi.uzh.ch/datasets/remode_test_data.zip . You can use a web browser or download tool to download it. After decompression, all images from 0 to 200 will be found in test _data/Images, and a text file will be found in the test _data directory, which records the pose corresponding to each image:

```

1 scene_000.png 1.086410 4.766730 -1.449960 0.789455 0.051299 -0.000779
    0.611661
2 scene_001.png 1.086390 4.766370 -1.449530 0.789180 0.051881 -0.001131
    0.611966
3 scene_002.png 1.086120 4.765520 -1.449090 0.788982 0.052159 -0.000735
    0.612198
4 ...

```

?? shows images at several moments. It can be seen that the scene is mainly composed of the ground, the table and the debris on the table. If the depth estimate is roughly correct, then we can at least see the difference between the depth value of the table and the ground. Below, we write a dense depth estimation program as explained earlier. For easy understanding, the program is written in C style and placed in a single file. This program is a bit long. In the book, we will focus on several important functions. The rest of the content is asked to read the source code of GitHub.

Listing 12.1: slambook/ch13/dense _monocular/dense _mapping.cpp (fragment)

```

1 #include <iostream>
2 #include <vector>
3 #include <fstream>
4 using namespace std;

```

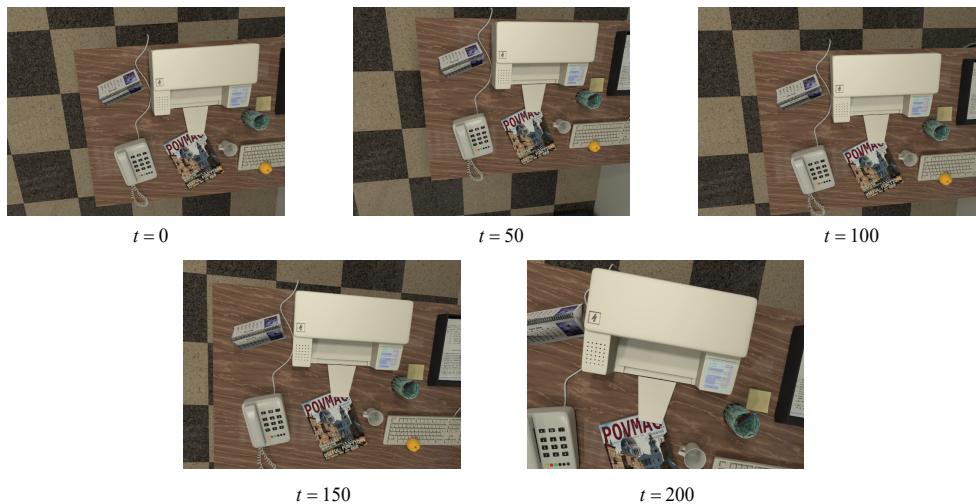


Figure 12-5: Dataset icon.

```

5 #include <boost/timer.hpp>
6
7 // for sophus
8 #include <sophus/se3.h>
9 using Sophus::SE3;
10
11 // for eigen
12 #include <Eigen/Core>
13 #include <Eigen/Geometry>
14 using namespace Eigen;
15
16 #include <opencv2/core/core.hpp>
17 #include <opencv2/highgui/highgui.hpp>
18 #include <opencv2/imgproc/imgproc.hpp>
19
20 using namespace cv;
21
22 / ****
23 * This program demonstrates dense depth estimation of a monocular camera
24 * with a known trajectory
25 * Use epipolar search + NCC matching method, corresponding to section 13.2
26 * of this book
27 * Please note that this program is not perfect, you can improve it
28 * completely.
29 **** /
30
31 // _____
32 // parameters
33 const int boarder=20; // 0 0 0 0
34 const int width=640; // 0 0
35 const int height=480; // 0 0
36 const double fx=481.2f; // 0 0 0 0
37 const double fy=-480.0f;
38 const double cx=319.5f;
39 const double cy=239.5f;
40 const int ncc_window_size=2; // window width taken by NCC
41 const int ncc_area=(2 * ncc_window_size + 1) * (2 * ncc_window_size + 1);
42 // NCC window area
43 const double min_cov=0.1; // convergence judgment: minimum variance

```

```

40 const double max_cov=10; // divergence decision: maximum variance
41 //
42 // _____
43 // important functions
44 // read data from the REMODE data set
45 bool readDatasetFiles (
46 const string & path,
47 vector <string> & color_image_files,
48 vector <SE3> & poses
49 );
50
51 // Update the depth estimate based on the new image
52 bool update (
53 const Mat & ref,
54 const Mat & curr,
55 const SE3 & T_C_R,
56 Mat & depth,
57 Mat & depth_cov
58 );
59
60 // epipolar search
61 bool epipolarSearch (
62 const Mat & ref,
63 const Mat & curr,
64 const SE3 & T_C_R,
65 const Vector2d & pt_ref,
66 const double & depth_mu,
67 const double & depth_cov,
68 Vector2d & pt_curr
69 );
70
71 // Update the depth filter
72 bool updateDepthFilter (
73 const Vector2d & pt_ref,
74 const Vector2d & pt_curr,
75 const SE3 & T_C_R,
76 Mat & depth,
77 Mat & depth_cov
78 );
79
80 // Calculate NCC score
81 double NCC (const Mat & ref, const Mat & curr, const Vector2d & pt_ref,
82 const Vector2d & pt_curr);
83
84 // Bilinear grayscale interpolation
85 inline double getBilinearInterpolatedValue (const Mat & img, const Vector2d
86 & pt){
87 uchar * d=& img.data[int (pt (1,0)) * img.step + int (pt (0,0))];
88 double xx=pt (0,0)-floor (pt (0,0));
89 double yy=pt (1,0)-floor (pt (1,0));
90 return ((1-xx) * (1-yy) * double (d[0]) +
91 xx * (1-yy) * double (d[1]) +
92 (1-xx) * yy * double (d[img.step]) +
93 xx * yy * double (d[img.step + 1]))/255.0;
94 }
95
96 // _____
97 // some gadgets
98 // show estimated depth map
99 bool plotDepth (const Mat & depth);
100
101 // pixel to camera coordinate system
102 inline Vector3d px2cam (const Vector2d px){

```

```

101 return Vector3d (
102 (px (0,0)-cx)/fx,
103 (px (1,0)-cy)/fy,
104 1
105 );
106 }
107
108 // Camera coordinate system to pixels
109 inline Vector2d cam2px (const Vector3d p_cam){
110 return Vector2d (
111 p_cam (0,0) * fx/p_cam (2,0) + cx,
112 p_cam (1,0) * fy/p_cam (2,0) + cy
113 );
114 }
115
116 // detect if a point is inside the image border
117 inline bool inside (const Vector2d & pt){
118 return pt (0,0)>=boarder && pt (1,0)>=boarder
119 && pt (0,0) + boarder < width && pt (1,0) + boarder <= height;
120 }
121
122 int main (int argc, char ** argv)
123 {
124 if (argc!=2)
125 {
126 cout << "Usage: dense_mapping path_to_test_dataset" << endl;
127 return -1;
128 }
129
130 // read data from the dataset
131 vector <string> color_image_files;
132 vector <SE3> poses_TWC;
133 bool ret=readDatasetFiles (argv[1], color_image_files, poses_TWC);
134 if (ret == false)
135 {
136 cout << "Reading image files failed!" << endl;
137 return -1;
138 }
139 cout << "read total" << color_image_files.size () << "files." << endl;
140
141 // first picture
142 Mat ref=imread (color_image_files[0], 0); // gray-scale image
143 SE3 pose_ref_TWC=poses_TWC[0];
144 double init_depth=3.0; // initial depth value
145 double init_cov2=3.0; // initial value of variance
146 Mat depth (height, width, CV_64F, init_depth); // depth map
147 Mat depth_cov (height, width, CV_64F, init_cov2); // depth map variance
148 Ranch
149 for (int index=1; index <color_image_files.size (); index++)
150 {
151 cout << "*** loop" << index << "***" << endl;
152 Mat curr=imread (color_image_files[index], 0);
153 if (curr.data == nullptr) continue;
154 SE3 pose_curr_TWC=poses_TWC[index];
155 SE3 pose_T_C_R=pose_curr_TWC.inverse () * pose_ref_TWC;
156 // Coordinate conversion relationship: T_C_W * T_W_R=T_C_R
157 update (ref, curr, pose_T_C_R, depth, depth_cov);
158 plotDepth (depth);
159 imshow ("image", curr);
160 waitKey (1);
161 }
162 Ranch
163

```

```

164 return 0;
165 }
166
167 // Update the entire depth map
168 bool update (const Mat & ref, const Mat & curr, const SE3 & T_C_R, Mat &
169   depth, Mat & depth_cov)
170 {
171 #pragma omp parallel for
172 for (int x=boarder; x < width-boarder; x++)
173 #pragma omp parallel for
174 for (int y=boarder; y < height-boarder; y++)
175 {
176 // traverse each pixel
177 if (depth_cov.ptr <double> (y)[x] <min_cov
178 || depth_cov.ptr <double> (y)[x] > max_cov) // depth has converged or
179   diverged
180   continue;
181 // search for a match on (x, y) on the epipolar line
182 Vector2d pt_curr;
183 bool ret=epipolarSearch (
184   ref,
185   curr,
186   T_C_R,
187   Vector2d (x, y),
188   depth.ptr <double> (y)[x],
189   sqrt (depth_cov.ptr <double> (y)[x]),
190   pt_curr
191 );
192 Ranch
193 if (ret == false) // match failed
194   continue;
195 Ranch
196 // uncomment this to show matches
197 // showEpipolarMatch (ref, curr, Vector2d (x, y), pt_curr);
198 Ranch
199 // Match successfully, update depth map
200 updateDepthFilter (Vector2d (x, y), pt_curr, T_C_R, depth, depth_cov);
201 }
202 }

203 // epipolar search
204 // For methods, see sections 13.2 and 13.3 of this book
205 bool epipolarSearch (
206   const Mat & ref, const Mat & curr,
207   const SE3 & T_C_R, const Vector2d & pt_ref,
208   const double & depth_mu, const double & depth_cov,
209   Vector2d & pt_curr
210 )
211 {
212 Vector3d f_ref=px2cam (pt_ref);
213 f_ref.normalize ();
214 Vector3d P_ref=f_ref * depth_mu; // P vector of reference frame
215 Ranch
216 Vector2d px_mean_curr=cam2px (T_C_R * P_ref); // pixels projected by mean
217   depth
218 double d_min=depth_mu-3 * depth_cov, d_max=depth_mu + 3 * depth_cov;
219 if (d_min <0.1) d_min=0.1;
220 Vector2d px_min_curr=cam2px (T_C_R * (f_ref * d_min)); // pixels projected
221   at minimum depth
222 Vector2d px_max_curr=cam2px (T_C_R * (f_ref * d_max)); // pixels projected
223   at maximum depth
224 Ranch
225 Vector2d epipolar_line=px_max_curr-px_min_curr; // polar line (line segment

```

```

        form)
222 Vector2d epipolar_direction=epipolar_line; // polar line direction
223 epipolar_direction.normalize ();
224 double half_length=0.5 * epipolar_line.norm (); // half length of the polar
225     line segment
226 if (half_length> 100) half_length=100; // we don't want to search too many
227     things
228 Ranch
229 // Uncomment this sentence to show the epipolar line (segment)
230 // showEpipolarLine (ref, curr, pt_ref, px_min_curr, px_max_curr);
231 Ranch
232 // Search on the epipolar line, centering on the average depth point,
233     taking half length on each side
234 double best_ncc=-1.0;
235 Vector2d best_px_curr;
236 for (double l=-half_length; l <= half_length; l +=0.7) // l +=sqrt (2)/2
237 {
238     Vector2d px_curr=px_mean_curr + l * epipolar_direction; // point to be
239         matched
240     if (! inside (px_curr))
241         continue;
242     // Calculate the NCC of the point to be matched and the reference frame
243     double ncc=NCC (ref, curr, pt_ref, px_curr);
244     if (ncc> best_ncc)
245     {
246         best_ncc=ncc;
247         best_px_curr=px_curr;
248     }
249 }
250
251 double NCC (
252 const Mat & ref, const Mat & curr,
253 const Vector2d & pt_ref, const Vector2d & pt_curr
254 )
255 {
256 // zero mean - normalized cross-correlation
257 // first calculate the mean
258 double mean_ref=0, mean_curr=0;
259 vector <double> values_ref, values_curr; // mean of reference frame and
260     current frame
261 for (int x=-ncc_window_size; x <= ncc_window_size; x++)
262 for (int y=-ncc_window_size; y <= ncc_window_size; y++)
263 {
264     double value_ref=double (ref.ptr <uchar> (int (y + pt_ref (1,0)))[int (x +
265         pt_ref (0,0))])/255.0;
266     mean_ref +=value_ref;
267 Ranch
268     double value_curr=getBilinearInterpolatedValue (curr, pt_curr + Vector2d (x
269         , y));
270     mean_curr += value_curr;
271 Ranch
272     values_ref.push_back(value_ref);
273     values_curr.push_back(value_curr);
274 }
275 Ranch
276     mean_ref /= ncc_area;
277     mean_curr /= ncc_area;
278 Ranch

```

```

277 // 0 0 Zero mean NCC
278 double numerator=0, denominator1=0, denominator2=0;
279 for ( int i=0; i<values_ref.size(); i++ )
280 {
281     double n=(values_ref[i]-mean_ref) * (values_curr[i]-mean_curr);
282     numerator += n;
283     denominator1 += (values_ref[i]-mean_ref)*(values_ref[i]-mean_ref);
284     denominator2 += (values_curr[i]-mean_curr)*(values_curr[i]-mean_curr);
285 }
286 return numerator/sqrt( denominator1*denominator2+1e-10 ); // 0 0 0 0 0 0
287 }
288
289 bool updateDepthFilter(
290     const Vector2d& pt_ref,
291     const Vector2d& pt_curr,
292     const SE3& T_C_R,
293     Mat& depth,
294     Mat& depth_cov
295 )
296 {
297     // 0 0 0 0 0 0 0
298     SE3 T_R_C=T_C_R.inverse();
299     Vector3d f_ref=px2cam( pt_ref );
300     f_ref.normalize();
301     Vector3d f_curr=px2cam( pt_curr );
302     f_curr.normalize();
303 Ranch
304     // 0 0 0 0 0 0 7 0 0 0 0 0
305     Vector3d t=T_R_C.translation();
306     Vector3d f2=T_R_C.rotation_matrix() * f_curr;
307     Vector2d b=Vector2d ( t.dot ( f_ref ), t.dot ( f2 ) );
308     double A[4];
309     A[0]=f_ref.dot ( f_ref );
310     A[2]=f_ref.dot ( f2 );
311     A[1]=-A[2];
312     A[3]=- f2.dot ( f2 );
313     double d=A[0]*A[3]-A[1]*A[2];
314     Vector2d lambdavec=
315         Vector2d ( A[3] * b ( 0,0 ) - A[1] * b ( 1,0 ),
316                     -A[2] * b ( 0,0 ) + A[0] * b ( 1,0 ) ) /d;
317     Vector3d xm=lambdavec ( 0,0 ) * f_ref;
318     Vector3d xn=t + lambdavec ( 1,0 ) * f2;
319     Vector3d d_esti=( xm+xn )/2.0; // 0 0 0 0 0 0 0 0
320     double depth_estimation=d_esti.norm(); // 0 0
321 Ranch
322     // 0 0 0 0 0 0 0 0 0 0 0 0
323     Vector3d p=f_ref*depth_estimation;
324     Vector3d a=p - t;
325     double t_norm=t.norm();
326     double a_norm=a.norm();
327     double alpha=acos( f_ref.dot(t)/t_norm );
328     double beta=acos( -a.dot(t)/(a_norm*t_norm));
329     double beta_prime=beta + atan(1/fx);
330     double gamma=M_PI - alpha - beta_prime;
331     double p_prime=t_norm * sin(beta_prime)/sin(gamma);
332     double d_cov=p_prime - depth_estimation;
333     double d_cov2=d_cov*d_cov;
334 Ranch
335     // 0 0 0 0
336     double mu=depth.ptr<double>( int(pt_ref(1,0)) )[ int(pt_ref(0,0)) ];
337     double sigma2=depth_cov.ptr<double>( int(pt_ref(1,0)) )[ int(pt_ref(0,0)) ];

```

```

338 Ranch
339     double mu_fuse=(d_cov2*mu+sigma2*depth_estimation)/( sigma2+d_cov2);
340     double sigma_fuse2=( sigma2 * d_cov2 )/( sigma2 + d_cov2 );
341 Ranch
342     depth.ptr<double>( int(pt_ref(1,0)) )[ int(pt_ref(0,0)) ]=mu_fuse;
343     depth_cov.ptr<double>( int(pt_ref(1,0)) )[ int(pt_ref(0,0)) ]=sigma_fuse2
344     ;
345 Ranch
346     return true;
347 }
348 // █ █ █ █ █ █ █ █

```

If the reader understands the previous section, it is not difficult to read the source code here. Nevertheless, we explain a few key functions:

1. The main function is very simple. It is only responsible for reading the image from the data set, and then handing it to the update function to update the depth map.
2. update function, we traverse each pixel of the reference frame, first find the epipolar match in the current frame, if it can match, then use the result of epipolar match to update the depth map estimation.
3. The principle of epipolar search is roughly the same as that described in the previous section, but some details have been added to the implementation: because the depth value is assumed to follow a Gaussian distribution, we use the mean as the center, and take $\pm 3\sigma$ as the radius , Then look for the epipolar projection in the current frame. Then, iterate through the pixels on this polar line (the step size is an approximate value of $\sqrt{2}/20.7$), and find the highest NCC point as the matching point. If the highest NCC is also lower than the threshold (here take 0.85), the match is considered to have failed.
4. The calculation of NCC uses the de-averaging method, that is, for the image block \mathbf{A}, \mathbf{B} , take:

$$\text{NCC}_z(\mathbf{A}, \mathbf{B}) = \frac{\sum_{i,j} (\mathbf{A}(i,j) - \bar{\mathbf{A}}(i,j)) (\mathbf{B}(i,j) - \bar{\mathbf{B}}(i,j))}{\sqrt{\sum_{i,j} (\mathbf{A}(i,j) - \bar{\mathbf{A}}(i,j))^2 \sum_{i,j} (\mathbf{B}(i,j) - \bar{\mathbf{B}}(i,j))^2}}. \quad (12.11)$$

5. The calculation method of triangulation is consistent with Section 7.5, and the calculation of uncertainty is consistent with the Gaussian fusion method and the previous section.

Although the program is a bit long, I believe the reader can understand it according to the above tips. Let's look at its actual operation effect.

Experimental result

After compiling this program, run with the data set directory as a parameter.*:

* Please note that dense depth estimation is time consuming. If your computer is older, please wait patiently for a while.

```
1 $ build/dense_mapping ~/dataset/test_data
2 read total 202 files.
3 *** loop 1 ***
4 *** loop 2 ***
5 . . .
```

The information output by the program is relatively simple, showing only the number of iterations, the current image and the depth map. Regarding the depth map, we show the result of multiplying the depth value by 0.4—that is, the depth of the pure white point (the value is 1.0) is about 2.5 meters. The darker the color, the smaller the depth value, which is the closer the object is to us. If you actually run the program, you should find that the depth estimation is a dynamic process—a process that gradually converges from a less certain initial value to a stable value. Our initial value uses a distribution with a mean and variance of 3.0. Of course, you can also modify the initial distribution to see how it affects the results.

From ??, it can be found that when the number of iterations exceeds a certain value, the depth map stabilizes and no new data is changed. Looking at the depth map after stabilization, we find that the difference between the floor and the table can be roughly seen, while the depth of objects on the table is close to the table. The whole estimate is mostly correct, but there are also a large number of incorrect estimates. They appear as inconsistencies in the depth map with the surrounding data and are over- or under-estimated. In addition, the location at the edge, because it is seen less frequently during exercise, has not been accurately estimated. In summary, we think that most of this depth map is correct, but it does not achieve the expected results. We will analyze the reasons for these situations in the next section and discuss what can be improved.

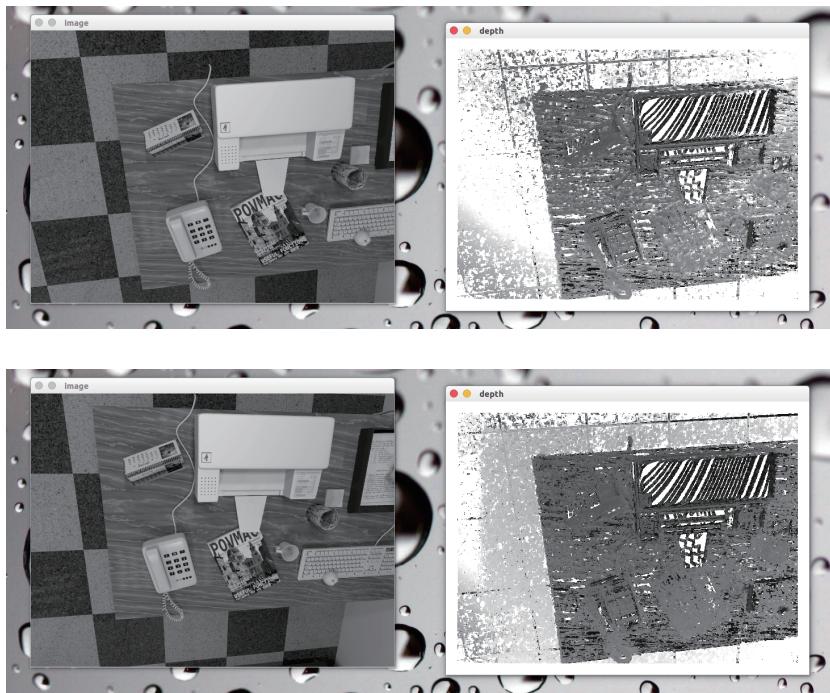


Figure 12-6: Screenshot while the demo program is running. The two graphs are the results of 10 and 30 iterations, respectively.

12.4 Experimental Analysis and Discussion

In the previous section we demonstrated a dense map of a mobile monocular camera, estimating the depth of each pixel of the reference frame. Our code is relatively simple and straightforward, without using many tricks, so common situations occur in practical engineering-simple is often not the most effective.

Due to the complexity of real data, programs that can work in the actual environment often require careful consideration and a lot of engineering skills, which makes every practical code extremely complex-they are difficult to explain to beginners, so We had to use a less efficient, but relatively readable and writeable implementation. Of course, we can put forward several suggestions for improving the demo program, but we do not intend to present the modified (very complicated) program directly to the reader.

Below we make a preliminary analysis of the results of the experiments in the previous section. We will analyze the results of the demonstration experiments from the perspectives of computer vision and filters.

12.4.1 The problem with pixel gradients

Observing the depth image, we will find an obvious fact. The correctness of the block matching depends on whether the image blocks are distinguishable. Obviously, if the image block is only black or white and lacks valid information, then in the NCC calculation, we are likely to mismatch it with a surrounding pixel. The reader is invited to observe the printer surface in the demo program. Because it is uniform white, it is very easy to cause mismatch, so the depth information of the printer surface is mostly incorrect-the space surface of the sample program has a streak-like depth estimate that should not be. According to our intuitive imagination, The printer surface must be smooth.

There is a problem involved here, which we have seen once in the direct method. When performing block matching (and NCC calculations), we must assume that the small block is unchanged, and then compare the small block with other small blocks. At this time, the small blocks with **obvious gradient** will have good discrimination, and it is not easy to cause mismatch. For **pixels with inconspicuous gradients**, since there is no discriminativeness in block matching, it will be difficult to effectively estimate its depth. Conversely, where the pixel gradient is obvious, the depth information we get is relatively accurate, such as magazines on the desktop, phones, and other objects with obvious **texture**. Therefore, the demo program reflects a very common problem in stereo vision: **dependence on object texture**. This problem is also extremely common in binocular vision, which shows that the reconstruction quality of stereo vision is very dependent on the environment texture.

Our demo program deliberately used well-textured environments, such as checkerboard-like floors, wood-grained tabletops, etc., and thus got a seemingly good result. However, in practice, places with uniform brightness such as walls and smooth objects will often appear, affecting our estimation of its depth. In a way, the problem is **cannot be improved and solved on the existing algorithm flow**-if we still only care about the neighborhood (small block) around a pixel.

Further discussing the pixel gradient problem, we will also find the connection between the pixel gradient and the epipolar line. The article [?] discussed their relationship in detail, but it is also intuitively reflected in our demo program.

Taking ?? as an example, we will give two more extreme cases: the pixel gradient

is parallel to the polar line direction, and the pixel gradient is perpendicular to the polar line direction. Let's look at the vertical situation first. In the vertical example, even if the small blocks have obvious gradients, when we do block matching along the epipolar line, we will find that the matching degrees are all the same, so no effective match can be obtained. Conversely, in the parallel example, we can accurately determine where the highest matching point appears. In practice, the gradient and the epipolar line are likely to be in between: they are neither completely vertical nor completely parallel. At this time, we say that when the angle between the pixel gradient and the epipolar line is large, the uncertainty of the epipolar line matching is large; and when the angle is small, the uncertainty of the matching becomes small. In the demo program, we uniformly treat these situations as a pixel error, which is actually not fine enough. Considering the relationship between epipolar lines and pixel gradients, a more accurate uncertainty model should be used. Specific adjustments and improvements are left as exercises.

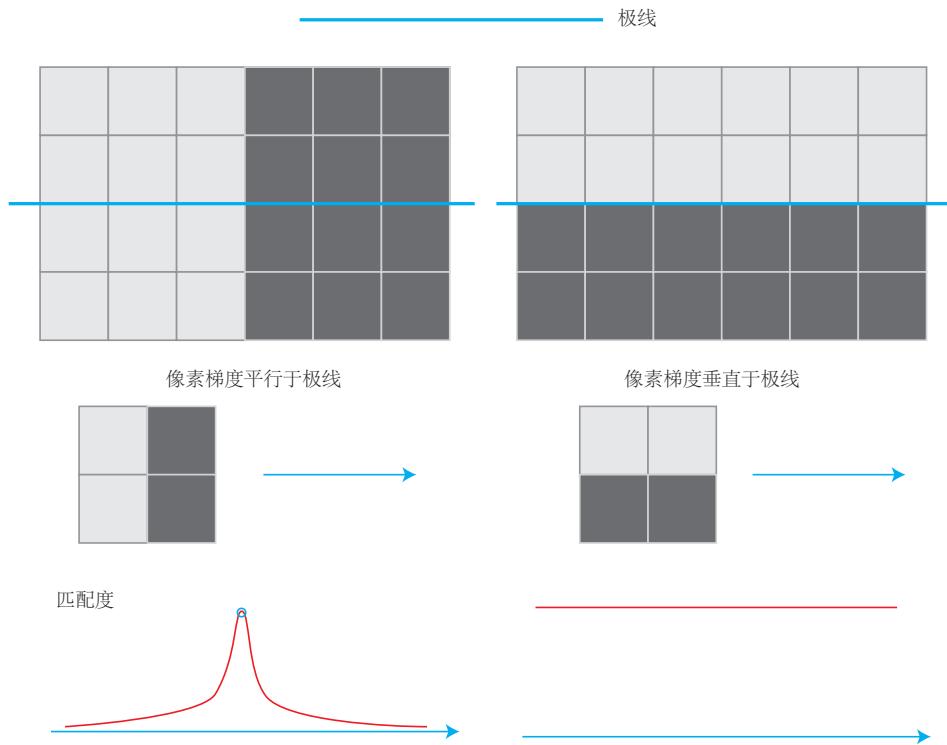


Figure 12-7: The relationship between pixel gradient and epipolar line.

12.4.2 inverse depth

From another perspective, we may ask: Is it appropriate to assume pixel depth as a Gaussian distribution? This relates to a parameterization problem (Parameterization).

In the previous content, we often described a point by its three world coordinates x, y, z , which is a parameterized form. We think that the three quantities x, y, z are random, and they obey the (three-dimensional) Gaussian distribution. However, this lecture uses the image coordinates u, v and the depth value d to describe a certain spatial point (that is, a densely constructed map). We think that u, v are not moving, and d follows a (one-dimensional) Gaussian distribution, which is another form of parameterization. Then we have to ask: Is there any difference between the two parameterization forms? Can we also assume that u, v obey the Gaussian distribution and form another parametric form?

Different parameterized forms actually describe the same quantity, that is, a certain three-dimensional space point. Considering that when we see a point on the camera, its image coordinates u, v are relatively certain. The uncertainty of u, v is very uncertain. At this time, if the world coordinates x, y, z are used to describe this point, then depending on the current pose of the camera, there may be a clear correlation between the three quantities x, y, z . Reflected in the covariance matrix, the non-diagonal elements are not zero. And if you parameterize a point with u, v, d , then its u, v , and d are at least approximately independent, and we can even think that u, v is also independent-so it The covariance matrix is approximated by a diagonal matrix, which is more concise.

Inverse depth is a widely used parameterization technique [? ?] that has appeared in SLAM research in recent years. In the demo program, we assume that the depth value satisfies a Gaussian distribution: $d \sim N(\mu, \sigma^2)$. But isn't this reasonable? Does the depth really approximate a Gaussian distribution? If you think about it, there are some problems with the normal distribution of depth:

1. What we actually want to express is: the depth of this scene is about 5 ~10 meters, there may be some more distant points, but the distance is definitely not less than the focal length of the camera (or the depth is not considered to be less than 0). This distribution does not form a symmetrical shape like the Gaussian distribution. Its tail may be slightly longer, while the negative range is zero.
2. In some outdoor applications, there may be points that are very far away or even infinitely far away. It is difficult to cover these points in our initial values, and describing them with a Gaussian distribution will have some numerical computational difficulties.

Therefore, the inverse depth came into being. People have found in simulations that it is more efficient to assume that the inverse of the depth, which is **inverse depth**, is Gaussian distribution [?]. Subsequently, in practical applications, the inverse depth also has better numerical stability, which has gradually become a general technique that exists in standard practices in existing SLAM schemes [? ? ?].

It is not complicated to change the demo program from positive depth to reverse depth. As long as in the derivation of the previous depth, change d to the inverse depth d^{-1} . We also leave this change as an exercise for the reader to complete.

12.4.3 Transform between images

Before block matching, it is also a common preprocessing method to do an image-to-image transformation. This is because we assume that the image patches remain the

* u, v depends on the resolution of the image.

same as the camera moves, and this assumption holds when the camera is panned (the sample data set is basically this example), but when the camera undergoes a significant rotation, it is difficult to continue. In particular, when the camera is rotated around the light center, an image with lower black and white may become an image with upper black and lower white, causing the correlation to directly become negative (although still the same block).

To prevent this, we usually need to take into account the motion between the reference frame and the current frame before block matching. According to the camera model, a pixel \mathbf{P}_R on the reference frame has the following relationship with the real three-dimensional point world coordinate \mathbf{P}_W :

$$d_R \mathbf{P}_R = \mathbf{K} (\mathbf{R}_{RW} \mathbf{P}_W + \mathbf{t}_{RW}). \quad (12.12)$$

Similarly, for the current frame, there is also a projection of \mathbf{P}_W on it, which is recorded as \mathbf{P}_C :

$$d_C \mathbf{P}_C = \mathbf{K} (\mathbf{R}_{CW} \mathbf{P}_W + \mathbf{t}_{CW}). \quad (12.13)$$

Substituting and eliminating \mathbf{P}_W , we get the pixel relationship between the two images:

$$d_C \mathbf{P}_C = d_R \mathbf{K} \mathbf{R}_{CW} \mathbf{R}_{RW}^T \mathbf{K}^{-1} \mathbf{P}_R + \mathbf{K} \mathbf{t}_{CW} - \mathbf{K} \mathbf{R}_{CW} \mathbf{R}_{RW}^T \mathbf{K} \mathbf{t}_{RW}. \quad (12.14)$$

When d_R, \mathbf{P}_R is known, the projection position of \mathbf{P}_C can be calculated. At this time, by giving \mathbf{P}_R two additional components, du, dv , you can get the increase of \mathbf{P}_C . The amount du_c, dv_c . In this way, an affine transformation is formed by calculating a linear relationship between the reference frame and the current frame image coordinate transformation in the local range:

$$\begin{bmatrix} du_c \\ dv_c \end{bmatrix} = \begin{bmatrix} \frac{du_c}{du} & \frac{du_c}{dv} \\ \frac{dv_c}{du} & \frac{dv_c}{dv} \end{bmatrix} \begin{bmatrix} du \\ dv \end{bmatrix} \quad (12.15)$$

According to the affine transformation matrix, we can transform the pixels of the current frame (or reference frame) and then perform block matching in order to obtain a better effect on rotation.

12.4.4 Parallelization: A Question of Efficiency

We have also seen in experiments that the estimation of dense depth maps is very time-consuming, because the points we need to estimate have suddenly changed from hundreds of feature points to hundreds of thousands of pixels, even now mainstream CPUs have It is impossible to calculate such a huge amount in real time. However, this problem also has another property: the depth estimates of these hundreds of thousands of pixels are independent of each other! This makes parallelization useful.

In the sample program, we traverse all pixels in a double loop and search them epipolarly one by one. When we use the CPU, the process is serial: the next pixel must be calculated after the previous pixel has been calculated. However, in fact, it is not necessary for the next pixel to wait for the calculation of the previous pixel to complete, because there is no obvious relationship between them, so we can use multiple threads to calculate each pixel separately and then unify the results. In theory, if we have 300,000 threads, the calculation time for this problem is the same as calculating one pixel.

GPU's parallel computing architecture is very suitable for such problems. Therefore, in single-dual and dual-purpose dense reconstruction, it is often seen to use

GPU for parallel acceleration. Of course, this book is not intended to involve GPU programming, so we only point out the possibility of using GPU acceleration here, and the specific practice is left to the reader for verification. According to some similar work, using GPU's dense depth estimation can be real-time on mainstream GPUs.

12.4.5 Other improvements

In fact, we can propose many improvements to this routine, such as:

1. Now each pixel is calculated completely independently, there may be situations where the pixel depth is small and the edge is large. We can assume that the adjacent depth changes in the depth map will not be too large, thus adding a spatial regular term to the depth estimation. This approach makes the resulting depth map smoother.
2. We do not explicitly deal with the case of Outlier. In fact, due to various factors such as occlusion, lighting, motion blur, it is impossible to maintain a successful match for each pixel. In the practice of the demo program, as long as the NCC is greater than a certain value, a successful match is considered to have occurred, and no mismatch is considered. Ranch There are several ways to handle mismatches. For example, the depth filter under the uniform-Gaussian mixture distribution proposed by the literature [?], which explicitly distinguishes the inner point from the outer point and performs probabilistic modeling, can better deal with the outer point data. However, this type of filter theory is more complicated. I don't want to cover too much in this book, and readers can read the original paper.

As can be seen from the above discussion, there are many possible improvements. If we improve each step carefully, in the end, we hope to get a good dense construction plan. However, as we discussed, there are some problems **there are theoretical difficulties**, such as the dependence on the environment texture, and the correlation between the pixel gradient and the epipolar direction (and the parallel case). These issues **difficult to solve by tweaking the code implementation**. So, so far, although binocular and mobile mono can build dense maps, we usually think that they are too dependent on the environment texture and lighting and are not reliable enough.

12.5 RGB-D Dense Construction

In addition to using monocular and binocular for dense reconstruction, the RGB-D camera is a better choice in scope. The depth estimation problem discussed in detail in the previous lecture can be obtained completely by hardware measurement in the sensor in the RGB-D camera, without consuming a large amount of computing resources to estimate. In addition, the structured light or flying time principle of RGB-D ensures the independence of depth data on texture. Even when facing a solid-colored object, as long as it can reflect light, we can measure its depth. This is also a great advantage of RGB-D sensors.

It is relatively easy to make dense maps using RGB-D. However, depending on the map format, there are several different mainstream mapping methods. The most

intuitive and simple method is to convert the RGB-D data into a Point Cloud based on the estimated camera pose, and then stitch them together to get a Point Cloud Map composed of discrete points.). On this basis, if we have further requirements on the appearance, and we want to estimate the surface of the object, we can use triangle mesh (Surfel) to construct the map. On the other hand, if you want to know the obstacle information of the map and navigate on the map, you can also create an Occupancy Map through Voxel.

We seem to have introduced many new concepts. Readers are requested not to worry, we will slowly introduce them one by one. For those who are suitable for experiments, we will provide several demo programs as usual. Since there is not much theoretical knowledge involved in RGB-D mapping, the following sections will be introduced directly in the practical part. GPU mapping is beyond the scope of this book, we will briefly explain its principle without making a demonstration.

12.5.1 Practice: Point Cloud Map

First, let's explain the simplest point cloud map. The so-called point cloud is a map represented by a set of discrete points. The most basic point contains the three-dimensional coordinates of x, y, z , and can also carry the color information of r, g, b . Since RGB-D cameras provide color and depth maps, it is easy to calculate RGB-D point clouds based on camera internal parameters. If the pose of the camera is obtained by some means, as long as the point clouds are directly added, the global point cloud can be obtained. In the ?? section of this book, an example of splicing point clouds through internal and external parameters of the camera was given. However, that example is mainly for the reader to understand the internal and external parameters of the camera, and in the actual mapping, we will also add some filtering to the point cloud to obtain better visual effects. In this program, we mainly use two types of filters: out-point removal filters, and down-sampling filters. The code of the sample program is as follows (because part of the code is the same as before, we mainly look at the changed part):

Listing 12.2: slambook/ch13/dense_RGBD/pointcloud_mapping.cpp

```

1 int main( int argc, char** argv )
2 {
3     // ...
4     // ...
5     // ...
6     // ...
7     // ...
8     // ...
9     PointCloud::Ptr pointCloud( new PointCloud );
10    for ( int i=0; i<5; i++ )
11    {
12        PointCloud::Ptr current( new PointCloud );
13        cout<<"\n" : "<<i+1<<endl";
14        cv::Mat color=colorImgs[i];
15        cv::Mat depth=depthImgs[i];
16        Eigen::Isometry3d T=poses[i];
17        for ( int v=0; v<color.rows; v++ )
18        for ( int u=0; u<color.cols; u++ )
19        {
20            unsigned int d=depth.ptr<unsigned short>( v )[u]; // ...
21            if ( d==0 ) continue; // ...
22            if ( d >= 7000 ) continue; // ...
23            Eigen::Vector3d point;

```

```
24     point[2]=double(d)/depthScale;
25     point[0]=(u-cx)*point[2]/fx;
26     point[1]=(v-cy)*point[2]/fy;
27     Eigen::Vector3d pointWorld=T*point;
28 Ranch
29     PointT p ;
30     p.x=pointWorld[0];
31     p.y=pointWorld[1];
32     p.z=pointWorld[2];
33     p.b=color.data[ v*color.step+u*color.channels() ];
34     p.g=color.data[ v*color.step+u*color.channels()+1 ];
35     p.r=color.data[ v*color.step+u*color.channels()+2 ];
36     current->points.push_back( p );
37 }
38 // depth filter and statistical removal
39 PointCloud::Ptr tmp ( new PointCloud );
40 pcl::StatisticalOutlierRemoval<PointT> statistical_filter;
41 statistical_filter.setMeanK(50);
42 statistical_filter.setStddevMulThresh(1.0);
43 statistical_filter.setInputCloud(current);
44 statistical_filter.filter( *tmp );
45 (*PointCloud) += *tmp;
46 }
47 Ranch
48     pointCloud->is_dense=false;
49     cout<<"\n \n \n \n "<<pointCloud->size()<<"\n \n ."<<endl;
50 Ranch
51     // voxel filter
52     pcl::VoxelGrid<PointT> voxel_filter;
53     voxel_filter.setLeafSize( 0.01, 0.01, 0.01 );           // resolution
54     PointCloud::Ptr tmp ( new PointCloud );
55     voxel_filter.setInputCloud( pointCloud );
56     voxel_filter.filter( *tmp );
57     tmp->swap( *PointCloud );
58 Ranch
59     cout<<"\n \n \n \n \n \n \n "<<pointCloud->size()<<"\n \n ."<<endl;
60 Ranch
61     pcl::io::savePCDFileBinary("map.pcd", *PointCloud );
62     return 0;
63 }
```

Our thinking has not changed much. The main differences are:

1. When generating a point cloud per frame, remove points with too large or invalid depth values. This is mainly because the effective range of Kinect is taken into account, and the depth value after the range will have a larger error.
2. Uses statistical filter methods to remove outliers. This filter counts the distribution of distance values between each point and the N points closest to it, and removes points with an excessively large mean distance. In this way, we keep those "sticky" points and remove isolated noise points.
3. Finally, the Voxel Filter is used for downsampling. Due to the overlap of visual fields in multiple perspectives, there will be a large number of very close points in the overlapping area. This will take up a lot of memory space in vain. Voxel filtering ensures that there is only one point in a certain size cube (or voxel), which is equivalent to downsampling the three-dimensional space, which can save a lot of storage space.

?? shows the comparison chart before and after filtering. On the left is a point cloud map generated by the lecture 5 program, and on the right is a filtered point cloud map. Looking at the white box, you can see that there are many isolated points caused by noise before filtering. After removing the statistical outliers, we eliminated these noises and made the entire map cleaner. On the other hand, in the voxel filter, we adjusted the resolution to 0.01, which means there is one point per cubic centimeter. This is a relatively high resolution, so we can't feel the difference in the map in the screenshot, but you can see that the number of points has been significantly reduced from the program output (from 900,000 points to 440,000 points, which is half removed about).



Figure 12-8: Comparison chart before and after filtering (you may need to zoom in to see it clearly, or try it yourself on your computer).

The point cloud map provides us with a more basic visual map that allows us to get an overview of what the environment looks like. It is stored in three dimensions, allowing us to quickly browse all corners of the scene and even roam through the scene. A big advantage of point clouds is that they can be efficiently generated directly from RGB-D images without additional processing. Its filtering operation is also very intuitive, and the processing efficiency is acceptable.

However, the use of point clouds to express maps is still very basic, and we may wish to see if the point cloud map can meet the needs of maps mentioned earlier.

1. Positioning requirements: Depends on how the front-end visual odometer is processed. If it is a feature point-based visual odometer, since the feature point information is not stored in the point cloud, it cannot be used for a feature point-based positioning method. If the front end is the ICP of the point cloud, then you can consider performing ICP on the local point cloud to the global point cloud to estimate the pose. However, this requires better accuracy of the global point cloud. In our way of processing point clouds, the point cloud itself is not optimized, so it is not enough.
2. Requirements for navigation and obstacle avoidance: It cannot be used directly for navigation and obstacle avoidance. A pure point cloud cannot represent the information of "there is an obstacle", nor can we make a query in the point cloud of "whether an arbitrary space point is occupied." However, it can be processed on the basis of point clouds to obtain a map form more suitable for navigation and obstacle avoidance.
3. Visualization and interaction: Has basic visualization and interaction capabilities. We can see what the scene looks like and we can walk around in the scene. From a visual point of view, because the point cloud contains only discrete points and no object surface information (such as normals), it is not in line with people's visualization habits. For example, an object in a point cloud map is the same when viewed from the front and from the back, and you can see what is behind it through the object: these are not in line with our daily experience, because we have no information on the surface of the object.

In summary, when we say that a point cloud map is "basic" or "primary", it means that it is closer to the raw data read by the sensor. It has some basic functions, but it is usually used for debugging and basic display, and it is not convenient to use it directly for applications. If we want maps to have more advanced features, point cloud maps are a good starting point. For example, for the navigation function, we can start from a point cloud and build an Occupancy Grid for the navigation algorithm to query whether a point can pass. For another example, the Poisson reconstruction [?] method commonly used in SfM can reconstruct the object mesh map from the basic point cloud to obtain the surface information of the object. In addition to Poisson reconstruction, Surfel is also a way to express the surface of an object. Using the surface element as the basic unit of the map, it can create a beautiful visual map [?].

?? shows an example of Poisson reconstruction and Surfel. You can see that their visual effects are significantly better than pure point cloud mapping, and they can all be constructed from point clouds. Most of the map forms converted from point clouds are provided in the PCL library, and interested readers can further explore the contents of the PCL library. This book, as an introductory material, does not introduce every map form in detail.

12.5.2 octree map

The following introduces a map form that is more commonly used in navigation and has better compression performance: **octree map**.

In the point cloud map, although we have a three-dimensional structure and voxel filtering to adjust the resolution, the point cloud has several obvious defects:



Figure 12-9: Schematic representation of Poisson reconstruction and Surfel.

- Point cloud maps are usually large, so pcd files are also large. An image with 640 pixels \times 480 pixels will generate 300,000 space points and requires a lot of storage space. Even after some filtering, the pcd file is very large. And the annoyance is that its "big" is not necessary. Point cloud maps provide a lot of unnecessary detail. We don't particularly care about the folds on the carpet and the shadows in the dark. Putting them on the map is a waste of space. Because of the space taken up, unless we reduce the resolution, we cannot model a larger environment with limited memory. However, reducing the resolution will cause the map quality to decrease. Is there any way to compress and store the map and discard some duplicate information?
- Point cloud maps cannot handle moving objects. Because we only have "add points" in our approach, but not "remove them when they disappear". In the actual environment, the ubiquity of moving objects makes point cloud maps impractical.

What we are going to introduce next is a flexible, compressed map form that can be updated at any time: Octo-map [?].

We know that it is a common practice to model three-dimensional space into many small squares (or voxels). If we cut each side of a small square into two pieces on average, the small square would become eight small squares of the same size. This step can be repeated until the final block size reaches the highest accuracy for modeling. In this process, the matter of "dividing a small square into eight of the same size" is regarded as "expanding from a node into eight children", then the entire process of subdividing from the largest space to the smallest space is An Octo-tree.

As shown in ??, the left side shows a large cube that is continuously divided evenly into eight pieces until it becomes the smallest one. Therefore, the entire large block can be regarded as the root node, and the smallest block can be regarded as the "leaf node". Therefore, in the octree, when we go up one layer from the next layer node, the volume of the map can be expanded by eight times. Let's do a simple calculation: if the block size of the leaf node is 1 cm^3 , when we limit the octree to 10 layers, the total volume that can be modeled is about $8^{10} \text{ cm}^3 = 1,073 \text{ m}^3$, which is enough to model a house. Because volume and depth have an exponential relationship, when we use larger depths, the modeled volume will grow very quickly.

The reader may be wondering, in the voxel filter of the point cloud, don't we also limit only one point in a voxel? Why do we say that point clouds occupy space, while octrees save space? This is because, in an octree, we store information in a node if it is occupied. However, the difference is that **no need to expand this node**

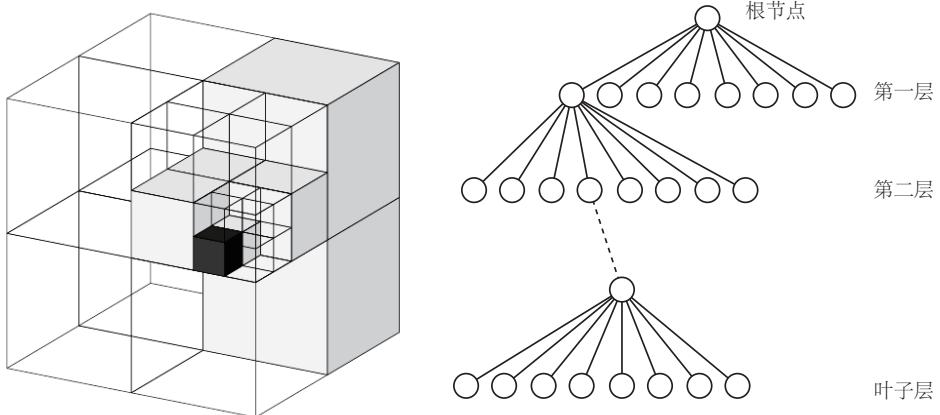


Figure 12-10: Octave schematic.

when all child nodes of a block are occupied or not occupied. For example, when the map is blank at the beginning, we only need a root node instead of a complete tree. When adding information to the map, most of the octree nodes do not need to be expanded to the leaf level because the actual objects are often connected together and the empty places are often connected together. Therefore, octrees save a lot of storage space than point clouds.

Earlier, the nodes of the octree stored information about whether it was occupied. From the point cloud point of view, we can naturally use 0 to indicate blank and 1 to occupy. This 0-1 representation can be stored in one bit to save space, but it seems a bit too simple. Due to the influence of noise, we may see that a point is 0 for a while and 1 for a while; or 0 for most of the time and 1 for a small part of the time; An "unknown" state. Can you describe this in more detail? We will choose to use **probability** to express whether a node is occupied or not. For example, use a floating point number $x \in [0, 1]$. This x starts with 0.5. If it is continuously observed to be occupied, then let this value increase continuously; conversely, if it is continuously observed to be blank, then let it continue to decrease.

In this way, we dynamically model the obstacle information in the map. However, there is a small problem with the current method: if you let x keep increasing or decreasing, it may run outside the $[0, 1]$ range, causing inconvenience in processing. So instead of directly describing a node's occupation with probability, we use log-odds to describe it. Let $y \in \mathbb{R}$ be the logarithm of probability and x be the probability of 0 ~1, then the transformation between them is described by logit transformation:

$$y = \text{logit}(x) = \log\left(\frac{x}{1-x}\right). \quad (12.16)$$

Its inverse transform is

$$x = \text{logit}^{-1}(y) = \frac{\exp(y)}{\exp(y) + 1}. \quad (12.17)$$

It can be seen that when y changes from $-\infty$ to $+\infty$, x changes from 0 to 1. When y takes 0, x takes 0.5. Therefore, we might as well store y to indicate whether the node is occupied. When "occupancy" is continuously observed, let y increase by one value; otherwise, let y decrease by one value. When querying probability, use inverse logit transformation to convert y to probability. In mathematical form, let a node be n and the observation data be z . Then the logarithm of the probability of a node from the beginning to the time of t is $L(n|z_{1:t})$, and the time of $t+1$ is

$$L(n|z_{1:t+1}) = L(n|z_{1:t-1}) + L(n|z_t). \quad (12.18)$$

It would be a little more complicated if written in probabilistic form instead of log-probabilistic form:

$$P(n|z_{1:T}) = \left[1 + \frac{1 - P(n|z_T)}{P(n|z_T)} \frac{1 - P(n|z_{1:T-1})}{P(n|z_{1:T-1})} \frac{P(n)}{1 - P(n)} \right]^{-1}. \quad (12.19)$$

With log probability, we can update the entire octree map based on RGB-D data. Suppose we observe a pixel with a depth of d in an RGB-D image, this illustrates one thing: we observed an occupation data at the spatial point corresponding to the depth value, and from the camera light The line from which the heart starts to this point should be (otherwise it will be blocked). With this information, the octree map can be updated well, and the structure of the motion can be processed.

12.5.3 Practice: Octree Map

The following demonstrates the mapping process of octomap through a program. First, readers are asked to install the octomap library: <https://github.com/OctoMap/octomap>. The Octomap library mainly contains octomap maps and octovis (a visualization program), both of which are cmake projects. It mainly depends on doxygen and can be installed by the following command:

```
1 sudo apt-get install doxygen
```

Let's directly demonstrate how to generate an octree map from the previous 5 images, and then draw it.

Listing 12.3: slambook/ch13/dense_RGBD/octomap_mapping.cpp

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 #include <opencv2/core/core.hpp>
6 #include <opencv2/highgui/highgui.hpp>
7
8 #include <octomap/octomap.h> // for octomap
9
10 #include <Eigen/Geometry>
11 #include <boost/format.hpp> // for formating strings
12
13 int main( int argc, char** argv )
14 {
15     // █ █ █ █ █ █ █ █
16     cout<<"█ █ █ █ █ █ █ Octomap ..."<<endl;
17 Ranch
18     // octomap tree
19     octomap::Octree tree( 0.05 ); // █ █ █ █ █
20 Ranch
21     for ( int i=0; i<5; i++ )
22     {
23         cout<<"█ █ █ █ : "<<i+1<<endl;
24         cv::Mat color=colorImgs[i];
25         cv::Mat depth=depthImgs[i];
26         Eigen::Isometry3d T=poses[i];
27 Ranch

```

```

28     octomap::Pointcloud cloud; // the point cloud in octomap
29 Ranch
30     for ( int v=0; v<color.rows; v++ )
31         for ( int u=0; u<color.cols; u++ )
32     {
33         unsigned int d=depth.ptr<unsigned short> ( v )[u]; // 0 0 0
34         if ( d==0 ) continue; // 0 0 0 0 0 0 0 0 0
35         if ( d >= 7000 ) continue; // 0 0 0 0 0 0 0 0 0
36         Eigen::Vector3d point;
37         point[2]=double(d)/depthScale;
38         point[0]=(u-cx)*point[2]/fx;
39         point[1]=(v-cy)*point[2]/fy;
40         Eigen::Vector3d pointWorld=T*point;
41         // 0 0 0 0 0 0 0 0 0
42         cloud.push_back( pointWorld[0], pointWorld[1], pointWorld[2] );
43     }
44 Ranch
45     // 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
46     tree.insertPointCloud( cloud, octomap::point3d( T(0,3), T(1,3), T(2,3)
47         ) );
48 Ranch
49     // 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
50     tree.updateInnerOccupancy();
51     cout<<"saving octomap ... "<<endl;
52     tree.writeBinary( "octomap.bt" );
53     return 0;
54 }
```

We used octomap :: OcTree to build the entire map. In fact, octomap provides many kinds of octrees: there are maps and occupancy information. You can also define which variables each node needs to carry. For simplicity, we use the most basic octree map without color information.

Ocotmap provides a point cloud structure inside. It is slightly simpler than PCL's point cloud, and only carries the spatial location information of points. Based on the RGB-D image and camera pose information, we first turn the coordinates of the points to world coordinates, then put them into the point cloud of the octomap, and finally hand them to the octree map. After that, octomap will update the internal occupation probability based on the projection information introduced previously, and finally save it as a compressed octree map. We save the generated map as octomap.bt file. When compiling octovis before, we actually installed a visualization program, octovis. Now, call it to open the map file, and you can see what the map actually looks like.

?? shows the results of the map we built. Since we did not add color information to the map, it will be gray when the map is opened at first. Press "1" to color according to the height information. Readers can familiarize themselves with the octovis interface, including viewing, rotating, and zooming maps.

On the right is the depth limit bar of the octree. Here you can adjust the resolution of the map. Since the default depth we use when constructing is 16 layers, here we show 16 layers, which is the highest resolution, that is, the side length of each small block is 0.05 meters. When we reduce the depth by one layer, the leaf nodes of the octree are raised by one layer, and the side length of each small block is doubled to 0.1 meters. As you can see, we can easily adjust the map resolution to suit different occasions.

Octomap also has some places to explore. For example, we can easily query the occupation probability of any point to design a method for navigating in the

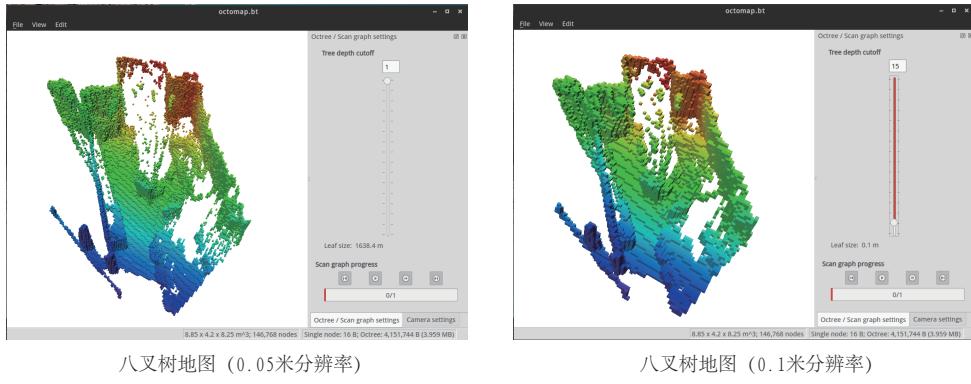


Figure 12-11: Octree display results at different resolutions.

map [?]. Readers can also compare the file sizes of point cloud maps and octree maps. The disk file of the point cloud map generated in the previous section is about 6.9MB, while the octomap is only 56KB, not even one percent of the point cloud map, which can effectively model larger scenes.

12.6 * TSDF maps and Fusion series

At the end of this lecture, we introduce a research direction that is very similar to SLAM but slightly different: real-time 3D reconstruction. This section involves GPU programming, and does not provide reference examples, so it is an optional reading material.

In the previous map model, **focused on positioning**. Map stitching is placed in the SLAM framework as a subsequent processing step. The reason this framework becomes mainstream is that the positioning algorithm can meet the real-time requirements, and the map processing can be processed at key frames without the need for real-time response. Positioning is usually lightweight, especially when using sparse features or sparse direct methods; correspondingly, the representation and storage of maps are heavyweight. Their scale and computational requirements are large and not conducive to real-time processing. Especially dense maps can often only be calculated at the key frame level.

However, in the current practice, we have not optimized dense maps. For example, when the same chair was observed in both images, we only superimposed the two point clouds based on the pose of the two images to generate a map. Because pose estimation is usually with errors, this direct stitching is often not accurate enough, for example, the point clouds of the same chair cannot be superimposed perfectly. At this time, two ghost images of the chair appear in the map-this phenomenon is sometimes called "ghost image".

This phenomenon is obviously not what we want. We want the reconstruction result to be smooth and complete, which is in line with our understanding of the map. Under this kind of thinking, a kind of practice that takes "building a picture" as the main body and locates the secondary position, that is, the real-time 3D reconstruction to be introduced in this section. As the main goal of 3D reconstruction is to reconstruct accurate maps, GPUs need to be used for acceleration, and even very advanced GPUs or multiple GPUs are required for parallel acceleration, usu-

ally requiring heavy computing equipment. In contrast, SLAM is moving towards lightweight and miniaturization. Some solutions even abandon the construction and loopback detection parts, and only retain the visual odometer. And real-time reconstruction is moving towards the reconstruction of large-scale, large-scale dynamic scenes.

Since the emergence of RGB-D sensors, real-time reconstruction using RGB-D images has formed an important development direction. Kinect Fusion [?], Dynamic Fusion [?], Elastic Fusion [?], Fusion4D [?], Volume Deform [?], and more. Among them, Kinect Fusion has completed the basic model reconstruction, but it is limited to small scenes; the subsequent work is to expand it to large, sports, and even deformed scenes. We see them as rebuilding a large class of work in real time, but due to the large variety, it is impossible to discuss in detail how each works. ?? shows part of the reconstruction results. You can see that these modeling results are very fine, much more delicate than just stitching point clouds.

Let's take the classic TSDF map as an example. TSDF is the abbreviation of Truncated Signed Distance Function, which may be translated as **Truncation Signed Distance Function**. Although it seems inappropriate to call "functions" "maps", we will temporarily call them TSDF maps, TSDF reconstructions, etc. until there is no better translation, as long as there is no deviation in understanding.

Similar to octrees, TSDF maps are also a web format (or, in other words, square) maps, as shown in ???. First select the three-dimensional space to be modeled, such as $3 \times 3 \times 3m^3$, divide this space into many small blocks according to a certain resolution, and store the information inside each small block. The difference is that TSDF maps are stored entirely in video memory instead of memory. Using the parallel nature of the GPU, we can compute and update each voxel in parallel, instead of having to serialize as the CPU traverses the memory area.

In each TSDF voxel, the distance between the small block and the nearest object surface is stored. If the patch is in front of the surface of the object, it has a positive value; conversely, if the patch is behind the surface, it is negative. Because the surface of the object is usually a very thin layer, the values that are too large and too small are taken as 1 and -1, so that the distance after truncation is obtained, which is called TSDF. Then by definition, the place where the TSDF is 0 is the surface itself—or, because of the numerical error, the place where the TSDF changes from negative to positive is the surface itself. In the lower part of ??, we see that a surface similar to a human face appears where TSDF changes the symbol.

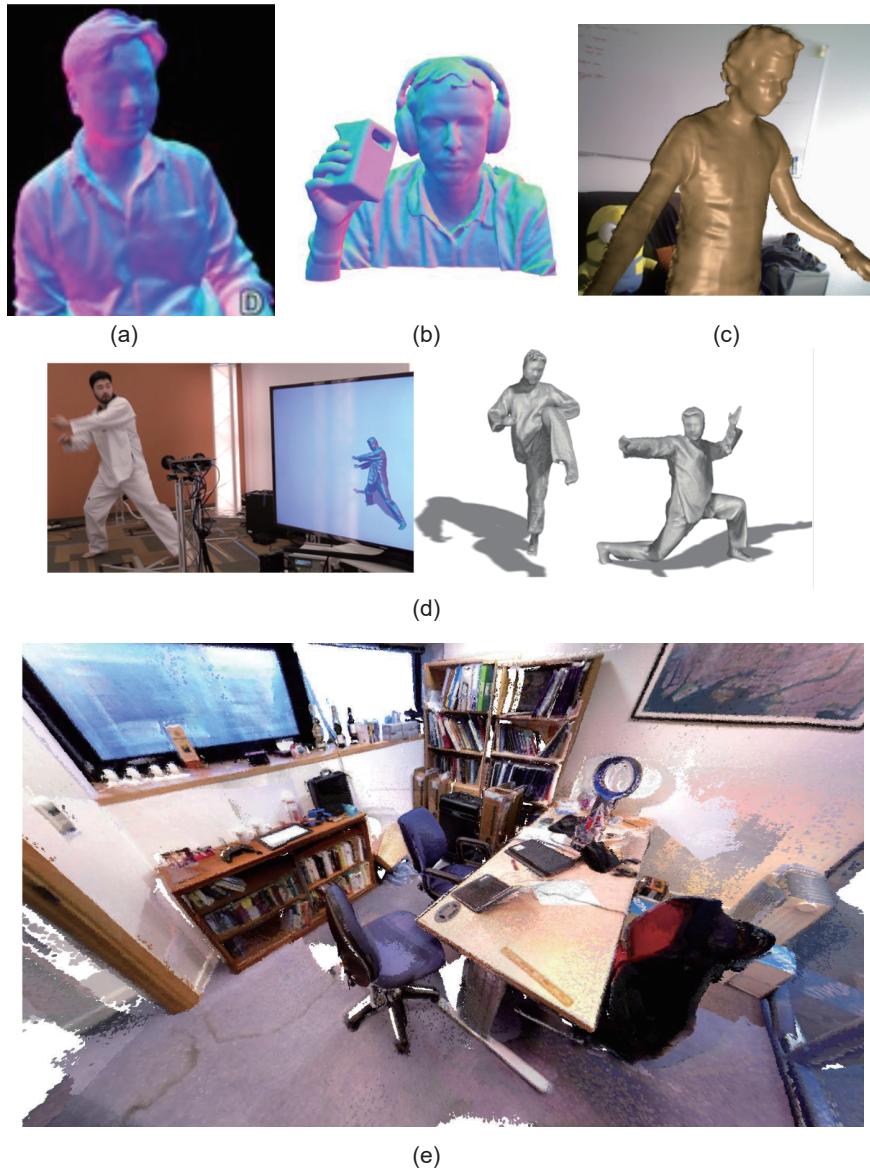
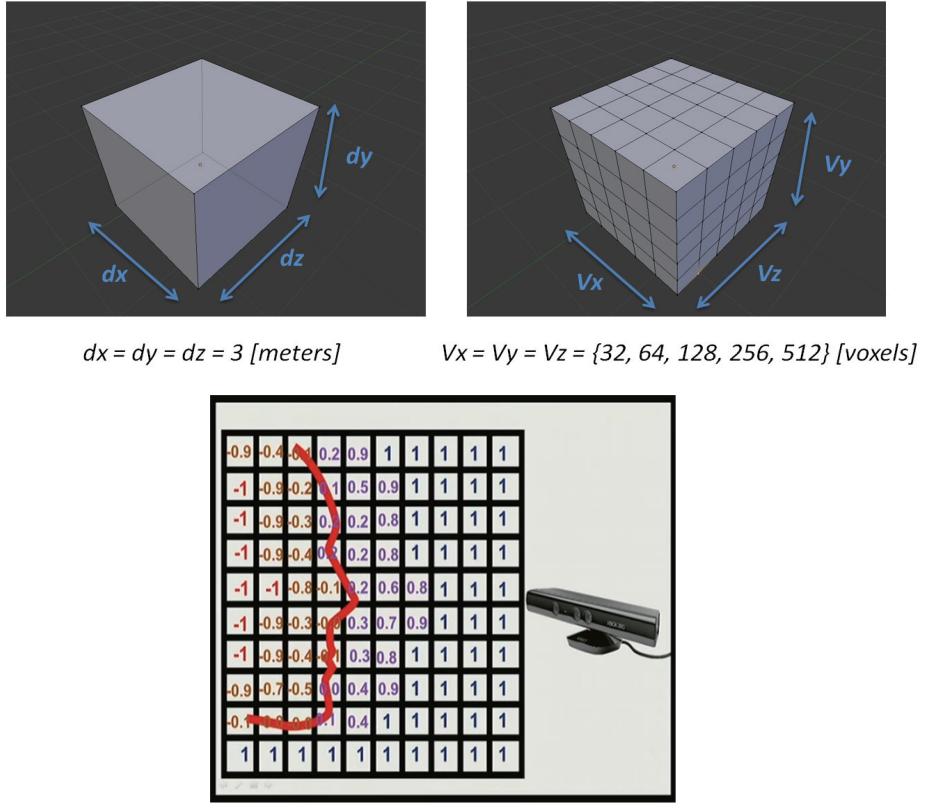


Figure 12-12: Various real-time 3D reconstruction models. (A) Kinect Fusion; (b) Dynamic Fusion; (c) Volume Deform; (d) Fusion4D; (e) Elastic Fusion.



相机观察到物体表面时形成的截断距离值

Figure 12-13: TSDF schematic.

TSDF also has two issues, "positioning" and "mapping", which are very similar to SLAM, but the specific form is slightly different from the previous lectures. Here, the positioning problem mainly refers to how to compare the current RGB-D image with the TSDF map in the GPU to estimate the camera pose. The mapping problem is how to update the TSDF map based on the estimated camera pose. Traditionally, we will also perform bilateral Bayesian filtering on the RGB-D image to remove noise from the depth map.

The positioning of TSDF is similar to the ICP introduced earlier, but due to the parallelization of the GPU, we can perform ICP calculations on the entire depth map and TSDF map without having to calculate feature points first, as in traditional visual odometry. At the same time, because TSDF has no color information, it means that we can **use only the depth map** and not use the color map to complete the pose estimation, which rids the visual mileage calculation method's dependence on lighting and texture to some extent, Making RGB-D reconstruction more robust *. On the other hand, the mapping part is also a process of updating the values in TSDF in parallel, making the estimated surface smoother and more reliable. Since we don't introduce too much GPU-related content, the specific method will not be elaborated. Please refer to relevant literature for interested readers.

* But then again, it is more dependent on depth maps.

12.7 Summary

This lecture introduces some common map types, especially dense map forms. We see that dense maps can be constructed based on monocular or binocular, while RGB-D maps are often easier and more stable. The maps in this lecture focus on metric maps, and the topological map form is relatively different from SLAM research, so it is not discussed in detail.

EXERCISE

1. comprehension (13.6).
2. Change the dense depth estimation in this tutorial to semi-dense. You can filter out the obvious gradients first.
- 3.*Change the monocular dense reconstruction code demonstrated in this lecture from positive depth to inverse depth, and add affine transformation. Has your experiment improved?
4. Can you demonstrate how to do navigation or path planning in an octree?
5. Research Literature [?] explores how TSDF maps perform pose estimation and update. How is it similar to the positioning mapping algorithm we talked about before?
- 6.*Investigate the principle and implementation of a uniform-Gaussian hybrid filter.

Chapter 13

Practice: Designing the front end

main target

1. actually designs a visual odometer front end.
2. Understand how the SLAM software framework is built.
3. Understand issues that tend to arise in front-end design and how to fix them.

This lecture is a relatively rare one that is composed of practical parts. We will use the knowledge from the previous two lectures to actually write a visual odometer program. You will manage local robot trajectories and landmarks, and experience how a software framework is composed. During the operation, we will encounter many problems: too fast camera movement, blurred images, mismatches ... will invalidate the algorithm. In order for the program to run stably, we need to deal with all the above situations, which will bring many useful discussions on the implementation of the project.

13.1 Building VO Framework

Knowing the principles of bricks and cement does not mean that you can build a great palace.

In my favorite "Minecraft" game, players only have some blocks with different colors and textures. Its nature is extremely simple, and all the player has to do is place these blocks on the open space. Understanding a block is simple, but when you actually pick them up, beginners often can only build simple matchbox houses, and experienced and creative players can use these simple blocks to build houses, gardens, and pavilions. Even the city (??) *.

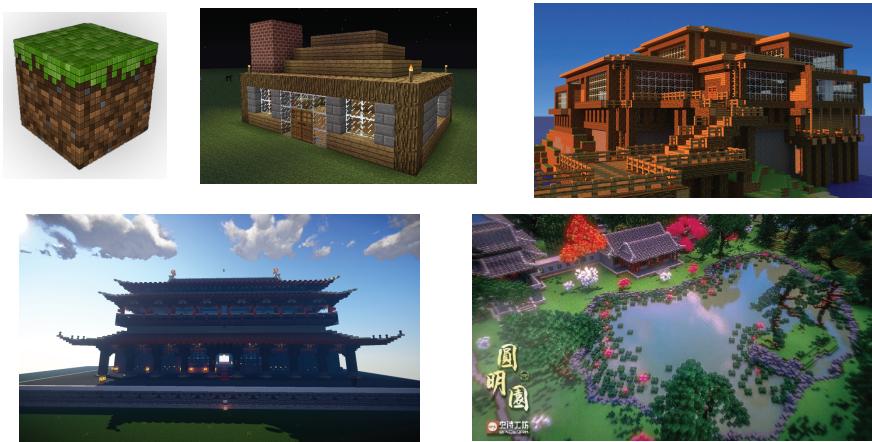


Figure 13-1: Starting from simple things, gradually build more and more complex but better works.

In SLAM, we believe that engineering implementation and understanding of algorithm principles should be at least equally important, and even more emphasis should be placed on how to write practically usable programs. The principles of algorithms are like blocks, we can clearly and clearly discuss their principles and properties, but just understanding a block does not make you build a real building; they require a lot of trial, time, and experience. We Encourage readers to work in a more practical direction-this is often very complicated. Just like in Minecraft, you need to master the structure of various columns, walls, and roofs, the carvings on the walls, and the calculation of the angles of geometric shapes. These are far from as simple as discussing the nature of each block.

The same is true of the specific implementation of SLAM. A practical program will have a lot of engineering design and tricks. It also needs to discuss how to deal with problems after each step. In principle, everyone's implementation of SLAM will be different, and most of the time we can't say which implementation is necessarily the best. However, we usually encounter some common problems: "how to manage map points", "how to deal with mismatches", "how to select key frames", and so on. We hope that readers will have some intuitive feelings about these possible problems-we think this feeling is very important.

* The lower left is the author's practice work. The bottom right comes from the Epicwork team work: "Yuanmingyuan".

Therefore, out of the emphasis on practice, in this chapter we will guide the reader through the process of building a SLAM framework. Just like architecture, we have to discuss trivial but important issues such as column spacing and aspect ratio of the facade. SLAM engineering is complicated. Even if we keep only the core parts, it will take up a lot of space and make this book too verbose. However, please note that although the project after completion is complicated, the intermediate "from simple to complex" process is worthy of detailed discussion and learning value. Therefore, we need to start with a simple data structure, first make a simple visual odometer, and then slowly add some additional features. In other words, we need to show the **from simple to complex** process to the reader, so that you can understand how a library slowly piles up like a snowman.

The code for this tutorial is in `slambook/project`. As the development process continues, we will make some changes to the project, so its content will also change. Therefore, we will also keep the intermediate code in the directory, named after the version number, so that readers can view and imitate at any time.

13.1.1 determining the program frame

According to the contents of the first two lectures, we know that there are three types of visual mileometer: monocular, binocular, and RGB-D. Monocular vision is relatively complicated, and RGB-D is the simplest, without initialization and scale problems. Based on the simple and complex guideline, let's start with RGB-D. In order to facilitate the reader's experiments, we will use the data set instead of the actual RGB-D camera (because the reader cannot guarantee a RGB-D camera).

First, let's take a look at how Linux programs are organized. When writing a small-scale library, we usually create some folders and store the source code, header files, documents, test data, configuration files, logs, and other categories in a organized manner. If a library contains a lot of content, we will also break the code into separate small modules for testing. Readers can refer to the organization of OpenCV or g2o to see how a large and medium-sized library is organized. For example, OpenCV has modules such as core, imgproc, and features2d, and each module is responsible for different tasks. g2o has several modules such as core, solvers, and types. But in small programs, we can also put everything together, called the SLAM library.

The SLAM library we are writing now is a small library. The goal is to help readers to integrate the various algorithms used in this book and write their own SLAM programs. Pick a project directory and create the following folders under it to organize the code files:

1. bin is used to store executable binaries.
2. include/myslam Stores the header files of the SLAM module, mainly .h files. The reason for this is that when you set the include directory to include and reference your own header files, you need to write include " myslam/xxx.h ", which is not easy to be confused with other libraries.
3. src Stores source code files, mainly .cpp files.
4. test Stores test files, which are also .cpp files.
5. lib Holds compiled library files.

6. config stores configuration files.
7. cmake _modules cmake files for third-party libraries, which are used when using libraries like g2o.

The above is our directory structure, as shown in ?? . Compared with the scattered main.cpp in each of the previous lectures, this approach seems more organized. Next, we will continue to add new files to these directories, gradually forming a complete program.

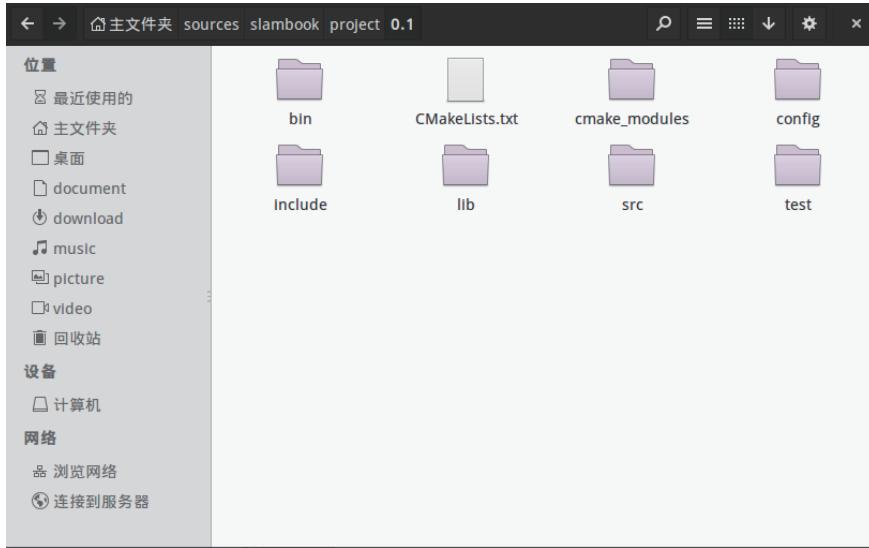


Figure 13-2: Directory of the project.

13.1.2 Determine the basic data structure

In order to make the program run, we need to design the data unit and the process of the program. This is like the pillars and bricks that make up a house. So, what are the most basic structures in a SLAM program? We abstract the following basic concepts:

1. **frame:** A frame is a unit of image captured by the camera. It mainly contains one image (a pair of images in the case of RGB-D). In addition, there are information such as feature points, poses, and internal parameters. Hell In visual SLAM we will talk about key-frames. Because the camera collects a lot of data, it is obviously impractical to store all the data. Otherwise, if the camera is left on the table, the memory usage of the program will become higher and higher until it is unacceptable. The common practice is to save some frames that we think are more important, and think that the camera track can be described by these key frames. How to choose key frames is a big problem, and based on engineering experience, there is little theoretical guidance. We will use a key frame selection method in this book, but readers can also consider coming up with new methods on their own. Hell
2. **Signpost:** Signpost points are feature points in the image. After camera movement, we can also estimate their 3D position. Usually, the landmark

points are placed in a **map**, and the new frame is matched with the landmark points in the map to estimate the camera pose.

The estimation of the pose and position of the frame is equivalent to a local SLAM problem. In addition, we need some tools to make the program write more smoothly. E.g:

1. **configuration file:** During the process of writing a program, you will often encounter various parameters, such as the camera's internal parameters, the number of feature points, the proportion selected during matching, and so on. You can write these numbers in the program, but that is not a good habit. You will often modify these parameters, but you must recompile the program after each modification. As its number increases, it becomes more difficult to modify. Therefore, a better way is to define a configuration file externally, and read the parameter values in the configuration file when the program runs. In this way, you only need to modify the content of the configuration file each time, without having to make any changes to the program itself.
2. **Coordinate Transformation:** You will often need to perform coordinate transformations between coordinate systems, for example, world coordinates to camera coordinates, camera coordinates to normalized camera coordinates, normalized camera coordinates to pixel coordinates, and so on. It would be more convenient to define a class to put these operations together.

Here we define the concepts of frames and road signs, which are represented by classes in C++. We try to ensure that a class has separate header and source files, and avoid putting many classes in the same file. Then, put the function declaration in the header file and the implementation in the source file (unless the function is short, it can also be written in the header file). We follow Google's naming conventions and consider writing programs in a way that even beginners can understand. Since our program is biased towards algorithms rather than software engineering, we do not discuss complex class inheritance relationships, interfaces, templates, etc., but focus more on **the correct implementation of the algorithm and whether it is easy to extend**. We will make the data members public, although this should be avoided in C++ software design. If the reader wants to, you can also change them to private or protected interfaces, and add settings and get interfaces. In a more complex algorithm, we will break it into several steps. For example, feature extraction and matching should be implemented in different functions. In this way, when we want to modify the algorithm flow, we do not need to modify the entire operation flow. Need to adjust the local processing method.

Now let's start writing VO. We set this version as 0.1, indicating that this is the beginning stage. We write a total of 5 classes: Frame is the frame, Camera is the camera model, MapPoint is the feature point/landmark point, Map manages the feature point, and Config provides configuration parameters. Their relationship is shown in ?? . We only write their data members and common methods now, and add them later when more content is used.

The Camera class is the simplest, let's implement it first.

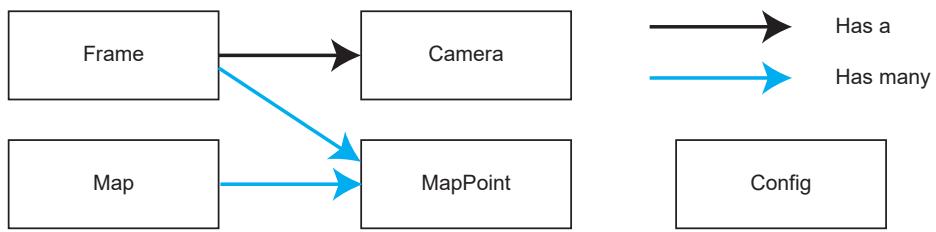


Figure 13-3: Schematic diagram of basic class relationships.

13.1.3 Camera class

The Camera class stores the internal and external parameters of the camera, and completes the coordinate transformation between the camera coordinate system, the pixel coordinate system, and the world coordinate system. Of course, in the world coordinate system you need a camera (variable) external parameter, which we pass in as a parameter.

Listing 13.1: slambook/project/0.1/include/myslam/camera.h

```

1 #ifndef CAMERA_H
2 #define CAMERA_H
3 #include "myslam/common_include.h"
4 namespace myslam
5 {
6 // Pinhole RGB-D camera model
7 class Camera
8 {
9 public:
10    typedef std::shared_ptr<Camera> Ptr;
11    float fx_, fy_, cx_, cy_, depth_scale_; // Camera intrinsics
12
13    Camera();
14    Camera( float fx, float fy, float cx, float cy, float depth_scale=0 ) :
15        fx_( fx ), fy_( fy ), cx_( cx ), cy_( cy ), depth_scale_(
16            depth_scale )
17    {}
18
19    // coordinate transform: world, camera, pixel
20    Vector3d world2camera( const Vector3d& p_w, const SE3& T_c_w );
21    Vector3d camera2world( const Vector3d& p_c, const SE3& T_c_w );
22    Vector2d camera2pixel( const Vector3d& p_c );
23    Vector3d pixel2camera( const Vector2d& p_p, double depth=1 );
24    Vector3d pixel2world ( const Vector2d& p_p, const SE3& T_c_w, double
25        depth=1 );
26    Vector2d world2pixel ( const Vector3d& p_w, const SE3& T_c_w );
27 };
28 }
29 #endif // CAMERA_H

```

The description is as follows (from top to bottom):

1. In this simple example, we give an ifndef macro definition that prevents duplicate references in header files. Without this macro, duplicate definitions of the class would occur when the header file was referenced in two places. Therefore, such a macro is defined in each program header file.
2. We wrap the class definition with the namespace namespace myslam (because it is our own SLAM, the namespace is called myslam). Namespaces can prevent us from accidentally defining functions with the same name in other libraries. Since the macro definitions and namespaces are written once in each file, we will only introduce them here and omit them later.
3. We put some common header files in the common _include.h file, so we can avoid writing a long list of include each time.
4. We define smart pointers as Camera pointer types, so use Camera :: Ptr when passing parameters in the future.

5. We use Sophus :: SE3 to express the pose of the camera. The Sophus library was introduced in Lie Algebra.

In the source file, give the implementation of the Camera method:

Listing 13.2: slambook/project/0.1/src/camera.cpp

```

1 #include "myslam/camera.h"
2 namespace myslam
3 {
4
5     Camera::Camera()
6     {
7     }
8
9     Vector3d Camera::world2camera ( const Vector3d& p_w, const SE3& T_c_w )
10    {
11        return T_c_w*p_w;
12    }
13
14    Vector3d Camera::camera2world ( const Vector3d& p_c, const SE3& T_c_w )
15    {
16        return T_c_w.inverse() *p_c;
17    }
18
19    Vector2d Camera::camera2pixel ( const Vector3d& p_c )
20    {
21        return Vector2d (
22            fx_ * p_c ( 0,0 )/p_c ( 2,0 ) + cx_,
23            fy_ * p_c ( 1,0 )/p_c ( 2,0 ) + cy_
24        );
25    }
26
27    Vector3d Camera::pixel2camera ( const Vector2d& p_p, double depth )
28    {
29        return Vector3d (
30            ( p_p ( 0,0 )-cx_ ) *depth/fx_,
31            ( p_p ( 1,0 )-cy_ ) *depth/fy_,
32            depth
33        );
34    }
35
36    Vector2d Camera::world2pixel ( const Vector3d& p_w, const SE3& T_c_w )
37    {
38        return camera2pixel ( world2camera ( p_w, T_c_w ) );
39    }
40
41    Vector3d Camera::pixel2world ( const Vector2d& p_p, const SE3& T_c_w,
42        double depth )
43    {
44        return camera2world ( pixel2camera ( p_p, depth ), T_c_w );
45    }
}

```

The reader can check whether these methods are consistent with the content of lecture 5. They complete the coordinate transformation between the pixel coordinate system, the camera coordinate system, and the world coordinate system.

13.1.4 Frame class

Let's consider the Frame class. The Frame class is the basic data unit and will be used in many places, but now is the initial design stage, we do not know what may

be added in the future. So the Frame class here only provides basic data storage and interfaces. If there are new content later, continue to add it.

Listing 13.3: slambook/project/0.1/include/myslam/frame.h

```

1 class Frame
2 {
3 public:
4     typedef std::shared_ptr<Frame> Ptr;
5     unsigned long id_; // id of this frame
6     double time_stamp_; // when it is recorded
7     SE3 T_c_w_; // transform from world to camera
8     Camera::Ptr camera_; // Pinhole RGB-D Camera model
9     Mat color_, depth_; // color and depth image
10
11 public: // data members
12     Frame();
13     Frame( long id, double time_stamp=0, SE3 T_c_w=SE3(), Camera::Ptr camera=
14         nullptr, Mat color=Mat(), Mat depth=Mat() );
15     ~Frame();
16
17     // factory function
18     static Frame::Ptr createFrame();
19
20     // find the depth in depth map
21     double findDepth( const cv::KeyPoint& kp );
22
23     // Get Camera Center
24     Vector3d getCamCenter() const;
25
26     // check if a point is in this frame
27     bool isInFrame( const Vector3d& pt_world );
28 };

```

In Frame, we define the ID, timestamp, pose, camera, and image. These should be the most important information contained in a frame. In the method, we extracted several important methods: creating a frame, finding the depth corresponding to a given point, obtaining the camera's optical center, determining whether a point is in the field of view, and so on. Their implementation is relatively trivial, please refer to frame.cpp for specific implementation of these functions.

13.1.5 MapPoint class

MapPoint represents a waypoint. We will estimate its world coordinates, and will match the feature points extracted from the current frame with the landmark points in the map to estimate the camera's movement, so it also needs to store its corresponding descriptor. In addition, we will record the number of times a point is observed and the number of times it is matched as an indicator of how good it is.

Listing 13.4: slambook/project/0.1/include/myslam/mappoint.h

```

1 class MapPoint
2 {
3 public:
4     typedef shared_ptr<MapPoint> Ptr;
5     unsigned long id_; // ID
6     Vector3d pos_; // Position in world
7     Vector3d norm_; // Normal of viewing direction
8     Mat descriptor_; // Descriptor for matching
9     int observed_times_; // being observed by feature matching
10    algo.

```

```

10 int correct_times_; // being an inliner in pose estimation
11
12 MapPoint();
13 MapPoint( long id, Vector3d position, Vector3d norm );
14
15 // factory function
16 static MapPoint::Ptr createMapPoint();
17 };

```

Similarly, the reader can browse `src/map.cpp` to see its implementation. So far we have only considered the initialization of these data members.

13.1.6 Map class

The Map class manages all the landmarks and is responsible for adding new landmarks and deleting bad landmarks. The matching process of VO only needs to deal with Map. Of course Map will also have many operations, but at this stage we only define the main data structures.

Listing 13.5: `slambook/project/0.1/include/myslam/map.h`

```

1 class Map
2 {
3 public:
4     typedef shared_ptr<Map> Ptr;
5     unordered_map<unsigned long, MapPoint::Ptr > map_points_; // all
6     landmarks
7     unordered_map<unsigned long, Frame::Ptr > keyframes_; // all
8     key-frames
9
10    Map() {}
11
12    void insertKeyFrame( Frame::Ptr frame );
13    void insertMapPoint( MapPoint::Ptr map_point );
14 };

```

The Map class actually stores each key frame and waypoint. It needs both random access and insertion and deletion at any time. Therefore, we use Hash to store it.

13.1.7 Config class

The Config class is responsible for reading parameter files, and can provide parameter values at any time in the program. So we write Config as Singleton. It has only one global object. When we set the parameter file, we create the object and read the parameter file, and then we can access the parameter value anywhere, and finally it is destroyed automatically at the end of the program.

Listing 13.6: `slambook/project/0.1/include/myslam/config.h`

```

1 class Config
2 {
3 private:
4     static std::shared_ptr<Config> config_;
5     cv::FileStorage file_;
6
7     Config () {} // private constructor makes a singleton
8 public:
9     ~Config(); // close the file when deconstructing

```

```

10 // set a new config file
11 static void setParameterFile( const std::string& filename );
12
13 // access the parameter values
14 template< typename T >
15 static T get( const std::string& key )
16 {
17     return T( Config::config_->file_[key] );
18 }
19
20 };

```

described as follows:

1. We declare the constructor as private to prevent objects of this class from being created elsewhere. It can only be constructed during setParameterFile. The actual constructed object is a smart pointer for Config: static shared _ptr <Config> config_. The reason for using smart pointers is that they can be destructed automatically, saving us from having to call another function to do the destruction.
2. For file reading, we use the FileStorage class provided by OpenCV. It can read a YAML file and can access any of these fields. Since the actual value of the parameter may be an integer, a floating point number, or a string, we use a template function get to obtain any type of parameter value.

The following is the implementation of Config. Note that we have defined the global pointer for the singleton pattern in this source file:

Listing 13.7: slambook/project/0.1/src/config.cpp

```

1 void Config :: setParameterFile (const std :: string & filename)
2 {
3     if ( config_ == nullptr )
4         config_ = shared_ptr<Config>(new Config);
5     config_->file_ = cv::FileStorage( filename.c_str(), cv::FileStorage::READ
6         );
7     if ( config_->file_.isOpened() == false )
8     {
9         std::cerr<<"parameter file "<<filename<<" does not exist."<<std::endl;
10        config_->file_.release();
11        return ;
12    }
13 Config::~Config()
14 {
15     if ( file_.isOpened() )
16         file_.release();
17 }
18 shared_ptr<Config> Config::config_ = nullptr;

```

In implementation, we just need to judge whether the parameter file exists. After defining the Config class, we can get the parameters in the parameter file anywhere. For example, when you want to define the camera's focal length f_x , follow these steps:

1. Add "Camera.fx: 500" to the parameter file.
2. is used in the code:

```

1 myslam :: Config :: setParameterFile ("parameter.yaml");
2 double fx = myslam :: Config :: get <double> ("Camera.fx");

```

You will get the value of f_x .

Of course, there is definitely more than one way to implement parameter files. We mainly consider this implementation from the perspective of program development convenience. Of course, the reader can also implement the parameter configuration in a simpler way.

So far, we have defined the basic data structure of the SLAM program and written several basic classes. It's like bricks and cement for building a house. You can call cmake to compile this version 0.1, although it has no substantial features yet. Next, let's consider adding the previously mentioned VO algorithm to the project, and do some tests to adjust the performance of each algorithm. Note that the author deliberately exposes certain design issues, so the implementation you see may not be the best (or good enough).

13.2 Basic VO: Feature Extraction and Matching

Now let's implement VO, first consider the feature point method. Its task is to calculate camera movement and feature point positions based on the input image. Earlier we discussed pose estimation between two or two frames, but we will find that the estimation of two frames alone is not enough. We will cache the feature points into a small map and calculate the position relationship between the current frame and the map. But the procedure will be a bit more complicated, so let's set a small target first, starting from the motion estimation between two or two frames.

13.2.1 two-frame visual odometry

If, as in the previous two lectures, you only care about the motion estimation between two frames, and do not optimize the position of the feature points. Then “stringing” the estimated poses can also get a motion trajectory. This method can be viewed as Pairwise structureless VO, which is the easiest to implement, but the effect is not good. Why not? Let's experience it together. Record the project as version 0.2.

The schematic diagram of VO work between two frames is shown in ?? . In this kind of VO, we have defined two concepts of Reference Frame and Current Frame. Taking the reference frame as the coordinate system, we match the current frame with it and estimate the motion relationship. Assume that the transformation matrix of the reference frame relative to the world coordinates is \mathbf{T}_{rw} , and the current frame and the world coordinate system are \mathbf{T}_{cw} . The motion and the transformation matrix of these two frames form a left multiplication relationship:

$$\mathbf{T}_{cr}, \quad \text{s.t.} \quad \mathbf{T}_{cw} = \mathbf{T}_{cr} \mathbf{T}_{rw}.$$

From $t - 1$ to t , we use $t - 1$ as a reference to find the movement at t . This can be obtained through feature point matching, optical flow, or direct methods, but here we **only care about motion, not care about structure**. In other words, as long as the motion is successfully obtained through the feature points, we no longer need the feature points for this frame. This approach is certainly flawed, but ignoring the large number of feature points can save a lot of calculations. Then, from the

time of t to $t + 1$, we use the time of t as a reference frame to consider the motion relationship between t and $t + 1$. In this way, a trajectory is obtained.

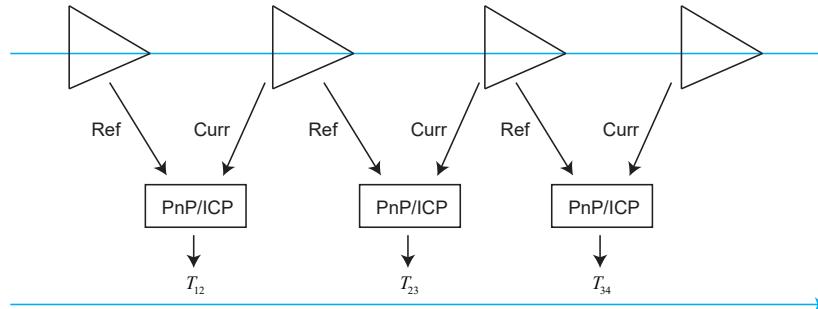


Figure 13-4: Two or two frames of VO.

The way this VO works is simple, but there are several implementations. We take the traditional matching feature point-PnP method as an example to achieve it again. I hope that the reader can combine the knowledge of the previous lectures to realize the VO of the optical flow/direct method or ICP for motion. In the way of matching feature points, the most important is the feature matching relationship between the reference frame and the current frame. Its process can be summarized as follows:

1. Extract keypoints and descriptors for the new current frame.
2. If the system is not initialized, use this frame as the reference frame, calculate the 3D position of the key point according to the depth map, and return to step 1.
3. Estimates the motion between the reference frame and the current frame.
4. Determines whether the above estimation was successful.
5. If successful, use the current frame as the new reference frame and return to step 1.
6. If it fails, record the number of consecutive lost frames. When the continuous loss exceeds a certain number of frames, the VO state is set to lost, and the algorithm ends. If not, go back to step 1.

The VisualOdometry class gives an implementation of the above algorithm.

Listing 13.8: slambook/project/0.2/include/myslam/visual_odometry.h

```

1 class VisualOdometry
2 {
3 public:
4     typedef shared_ptr<VisualOdometry> Ptr;
5     enum VOState {
6         INITIALIZING=-1,
7         OK=0,
8         LOST
9     };
10    VOState      state_; // current VO status
11
  
```

```

12 Map::Ptr map_; // map with all frames and map points
13 Frame::Ptr ref_; // reference frame
14 Frame::Ptr curr_; // current frame
15
16 cv::Ptr<cv::ORB> orb_; // orb detector and computer
17 vector<cv::Point3f> pts_3d_ref_; // 3d points in reference frame
18 vector<cv::KeyPoint> keypoints_curr_; // keypoints in current frame
19 Mat descriptors_curr_; // descriptor in current frame
20 Mat descriptors_ref_; // descriptor in reference frame
21 vector<cv::DMatch> feature_matches_;
22
23 SE3 T_c_r_estimated_; // the estimated pose of current frame
24 int num_inliers_; // number of inlier features in icp
25 int num_lost_; // number of lost times
26
27 // parameters
28 int num_of_features_; // number of features
29 double scale_factor_; // scale in image pyramid
30 int level_pyramid_; // number of pyramid levels
31 float match_ratio_; // ratio for selecting good matches
32 int max_num_lost_; // max number of continuous lost times
33 int min_inliers_; // minimum inliers
34
35 double key_frame_min_rot; // minimal rotation of two key-frames
36 double key_frame_min_trans; // minimal translation of two key-frames
37
38 public: // functions
39 VisualOdometry();
40 ~VisualOdometry();
41
42 bool addFrame( Frame::Ptr frame ); // add a new frame
43
44 protected:
45 // inner operation
46 void extractKeyPoints();
47 void computeDescriptors();
48 void featureMatching();
49 void poseEstimationPnP();
50 void setRef3DPoints();
51
52 void addKeyFrame();
53 bool checkEstimatedPose();
54 bool checkKeyFrame();
55 };

```

There are a few things to explain about this VisualOdometry class:

1. VO itself has several states: setting the first frame, tracking smoothly or missing, you can think of it as a Finite State Machine (FSM). Of course, there can be more states, for example, the monocular VO has at least one initialization state. In our implementation, consider the three simplest states: initialized, normal, and lost.
2. We define some intermediate variables in the class, which can save complicated parameter passing. Because they are defined inside the class, they can be accessed by various functions.
3. The parameters in feature extraction and matching are read from the parameter file. E.g:

¹ VisualOdometry :: VisualOdometry ():

```

2| state_ (INITIALIZING), ref_ (nullptr), curr_ (nullptr), map_ (new Map)
3|   , num_lost_ (0), num_inliers_ (0)
4| {
5|     num_of_features_ = Config::get<int> ( "number_of_features" );
6|     scale_factor_ = Config::get<double> ( "scale_factor" );
7|     level_pyramid_ = Config::get<int> ( "level_pyramid" );
8|     match_ratio_ = Config::get<float> ( "match_ratio" );
9|   ...
9| }
```

4. addFrame function is an interface called externally. When using VO, after loading the image data into the Frame class, call addFrame to estimate its pose. This function performs different operations depending on the state of VO:

```

1| bool VisualOdometry :: addFrame (Frame :: Ptr frame)
2| {
3|   switch ( state_ )
4|   {
5|     case INITIALIZING:
6|     {
7|       state_ = OK;
8|       curr_ = ref_ = frame;
9|       map_ ->insertKeyFrame ( frame );
10|      // extract features from first frame
11|      extractKeyPoints();
12|      computeDescriptors();
13|      // compute the 3d position of features in ref frame
14|      setRef3DPoints();
15|      break;
16|    }
17|    case OK:
18|    {
19|      curr_ = frame;
20|      extractKeyPoints();
21|      computeDescriptors();
22|      featureMatching();
23|      poseEstimationPnP();
24|      if ( checkEstimatedPose() == true ) // a good estimation
25|      {
26|        curr_ ->T_c_w_ = T_c_r_estimated_ * ref_ ->T_c_w_; // T_c_w =
27|        T_c_r*T_r_w
28|        ref_ = curr_;
29|        setRef3DPoints();
30|        num_lost_ = 0;
31|        if ( checkKeyFrame() == true ) // is a key-frame
32|        {
33|          addKeyFrame();
34|        }
35|      }
36|      else // bad estimation due to various reasons
37|      {
38|        num_lost_++;
39|        if ( num_lost_ > max_num_lost_ )
40|        {
41|          state_ = LOST;
42|        }
43|        return false;
44|      }
45|      break;
46|    }
47|    case LOST:
48|  }
```

```

47  {
48      cout<<"vo has lost."<<endl;
49      break;
50  }
51 }
52 return true;
53 }
```

It is worth mentioning that, for various reasons, the above-mentioned VO algorithm we designed may fail at each step. For example, it is difficult to mention features in the picture, feature points lack depth values, mismatches, errors in motion estimation, and so on. Therefore, to design a robust VO, it is necessary (explicitly) to take into account all the above possible mistakes—that naturally makes the program very complicated. In checkEstimatedPose, we do a simple test based on the number of inliers and the size of the movement: we think that the inside point cannot be too small, and the movement cannot be too large. Of course, readers can also think about other ways to detect problems and try the results.

We have omitted the rest of the implementation of the VisualOdometry class, and the reader can find all the source code on GitHub. Finally, we add the test program of this VO to the test and use the data set to observe the estimated motion effect:

Listing 13.9: slambook/project/0.2/test/run_vo.cpp

```

1 int main (int argc, char ** argv)
2 {
3     if ( argc != 2 )
4     {
5         cout<<"usage: run_vo parameter_file"<<endl;
6         return 1;
7     }
8
9     myslam::Config::setParameterFile ( argv[1] );
10    myslam::VisualOdometry::Ptr vo ( new myslam::VisualOdometry );
11
12    string dataset_dir = myslam::Config::get<string> ( "dataset_dir" );
13    cout<<"dataset: "<<dataset_dir<<endl;
14    ifstream fin ( dataset_dir+ "/associate.txt" );
15    if ( !fin )
16    {
17        cout<<"please generate the associate file called associate.txt!"<<endl;
18        return 1;
19    }
20
21    vector<string> rgb_files, depth_files;
22    vector<double> rgb_times, depth_times;
23    while ( !fin.eof() )
24    {
25        string rgb_time, rgb_file, depth_time, depth_file;
26        fin>>rgb_time>>rgb_file>>depth_time>>depth_file;
27        rgb_times.push_back ( atof ( rgb_time.c_str() ) );
28        depth_times.push_back ( atof ( depth_time.c_str() ) );
29        rgb_files.push_back ( dataset_dir+ "+" +rgb_file );
30        depth_files.push_back ( dataset_dir+ "+" +depth_file );
31
32        if ( fin.good() == false )
33            break;
34    }
35 }
```

```

36     myslam::Camera::Ptr camera ( new myslam::Camera );
37
38     // visualization
39     cv::viz::Viz3d vis("Visual Odometry");
40     cv::viz::WCoordinateSystem world_coor(1.0), camera_coor(0.5);
41     cv::Point3d cam_pos( 0, -1.0, -1.0 ), cam_focal_point(0,0,0), cam_y_dir
42         (0,1,0);
43     cv::Affine3d cam_pose = cv::viz::makeCameraPose( cam_pos, cam_focal_point
44         , cam_y_dir );
45     vis.setViewerPose( cam_pose );
46
47     world_coor.setRenderingProperty(cv::viz::LINE_WIDTH, 2.0);
48     camera_coor.setRenderingProperty(cv::viz::LINE_WIDTH, 1.0);
49     vis.showWidget( "World", world_coor );
50     vis.showWidget( "Camera", camera_coor );
51
52     cout<<"read total "<<rgb_files.size() <<" entries"<<endl;
53     for ( int i=0; i<rgb_files.size(); i++ )
54     {
55         Mat color = cv::imread ( rgb_files[i] );
56         Mat depth = cv::imread ( depth_files[i], -1 );
57         if ( color.data==nullptr || depth.data==nullptr )
58             break;
59         myslam::Frame::Ptr pFrame = myslam::Frame::createFrame();
60         pFrame->camera_ = camera;
61         pFrame->color_ = color;
62         pFrame->depth_ = depth;
63         pFrame->time_stamp_ = rgb_times[i];
64
65         boost::timer timer;
66         vo->addFrame ( pFrame );
67         cout<<"VO costs time: "<<timer.elapsed()<<endl;
68
69         if ( vo->state_ == myslam::VisualOdometry::LOST )
70             break;
71         SE3 Tcw = pFrame->T_c_w_.inverse();
72
73         // show the map and the camera pose
74         cv::Affine3d M_C
75         cv::Affine3d::Mat3(
76             Tcw.rotation_matrix()(0,0), Tcw.rotation_matrix()(0,1), Tcw.
77                 rotation_matrix()(0,2),
78             Tcw.rotation_matrix()(1,0), Tcw.rotation_matrix()(1,1), Tcw.
79                 rotation_matrix()(1,2),
80             Tcw.rotation_matrix()(2,0), Tcw.rotation_matrix()(2,1), Tcw.
81                 rotation_matrix()(2,2)
82         ),
83         cv::Affine3d::Vec3(
84             Tcw.translation()(0,0), Tcw.translation()(1,0), Tcw.translation()
85             (2,0)
86         )
87     };
88 }
```

In order to run this program, several things need to be done:

1. Because we use the Viz module of OpenCV 3 to display the estimated pose,

please make sure you have OpenCV 3 installed and the viz module has been compiled and installed.

2. Prepare one of the TUM datasets. For simplicity, I recommend fr1 _xyz. Please use associate.py to generate a pairing file associate.txt. The TUM dataset format is described in the ?? section.
3. Fill in the path of your data set in config/default.yaml, refer to the author's writing. Then, use

```
1 bin/run_vo config/default.yaml
```

Run the program and you can see the real-time demo, as shown in ?? .

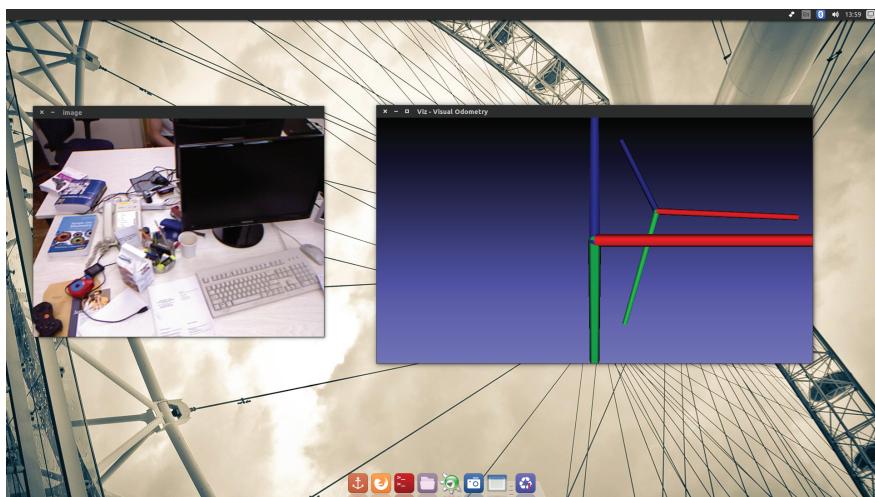


Figure 13-5: Vo demo for version 0.2.

In the demo program, you can see the image of the current frame and its estimated position. We have drawn the world axis (large) and the current frame (small). The corresponding relationship between the color and the axis is: blue—Z, red—X, green—Y. You can intuitively feel the movement of the camera, which is roughly consistent with our human feelings, although there is still a certain gap between the effect and the expectations. The program also outputs the time for a single calculation of VO. On the author's machine, it can run at a speed of about 30ms. Reducing the number of feature points can increase the calculation speed. The reader can modify the operating parameters and data set to see how it performs in various situations.

13.2.2 discussion

In this section, we have implemented a simple visual odometry between two or two frames, but its effect is not ideal in terms of speed or accuracy. It seems that this seemingly simple idea does not give good results. Let's consider the possible reasons:

1. For pose estimation, we used the PnP solution obtained by RANSAC. Because RANSAC uses only a few random points to calculate PnP, although the inlier

can be determined, this method is susceptible to noise. In the case of noise at 3D-2D points, we need to use RANSAC's solution as the initial value, and then use nonlinear optimization to find an optimal value. The next section will show that this approach is better than the current one. Hell

2. Since the current VO is unstructured, the 3D positions of feature points are treated as true values to estimate motion. But in fact, the depth map of RGB-D must have certain errors, especially those where the depth is too close or too far. And, because feature points are often located at the edges of objects, depth measurements in those places are often inaccurate. So the current method is not accurate enough, we need to optimize the feature points together. Hell
3. only considers the reference frame/current frame, on the one hand, the pose estimation is too dependent on the reference frame. If the reference frame quality is too poor, for example, in the case of severe occlusion and lighting changes, the tracking is easily lost. In addition, when the reference frame pose is inaccurate, a significant **drift** will occur. On the other hand, using only two frames of data obviously does not make full use of all the information. A more natural way is to compare the current frame and the map, rather than the current frame and the reference frame. Therefore, we need to care about **how to match the current frame with the map, and how to optimize the map points.** Hell
4. Since the running time of each step is output, we can have a general understanding of the amount of calculation (??). Hell Hell It can be seen that

Table 13-1: Elapsed steps of a cycle

Item	Feature Extraction	Descriptor Calculation	Feature Matching	PnP Solving	Others
Time	0.0102	0.0087	0.0118	0.0011	0.0001

the extraction and matching of feature points occupy most of the calculation time, and the seemingly complex PnP optimization, the amount of calculation is basically negligible compared to it. Therefore, how to improve the speed of feature extraction and matching algorithms will be an important topic of feature point methods. One predictable way is to use direct method/optical flow, which can effectively avoid heavy feature calculation work. The direct method and optical flow method have been discussed before in this book, and the reader may wish to try it out by themselves.

13.3 Improved: Optimizing PnP results

Next, we follow the previous content and try some ways to improve VO. In this section, we will try to estimate the camera pose using RANSAC PnP plus iterative optimization to see if it improves the effect of the previous section.

The solution of nonlinear optimization problems has been introduced in lectures 6 and 7. Since the goal of this section is to estimate the pose rather than the structure, we use the camera pose ξ as the optimization variable to construct the optimization problem by minimizing reprojection errors. As before, we customize an optimized edge in g2o. It only optimizes one pose, so it is a unary edge.

Listing 13.10: slambook/project/0.3/include/myslam/g2o_types.h

```

1 class EdgeProjectXYZ2UVPoseOnly: public g2o::BaseUnaryEdge<2, Eigen::  
2   Vector2d, g2o::VertexSE3Expmap >  
3 {  
4   public:  
5     EIGEN_MAKE_ALIGNED_OPERATOR_NEW  
6  
7     virtual void computeError();  
8     virtual void linearizeOplus();  
9  
10    virtual bool read( std::istream& in ){}  
11    virtual bool write(std::ostream& os) const {};  
12  
13    Vector3d point_;  
14    Camera* camera_;  
15 };

```

Put the 3D point and camera model into its member variables to facilitate the calculation of reprojection error and Jacobian matrix:

Listing 13.11: slambook/project/0.3/src/g2o_types.cpp

```

1 void EdgeProjectXYZ2UVPoseOnly::computeError()  
2 {  
3   const g2o::VertexSE3Expmap* pose = static_cast<const g2o::VertexSE3Expmap*>  
4     (_vertices[0] );  
5   _error = _measurement - camera_->camera2pixel (   
6     pose->estimate().map(point_)  
7   );  
8 }  
9  
10 void EdgeProjectXYZ2UVPoseOnly::linearizeOplus()  
11 {  
12   g2o::VertexSE3Expmap* pose = static_cast<g2o::VertexSE3Expmap*> (   
13     _vertices[0] );  
14   g2o::SE3Quat T ( pose->estimate() );  
15   Vector3d xyz_trans = T.map ( point_ );  
16   double x = xyz_trans[0];  
17   double y = xyz_trans[1];  
18   double z = xyz_trans[2];  
19   double z_2 = z*z;  
20  
21   _jacobianOplusXi ( 0,0 ) = x*y/z_2 *camera_->fx_;  
22   _jacobianOplusXi ( 0,1 ) = - ( 1+ ( x*x/z_2 ) ) *camera_->fx_;  
23   _jacobianOplusXi ( 0,2 ) = y/z * camera_->fx_;  
24   _jacobianOplusXi ( 0,3 ) = -1./z * camera_->fx_;  
25   _jacobianOplusXi ( 0,4 ) = 0;  
26   _jacobianOplusXi ( 0,5 ) = x/z_2 * camera_->fx_;  
27  
28   _jacobianOplusXi ( 1,0 ) = ( 1+y*y/z_2 ) *camera_->fy_;  
29   _jacobianOplusXi ( 1,1 ) = -x*y/z_2 *camera_->fy_;  
30   _jacobianOplusXi ( 1,2 ) = -x/z *camera_->fy_;  
31   _jacobianOplusXi ( 1,3 ) = 0;  
32   _jacobianOplusXi ( 1,4 ) = -1./z * camera_->fy_;  
33   _jacobianOplusXi ( 1,5 ) = y/z_2 *camera_->fy_;  
34 }

```

PoseEstimationPnP	RANSAC PnP	g2o
-------------------	------------	-----

Listing 13.12: slambook/project/0.3/src/visual_odometry.cpp

```

1 void VisualOdometry::poseEstimationPnP()
2 {
3     ...
4     // using bundle adjustment to optimize the pose
5     typedef g2o::BlockSolver<g2o::BlockSolverTraits<6,2>> Block;
6     Block::LinearSolverType* linearSolver = new g2o::LinearSolverDense<Block
7         ::PoseMatrixType>();
8     Block* solver_ptr = new Block( linearSolver );
9     g2o::OptimizationAlgorithmLevenberg* solver = new g2o::
10        OptimizationAlgorithmLevenberg ( solver_ptr );
11     g2o::SparseOptimizer optimizer;
12     optimizer.setAlgorithm ( solver );
13
14     g2o::VertexSE3Expmap* pose = new g2o::VertexSE3Expmap();
15     pose->setId ( 0 );
16     pose->setEstimate ( g2o::SE3Quat (
17         T_c_r_estimated_.rotation_matrix(), T_c_r_estimated_.translation()
18     ) );
19     optimizer.addVertex ( pose );
20
21     // edges
22     for ( int i=0; i<inliers.rows; i++ )
23     {
24         int index = inliers.at<int>(i,0);
25         // 3D -> 2D projection
26         EdgeProjectXYZ2UVPoseOnly* edge = new EdgeProjectXYZ2UVPoseOnly();
27         edge->setId(i);
28         edge->setVertex(0, pose);
29         edge->camera_ = curr_->camera_.get();
30         edge->point_ = Vector3d( pts3d[index].x, pts3d[index].y, pts3d[index].z
31             );
32         edge->setMeasurement( Vector2d(pts2d[index].x, pts2d[index].y) );
33         edge->setInformation( Eigen::Matrix2d::Identity() );
34         optimizer.addEdge( edge );
35     }
36
37     T_c_r_estimated_ = SE3 (
38         pose->estimate().rotation(),
39         pose->estimate().translation()
40     );

```

The reader is asked to run this program and compare the previous results. You will notice that the estimated motion has stabilized a lot. At the same time, because the new optimization is still unstructured and small in scale, the impact on the calculation time can be ignored. The overall visual odometer calculation time is still around 30ms.

discussion

We find that the quality of the estimated results is significantly improved compared with pure RANSAC PnP after the iterative optimization method is introduced. Although we still only use the information between two or two frames, the resulting

motion is more accurate and smooth. From this improvement we see the importance of optimization. However, the 0.3 version of VO is still affected by the limitations of matching between two frames. Once a frame in a video sequence is lost, subsequent frames cannot match the previous frame. Next, we introduce maps into VO.

13.4 Improvement: Partial Map

In this section, we put the feature points matched by VO into the map, and match the current frame with the map points to calculate the pose. The difference between this approach and the previous is shown in ?? .

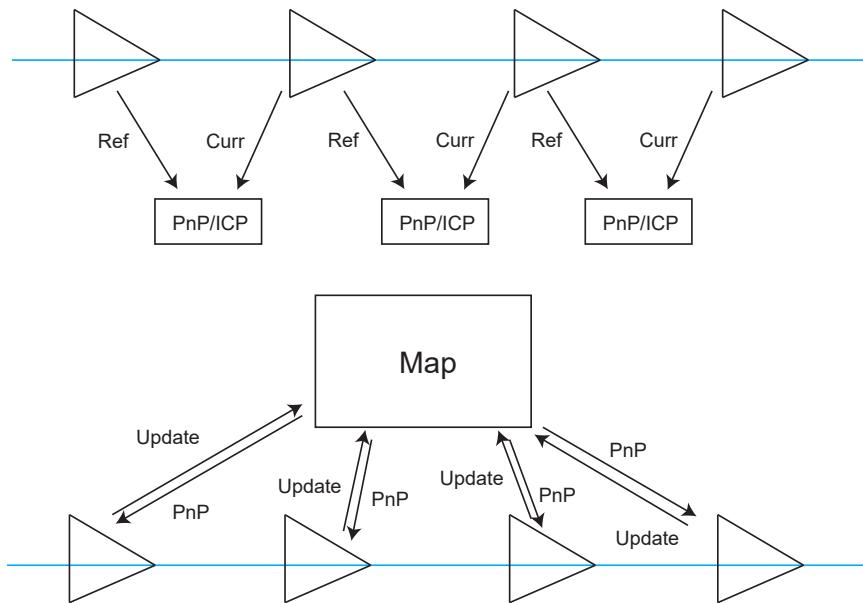


Figure 13-6: The difference between the working principle of two or two frames VO and map VO.

In the comparison between two frames, we only calculate the feature matching and motion relationship between the reference frame and the current frame, and set the current frame as the new reference frame after the calculation. In VO using maps, each frame contributes some information to the map, such as adding new feature points or updating the position estimates of old feature points. The location of feature points in the map is often in world coordinates. Therefore, when the current frame arrives, we find the feature matching and motion relationship between it and the map, that is, directly calculate T_{cw} .

The benefit of this is that we are able to maintain a constantly updated map. As long as the map is correct, even if something goes wrong in the middle of the frame, there is still hope to find the correct position of those subsequent frames. Please note that we have not discussed the problem of SLAM in detail at this time, so the map here is only a temporary concept, which refers to the set of feature points formed by caching feature points of each frame to a place .

Maps can be divided into **Local** maps and **Global** maps. Because of different uses, maps are often discussed separately. As the name suggests, the local map describes nearby feature point information-we only keep feature points that are closer to the camera's current location, and discard feature points that are far away or out of the field of view. These feature points are used to find the camera position by matching the current frame, so we hope it can be done faster. On the other hand, the global map records all feature points since SLAM was run. Obviously, it is larger in size and is mainly used to express the entire environment, but positioning it directly on the global map places too much burden on the computer. It is mainly used for loop detection and map expression.

In visual odometers, we are more concerned with local maps that can be used directly for positioning (if we decide to use maps). So in this lecture we will maintain a local map. As the camera moves, we add new feature points to the map. We still want to remind readers: whether to use maps depends on your contradictory grasp of precision-efficiency. We can completely use pairless unstructured VO for efficiency reasons; we can also build local maps and even consider map optimization for better accuracy.

One troublesome part of a local map is maintaining its scale. In order to ensure real-time, we need to ensure that the map size is not too large (otherwise matching will consume a lot of time). In addition, there are some acceleration methods for single frame and map feature matching, but due to the technical complexity, we will not give them in our routine.

Now, let's implement the map point class. We slightly improved the MapPoint class that was not used before, mainly its constructor and generator functions.

Listing 13.13: slambook/project/0.4/include/myslam/mappoint.h

```

1 class MapPoint
2 {
3     public:
4     typedef shared_ptr<MapPoint> Ptr;
5     unsigned long id_; // ID
6     static unsigned long factory_id_; // factory id
7     bool good_; // whether a good point
8     Vector3d pos_; // Position in world
9     Vector3d norm_; // Normal of viewing direction
10    Mat descriptor_; // Descriptor for matching
11
12    list<Frame*> observed_frames_; // key-frames that can observe this
13    point
14
15    int matched_times_; // being an inliner in pose estimation
16    int visible_times_; // being visible in current frame
17
18    MapPoint();
19    MapPoint(
20        unsigned long id,
21        const Vector3d& position,
22        const Vector3d& norm,
23        Frame* frame=nullptr,
```

```

23     const Mat& descriptor=Mat()
24 );
25
26     inline cv::Point3f getPositionCV() const {
27         return cv::Point3f( pos_(0,0), pos_(1,0), pos_(2,0) );
28     }
29
30     static MapPoint::Ptr createMapPoint();
31     static MapPoint::Ptr createMapPoint (
32         const Vector3d& pos_world,
33         const Vector3d& norm_,
34         const Mat& descriptor,
35         Frame* frame
36     );
37 };

```

The main modification is on the VisualOdometry class. Due to changes in the workflow, we have modified several of its main functions, such as adding and deleting maps in each loop, counting the number of times each map point has been observed, etc. *. These things are trivial, so we recommend readers to take a closer look at the source code provided on GitHub. Focus on the following:

1. After extracting the feature points of the first frame, put all the feature points of the first frame into the map:

```

1 void VisualOdometry::addKeyFrame()
2 {
3     if ( map_->keyframes_.empty() )
4     {
5         // first key-frame, add all 3d points into map
6         for ( size_t i=0; i<keypoints_curr_.size(); i++ )
7         {
8             double d = curr_->findDepth ( keypoints_curr_[i] );
9             if ( d < 0 )
10                 continue;
11             Vector3d p_world = ref_->camera_->pixel2world (
12                 Vector2d ( keypoints_curr_[i].pt.x, keypoints_curr_[i].pt.y ),
13                 curr_->T_c_w_, d
14             );
15             Vector3d n = p_world - ref_->getCamCenter();
16             n.normalize();
17             MapPoint::Ptr map_point = MapPoint::createMapPoint(
18                 p_world, n, descriptors_curr_.row(i).clone(), curr_->get()
19             );
20             map_->insertMapPoint( map_point );
21         }
22     }
23     map_->insertKeyFrame ( curr_ );
24     ref_ = curr_;
}

```

2. OptimizeMap

```

1 void VisualOdometry::optimizeMap()
2 {
3     // remove the hardly seen and no visible points
4     for ( auto iter = map_->map_points_.begin(); iter != map_->
map_points_.end(); )

```

* Of course, from a C ++ design perspective In general, retaining the previous approach and using inheritance will reuse existing code more effectively.

```

5   {
6     if ( !curr_->isInFrame(iter->second->pos_) )
7     {
8       iter = map_->map_points_.erase(iter);
9       continue;
10    }
11   float match_ratio = float(iter->second->matched_times_)/iter->
12     second->visible_times_;
13   if ( match_ratio < map_point_erase_ratio_ )
14   {
15     iter = map_->map_points_.erase(iter);
16     continue;
17   }
18   double angle = getViewAngle( curr_, iter->second );
19   if ( angle > M_PI/6. )
20   {
21     iter = map_->map_points_.erase(iter);
22     continue;
23   }
24   if ( iter->second->good_ == false )
25   {
26     // TODO try triangulate this map point
27   }
28   iter++;
29 }

30 if ( match_2dkp_index_.size()<100 )
31 addMapPoints();
32 if ( map_->map_points_.size() > 1000 )
33 {
34   // TODO map is too large, remove some one
35   map_point_erase_ratio_ += 0.05;
36 }
37 else
38   map_point_erase_ratio_ = 0.1;
39 cout<<"map points: "<<map_->map_points_.size()<<endl;
40 }
```

We have intentionally left some places blank, please ask interested readers to complete it by themselves. For example, you can use triangulation to update the world coordinates of feature points, or consider strategies to better manage map size dynamically. These questions are open.

3. Feature matching code. Before matching, we take some candidate points (points that appear in the field of view) from the map and then match them with the feature descriptors of the current frame.

```

1 void VisualOdometry::featureMatching()
2 {
3   boost::timer timer;
4   vector<cv::DMatch> matches;
5   // select the candidates in map
6   Mat desp_map;
7   vector<MapPoint::Ptr> candidate;
8   for ( auto& allpoints: map_->map_points_ )
9   {
10     MapPoint::Ptr& p = allpoints.second;
11     // check if p in curr frame image
12     if ( curr_->isInFrame(p->pos_) )
13     {
14       // add to candidate
15       p->visible_times_++;
16     }
17   }
18 }
```

```

16     candidate.push_back( p );
17     desp_map.push_back( p->descriptor_ );
18 }
19
20 matcher_flann_.match ( desp_map, descriptors_curr_, matches );
21 // select the best matches
22 float min_dis = std::min_element (
23     matches.begin(), matches.end(),
24     [] ( const cv::DMatch& m1, const cv::DMatch& m2 )
25     {
26         return m1.distance < m2.distance;
27     } )->distance;
28
29 match_3dpts_.clear();
30 match_2dkp_index_.clear();
31 for ( cv::DMatch& m : matches )
32 {
33     if ( m.distance < max<float> ( min_dis*match_ratio_, 30.0 ) )
34     {
35         match_3dpts_.push_back( candidate[m.queryIdx] );
36         match_2dkp_index_.push_back( m.trainIdx );
37     }
38 }
39 cout<<"good matches: "<<match_3dpts_.size() <<endl;
40 cout<<"match cost time: "<<timer.elapsed() <<endl;
41
42 }
```

In addition to existing maps, we have introduced the concept of "key-frames". Key frames are used in many visual SLAMs, but this concept is mainly used by the backend, so we will discuss the detailed processing of key frames in the next few lectures. In practice, we certainly don't want to do detailed optimization and loop detection for each image, because it is too resource-intensive after all. At least when the camera is standing still, we don't want the entire model (both maps and trajectories) to get bigger and bigger. Therefore, the main object of back-end optimization is key frames.

The key frames are some special frames during camera movement. Here, the meaning of "special" can be specified by us. In common practice, every time the camera moves through a certain interval, a new key frame is taken and saved *. The poses of these key frames will be carefully optimized, and those things located between the two key frames, except for contributing some map points to the map, are naturally ignored.

The implementation in this section will also extract some key frames to make some data preparations for back-end optimization. Now the reader can compile this project and see the results of its operation. The routines in this section project the points of the local map onto the image plane and display them. If the pose estimates are correct, they should look like **fixed in space**. Conversely, if you feel that a certain feature point moves unnaturally, it may be that the camera pose estimation is not accurate enough, or the position of the feature point is not accurate enough.

We did not provide map optimization in version 0.4, and readers are recommended to try it out by themselves. The principles used are mainly least squares and triangulation, which have already been introduced in the first two lectures and will not be too difficult.

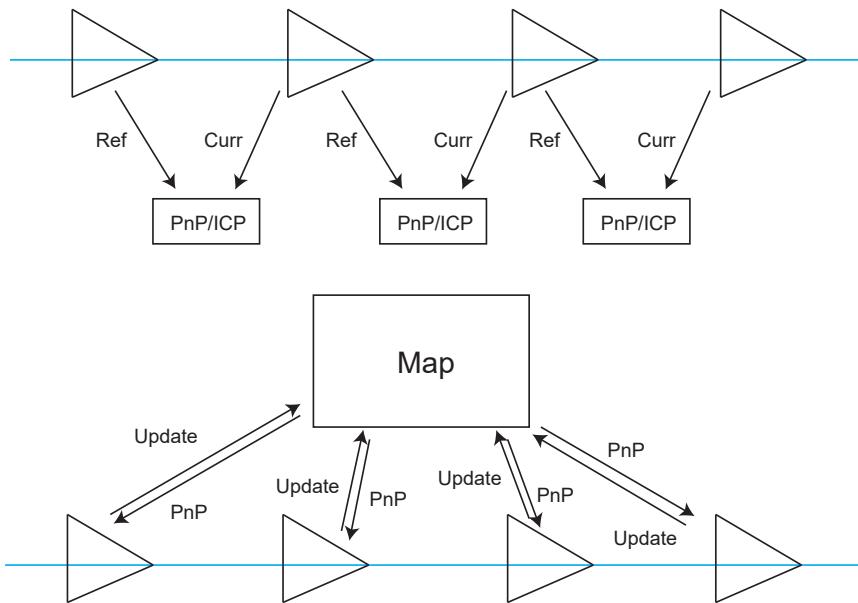


Figure 13-7: 0.4 version of VO's running screenshot, marked the landmark projection points at two different times.

13.5 Summary

As a practice, this lecture leads the reader to implement a simple visual odometer from scratch, in order to allow the reader to have an empirical understanding of the algorithms introduced in the previous two lectures. Without this lecture, it is difficult for you to personally experience questions such as "how many ORB feature

* This is easy to implement in Lie algebra, please think about how to achieve it.

points can the VO of a feature point handle in real time”[†]. We see that the visual odometer can estimate the camera movement and the position of feature points in a local time, but this local approach has obvious disadvantages:

1. is easy to lose. Once lost, we either ”wait for the camera to turn back” (save the reference frame and compare it with the new frame) or reset the entire VO to track the new image data.
2. Trajectory drift. The main reason is that the error of each estimation will accumulate to the next estimation, resulting in inaccurate long-term trajectories. A larger local map can alleviate this phenomenon, but it always exists.

It is worth mentioning that if you only care about the movement in a short period of time, or the accuracy of the VO has met the needs of the application, then sometimes you may only need a visual odometer instead of a complete SLAM. For example, in some drone control or AR game applications, users do not need a globally consistent map, so the portable VO may be a better choice. However, the goal of this book is to introduce the entire SLAM, so we have to go further and see how backend and loopback detection work.

EXERCISE

1. Do you understand the C ++ techniques used in this book? If you don’t understand something, use search engines to supplement your knowledge, including: range-based for loops, lambda expressions, smart pointers, singleton patterns in design patterns, and more.
2. Based on version 0.3 or 0.4, add code to optimize the map. Alternatively, you can also perform triangulation according to the PnP result to eliminate the error of the RGB-D depth value.
3. Observe how this code handles mismatches. What is RANSAC? Read the literature [?] or search for related materials to learn more.

[†] Of course, it depends on the performance of your machine.

Chapter 14

SLAM: present and future

main target

1. Learn about the classic SLAM implementation.
2. Through experiments, compare the similarities and differences of various SLAM schemes.
3. Explore the future direction of SLAM.

The working principle of each module in a SLAM system was introduced earlier, which is the result of many years of work by researchers. At present, in addition to these theoretical frameworks, we have also accumulated many excellent open source SLAM solutions. However, since most of their implementations are complex and not suitable for beginners, we have introduced them at the end of this book. I believe that the reader should understand the basic principles by reading the previous content.

14.1 current open source solution

This lecture is the summary of the entire book. We will take the reader to see how much the existing SLAM scheme can do. In particular, we focus on solutions that provide open source implementations. In the field of SLAM research, it is not easy to see open source solutions. Often, the content of the theory in the paper is only 20

The first half of this lecture will take the reader through the current visual SLAM solution and comment on its historical status and advantages and disadvantages. ?? enumerates some common open source SLAM schemes, readers can choose the schemes of interest for research and experiment. Due to space limitations, we have selected only a few representative programs, which is certainly not comprehensive. In the second half, we will explore some possible future development directions and present some current research results.

Table 14-1: Commonly used open source SLAM solutions

Scheme name	sensor form	address
MonoSLAM	Monocular	https://github.com/hanmekim/ SceneLib2
PTAM	Monocular	http://www.robots.ox.ac.uk/~gk/ PTAM/
ORB-SLAM	Monocular-based	http://webdiis.unizar.es/~raulmur/ orbslam/
LSD-SLAM	Monocular-based	http://vision.in.tum.de/research/ vslam/lسدslam
SVO	Monocular	https://github.com/uzh-rpg/rpg_svo
DTAM	RGB-D	https://github.com/anuranbaka/ OpenDTAM
DVO	RGB-D	https://github.com/tum-vision/dvo_ slam
DSO	Monocular	https://github.com/JakobEngel/dso
RTAB-MAP	Binocular/RGB-D	https://github.com/introlab/ rtabmap
RGBD-SLAM-V2	RGB-D	https://github.com/felixendres/ rgbdslam_v2
Elastic Fusion	RGB-D	https://github.com/mp3guy/ ElasticFusion
Hector SLAM	Laser	http://wiki.ros.org/hector_slam
GMapping	Laser	http://wiki.ros.org/gmapping
OKVIS	Multi-head + IMU	https://github.com/ethz-asl/okvis
ROVIO	Monocular + IMU	https://github.com/ethz-asl/rovio

14.1.1 MonoSLAM

When it comes to visual SLAM, the first thing that many researchers think of is A. J. Davison's monocular SLAM job [? ?]. Professor Davison is a pioneer in the field of visual SLAM research. The MonoSLAM he proposed in 2007 is the first real-time monocular visual SLAM system [?], which is considered to be the birthplace

of many jobs *. MonoSLAM uses extended Kalman filtering as the back end and tracks very sparse feature points on the front end. Because EKF occupies a clear dominant position in early SLAM, MonoSLAM is also based on EKF. The current state of the camera and all landmark points are used as state quantities to update its mean and covariance.

?? shows what MonoSLAM looks like at runtime. It can be seen that the monocular camera tracks very sparse feature points in an image (and uses active tracking technology). In EKF, the position of each feature point follows a Gaussian distribution, so we can express its mean and uncertainty in the form of an ellipsoid. In the right half of the figure, we can find some small balls distributed in space. The longer they appear in a certain direction, the more uncertain their position in that direction. We can imagine that if a feature point converges, we should be able to see it change from a long ellipsoid (very uncertain in the direction of camera Z) to a small point.

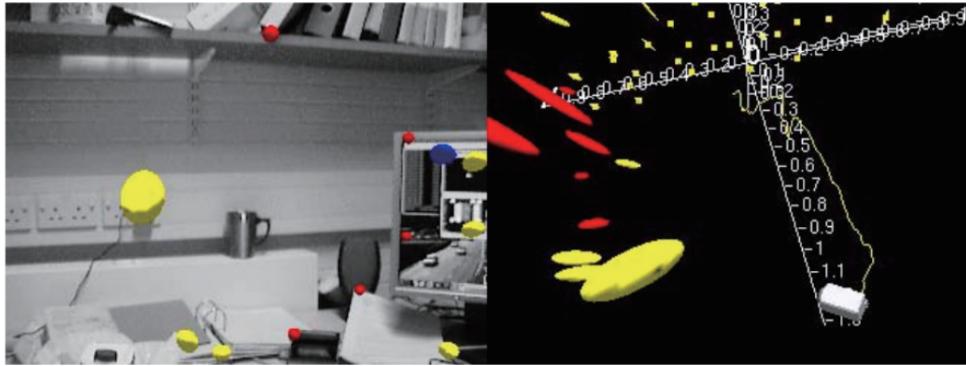


Figure 14-1: MonoSLAM runtime screenshot. Left: Representation of tracking feature points in the image; Right: Representation of feature points in 3D space.

Although this approach seems to have many disadvantages today, it was already a milestone work at that time, because before that the visual SLAM system could not basically run online. It could only rely on a robot to carry a camera to collect data and then perform offline positioning. And mapping. The advances in computer performance and the processing of images in a sparse manner add up to a SLAM system that can run online. From a modern point of view, MonoSLAM has situations such as narrow application scenarios, limited number of road signs, and sparse feature points are very easy to lose. The development of MonoSLAM has also been stopped and replaced by more advanced theory and programming tools. But this does not prevent us from understanding and respecting the work of our predecessors.

14.1.2 PTAM

In 2007, Klein and others proposed PTAM (Parallel Tracking and Mapping) [?], which is also an important event in the development of visual SLAM. The significance of PTAM lies in the following two points:

* this It is a continuation of his doctoral work. He is also working on miniaturization and low power of SLAM.

1. PTAM proposes and implements the parallelization of the tracking and mapping process. We now know that the tracking part needs to respond to the image data in real time, and the optimization of the map does not need to be calculated in real time. Back-end optimization can be performed slowly in the background, and then thread synchronization can be performed when necessary. This is the first time that the concept of front-end and back-end is distinguished in visual SLAM, which led to the design of many later visual SLAM systems (most of the SLAM we see now are divided into front-end and back-end).
2. PTAM is the first solution to use non-linear optimization instead of using traditional filters as the back end. It introduces a keyframe mechanism: instead of processing each image in detail, we string together several key images and then optimize their trajectories and maps. Most of the early SLAMs used EKF filters or their variants, as well as particle filters. After PTAM, visual SLAM research gradually turned to the back end dominated by nonlinear optimization. Because people did not realize the sparseness of back-end optimization before, they felt that the optimized back-end cannot process such large-scale data in real time, and PTAM is a significant counter-example.

PTAM is also an augmented reality software that demonstrates cool AR effects (as shown in ??). Based on the camera pose estimated by PTAM, we can place a virtual object on a virtual plane, which looks just like in a real scene.

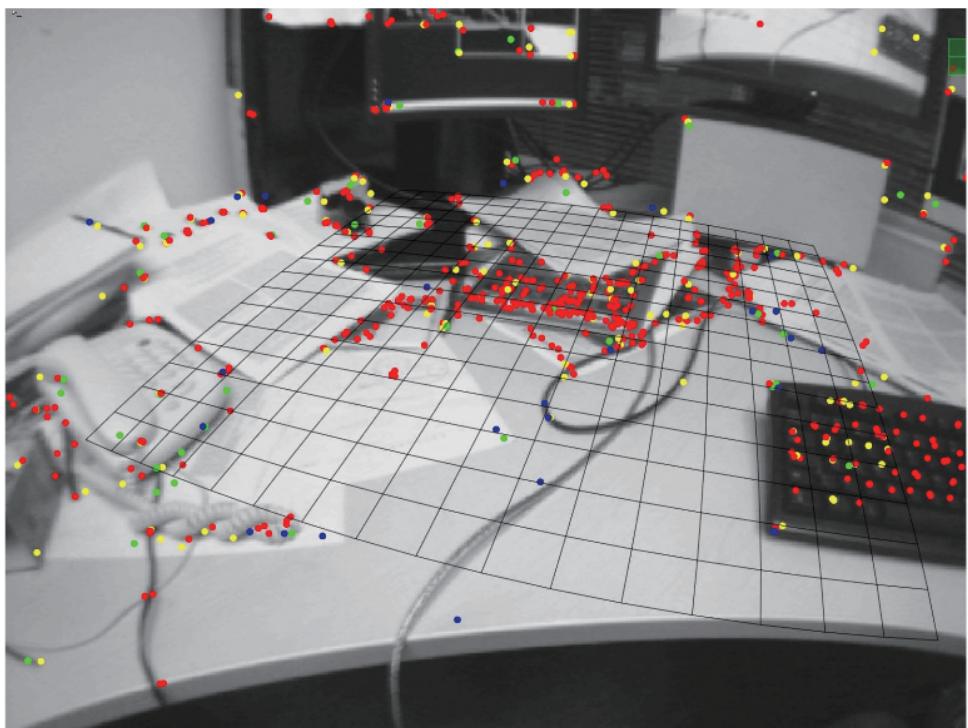


Figure 14-2: PTAM demo screenshot. It can provide real-time positioning and mapping, and can also overlay virtual objects on the virtual plane.

However, from a modern perspective, PTAM is also considered to be one of the early SLAM work combined with AR. Similar to many early work, there are obvious flaws: small scenes, tracking is easy to lose, etc. These were corrected in subsequent plans.

14.1.3 ORB-SLAM

After introducing several historical solutions, let's look at some modern SLAM systems. ORB-SLAM is a very famous [?] among the successors of PTAM (see ??). It was proposed in 2015, and it is one of the very complete and very easy to use systems in modern SLAM systems (if not the most complete and easy to use). ORB-SLAM represents a peak of mainstream feature point SLAM. Compared with previous work, ORB-SLAM has the following obvious advantages:

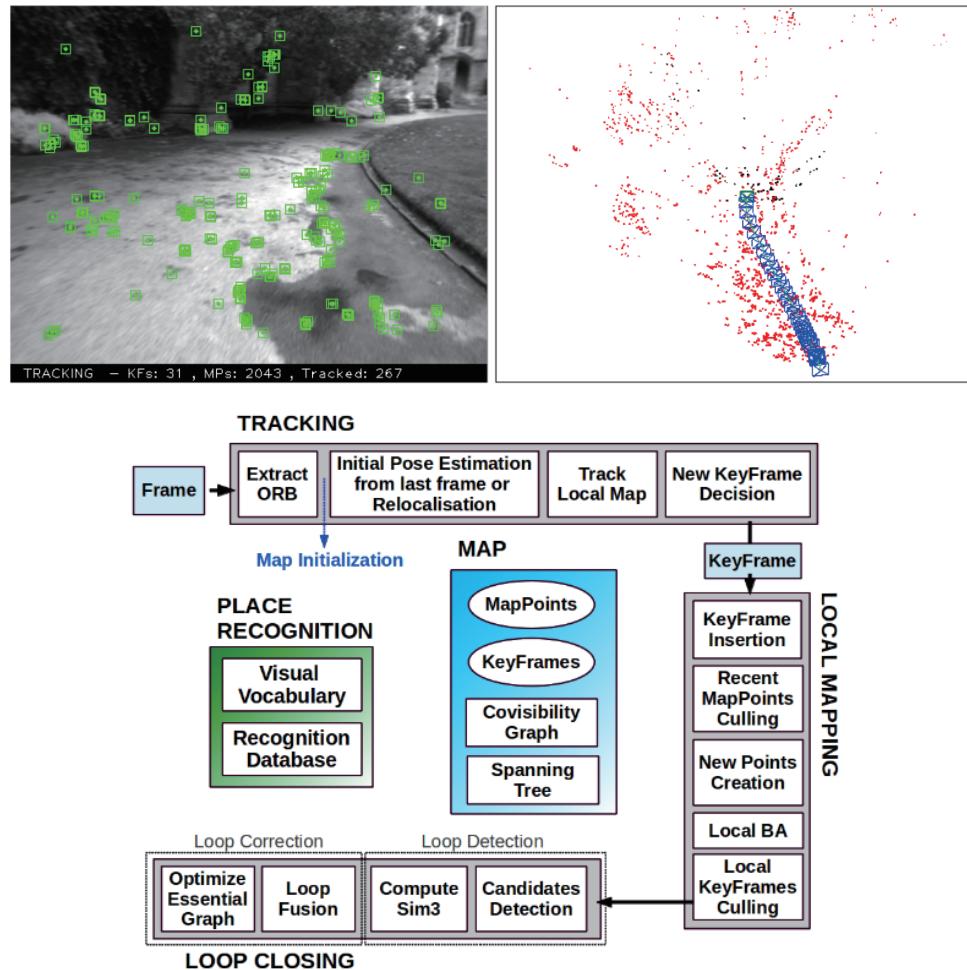


Figure 14-3: ORB-SLAM Run screenshot. The left side is the image and the feature points tracked, and the right side is the camera trajectory and the modeled feature point map. Below is its signature three-threaded structure.

1. supports three modes: monocular, binocular, and RGB-D. This makes it pos-

sible to test it on ORB-SLAM, no matter what kind of common sensor we have, it has good versatility.

2. The entire system is calculated around ORB features, including visual odometers and ORB dictionaries for loop detection. It shows that the ORB feature is an excellent compromise between efficiency and accuracy of current computing platforms. ORB is not as time-consuming as SIFT or SURF, and can be calculated in real time on the CPU; compared with simple corner features such as Harris corners, it also has good rotation and scaling invariance. In addition, ORB provides descriptors that enable us to perform loop detection and relocation during large-scale movements.
3. ORB loop detection is its highlight. The excellent loop detection algorithm ensures that ORB-SLAM effectively prevents accumulated errors and can be quickly recovered after being lost. This is because many existing SLAM systems are not perfect. For this reason, ORB-SLAM must load a large ORB dictionary file *.
4. ORB-SLAM innovatively uses three threads to complete SLAM: a tracking thread that tracks feature points in real time, an optimization thread for local bundle adjustment (Co-visibility Graph, commonly known as), and Loop-back detection and optimization thread (Essential Graph commonly known as). Among them, the Tracking thread is responsible for extracting ORB feature points for each new image, and comparing with the nearest key frame, calculating the position of the feature points and roughly estimating the camera pose. The small graph thread solves a Bundle Adjustment problem, which includes feature points and camera poses in local space. This thread is responsible for solving more detailed camera poses and feature point spatial positions. However, with only the first two threads, only a better visual odometer has been completed. The third thread, the big picture thread, performs loop detection on the global map and key frames to eliminate accumulated errors. Because there are too many map points in the global map, the optimization of this thread does not include map points, but only pose maps composed of camera poses. Hell Following the dual-threaded structure of PTAM, the three-threaded structure of ORB-SLAM has achieved very good tracking and mapping effects, which can ensure the global consistency of the trajectory and the map. This three-thread structure will also be recognized and adopted by subsequent researchers.
5. ORB-SLAM has been optimized a lot around feature points. For example, based on the feature extraction of OpenCV, a uniform distribution of feature points is ensured. When optimizing the pose, a method of cyclic optimization 4 times to get more correct matches is adopted. A more relaxed key frame selection strategy than PTAM and many more. These small improvements make ORB-SLAM far more robust than other solutions: even for poor scenes and poor calibration internal parameters, ORB-SLAM can work smoothly.

These advantages make the ORB-SLAM reach its peak in the feature point SLAM. Many researches use ORB-SLAM as the standard, or carry out subsequent

* before running. Currently the open source version of ORB-SLAM uses a dictionary in text format, which can be speeded up by changing to a binary format dictionary.

development based on it. Its code is known for its legibility and well-annotated for further understanding by later researchers.

Of course, ORB-SLAM also has some shortcomings. First, because the entire SLAM system uses feature points for calculations, we must calculate the ORB features for each image again, which is very time-consuming. ORB-SLAM's three-thread structure also brings a heavy burden on the CPU, making it only possible to perform real-time operations on CPUs of the current PC architecture, and it is difficult to port to embedded devices. Secondly, ORB-SLAM maps are sparse feature points. At present, there is no open storage and relocation function after reading the map (although it is not difficult in terms of implementation). According to our analysis in the mapping section, the sparse feature point map can only meet our needs for positioning, and cannot provide many functions such as navigation, obstacle avoidance, and interaction. However, if we only use ORB-SLAM to deal with the positioning problem, it seems to be a bit too heavyweight. In contrast, other solutions provide a more lightweight positioning, allowing us to run SLAM on low-end processors, or to allow the CPU to spare other tasks.

14.1.4 LSD-SLAM

LSD-SLAM (Large Scale Direct monocular SLAM) is a SLAM work proposed by J. Engel et al. In 2014 [? ?]. Analogous to ORB-SLAM is the feature point, and LSD-SLAM marks the successful application of monocular direct method in SLAM. The core contribution of LSD-SLAM is to apply the direct method to semi-dense monocular SLAM. Not only does it not need to calculate feature points, it can also construct a semi-dense map-here semi-dense means mainly estimated pixel locations with obvious gradients. Its main advantages are as follows:

1. The direct method of LSD-SLAM is pixel-specific. The author creatively proposed the relationship between pixel gradient and direct method, and the angular relationship between pixel gradient and epipolar direction in dense reconstruction. These are discussed in lectures 8 and 13 of this book. However, LSD-SLAM performs semi-dense tracking on monocular images, and the implementation principle is more complicated than the book's routines.
2. LSD-SLAM implements reconstruction of semi-dense scenes on the CPU, which was rarely seen in previous solutions. Feature point-based methods can only be sparse, and most of the dense reconstruction schemes use RGB-D sensors, or use GPUs to build dense maps [?]. Based on years of research on direct method, TUM computer vision group has realized this kind of real-time semi-dense SLAM on CPU.
3. also said before that LSD-SLAM's semi-dense tracking uses some delicate methods to ensure the real-time and stability of the tracking. For example, LSD-SLAM does not use a single pixel or an image block, but takes 5 points at equal distances on the epipolar line to measure its SSD. In depth estimation, LSD-SLAM first initializes the depth with a random number. After the estimation, the depth mean is normalized to adjust the scale. When measuring the depth uncertainty, not only the geometric relationship of the triangulation, but also the angle between the epipolar line and the depth is taken into account. ; Constraints between key frames use similar transformation groups and corresponding Lie algebras $\zeta \in \mathfrak{sim}(3)$ to explicitly express the scale, which

can be used in back-end optimization. Taking scenes of different scales into account reduces the scale drift phenomenon.

?? shows the operation of LSD. We can observe how this delicate semi-dense map is a form between sparse and dense maps. Semi-dense maps model the parts with obvious gradients in the grayscale map, which are displayed in the map, and a large part of them are the edges or textured parts of the surface. LSD-SLAM tracks them and builds key frames, and finally optimizes to obtain such a map. It seems to have more information than a sparse map, but it does not have a complete surface like a dense map (dense maps generally consider it impossible to achieve real-time performance with only the CPU).

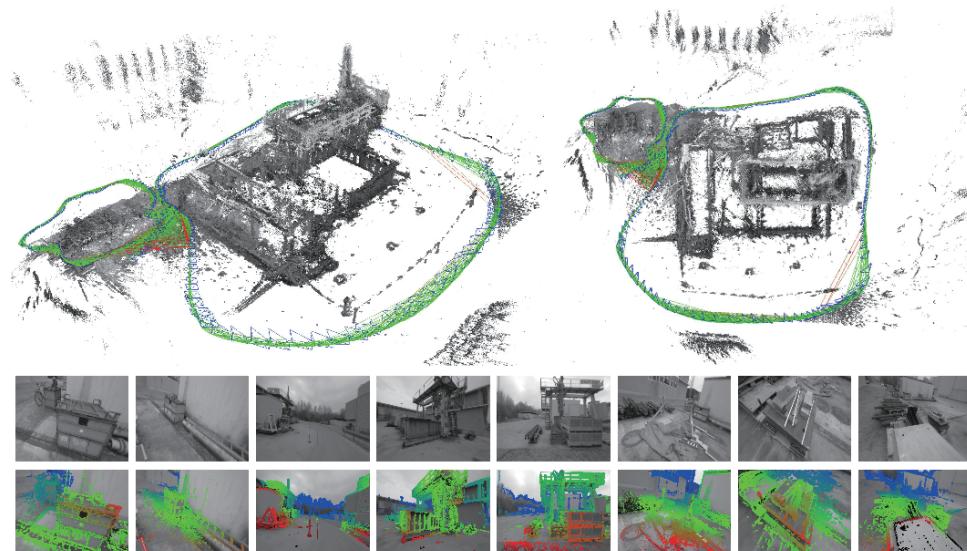


Figure 14-4: LSD-SLAM run picture. The upper part is the estimated trajectory and map, and the lower part is the modeled part in the image, that is, the part with better pixel gradient.

Because LSD-SLAM uses the direct method for tracking, it has both the advantages of the direct method (not sensitive to missing regions of features) and the disadvantages of the direct method. For example, LSD-SLAM is very sensitive to camera internal parameters and exposure, and is easily lost when the camera moves quickly. In addition, in the loop detection part, since there is currently no loop detection method based on the direct method, LSD-SLAM must rely on the feature point method for loop detection, and it has not completely got rid of the calculation of feature points.

14.1.5 SVO

SVO is short for Semi-direct Visual Odometry [?]. It is a visual odometer based on sparse direct method proposed by Forster et al. In 2014. According to the author, it should be called "semi-direct" method, but according to the conceptual framework of this book, it is better to call "sparse direct method". **Semi-direct** means in the original text a mixture of feature points and direct methods: SVO tracks some key

points (corners, no descriptors), and then uses the information around these key points like direct methods Estimate camera movement and its position (as shown by ??). In the implementation, SVO uses the small blocks of 4×4 around the key points for block matching to estimate the camera's own motion.

Compared with other solutions, the biggest advantage of SVO is that it is extremely fast. Because it uses a sparse direct method, it does not have to laboriously calculate descriptors, and it does not have to process as much information as dense and semi-dense. Therefore, it can achieve real-time performance even on low-end computing platforms, while on PC Can reach the speed of more than 100 frames per second. In the subsequent SVO 2.0, the speed reached an astonishing 400 frames per second. This makes SVO ideal for applications where computing platforms are limited, such as the positioning of drones and handheld AR/VR devices. UAV is also the target application platform for the author to develop SVO.



Figure 14-5: SVO A picture of the key points.

Another innovation of SVO is to propose the concept of a depth filter and derive a depth filter based on a uniform-Gaussian mixture distribution. This is mentioned in lecture 13 of this book, but because the principle is more complicated, we have not explained it in detail. SVO uses this filter for the position estimation of key points, and uses inverse depth as a parameterized form to enable it to better calculate the position of feature points.

The open source version of SVO code is clear and easy to read, which is very suitable for readers to analyze as the first SLAM instance. However, there are some problems with the open source version of SVO:

1. Because the target application platform is a drone's overhead camera, the objects in its field of view are mainly the ground, and the camera's movement is mainly horizontal and vertical movement. Many details of SVO are designed around this application, which makes it in Poor performance in head-up cameras. For example, SVO uses the method of factoring the \mathbf{H} matrix instead of

the traditional \mathbf{F} or \mathbf{E} matrix during monocular initialization, which requires the feature points to be on the plane on. This assumption is true for a head-up camera, but it is usually not true for a head-up camera, which may cause initialization to fail. For another example, SVO uses the translation amount as a strategy for determining new key frames when selecting key frames, without considering the amount of rotation. This is also effective in a drone top-down configuration, but it is easily lost in a head-up camera. Therefore, if you want to use SVO in a head-up camera, you must modify it yourself.

2. SVO has abandoned the back-end optimization and loop detection part for speed and light weight, and basically has no mapping function. This means that there is necessarily a cumulative error in the pose estimation of SVO, and it is not easy to relocate after loss (because there are no descriptors for loop detection). So, instead of calling it a complete SLAM, we call it a VO.

14.1.6 RTAB-MAP

After introducing several monocular SLAM solutions, let's look at some SLAM solutions on RGB-D sensors. Compared to monocular and binocular, the principle of RGB-D SLAM is much simpler (although not necessarily implemented), and it can build dense maps on the CPU in real time.

RTAB-MAP (Real Time Appearance-Based Mapping) [?] is a classic solution in RGB-D SLAM. It implements everything that should be in RGB-D SLAM: feature-based visual odometry, bag-of-words-based loop detection, back-end pose map optimization, and point cloud and triangular mesh maps. Therefore, RTAB-MAP provides a complete (but somewhat large) RGB-D SLAM scheme. At present, we can directly obtain its binary program from ROS. In addition, we can also obtain its application on Google Project Tango (such as ??).

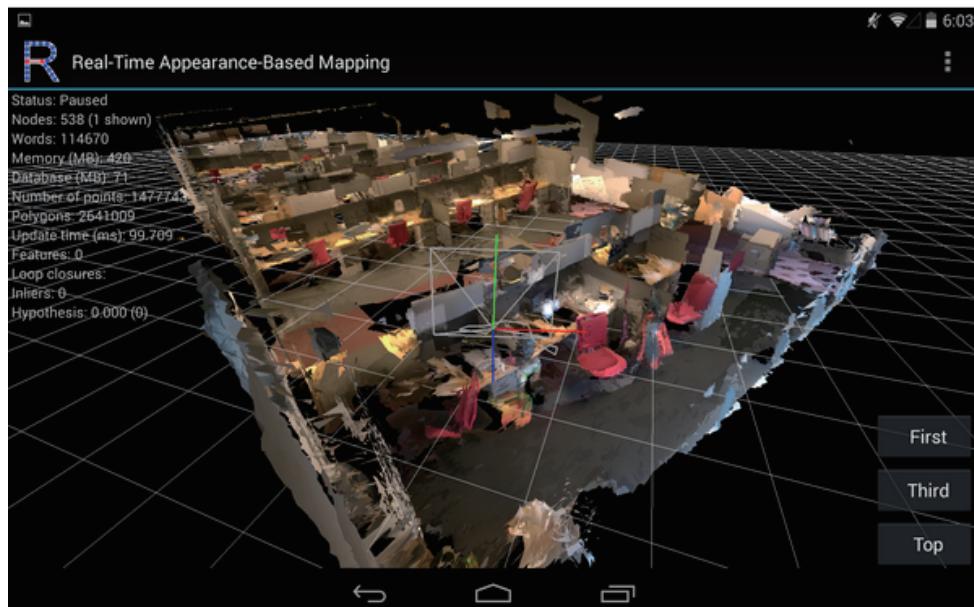


Figure 14-6: RTAB-MAP running example on Google Project Tango.

RTAB-MAP supports some common RGB-D and binocular sensors, such as Kinect, Xtion, etc., and provides real-time positioning and mapping functions. However, due to the high degree of integration, it is difficult for other developers to carry out secondary development based on it, so RTAB-MAP is more suitable for SLAM applications rather than research.

14.1.7 other

In addition to these open source solutions, readers can find many other research on websites such as openslam.org, such as DVO-SLAM [?], RGBD-SLAM-V2 [?], DSO [?], and some Kinect Fusion related work, etc. With the development of the times, newer and better open-source SLAM works will also appear in people's vision. I will not introduce them one by one due to space limitations. Hell

14.2 Future SLAM Topics

After reading the existing schemes, let's discuss some future development directions *. Generally speaking, there are two major types of SLAM's future development trends: one is to move towards lightweight and miniaturization, so that SLAM can run well on small devices such as embedded or mobile phones, and then consider applications using it as the underlying function . After all, in most occasions, our real purpose is to realize the functions of robots and AR/VR devices, such as sports, navigation, teaching, and entertainment, and SLAM provides its own pose estimation for upper-level applications. In these applications, we do not want SLAM to occupy all computing resources, so there is a very strong demand for SLAM miniaturization and lightweight. On the other hand, it uses high-performance computing equipment to achieve precise 3D reconstruction and scene understanding. In these applications, our goal is to reconstruct the scene perfectly, without much limitation on the portability of computing resources and devices. Since GPUs can be used, this direction and deep learning also have a combination.

14.2.1 Vision + Inertial Navigation SLAM

First, we want to talk about a direction with a strong application background: vision-inertial navigation fusion SLAM solution. Whether it is an actual robot or a hardware device, it usually does not carry only one kind of sensor, but often a fusion of multiple sensors. Researchers in academia love the "Big Clean Problem", such as visual SLAM with a single camera. But friends in the industry are more focused on making the algorithm more practical, and have to face some complicated and trivial scenarios. In this application context, SLAM using fusion of vision and inertial navigation has become a focus of attention.

The inertial sensor (IMU) can measure the angular velocity and acceleration of the sensor body, which is considered to have obvious complementarity with the camera sensor, and has great potential to obtain a more complete SLAM system after fusion [?]. Why do you say that?

1. Although the IMU can measure angular velocity and acceleration, these quantities have significant drift (Drift), which makes the pose data obtained by two

* There is a part of my understanding here, which may not be completely correct.

integrations very unreliable. For example, if we put the IMU on a table without moving it, the position obtained by integrating its readings will drift out of the air. However, for fast movements in a short time, the IMU can provide some good estimates. This is exactly the weakness of the camera. Hell

When the motion is too fast, the camera (of the rolling shutter) will have motion blur, or there is too little overlap between the two frames to allow feature matching, so pure visual SLAM is very afraid of fast motion. With IMU, we can maintain a good pose estimation even during the period when the camera data is invalid, which is not possible with pure visual SLAM. Hell

2. Compared to IMU, camera data is basically free of drift. If the camera is fixed in place, the pose estimation of the visual SLAM (in a static scene) is also fixed. Therefore, camera data can effectively estimate and correct drift in IMU readings, so that pose estimation after slow motion is still valid. Hell
3. When the image changes, we basically cannot know whether the camera itself has moved or the external conditions have changed, so pure visual SLAM is difficult to handle dynamic obstacles. The IMU can sense its own motion information and mitigate the impact of dynamic objects to some extent.

All in all, we see that the IMU provides a better solution for fast motion, and the camera can solve the drift problem of the IMU under slow motion-in this sense, they are complementary.

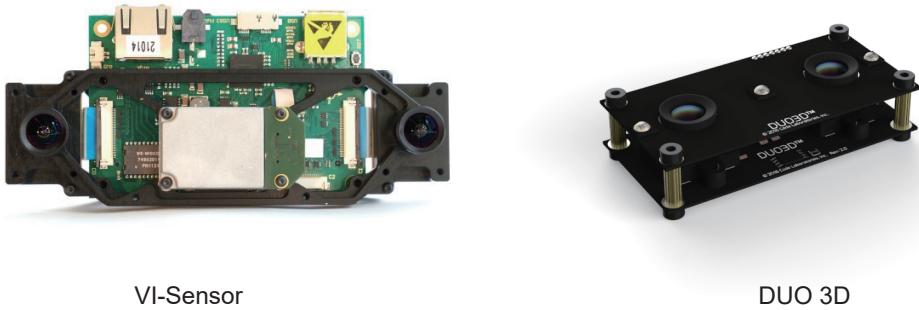


Figure 14-7: More and more cameras are integrating IMU devices.

Of course, although it sounds good, whether it is theory or practice, VIO (Visual Inertial Odometry) is quite complicated. Its complexity is mainly derived from the fact that the IMU measures acceleration and angular velocity, so it has to introduce kinematic calculations. At present, the framework of VIO has been shaped into two categories: Loosely Coupled and Tightly Coupled [?]. Loose coupling means that the IMU and the camera separately perform their own motion estimation, and then fuse their pose estimation results. Tight coupling refers to combining the state of the IMU with the state of the camera, constructing the equations of motion and observations, and then performing the state estimation-this is very similar to the theory we introduced earlier. We can predict that the tight coupling theory will also be divided into two directions based on filtering and optimization. In terms of filtering, traditional EKF [?] and improved MSCKF (Multi-State Constraint KF) [?] have achieved certain results, and researchers have also performed EKF In-depth discussion (such as observability [?]); there is also a corresponding solution for optimization [? ?]. It is worth mentioning that although the optimization method has occupied the mainstream in pure visual SLAM, in VIO, because the data frequency of IMU is very high, the amount of calculation required to optimize the state is larger, so it is still in the filtering and Optimization coexisting phase [? ?]. Because it is too complicated and limited in space, I can only introduce this direction here.

VIO provides a very effective direction for the miniaturization and low cost of SLAM in the future. And combined with the sparse direct method, we are expected to achieve good SLAM or VO effects on low-end hardware, which is very promising.

14.2.2 Semantic SLAM

Another major direction of SLAM is to combine it with deep learning technology. So far, SLAM solutions are at the level of feature points or pixels. We don't know what these feature points or pixels come from. This makes SLAM in computer vision not very similar to our human approach. At least we never see the feature points ourselves, and we don't judge the direction of our own movement based on the feature points. What we see are objects, judging their distance through the left and right eyes, and then inferring the camera's movement based on their movement in the image.

Long ago, researchers tried to incorporate object information into SLAM. For example, the literature [? ? ? ?] has combined object recognition with visual SLAM to build a map with object labels. On the other hand, by introducing the label information into the objective function and constraints of the BA or optimization side, we can combine the position of the feature points with the label information to optimize [?]. These works can be called semantic SLAM. In summary, the combination of SLAM and semantics has two main aspects: [?]:

1. semantics help SLAM. Traditional object recognition and segmentation algorithms often only consider one picture, and in SLAM we have a moving camera. If we put object labels on the pictures during the exercise, we can get a labeled map. In addition, object information can also bring more conditions for loop detection and BA optimization.
2. SLAM helps semantics. Both object recognition and segmentation require a lot of training data. In order for the classifier to recognize objects from various angles, it is necessary to collect data on the objects from different perspectives and then perform manual calibration, which is very hard. In SLAM, because we can estimate the camera's motion, we can automatically calculate the position of the object in the image, saving the cost of manual calibration. If there is automatically generated sample data with high-quality annotations, it can greatly speed up the training process of the classifier.



Figure 14-8: Some results of semantic SLAM, the left and right images are from the literature [? ?].

Before the widespread application of deep learning, we could only use traditional tools such as support vector machines and conditional random fields to segment and identify objects or scenes, or directly compare observation data with samples in the database [? ?], trying to build a semantic map [? ? ? ?]. Because these tools have limitations on the accuracy of their classifications, their results are often unsatisfactory. With the development of deep learning, we started to use the network to recognize, detect and segment images more and more accurately [? ? ? ? ?]. This laid a better foundation for building accurate semantic maps [?]. We are seeing that scholars are gradually introducing neural network methods to object recognition and segmentation in SLAM, and even pose estimation and loop detection in SLAM itself [? ? ?]. Although these methods have not yet become mainstream, combining SLAM with deep learning to process images is also a promising research direction.

14.2.3 Future of SLAM

Besides, SLAM [? ? ?] based on line/area features, SLAM [? ? ?] in dynamic scenes, SLAM [? ? ?], etc., are all places where researchers are interested and working hard. According to the literature [?], visual SLAM has gone through three major eras: asking questions, finding algorithms, and improving algorithms. We are currently in the third era, facing how to further improve in the existing framework, so that the visual SLAM system can run stably under various interference conditions. This step requires the unremitting efforts of many researchers.

Of course, no one can predict the future, and we are not sure if one day, the entire framework will be overthrown and overwritten by new technologies. But even then, our contribution today will still be meaningful. Without today's research, there will be no future development. Finally, I hope that after reading this book, the reader will have a full understanding of the existing SLAM system. We also look forward to your contribution to SLAM research!

EXERCISE

1. Choose any of the open source SLAM systems mentioned in this lecture, compile and run it on your machine, and experience the process intuitively.
2. You should already be able to understand most SLAM related papers. Pick up paper and pen and start your research!