

SLAM in Robotics and Autonumous Driving

Xiang Gao

Last update: July 9, 2025

To my beloved Lilian and Shenghan

Contents

I Basic Knowledge	1
1 Autonomous Driving	3
1.1 Autonomous Driving Technologies	3
1.1.1 Autonomous Driving Capabilities and Grading	3
1.1.2 Typical L4 Tasks	6
1.2 Localization and Mapping in Autonomous Driving	10
1.2.1 Why L4 Needs Localization and Mapping	10
1.2.2 Contents and Production of High-Definition Maps	12
1.3 Introduction to SLAM in this Book.	14
2 Quick Review of Basic Mathematical Concepts	17
2.1 Geometry	19
2.1.1 Coordinate Systems	19
2.1.2 Rotation Vectors	22
2.1.3 Quaternions	23
2.1.4 Lie Group and Lie Algebra	26
2.1.5 BCH Linear Approximation on $\text{SO}(3)$	27
2.2 Kinematics.	27
2.2.1 Kinematics from the Perspective of Lie Groups	28
2.2.2 Kinematics from the Perspective of Quaternions	29
2.2.3 Conversion between Lie Algebra of Quaternions and Rotation Vectors	30
2.2.4 Other Kinematic Representations	32
2.2.5 Linear Velocity and Acceleration	33
2.2.6 Perturbation and Jacobian Matrices	35
2.3 Kinematics Example: Circular Motion	37
2.4 Filters and Optimization	38
2.4.1 State Estimation and Least Squares	39
2.4.2 Kalman Filter	39
2.4.3 Nonlinear Systems	40
2.4.4 Graph Optimization	42
2.5 Summary	43
3 Inertial Navigation and Integrated Navigation	45
3.1 Kinematics of IMU Systems	47
3.1.1 Kinematics	47
3.1.2 Explanation of IMU Measurements	48
3.1.3 Noise Model in IMU Measurement Equations	49

3.1.4	Discrete-Time Noise Model for IMU	51
3.1.5	IMUs in Reality	52
3.2	Trajectory Estimation Using IMU	54
3.2.1	Short-Term Trajectory Estimation Using IMU Data	54
3.2.2	Code Experiment for IMU Recursion	55
3.3	Satellite Navigation	57
3.3.1	Practical RTK Installation and Data Reception	60
3.3.2	Common World Coordinate Systems	62
3.3.3	Display of RTK Measurements	63
3.3.4	RTK Measurement Visualization	66
3.4	Implementing Integrated Navigation Using Error-State Kalman Filter	68
3.4.1	Mathematical Derivation of ESKF	68
3.4.2	Discrete-Time ESKF Kinematic Equations	73
3.4.3	Motion Process of ESKF	74
3.4.4	Update Process of ESKF	75
3.4.5	Post-Processing of ESKF Error State	76
3.5	Implementation of ESKF-based Integrated Navigation	77
3.5.1	Implementation of the ESKF Filter	78
3.5.2	Implementing the Prediction Process	79
3.5.3	Implementing RTK Observation Process	80
3.5.4	Initialization of the ESKF System	82
3.5.5	Implementation of Static Initialization	83
3.5.6	Running the ESKF	84
3.5.7	Velocity Observations	88
3.6	Summary	90
4	Pre-integration	93
4.1	Pre-integration of IMU States	95
4.1.1	Definition of Pre-integration	95
4.2	Pre-integration of IMU States	96
4.2.1	Definition of Pre-integration	96
4.2.2	Pre-integration Measurement Model	97
4.2.3	Bias Update	101
4.2.4	Preintegration Model Reduced to Graph Optimization	103
4.2.5	Jacobians of Preintegration	104
4.2.6	Jacobians of Preintegration	105
4.2.7	Summary	106
4.3	Practice: Implementation of Preintegration	107
4.3.1	Implementing the Preintegration Class	107
4.4	Practice: Implementation of Preintegration	108
4.4.1	Implementing the Single IMU Integration	108
4.4.2	Graph Optimization Vertices for Pre-integration	110
4.4.3	Graph Optimization Edges for Pre-integration Scheme	111
4.4.4	Implementation of GINS Based on Pre-integration and Graph Optimization	115
4.5	Summary	120

II Laser Localization and Mapping	123
5 Basic Point Cloud Processing	125
5.1 Mathematical Models of Laser Sensors and Point Clouds	127
5.1.1 Mathematical Model of Laser Sensors	127
5.1.2 Representation of Point Clouds	129
5.1.3 Packet Representation	131
5.1.4 Bird's-Eye View and Range Image	132
5.1.5 Bird's-Eye View and Range Image	134
5.1.6 Alternative Representations	137
5.2 Nearest Neighbor Problem	138
5.2.1 Brute-Force Nearest Neighbor Search	138
5.2.2 Grid and Voxel Methods	141
5.2.3 Binary Trees and K-d Trees	149
5.2.4 Construction of K-d Trees	150
5.2.5 Searching in K-d Trees	150
5.2.6 Implementation of K-d Tree Construction	154
5.2.7 Quadtree and Octree	159
5.2.8 Other Tree-Based Methods	165
5.2.9 Summary	165
5.3 Fitting Problems	166
5.3.1 Plane Fitting	167
5.3.2 Plane Fitting Implementation	169
5.3.3 Line Fitting	170
5.3.4 Implementation of Line Fitting	172
5.4 Summary	174
6 2D Laser Localization and Mapping	175
6.1 Basic Principles of 2D Laser SLAM	177
6.2 Scan Matching Algorithms	179
6.2.1 Point-to-Point Scan Matching	179
6.2.2 Implementation of Point-to-Point ICP (Gauss-Newton)	183
6.2.3 Point-to-Line Scan Matching	186
6.2.4 Implementation of Point-to-Line ICP (Gauss-Newton)	187
6.2.5 Likelihood Field Method	189
6.2.6 Implementation of Likelihood Field Method (Gauss-Newton)	191
6.2.7 Implementation of Likelihood Field Method (g2o)	193
6.3 Occupancy Grid Map	195
6.3.1 Principle of Occupancy Grid Map	195
6.3.2 Map Generation Based on Bresenham's Algorithm	197
6.4 Submaps	200
6.4.1 Principle of Submaps	200
6.4.2 Implementation of Submaps	200
6.5 Loop Closure Detection and Correction	203
6.5.1 Multi-Resolution Loop Closure Detection	204
6.5.2 Submap-Based Loop Closure Correction	206
6.5.3 Discussion	210
6.6 Summary	212

7 3D LiDAR Localization and Mapping	215
7.1 Working Principles of Multi-beam LiDAR	217
7.1.1 Mechanical LiDAR	217
7.1.2 Solid-State LiDAR	218
7.2 Scan Matching for Multi-beam LiDAR	219
7.2.1 Point-to-Point ICP	219
7.3 Experimental Evaluation	222
7.3.1 Point-to-Line and Point-to-Plane ICP	224
7.3.2 Point-to-Line ICP Implementation	224
7.3.3 NDT Method	226
7.3.4 Comparative Analysis of Registration Methods vs. PCL Implementations	231
7.4 Direct Method LiDAR Odometry.	232
7.4.1 Building LiDAR Odometry with NDT	233
7.4.2 Incremental NDT Odometry	236
7.4.3 Incremental NDT Odometry	238
7.5 Laser Odometry with Feature-based Methods.	242
7.5.1 Feature Extraction	242
7.5.2 Feature Extraction Based on LiDAR Beams	243
7.5.3 Implementation of Feature Extraction	244
7.6 Loosely Coupled LIO Systems	253
7.6.1 Coordinate Frame Conventions	254
7.6.2 Motion and Observation Equations for Loosely Coupled LIO	254
7.6.3 Data Preparation for Loosely Coupled System	254
7.6.4 Main Workflow of Loosely Coupled System	257
7.6.5 Registration Module of the Loosely Coupled LIO System	260
7.7 Summary	261
III Applications	263
8 Tightly Coupled LIO System	265
8.1 Principles and Advantages of Tight Coupling	267
8.2 LIO System Based on IEKF	267
8.2.1 State Variables and Motion Equations of IEKF	267
8.2.2 Iterative Process in the Observation Equation	268
8.2.3 Efficient Handling of High-Dimensional Observations	270
8.3 Implementation of IEKF-based LIO	272
8.4 LIO Based on Preintegration	275
8.4.1 Principles of Preintegrated LIO	275
8.4.2 Code Implementation	277
8.5 Summary	282
9 Mapping for Autonomous Vehicles	283
9.1 Point Cloud Mapping Pipeline	285
9.2 Frontend Implementation.	286
9.3 Keyframe Extraction	287
9.4 Backend Pose Graph Optimization and Outlier Detection	290

9.5 Loop Closure Detection	292
9.6 Map Export	297
9.7 Summary	298
10 Real-Time Localization System	301
10.1 Design Scheme for Point Cloud Fusion Localization.	303
10.2 Algorithm Implementation	304
10.2.1 RTK Initialization Search	304
10.3 Algorithm Implementation	305
10.3.1 Peripheral Test Code	307
10.4 Summary	308
Bibliography	311

Preface

About this book

Well, autonomous driving is a such cool stuff, isn't it?

You've probably seen scenes of self-driving cars in science fiction movies. In these vehicles, the steering wheel turns by itself, the throttle and brakes are controlled automatically, freeing people from the monotony of driving so they can enjoy their time more freely. In fact, some Level 2 vehicles have already achieved partial self-driving capabilities in simple road conditions. They can help drivers keep the vehicle centered in the lane or maintain a certain distance from the preceding vehicle. These systems are called **Advanced Driver Assistance Systems (ADAS)**. And for more advanced self-driving systems (Level 4), computers can fully take over, not only assisting drivers but also controlling buses, delivery vehicles, robots, robotic dogs, and even bicycles, enabling many functionalities we never imagined. Over time, these sci-fi-like scenes have gradually become reality. Self-driving jobs have also emerged as a new industry, attracting young talents from all sectors of society. It's truly an exciting development!

The field of autonomous driving encompasses many emerging technologies, with Simultaneous Localization and Mapping (SLAM) being a key focus. Since completing my PhD, I have been involved in the research and development of SLAM in the autonomous driving industry. It's an intriguing area because whether it's large passenger vehicles, small low-speed vehicles, or even sweepers, SLAM is certainly a fundamental technology. Most of the automation features we see are actually embedded within the vehicle's map data. For instance, the map might instruct the vehicle to turn left at the upcoming intersection and merge into the right lane of the opposite road; or, for cleaning a plaza ahead, the vehicle should drive along the right boundary in circles while avoiding the flower bed area in the middle. To achieve these functionalities, we need to integrate various types of data such as GPS, inertial navigation, laser point clouds, visual images, etc., to construct maps and then perform localization on these maps.

If you work in this area, you'll find that it brings together people with diverse backgrounds. People working on inertial navigation are familiar with strapdown inertial navigation systems. They enjoy writing matrix and vector computation programs on a small embedded CPU, often scratching their heads over errors of one or two points of accuracy. Those involved in processing laser point clouds engage in meticulous map reconstruction, displaying beautiful three-dimensional point clouds on the screen. Meanwhile, those working on vision spend their days manipulating images on the imaging plane, producing impressive results but not focusing much on accuracy issues. Well, I mean, accuracy is of course an important issue, but the accuracy on a two-dimensional imaging plane is not the same as the accuracy of three-dimensional world points or localization accuracy. Cameras usually don't have fixed accuracy metrics like other sensors. They can either focus on nearby objects for local high precision or distant objects for a broad field of view, just like the difference

between a microscope and a telescope. Thanks to the collaboration among these individuals and effective communication from management, vehicles are able to navigate the roads stably. However, most of the time, we aren't entirely clear about what other people are doing or how they're doing it. This is one of the motivations for me to write this book.

I hope to introduce readers to the localization and mapping technologies related to autonomous driving and robotics in this book, which include the sensors we use in our daily lives. Although there is currently no unified opinion on what constitutes a vehicle or a robot, they can even be seen as wheeled smartphones. However, these intelligent machines all use similar sensors, and the underlying theories are basically the same. I hope that through this book, researchers in this field can enhance mutual understanding, or colleagues and students outside the field can understand what work we are doing. I believe many people will be interested in these technologies.

Content of This Book

This book introduces SLAM-related technologies used in autonomous driving and robotics. The SLAM discussed here is quite general. We will cover sensors and processing methods related to localization and mapping. A typical autonomous vehicle will include various sensors such as IMU, wheel encoders, vehicle speed sensors, and multi-line laser sensors, so our localization and mapping will also involve methods for these sensors. Therefore, readers will encounter topics like basic algorithms for inertial navigation, filters represented by Kalman filtering, laser point cloud matching methods, trajectory fusion algorithms, and more in this book. Overall, we will introduce these contents in the following order:

1. The first part covers **Basic Mathematical Knowledge**. We will start with basic coordinate system definitions, rotational geometry, and quickly introduce some mathematical background knowledge used in this book. Since most of this background knowledge can be found in other books and materials, we will only provide a brief introduction. Chapter 1 provides an overview of autonomous driving, Chapter 2 introduces basic geometry and kinematics, Chapter 3 covers the error Kalman filter used for integrated navigation, and Chapter 4 introduces pre-integration systems and optimization methods. Readers don't need to worry about the specialized terminology mentioned here; we will delve into them in detail in specific chapters.
2. The second part is about **Laser Localization and Mapping**. This section introduces 2D and 3D laser localization technologies, with the former mainly used in robots represented by sweepers, and the latter being one of the foundational technologies for autonomous driving vehicles. We will detail representative techniques for processing laser point clouds and demonstrate their applications through code implementation. Chapter 5 of Part 2 introduces basic point cloud processing algorithms (nearest neighbor structures, KD trees, etc.), Chapter 6 discusses 2D laser localization and mapping, and Chapter 7 covers 3D laser localization and mapping.
3. The third part focuses on **Application**. We will discuss the process of building high-precision point cloud maps for autonomous driving and how to use point cloud maps for real-time localization. Chapter 8 introduces tightly coupled laser-inertial odometry methods, Chapter 9 presents offline point cloud mapping systems, and Chapter 10 introduces online fusion localization systems.

Similar to my previous book [1]¹, this book emphasizes the unity of theory and practice and pays close attention to the **implementation** of principles in code. All algorithms men-

¹<https://github.com/gaoxiang12/slambook-en>

tioned in this book will have code implementations provided in the corresponding chapters. Readers will work with us to implement those important and foundational algorithmic structures in this field from scratch, using modern programming techniques and fully exploiting parallelization principles to make our algorithms run faster than classical implementations. Consequently, we will not limit ourselves to specific implementations of open-source code. For example, we will avoid discussing what LOAM [2] does from line X to line Y or which library Cartographer [3] references in a particular cpp file. We will refrain from discussing engineering details like thread pools or parameter file formats. Yes, such discussions can be too detailed, and everyone's implementation may differ. We will strive to retain only the core algorithmic code, allowing readers to debug and understand the entire process themselves.

In terms of style, I will continue to use my familiar writing style. Readers who are familiar with me should quickly adapt, while those who are not should not find it overly challenging. I hope my writing is as clear and straightforward as a conversation. Throughout the introduction of content, I hope the reading process reflects a complete train of thought, rather than simply compiling information together. Although this writing style may result in some verbosity, I believe it is beneficial.

The majority of the key contents in this book will be accompanied by corresponding implementation code. This is one of the major features of this book. I believe that for a comprehensive book, providing code that demonstrates the concepts is always a wise choice. However, despite our efforts to streamline the code, the code section of this book is still much larger than my previous book.

Here is our code repository:

```
https://github.com/gaoxiang12/slam\_in\_autonomous\_driving
```

All code and data for this book are open-source and freely accessible to readers. The PDF file of this book will be continuously updated within the code repository. We use C++ as the primary programming language. Please don't ask me why I didn't use more concise languages like Python or Matlab because the programs running in actual vehicles or robots are still primarily C++ programs, and I don't want our experiments to deviate too far from industrial applications. Please note that the code, errata, and other files for this book will be updated on GitHub first, while the published version of the book may have a certain lag time due to the printing schedule of the publishing house. If readers find any discrepancies between the content in the book and the code repository, please consider the implementation in the code repository as authoritative.

We welcome readers to ask questions or answer questions from other readers in the code repository of this book. We encourage readers to communicate in English to facilitate sharing your experiences with international friends.

How to Use This Book

The content of this book follows a process of gradually deepening complexity, but even basic topics like those in Chapter 2 require some groundwork. Personally, I hope this book serves as a sequel to my previous book [1]. You should at least read the first 6 chapters of that book to familiarize yourself with some basic mathematical principles and the basic usage of optimization libraries. However, if readers have not read that book, you should at least have knowledge in the following areas:

- Basic undergraduate-level mathematics such as calculus, linear algebra, and probability theory.

- Mathematics at the graduate level: optimization, matrix theory, a small amount of knowledge about Lie groups and Lie algebras.
- Computer science: Linux system operations, C++ programming language.

If readers find certain parts of this book difficult to understand, they can refer to corresponding reference books for supplementary learning. You can of course skip some chapters since they are relatively separated by the contents. Overall, this book will be slightly more challenging and the pace of introduction will be somewhat faster.

The code for this book is organized by chapter. For example, the code for Chapter 3 will be located in `src/ch3`. The code for each chapter will be compiled into separate library files and executable files. Additionally, shared code will be placed in `src/common` (such as some common structures, message definitions, UI, etc.). There is a certain degree of dependency between the code of different chapters, with later chapters reusing the results of earlier chapters. The code for this book needs to be compiled using ROS, but the actual running and testing processes do not require the use of ROS mechanisms; only ROS data packages are used for storage. Readers only need to understand the installation process of ROS and do not need to familiarize themselves with the details of ROS in advance.

Notations

The mathematical symbols in this book follow the international standard. In general, scalars are expressed in slanted font, such as a ; matrices and vectors are represented in bold font, such as \mathbf{A} ; special sets are denoted in hollow sans-serif font, such as \mathbb{R} ; and Lie algebra-related sets are expressed in Gothic font, such as $\mathfrak{so}(3)$. We aim to maintain consistency throughout the book in terms of symbols, with additional explanations provided where ambiguity may arise.

Relationship with Other Books and Papers

Autonomous driving localization technology involves many active research fields. For example, Professor Barfoot's "State Estimation for Robotics" [4] focuses on introducing state estimation theory. Its Chinese translation was also translated by our team. On the one hand, it provides a comparative introduction to the differences and similarities between traditional filtering theory and modern optimization theory, and on the other hand, it provides an excellent introduction to Lie groups and Lie algebra for engineering readers. This book will partially use some conclusions from the book on state estimation, mainly the part related to Lie groups and Lie algebras, to support some of our formula derivations.

Professor Ma's "An invitation to 3-d vision: from images to geometric models" [5] is also an excellent book that introduces knowledge of 3D vision, with many similarities in the basic knowledge of 3D geometry.

Joan Sola's "Quaternion Kinematics for the Error-State Kalman Filter" [6] provides a very concise and precise theory of quaternion-based error Kalman filters. Although it is not lengthy, it discusses quaternions and Kalman filters very thoroughly, and most derivations in this field are based on this material. This book will also use some of its results, but we will mainly derive various filter formulas based on Lie groups rather than quaternion forms.

Professor Thrun's "Probabilistic Robotics" [7] is also a well-known classic book in the field of robotics. It introduces some results related to SLAM in the field of robotics, and provides a very detailed introduction to traditional filters, 2D grid maps, and other content. This book will also introduce 2D grid localization and mapping methods, with the theoretical part also referencing this book's content.

Professor Qin Yongyuan and Professor Yan Gongmin's works in the field of inertial navigation, including "Inertial Navigation" [8], "Kalman Filter Algorithm and Combination Navigation Principle for Strapdown Inertial Navigation" [9], and "Inertial Instrument Testing and Data Analysis" [10], are classic textbooks in this field, and many teachers and students studying inertial navigation will refer to their derivation process. This book also refers to these books in the field of inertial navigation, but compared to specialized textbooks on inertial navigation, the content introduced in this book will be relatively basic. We mainly introduce the basic principles of inertial navigation, without involving complex parameter compensation or discussions on various subdivided motion states. However, in contrast, the preintegration principle and nonlinear optimization part introduced in this book are not fully introduced in these traditional textbooks.

Finally, compared to the books and materials mentioned above, the biggest feature of this book is still the unity of code and theory. It can be said that most books are for reading, while this book can be **executed**. I believe that understanding many algorithmic aspects requires readers to participate in the debugging and running process.

Environment

This book uses Ubuntu 20.04 as the experimental environment. Readers can use their personal computers as development environments. If familiar with Docker, they can also use Docker environments. The book primarily utilizes **C++17** as the C++ standard, which may be relatively new to some readers. Older machines or environments may not necessarily support it well. We recommend readers to use software environments above Ubuntu 20.04 to run the code in this book; otherwise, you may need to address some minor issues regarding C++ standard support.

This book comes with a considerable amount of test data, which is quite large (approximately 270GB). We suggest that readers allocate at least 100GB of space to run the code in this book. Readers can download the test data through the links provided in the book's repository.

Acknowledgement

1. Considering the confidentiality of geographical information, this book avoids using domestic data and tends to use open-source datasets worldwide. Readers can consider the trajectories or point clouds provided in the book as data in a general spatial coordinate system, without concerning themselves with the actual geographical locations of this data.
2. Similarly, unless necessary, the data provided in this book will not specify geographical information such as place names or ranges. Readers can regard them as general road, plaza, or building scenes.
3. Some of the images used in this book are sourced from internet search engines and are used solely for educational purposes, with no intention of infringing on the original authors' copyrights. Some of the images used in this book may contain logos of commercial companies or may be images used in promotional materials for some companies. These images are sourced from public search engines and do not imply any cooperation or competition relationship between the authors and the companies. The author will strive to obtain authorization for images that may have commercial copyrights. If there is any dispute, please inform us.

4. Each chapter of this book uses datasets from different sources, mainly including the NCLT dataset from the University of Michigan [11], the UTBM dataset from Mont-béliard, France [12], and the UrbanLoco dataset mainly from Hong Kong, China [13] (ULHK), among others. This book has designed a unified interface for them programmatically, making it convenient for readers to test the performance of algorithms on different datasets.
5. The English version of this book is translated with the help of ChatGPT/DeepSeek and I would like to thank their great work here.

Part I

Basic Knowledge



Chapter 1

Autonomous Driving

1.1 Autonomous Driving Technologies

1.1.1 Autonomous Driving Capabilities and Grading

Autonomous driving, as the name suggests, is the study of enabling vehicles to drive by itself. If you were to design an autonomous driving system, where would you start?

Although the internal structure of a car is highly complex, what humans actually need to operate is simply looking ahead, manipulating the steering wheel, accelerator, and brake pedals. If a computer program also learns to send signals to the steering wheel and pedals based on information from camera images, does it qualify as learning autonomous driving? If so, how should this program be designed?

In a naive conception, to enable a car to autonomously drive, one should first observe how humans perform driving behaviors. Humans primarily use vision to judge the relationships between their own vehicles and surrounding vehicles, pedestrians, and roads. They determine the direction of travel by observing lane markings and then use maps to determine long-term route planning. Similarly, autonomous vehicles should possess these abilities. Let's make a simple analogy:

1. Autonomous vehicles should be able to identify the types of surrounding vehicles and pedestrians in real time, recognize road signs and signals, such as common lane markings, traffic lights, and traffic signs. This is called the **perception** capability of the vehicle [14, 15].
2. The vehicle should be able to determine its own direction and position, as well as the positional relationship between itself and the aforementioned elements. This is also known as the **localization** capability of the vehicle [16–18].
3. The vehicle should be able to control the throttle, brake, steering wheel, and other actuators based on the aforementioned signal recognition results, and plan short-term and long-term driving routes. This is called the **planning and control** (P&C) capability of the vehicle [19, 20].

However, despite addressing the same problems, the capabilities of humans and computers differ greatly. Throughout the long history of evolution, humans have developed extremely powerful spatial **perception** abilities. We can understand the vast majority of objects in our field of vision in an instant, with minimal errors. We also possess strong

learning abilities; even when encountering unfamiliar objects, we instinctively avoid them. We can quickly understand the structure of the road ahead in any weather and scene, and even drive normally on roads without lane markings. We can also communicate with surrounding vehicles through lights and sounds, and predict their actions based on the behavior of other vehicles. In some defensive driving techniques, we can even infer potential dangers in blind spots. Due to these powerful understanding abilities, we can drive vehicles freely based solely on vision, without precise position and attitude information, unlike autonomous vehicles, which require expensive ranging devices like LiDAR, high-definition maps, and high-precision positioning to precisely control vehicle behavior (Figure 1-1).

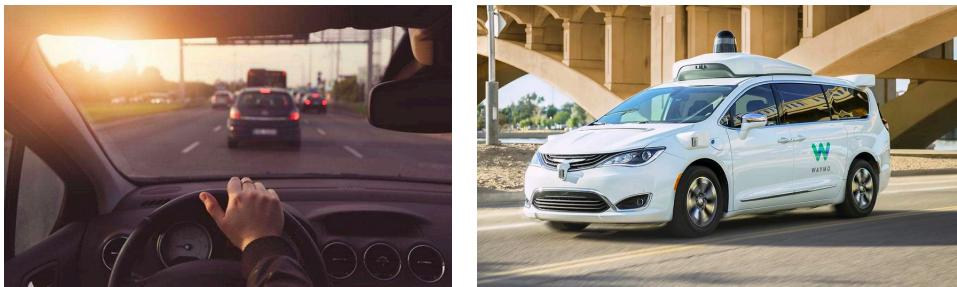


Figure 1-1: Cars driven by humans versus cars driven autonomously. Humans can drive solely based on vision, while autonomous vehicles currently rely on high-precision ranging devices and backend high-precision map services.

If we were to draw a comparison between birds and airplanes, our human driving abilities would resemble the effortless and natural flight of birds in the sky. However, for aircraft designers, should they make airplanes flap their wings like birds? The reality is quite different. Aircraft possess sophisticated control devices to manipulate airflow over their wings to generate the necessary lift; they also have precise measuring instruments to determine their own attitude and use modern control methods to maintain the desired orientation. The relationship between autonomous and human-driven vehicles is very similar to that of airplanes and birds. We can draw upon certain human abilities to design autonomous driving systems, but the resulting autonomous vehicles will inevitably differ significantly from human-driven ones. We don't necessarily need autonomous vehicles to behave exactly like human-driven ones. Autonomous vehicles should have their own design and operational logic.

In fact, autonomous driving vehicles are already capable of providing automated driving experiences. In several major cities in China (such as Beijing, Shanghai, Changsha, Chongqing, Wuhan, Shenzhen, among others), autonomous taxis (Robotaxis) has been opened to the public and can be experienced at any time. If you were to sit in an autonomous vehicle, you would notice that the steering wheel turns automatically, and braking and acceleration do not require human intervention. You might realize that if the vehicle were entirely controlled by a computer, there would be no need to install those steering wheels, pedals, or central control panels in the car. Additionally, you can see on the vehicle screen the planned route, surrounding vehicles and pedestrians, and data provided by HD maps. Since 2018, these vehicles have undergone several years of pilot testing but have not yet reached mainstream consumers.

On the other hand, if you were looking to buy a vehicle soon, most vehicles would advertise their autonomous driving features. They can automatically maintain a fixed speed on highways, eliminating the need for you to operate the accelerator; they can also automatically steer to keep the vehicle in the center of the lane, relieving you of steering duties. Some ve-

hicles even offer features like lane change assistance or automatic lane changing, so you can leave lane changing to the vehicle. They can even handle automatic following in congested traffic conditions. These features indeed assist drivers in vehicle operation. Moreover, these vehicles are tangible and can be purchased at widely accepted prices. Most of them use purely visual or predominantly visual sensor solutions.

Are all of these considered “autonomous driving”? If so, why is it difficult to purchase the former now, while the latter can appear directly on the consumer market?

This is precisely the situation that autonomous driving faces today: if we want to achieve driving capabilities similar to humans and have vehicles drive entirely on their own without the need for drivers, it requires a hefty price to implement such functionality. This cost could be attributed to laser radar sensors, backend map services, machine costs, research and development investments, and so on. When vehicles no longer require drivers, many business models will change accordingly. Taxi companies will no longer need drivers, food delivery companies will no longer need delivery personnel, and all operational personnel will only need to maintain autonomous driving vehicles. On the other hand, if we want to keep the price of vehicles within the reach of consumer-grade vehicles by using inexpensive, practical sensors, then we must acknowledge the current lack of reliability in existing algorithms, necessitating human driver supervision and intervention at any time, unable to completely replace human drivers. In fact, the entire autonomous driving industry is currently exploring along these two paths. The former path is highly likely to lead to fully autonomous driving, but currently, it seems too expensive to be directly targeted at consumers; the latter path is known as the “incremental” route, which is being used by various vehicle manufacturers, but it still has a significant distance from fully autonomous driving and is not suitable for tasks that require fully autonomous driving capabilities.

This divergence in paths reminds us that sometimes when discussing autonomous driving, different parties may not necessarily be talking about the same tasks. In fact, if we only care about “some” autonomous driving tasks, such as automatic following, lane keeping, and automatic lane changing, they do not require very complex technology or sensors. Although we sometimes refer to these vehicles as “autonomous driving” vehicles, and their manufacturers are also willing to claim that they are “fully autonomous driving,” they still belong to the category of “assisted driving” functions according to standards. This also reminds us that there should be a clear and standardized delineation of autonomous driving capabilities.

In the international arena, researchers have long classified vehicles into five levels of autonomy, from Level 1 to Level 5 (SAE classification¹), as shown in Table 1-1. Similar to this, China has its own “Classification of Automotive Driving Automation”², the summary of which is presented in 1-2. Generally, the various standards for classifying autonomous driving capabilities are primarily based on the following two points:

1. Whether the system requires human intervention, also known as **intervention**. **Assisted driving systems** require the driver to take over when intervention is needed, while **autonomous driving systems** strive to operate without the need for human intervention, allowing the removal of driver equipments such as steering wheels and pedals. This is the key distinction between Level 2 and above Level 3 autonomous driving capabilities.
2. Whether the system operates in limited scenarios or can function in most normal scenarios relative to human drivers. This is the key difference between Level 4 and Level 5.

¹Classification by the Society of Automotive Engineers, see SAE Standard J3016-202104: https://www.sae.org/standards/content/j3016_202104.

²See: GB/T 40429-2021.

Table 1-1: SAE Automated Driving Levels

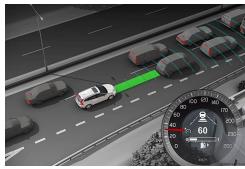
Level	L0	L1	L2	L3	L4	L5
Responsiblity	Driver			Computer		
Intervention	at all times		when required		not required	
Typical Func.	AEB BSD LDW ALC	LCC ACC	LCC+ACC	Traffic Jam Driving Auto Parking Auto Summoning	Robotaxi Robotruck	all conditions

Abbreviations: LDW: Lane Departure Warning, ACC: Adaptive Cruise Control, LCC: Lane Centering Control, BSD: Blind Spot Detection, AEB: Automatic Emergency Braking, ALC: Auto Lane Change.

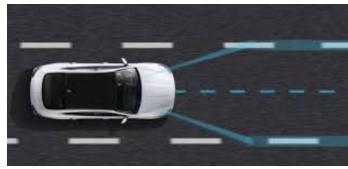
Table 1-2: China Classification of Automotive Driving Automation

Driving Level	Name	Main Contents
L0	Emergency Assist.	Detection and response capabilities for certain events
L1	Partial Driver Assist.	Continuous execution of lateral and longitudinal control
L2	Combined Driver Assist.	Continuous execution of lateral and longitudinal control
L3	Conditionally Auto. Driving	Continuous execution of all driving tasks
L4	Highly Auto. Driving	User may refrain from taking over
L5	Fully Auto. Driving	Can autonomously drive in any environment

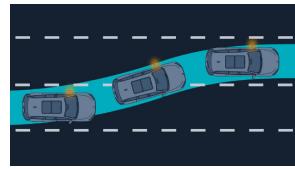
Therefore, although there are five to six levels in terms of classification, for professionals in the field of autonomous driving, the main concern lies in Levels 2 and 4. Level 2 autonomous driving vehicles can be directly targeted at consumers, enhancing driving comfort to a certain extent based on traditional vehicles. The functionalities currently achieved at Level 2 are not far from current laws and regulations and are gradually being popularized in some new car models. On the other hand, Level 4 vehicles should be capable of unmanned driving in most scenarios, able to address some business needs for automation, which is also considered by many researchers as a form of “driverless” mode. Although within the broad definition, both Level 2 and Level 4 are part of autonomous driving, in practical terms, there are fundamental differences in module design and implementation between them. Level 4 autonomous driving is most concerned with the **intervention rate**, requiring that the vehicle cannot be taken over by human without cause, thus placing high demands on various algorithm performances. On the other hand, all functions of Level 2 autonomous driving can be taken over by humans, thus emphasizing the recognition of which scenarios are **valid scenarios**, where Level 2 functionalities can be activated. Due to human intervention, Level 2 is much more tolerant of most algorithm metrics, placing greater emphasis on the presence of functions rather than complete automation.



ACC: Adaptive Cruise Control



LCC: Lane Centering Control



ALC: Automatic Lane Change

Figure 1-2: Typical L2 tasks: ACC, LCC, ALC.

1.1.2 Typical L4 Tasks

L2 or L4, that is the question.



Figure 1-3: Some applications of L4 passenger vehicles: autonomous taxis, buses, trucks, mining trucks

However, this question shouldn't be answered by technical personnel. We should first ask, does a certain type of vehicle really need to drive completely autonomously? In some scenarios, the answer is “yes”. **Automation** is the core functionality of these vehicles. Without full automation, these vehicles lose their *raison d'être*. In other scenarios, we might say “not necessarily.” What we need more is safe driving with computers helping to **alleviate burdens**. If computers can provide more advanced functions, we are willing to accept them, but we also need to consider the cost of these functions. If the cost is too high, consumers won't buy it.

The former belongs to typical L4 applications, and the entity doesn't have to be limited to **vehicles**. In a broad sense, as long as a chassis carries sensors and has a certain level of automation, it can be considered a form of autonomous driving vehicle. From this perspective, whether the entity is a car or carries passengers is not the key to distinguishing autonomous driving. Whether the tasks they perform require **full automation** is the key to distinguishing their autonomous driving capabilities. For example, an autonomous delivery vehicle's primary function is to autonomously deliver items to users. If this function isn't fully automated and still requires a driver's cooperation, then the business loses its main feature. An autonomous cleaning vehicle's main function is to automatically cover cleaning in fixed scenes. If this process still requires human involvement, it's not meaningful. For these businesses, **removing the driver** is their most important feature, so they belong to core L4 applications. These vehicles are not designed with driver cabins or driver positions (Figure 1-4).

On the other hand, we can also ask, do taxis need autonomous driving? Do trucks need autonomous driving? If there are no drivers, taxis can be operated solely by taxi companies, and trucks can also be operated solely by logistics companies, without the need to recruit drivers, only requiring vehicle maintenance. This business model is called Robotaxi and Robotruck, which is a completely different way from existing commercial models (see Figure 1-3). It imposes strict requirements on autonomous driving. Once a vehicle experiences a failure and requires human intervention, in the absence of a driver on board, it's difficult



Figure 1-4: Low-speed L4 applications: sweepers, delivery bots, patrol robot

to have a human driver intervene in a timely manner, and a takeover event could easily turn into an accident.

Applications such as Robotaxi, Robotruck, Robobus, etc., technically also belong to L4 autonomous driving. Compared to autonomous driving vehicles for cleaning, delivery, inspection, etc., they have higher requirements for autonomous driving safety, stricter requirements for overall vehicle system stability, lower tolerance for risks and failures, and require more perception, high-precision positioning, mapping, and other technical support. If a low-speed vehicle experiences a failure, it won't directly lead to casualties, and most of the time, it can be remotely managed by technical personnel. However, if a passenger-carrying vehicle experiences an accident, the consequences can easily have substantial impacts at the company or even the community level. So, we ask, can the current level of technology support applications like Robotaxi? Unfortunately, this question currently does not have a definitive answer.

On one hand, autonomous driving systems are complex systems, unlike traditional electronic or mechanical switches, where it's easy to provide a functional safety verification plan or give clear reasons for failures. If an autonomous driving vehicle fails to recognize a vehicle in front and collides, within the current theoretical framework, we can't explain **why the system failed to recognize the vehicle ahead**. It's just a phenomenon that occurs in reality. Perhaps if a certain value in a certain program is increased by 0.001, this phenomenon won't occur, but it may make it impossible to recognize vehicles of a different color in different weather conditions. There are hundreds of millions of parameters like this, none of them have names, and they are connected and calculated in a way arbitrarily stipulated by humans. It's difficult to attribute the occurrence of a specific result to a parameter being too large or too small, or the calculation sequence between them not being reasonable enough. A brake system is almost impossible to fail, but a perception system is almost impossible to be 100% correct. In short, autonomous driving systems are difficult to precisely analyze what failure at a certain point may lead to what phenomenon, and provide convincing reasons, like traditional mechanical and electronic systems.

On the other hand, if it's difficult to verify the safety of autonomous driving vehicles from a theoretical level, can we statistically measure the stability of autonomous driving systems at an experimental level? This is indeed what many autonomous driving companies are doing now. Most L4 autonomous driving companies will statistically analyze the relationship between vehicle mileage and takeover times, for example, calculating the **miles per intervention** (MPI)³, to measure the stability of the system. In the 2021 report of the California Department of Motor Vehicles (DMV) on autonomous driving, the MPI of some Chinese companies has reached the level of thousands to tens of thousands of kilometers (see Table 1-3)⁴. We generally believe that MPI is indeed an indicator of the overall autonomous driving capability of a vehicle, but so far, there is no very open and fair MPI testing method, and what we can see more are self-testing reports from various companies. They lack unified

³Sometimes also called miles per disengagement, MPD.

⁴Source: <http://www.evinchina.com/articleshow-217.html>

Table 1-3: 2021 report of the California Department of Motor Vehicles (DMV) on autonomous driving

Company	Num. of cars	Num. of Invent.	Miles	MPI
Waymo	693	292	2325843	7965
Cruise	138	21	876105	41719
Pony.ai	38	21	305617	14553
Zoox	85	21	155125	7387
Nuro	15	23	59100	2570
Mercedes-Benz	17	272	58613	215
WeRide	14	3	57966	19322
AutoX	44	1	50108	50108
DiDi	12	1	40745	40745
Argo AI	13	1	36734	36734
DeepRoute.ai	2	2	30872	15436
Nvidia	6	82	28004	342
Toyota	4	419	13959	33
Apple	37	663	13272	20
Aurora	7	9	12647	1405
Lyft	23	23	11200	487
Almotive	2	106	2976	28
Gatik ai	3	6	1924	321
Qualcomm	3	143	1635	11
Apollo	5	1	1468	1468
SF Motors	2	61	875	14
Nissan	5	17	508	39
Valeo	2	205	336	2
Easymile	1	222	320	1
Udelv	1	46	60	1
Inceptio.ai	2	0	39	-
UATC	3	31	14	0.5

testing environments and standardized criteria for when to intervene. In terms of quantity and mileage, compared to traditional mass-produced vehicles, most L4 autonomous driving companies only have fleets consisting of dozens or hundreds of vehicles, and their road test scenarios are usually relatively simple. Compared to traditional manufacturers with monthly sales of tens of thousands of vehicles, their accumulated test mileage and number of scenarios are very limited.

The dilemma of whether to have vehicles first or autonomous driving technology first is facing all L4 autonomous driving companies. Without a sufficient number of vehicles, it's difficult to prove that an autonomous driving system is stable enough, and thus challenging to truly implement businesses like Robotaxi; however, if we focus solely on the vehicle itself without paying attention to autonomous driving technology, it's also challenging to attract enough technical talents and cultivate the team's technical capabilities. Several years ago, it was a common problem that L4 technology companies didn't understand vehicles, and vehicle manufacturers didn't understand autonomous driving. Doing both requires enormous scale and determination, while collaboration between two different companies requires a high level of trust. Fortunately, major vehicle manufacturers have begun to pay attention to autonomous driving-related businesses (although currently focused on the easier-to-implement L2 business), and L2-related functionalities have been integrated into many vehicle models. Many practitioners also believe that as autonomous driving technology matures, L2 functionalities will gradually become richer and gradually approach L4 capabilities. Suppliers of autonomous driving-related sensors, algorithms, chips, and hardware will also play active roles in various vehicle models.

Even after overcoming technical issues, businesses like Robotaxi still face practical legal and social issues. After all, if L4 vehicles are widely deployed, other drivers will face the challenge of how to interact with driverless vehicles⁵. Will driverless vehicles understand the intentions of other vehicles changing lanes or overtaking? Will they avoid vehicles driving in the wrong direction? Will they detour around road sections undergoing sudden construction? Will they recognize children who have fallen on the road? Legally, how should the responsibility of driverless vehicles be defined if they collide with other vehicles? Should the responsibility lie with the developer of the driverless vehicle? Should the developers be held accountable if the collision occurs because the perception system failed to correctly identify pedestrians, or because the map annotation personnel incorrectly annotated the speed limit information of a road section, or because the satellite signal was weak that day, causing the vehicle to enter the wrong lane? These are questions that may be encountered in reality but are difficult to answer. In vehicles with human drivers, most safety responsibilities ultimately fall on the driver. Once the subject becomes a driverless vehicle, these responsibilities are difficult to distribute among various modules. It's unlikely that any company currently has the ability to assume these responsibilities on behalf of autonomous driving development companies. In short, discussions on many legal, ethical, and social issues related to autonomous driving will continue for many years to come.

However, autonomous driving still represents the direction of future technological advancement. Its overall prospects are promising, albeit with inevitable challenges along the way. Future passenger vehicles, low-speed vehicles, and robots will become increasingly intelligent, taking on more and more tasks in daily life. A series of technical issues derived from autonomous driving will also be fully addressed and discussed by researchers in various fields. Many things that appeared in science fiction movies will gradually become familiar landscapes in the coming years. For example, restaurant robots that the author fantasized about during their studies were once among the representatives of science fiction, but now they are widely present in major shopping malls, and major suppliers have begun price wars. Will autonomous driving vehicles also experience this situation in the coming years? We wait and see.

1.2 Localization and Mapping in Autonomous Driving

1.2.1 Why L4 Needs Localization and Mapping

This book primarily discusses the localization and mapping technologies in L4 autonomous driving. Before delving into the details, readers might ask a fundamental question: why does autonomous driving require localization and mapping?

That's an excellent question. My answer is, if high-precision localization and mapping weren't necessary for autonomous driving, that would be ideal. Unfortunately, at the current technological level, achieving low intervention rates in L4 autonomous driving still requires the use of high-precision localization and mapping. Conversely, if we're discussing L2 autonomous driving, which doesn't prioritize intervention rates, high-precision localization and mapping may not be necessary (although some L2 systems are also using high-precision maps) [21–23]. This contradicts the current level of intelligence and reliability. The smarter it is, the less reliable it becomes; the more reliable it is, typically meaning simpler in structure, the less intelligent it is. If we choose to accept the results of AI, we must also accept the mistakes made by AI. As for when L4 is needed and when L2 is needed, we've discussed this earlier.

⁵Throughout this book, the term **driverless vehicles** generally refers to L4 autonomous driving vehicles.

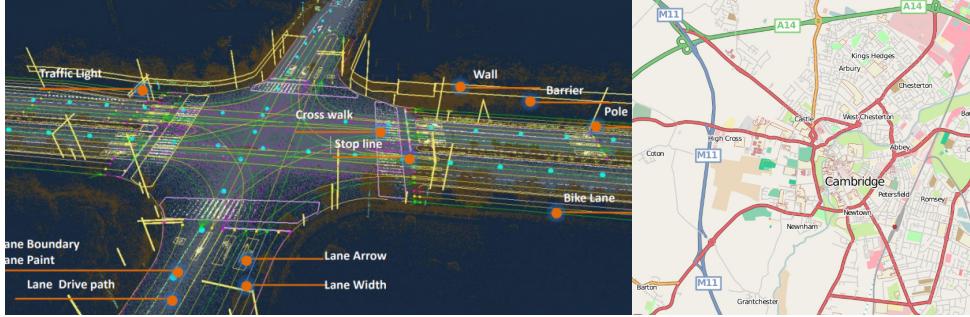


Figure 1-5: Differences between high-definition maps and traditional electronic navigation maps. Navigation maps represent roads and intersections using polygons and vectors, while high-definition maps also accurately mark lane positions, stop line positions, and provide detailed information about surrounding objects.

Why does L4 need high-definition maps? This is because L4 and L2 have different goals. L2 autonomous driving doesn't concern itself with intervention rates, whereas achieving low intervention rates is the primary goal of L4, which determines that their technological paths have fundamental differences. L4-related technologies exhibit strong determinism. Many behaviors that may seem acceptable at the L2 level, such as turning into the wrong lane at an intersection or misreading a traffic light, would result in intervention in L4 and are not allowed to occur. Consequently, L2 leans more toward real-time perception and may even use perception results directly to construct a bird's eye view (BEV) [24–26], while L4 relies on offline maps [27–29]. In simple terms, L2 is more like “driving while looking at the road,” whereas L4 is “driving in the map in the mind.” “Driving while looking at the road” has the advantage of very direct logical relationships, closer to human behavior, but the disadvantage is dealing with the limitations and uncertainties of computer detection results. Vehicle cameras typically only see the ground lane lines a few dozen meters ahead. They may also be obscured by other vehicles, submerged in puddles, or, due to lighting conditions, shadows of roadside guardrails may be mistaken for lane lines. These outcomes could affect the vehicle’s autonomous driving behavior, and these uncertainties must be considered at the control and decision-making levels. In contrast, high-definition maps in L4 are meticulously annotated by humans, with accurate positions for each lane. It’s predetermined which lane they will continue onto, which direction to turn at intersections, and which traffic light to look at in three-dimensional space [30]. Even if no lane lines are visible in real-time images, L4 vehicles can accurately follow straight lines in the map [31]. This approach comes with two costs: first, we need to create such a high-definition map in advance; second, we need to know our accurate position in this map [32, 33].

High-precision localization and mapping represent a strict, accurate concept. On the flip side, it brings about rigid, cumbersome business burdens. Maps essentially transform those limited, uncertain perception elements into static, precise data information through manual or post-processing methods (Figure 1-5). Maps carry out similar tasks to real-time perception but can provide correct results across unlimited ranges, greatly reducing the burden of perception [34]. Therefore, some have said that maps are a cheating sensor, essentially providing the answers directly to autonomous driving vehicles. With high-definition maps, the vehicle’s burden of perception can be significantly alleviated. We only need to focus on dynamic pedestrian and vehicle information, without worrying about the shape and topology of road lanes. However, the balance between maps and perception is constantly changing. Some companies’ high-definition maps are richer than others, even including information

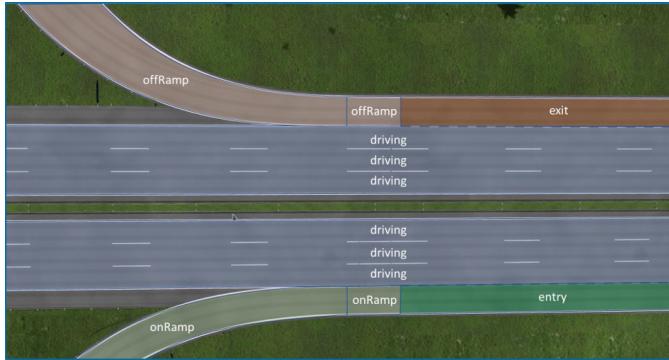


Figure 1-6: Common lane information in high-definition maps

about obstacles, flowerbed shapes, etc., while others' solutions require the perception module to detect this information. Perhaps in the near future, the balance between maps and perception will change with technological iterations.

In current L4 autonomous driving solutions, most task elements are tied to maps. When users want to drive from point A to point B in a city, the autonomous driving vehicle first generates a lane-level path from point A to point B on the map. This path is different from the common navigation systems we're familiar with; it's at the lane level. The navigation system calculates which road, which intersection, and which lane to turn into. When executing autonomous driving tasks, the vehicle also strives to ensure that the actual executed path aligns with the results of high-definition map navigation. For this reason, the vehicle needs to know its real-time position on the map, which requires high-precision localization within lanes.

1.2.2 Contents and Production of High-Definition Maps

High-definition maps are essentially **structured data** [35]. Their basic elements consist of sections of lanes in the real world, as depicted in Figure 1-6. Various questions about lanes can be asked, such as:

1. What is the geometric shape of this lane? Is it straight, with bends, or curved?
2. Which lane is to its left, and which is to its right?
3. What is the speed limit? Is it a straight lane or a turning lane?
4. Is it a motorized lane or a non-motorized lane?
5. Which lanes is it connected to? Are they sequentially connected, or are there forks or mergers?

Questions of this sort are easily described and stored using **structures** in programs. Various programming and markup languages support struct syntax, making high-definition map software compatible with multiple languages, including JSON, Protocol Buffers (protobuf), XML, and others. Describing a complete lane's information can be quite extensive and challenging. Researchers worldwide have therefore developed standards for high-definition maps. Common standards include OpenDrive [36], LaneLet2 [37], and Apollo OpenDrive.

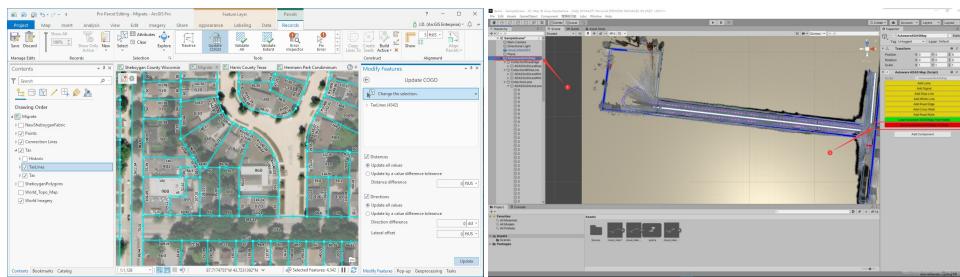


Figure 1-7: Common high-definition map editing software: ArcGIS and Autoware Map Tool

These standards define ways to describe a lane, an intersection, and specifics of various traffic lights. Readers can refer to these standards for a comprehensive understanding of field information.

Most geometric elements in high-definition maps are described by points. For instance, a lane can be described by its center reference line plus its width, or by lines marking its left and right boundaries, which are composed of a series of lower-level points. The coordinates of each point can be expressed in terms of latitude and longitude or other global coordinate systems discussed later. Typically, they are represented by floating-point numbers. On the other hand, area-like elements can be described by polygons formed by multiple points, such as parking lots or buildings. Once this information is exported to files, it can be used for map rendering or for vehicle navigation and control.

Since high-definition maps are essentially made up of these lines and information, can't we generate them freely? Certainly, we can. We can even draw a road on paper and say its speed limit is 60 kilometers per hour; this can be considered a high-definition map, albeit with limited practical use. High-definition maps on computers are usually generated using specialized drawing software (such as ArcGIS, Autoware Map Tool, as seen in Figure 1-7), and some companies may develop their own drawing software. You can certainly start from a blank area and draw a virtual map. However, if we want the map to correspond to the real world, we need to find a way to first obtain the three-dimensional structure or two-dimensional aerial view of the real world. These serve as the data source for real-world high-definition maps.

The two-dimensional or three-dimensional data from the real world mainly come from the following sources:

1. Satellite imagery from remote sensing satellites. Satellite imagery can be obtained in various mapping software, but the highest resolution of civilian satellite imagery is at the level of a few meters, and images become blurred at around 10 meters. While they cover the entire globe and can be used for annotating electronic navigation maps, they are insufficient for high-definition maps, making it difficult to discern the specific positions of lanes and road surface details (see Figure 1-8, top left).
2. Aerial imagery from drones. Drones equipped with high-precision positioning devices can take top-down photographs at any altitude, which are then stitched together to create aerial imagery. This imagery can be highly detailed, but its coverage is limited, and flying drones is prohibited in many areas.
3. Three-dimensional map reconstruction using sensors carried by autonomous vehicles. The most common method is to use onboard LiDAR sensors to construct three-dimensional point clouds of the scene. These point clouds reflect the three-dimensional

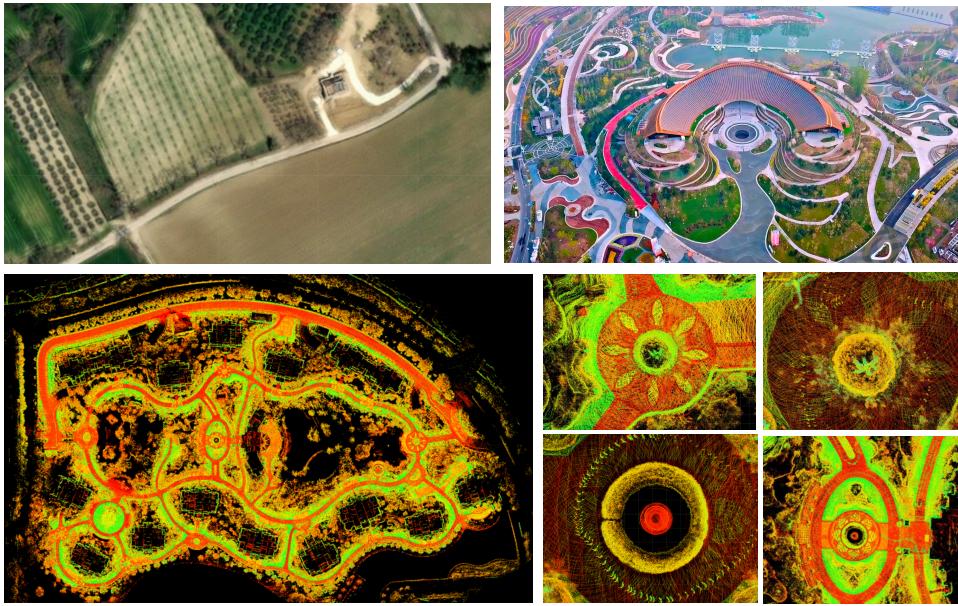


Figure 1-8: Data sources for high-definition maps: satellite imagery, drone aerial imagery, laser-generated point cloud maps (global and local)

structure and brightness information of the scene and can effectively serve as a reference for map drawing. Since autonomous vehicles have relatively unrestricted ranges of travel, most autonomous driving companies use this method to construct point cloud maps [38, 39].

Figure 1-8 shows several different data sources. They follow a simple logic: the closer they are, the clearer they appear. Compared to remote sensing satellites orbiting tens of thousands of kilometers above the Earth, drones can hover tens of meters above the ground, while cars can directly capture images or measure distances in front of objects. Satellite images often struggle to discern road surface details, while drone images and LiDAR point clouds can reflect the texture of the road, the shapes of tree trunks and trees, and various small objects in the scene. Based on vehicle point clouds, we can annotate various detailed objects. If time permits, we can even annotate details such as the texture of tiles and the positions of tree trunks in high-definition maps.

1.3 Introduction to SLAM in this Book

The next question is how to use sensors onboard vehicles for high-precision point cloud reconstruction? What principles underlie this type of three-dimensional reconstruction? Apart from annotating maps, what other purposes do they serve? We will explore all of these questions throughout the content of this book. We will see that high-precision point clouds are the result of the combined action of a series of sensors. Their prices range from hundreds to hundreds of thousands, each serving different purposes. The core of the three-dimensional reconstruction system lies in estimating the vehicle's position and attitude at each moment, involving the vehicle's own **kinematic theory** and the **state estimation theory** used to estimate the vehicle's state using sensors. The following chapters of this book will introduce

various sensor principles and methods in a certain order. The rough sequence is as follows:

1. Firstly, we need to introduce some basic geometric knowledge, including the sensor coordinate systems on the vehicle and the coordinate systems of the Earth. At the same time, we will review the basic knowledge of state estimation theory, including the Kalman filter and nonlinear optimization theory. This part of the knowledge has been introduced in my previous book [1], so we will not elaborate further but only review. In particular, this book will mainly use properties on $\text{SO}(3)$ when dealing with rotation variables. Readers need to maintain a certain proficiency in this part of the content. This is the content of Chapter 2 of this book.
2. Chapters 3 and 4 will introduce two mainstream methods for processing **inertial measurement unit** (IMU) data. Chapter 3 introduces the classical Error State Kalman Filter (ESKF) and handles rotations in $\text{SO}(3)$, while Chapter 4 mainly introduces preintegration methods. Since IMUs do not directly measure the physical state of the vehicle (translation and rotation) but rather measure their derivatives along the time axis (angular velocity and acceleration), we must introduce the differential relationship of the vehicle state and their various properties after integration. These are the main contents of Chapters 3 and 4.
3. Chapters 5 to 7 introduce the content of laser SLAM. Chapter 5 mainly covers basic point cloud processing methods, including how to represent laser point clouds, how to find their nearest neighbors, and how to rasterize them. We will implement some classical data structures ourselves. Chapters 6 and 7 respectively introduce 2D and 3D laser SLAM methods. We will first implement some laser registration methods: 2D and 3D ICP, NDT, probability grid, etc., then manage them using sub-map methods, and finally add loop detection to form a complete SLAM system. Chapter 7 also introduces loosely coupled laser-inertial odometry.
4. Chapters 8 to 10 introduce typical SLAM applications. Chapter 8 implements a tightly coupled laser-inertial odometry, each using an iterative error Kalman filter and a preintegration optimizer. Chapter 9 introduces offline point cloud map construction methods. We will rewrite some key algorithms into easily parallelized offline programs. Chapter 10 introduces methods for high-precision localization in existing point cloud maps. We will divide the point cloud map into small blocks in space and then use filters to achieve fusion localization of point clouds and inertial data.

The above is the main content of this book. We mainly focus on the application of SLAM in autonomous driving around the aspects of inertial navigation and laser point clouds. Most vehicles on open roads or park roads can be mapped and located in this way. However, this book will not delve into the annotation part of maps in detail because they are mainly drawn manually and do not involve much algorithmic content⁶.

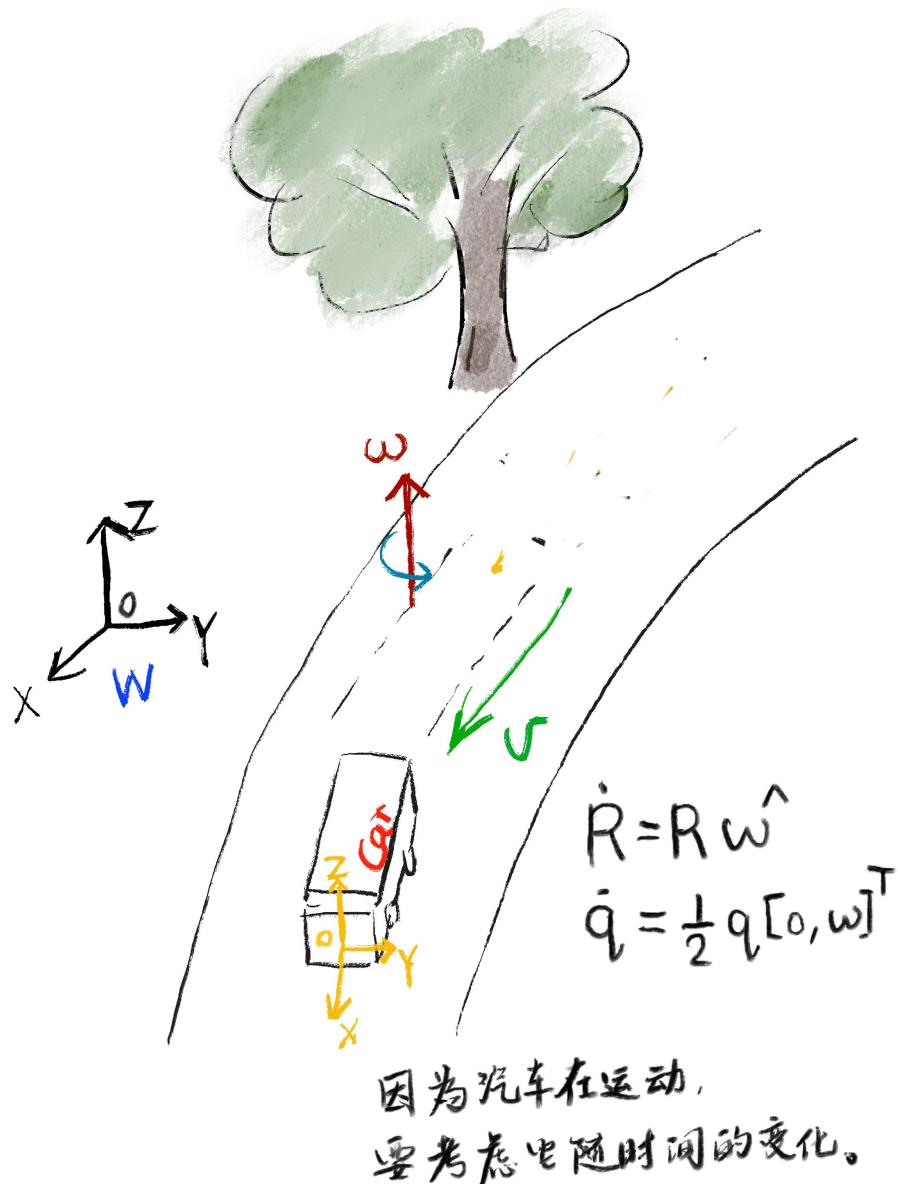
Except for this chapter, the end of each chapter will include a certain number of exercises. Readers should allocate time for exercises according to their own learning progress.

⁶The automatic generation of high-definition maps is also a key research direction in the field of autonomous driving, but the methods are quite different from the content covered in this book, and the results are not yet mature. We will not delve into this topic. Readers can refer to related literature, such as [40, 41].

Chapter 2

Quick Review of Basic Mathematical Concepts

Before delving into various sensor processing methods, let's review some fundamental mathematical concepts. This book largely follows the notation conventions established in "Introduction to Visual SLAM"[1]. To avoid redundancy, we must assume that readers are already familiar with the basic geometric knowledge presented in that book. This book does not elaborate on processes such as quaternion-to-rotation-matrix transformations in detail; instead, it briefly mentions their conclusions for readers to refer back to at any time. For topics not extensively covered in [1], this chapter provides additional explanations and derivations as appropriate.



2.1 Geometry

2.1.1 Coordinate Systems

To describe the position and orientation of an autonomous vehicle, we should first define various coordinate systems for it. Firstly, we assume the existence of a fixed coordinate system in the world, known as the **world coordinate system** or **inertial coordinate system**. There are several ways to define this coordinate system in the real world, but in principle, it can be simply considered as a fixed coordinate system. When the vehicle moves in the world coordinate system, there exists a transformation relationship between the vehicle's own coordinate system (referred to as the **body coordinate system** or **body frame**) and the world system. This transformation relationship changes over time, allowing us to define the vehicle's **linear velocity**, **angular velocity**, **acceleration**, and other physical quantities. This constitutes the motion process of the vehicle.

However, explaining what linear velocity, angular velocity, and especially attitude mean from a mathematical perspective is not so intuitive. The attitude of the vehicle is usually described by a **rotation matrix** or **quaternion**. When they change over time, how many-dimensional vectors should be used to describe the angular velocity? How does the angular velocity vector act on the rotation matrix or quaternion? Are there any formal differences in their various definitions? Are they essentially the same? These are the questions this chapter aims to answer.

A three-dimensional coordinate system is composed of three vectors in space. Typically, we choose a set of unit orthogonal vectors to form a reference frame. For instance, if $(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$ are the three vectors of the world coordinate system, it means that these three vectors have a length of 1 and their inner products are 0. In this case, we say that a coordinate system (reference frame) $E = \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ has been chosen. Then, any three-dimensional spatial vector \mathbf{a} can be represented in this reference frame as:

$$\mathbf{a} = a_1 \mathbf{e}_1 + a_2 \mathbf{e}_2 + a_3 \mathbf{e}_3, \quad (2.1)$$

where (a_1, a_2, a_3) are the coordinates of the vector \mathbf{a} .

Please note that even without specifying a reference frame and coordinates, various operations can be performed between vectors. For example, the following operations can be performed between two vectors \mathbf{a} and \mathbf{b} :

- Addition and subtraction.** The result of vector addition or subtraction is still a vector, following the parallelogram rule:

$$\mathbf{c} = \mathbf{a} \pm \mathbf{b}. \quad (2.2)$$

If the vectors have coordinates, the components are simply added or subtracted.

- Scalar multiplication.** Multiplying a vector by any scalar $k \in \mathbb{R}$ scales the vector:

$$\mathbf{b} = k\mathbf{a}, \quad (2.3)$$

resulting in another vector. When \mathbf{a} has coordinates, these coordinates are scaled accordingly.

- Taking the length.** We can compute the length of a vector, denoted as:

$$\|\mathbf{a}\|. \quad (2.4)$$

The length yields a scalar value. Mathematically, a vector's length can be zero or negative, for instance, in Minkowski space, but in the physical world of autonomous driving, we are concerned with vectors in Euclidean space, where their length is always non-negative.

4. **Dot product.** The dot product of two vectors yields the product of their lengths times the cosine of the angle between them, resulting in a scalar:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \langle \mathbf{a}, \mathbf{b} \rangle, \quad (2.5)$$

where if vectors have coordinates, the dot product results from the sum of the products of their respective components.

5. **Cross product.** The cross product of two vectors is another vector whose direction is perpendicular to the plane formed by the two vectors, and its magnitude is the product of their lengths times the sine of the angle between them. If vectors \mathbf{a}, \mathbf{b} are defined in the $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ frame, the cross product is written as:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \mathbf{b} \triangleq \mathbf{a}^\wedge \mathbf{b}. \quad (2.6)$$

The cross product can also be expressed as the usual matrix-vector multiplication, which requires expressing the first vector in a **skew-symmetric** matrix form¹. We use the $^\wedge$ symbol to define this transformation:

$$\mathbf{a}^\wedge = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} = \mathbf{A}. \quad (2.7)$$

Note that this operator is a **one-one mapping**, meaning that for any vector, there exists a unique corresponding skew-symmetric matrix, and vice versa. We use the $^\vee$ symbol to denote the mapping from skew-symmetric matrix to vector:

$$\mathbf{A}^\vee = \mathbf{a}. \quad (2.8)$$

The skew-symmetric matrix operator is a symbol widely used in the subsequent text; readers should pay attention to this notation. In other literature, it might also be denoted as $\mathbf{a}_\times, \mathbf{a}^\times, [\mathbf{a}]_\times, \hat{\mathbf{a}}$ [42], all of which have the same meaning. This book uniformly adopts the \wedge and \vee symbols on the upper right, as they appear more concise.

Lastly, even when a reference frame is not specified, vectors can undergo the aforementioned operations. Their results are independent of the choice of reference frame. If a reference frame and coordinates are specified, then the aforementioned computations can also be represented using numerical values of coordinates.

An autonomous vehicle is equipped with various types of sensors. We typically assume that each sensor has its own reference frame, and their respective axis directions are defined according to the usage habits of each sensor. For example, in Figure 2-1, the IMU, 64-line lidar, and camera of the vehicle all define their own reference frames. The vehicle body generally uses the **front-left-up**² or **right-front-up** order to define its coordinate system, while

¹A skew-symmetric matrix satisfies $\mathbf{A}^\top = -\mathbf{A}$.

²The convention "front-left-up" refers to the X axis pointing forward, the Y axis pointing left, and the Z axis pointing up, following the right-hand rule. The convention for "right-front-up" is analogous.

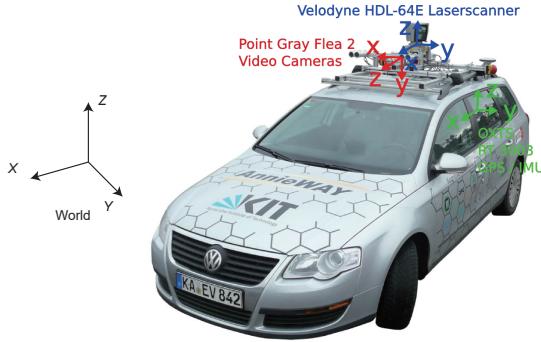


Figure 2-1: The body and world coordinate systems of a typical autonomous vehicle sensors

the camera coordinate system commonly adopts the **right-down-front** order. Consequently, there exist rotation and translation relationships between the coordinate systems of various sensors, which we characterize using rotation matrices and translation vectors.

Assuming a point \mathbf{p} in the world coordinate system has coordinates \mathbf{p}_w , and its coordinates in the vehicle body coordinate system are \mathbf{p}_b , then we define the rotation matrix \mathbf{R}_{wb} and the translation vector \mathbf{t}_{wb} , such that:

$$\mathbf{p}_w = \mathbf{R}_{wb}\mathbf{p}_b + \mathbf{t}_{wb}, \quad (2.9)$$

It is crucial for readers to understand the approach here. The key points are as follows:

1. Firstly, we define the transformation relationship between **coordinates**. \mathbf{R}_{wb} and \mathbf{t}_{wb} are used to handle coordinate transformations between vectors. Some materials deal with transformations between **coordinate axes** (or bases), interpreting rotation and translation as a transformation of a **coordinate axis** from one position to another. This definition is opposite to that of this book³, so please be careful.
2. We can directly write \mathbf{R}_{wb} , \mathbf{t}_{wb} as a **transformation matrix** \mathbf{T}_{wb} , expressing coordinate transformations in homogeneous form:

$$\mathbf{p}_w = \mathbf{T}_{wb}\mathbf{p}_b. \quad (2.10)$$

This transforms the discussion into properties of the transformation matrix \mathbf{T} . The specific form of \mathbf{T} is:

$$\mathbf{T}_{wb} = \begin{bmatrix} \mathbf{R}_{wb} & \mathbf{t}_{wb} \\ \mathbf{0} & 1 \end{bmatrix} \in \mathbb{R}^{4 \times 4}. \quad (2.11)$$

s However, since the subsequent discussion will involve the IMU, which does not directly measure the differential of \mathbf{T} , we prefer to separate \mathbf{R} and \mathbf{t} rather than express them in the form of a transformation matrix.

3. Our subscript reading order is **from right to left**, meaning the subscript wb is right-multiplied with b to obtain variables in the w system. This makes writing and reading more fluid and intuitive. Different books handle the superscript and subscript of coordinate systems differently. Some write them on the left, some write them above, and some books even have four different markings (superscript, subscript, left, right)

³One deals with transformations of coordinates, while the other deals with transformations of bases. Readers should be cautious.

for one variable. This book uniformly uses the subscript *wb* to define various variables. Since all variables have the subscript *wb*, we omit these subscripts in the vast majority of content to strive for simplicity. We also need to discuss various variables at different times or iteration numbers, which will introduce subscripts related to time or iteration numbers. If combined with coordinate system subscripts, readers would have to face a large number of formulas with various superscripts and subscripts.

All three-dimensional rotation matrices form the **Special Orthogonal Group** ($\text{SO}(3)$). It is a 3×3 real matrix that satisfies:

- The rotation matrix is an orthogonal matrix: $\mathbf{R}^\top = \mathbf{R}^{-1}$.
- The determinant of the rotation matrix is 1: $\det(\mathbf{R}) = 1$.

Additionally, a rotation matrix can also be represented by **quaternions** or **rotation vectors**. Below, we will review their definitions and conversion relationships.

2.1.2 Rotation Vectors

Rotation vectors, also known as **angle-axis**, correspond to the Lie algebra $\mathfrak{so}(3)$ of $\text{SO}(3)$. Since $\mathfrak{so}(3)$ is the tangent space of $\text{SO}(3)$, as we will see later, rotation vectors can also be used to express angular velocities.

Let's denote a rotation vector as $\mathbf{w} \in \mathbb{R}^3$, and it can be decomposed into direction and magnitude: $\mathbf{w} = \theta\mathbf{n}$. The conversion relationship from rotation vector to rotation matrix can be described by **Rodrigues' formula** or the exponential map on $\text{SO}(3)$:

$$\mathbf{R} = \cos \theta \mathbf{I} + (1 - \cos \theta) \mathbf{n} \mathbf{n}^\top + \sin \theta \mathbf{n}^\wedge = \exp(\mathbf{w}^\wedge). \quad (2.12)$$

Here, \exp can also be expanded using Taylor series and simplified to the formula on the left. To simplify notation, we denote the uppercase Exp as:

$$\text{Exp}(\mathbf{w}) = \exp(\mathbf{w}^\wedge), \quad (2.13)$$

which eliminates one $^\wedge$ symbol, making the formula appear more concise in complex expressions.

Conversely, the conversion relationship from rotation matrix to rotation vector can be described by the logarithmic map:

$$\mathbf{w} = \log(\mathbf{R})^\vee = \text{Log}(\mathbf{R}). \quad (2.14)$$

The computation method for the angle-axis is as follows. For the angle θ , we have:

$$\theta = \arccos \left(\frac{\text{tr}(\mathbf{R}) - 1}{2} \right). \quad (2.15)$$

And the axis \mathbf{n} is the unit eigenvector of \mathbf{R} corresponding to the eigenvalue 1:

$$\mathbf{R}\mathbf{n} = \mathbf{n}. \quad (2.16)$$

2.1.3 Quaternions

Three-dimensional rotations can also be described by unit quaternions. Quaternions, also known as expanded complex numbers, consist of a real part and three imaginary parts. This book uses Hamiltonian quaternions⁴, defined as:

$$\mathbf{q} = q_0 + q_1 i + q_2 j + q_3 k, \quad (2.17)$$

where q_0 is the real part, and q_1, q_2, q_3 are the imaginary parts. The imaginary units i, j, k satisfy the following multiplication rules:

$$\begin{cases} i^2 = j^2 = k^2 = -1 \\ ij = k, ji = -k \\ jk = i, kj = -i \\ ki = j, ik = -j \end{cases}. \quad (2.18)$$

To simplify notation, the three imaginary parts can be represented as a vector, and the quaternion can be expressed as a combination of scalar part s and vector part \mathbf{v} :

$$\mathbf{q} = [s, \mathbf{v}]^\top. \quad (2.19)$$

Using the vector part, a compact form of quaternion multiplication can be written.

Following the multiplication rules of quaternions, several commonly used quaternion calculation methods can be derived. We list them below.

1. Addition and Subtraction

The addition and subtraction of quaternions \mathbf{q}_a and \mathbf{q}_b are given by:

$$\mathbf{q}_a \pm \mathbf{q}_b = [s_a \pm s_b, \mathbf{v}_a \pm \mathbf{v}_b]^\top. \quad (2.20)$$

2. Multiplication

Multiplication involves multiplying each term of \mathbf{q}_a by each term of \mathbf{q}_b and then summing them up, while following the rules defined by Equation (2.18). It can be expressed as:

$$\begin{aligned} \mathbf{q}_a \mathbf{q}_b &= s_a s_b - x_a x_b - y_a y_b - z_a z_b \\ &\quad + (s_a x_b + x_a s_b + y_a z_b - z_a y_b) i \\ &\quad + (s_a y_b - x_a z_b + y_a s_b + z_a x_b) j \\ &\quad + (s_a z_b + x_a y_b - y_a x_b + z_a s_b) k. \end{aligned} \quad (2.21)$$

Though slightly complex, this form is well-structured. Expressing it in vector form and using inner and outer product operations results in a more concise expression:

$$\mathbf{q}_a \mathbf{q}_b = [s_a s_b - \mathbf{v}_a^\top \mathbf{v}_b, s_a \mathbf{v}_b + s_b \mathbf{v}_a + \mathbf{v}_a \times \mathbf{v}_b]^\top. \quad (2.22)$$

Under this definition of multiplication, the product of two real quaternions is still real, which is consistent with complex numbers. However, note that quaternion multiplication is usually non-commutative due to the presence of the cross-product term, unless \mathbf{v}_a and \mathbf{v}_b are collinear in \mathbb{R}^3 , in which case the cross-product term becomes zero.

⁴According to different tastes, there are slight variations in the definition of quaternions. Hamiltonian is the most common and intuitive way of defining quaternions.

This book does not deliberately distinguish between standard multiplication and quaternion multiplication. Some materials may use symbols such as \otimes to differentiate quaternion multiplication, but this book consistently uses standard multiplication. Quaternions are not multiplied with ordinary vectors or matrices, so **the meaning of multiplication should be clear**.

3. Norm

The norm of a quaternion is defined as:

$$\|\mathbf{q}_a\| = \sqrt{s_a^2 + x_a^2 + y_a^2 + z_a^2}. \quad (2.23)$$

It can be verified that the norm of the product of two quaternions equals the product of their norms, which ensures that the product of unit quaternions remains a unit quaternion:

$$\|\mathbf{q}_a \mathbf{q}_b\| = \|\mathbf{q}_a\| \|\mathbf{q}_b\|. \quad (2.24)$$

4. Conjugate

The conjugate of a quaternion is obtained by negating the imaginary parts:

$$\mathbf{q}_a^* = s_a - x_a i - y_a j - z_a k = [s_a, -\mathbf{v}_a]^\top. \quad (2.25)$$

Multiplying a quaternion by its conjugate yields a real quaternion with a real part equal to the square of its norm:

$$\mathbf{q}^* \mathbf{q} = \mathbf{q} \mathbf{q}^* = [s^2 + \mathbf{v}^\top \mathbf{v}, \mathbf{0}]^\top. \quad (2.26)$$

5. Inverse

The inverse of a quaternion is given by:

$$\mathbf{q}^{-1} = \mathbf{q}^* / \|\mathbf{q}\|^2. \quad (2.27)$$

According to this definition, the product of a quaternion and its inverse yields a real quaternion **1**:

$$\mathbf{q} \mathbf{q}^{-1} = \mathbf{q}^{-1} \mathbf{q} = \mathbf{1}. \quad (2.28)$$

If \mathbf{q} is a unit quaternion, its inverse and conjugate are the same. Moreover, the inverse of a product obeys a property similar to matrices:

$$(\mathbf{q}_a \mathbf{q}_b)^{-1} = \mathbf{q}_b^{-1} \mathbf{q}_a^{-1}. \quad (2.29)$$

6. Scalar Multiplication

Similar to vectors, quaternions can be multiplied by scalars:

$$k \mathbf{q} = [ks, k\mathbf{v}]^\top. \quad (2.30)$$

Representing Rotation with Quaternions

Rotation of a point can be expressed using quaternions. Let's assume a three-dimensional point in space $\mathbf{p} = [x, y, z] \in \mathbb{R}^3$, and a rotation specified by a unit quaternion \mathbf{q} . The point \mathbf{p} undergoes a rotation to become \mathbf{p}' . If described using matrices, then $\mathbf{p}' = \mathbf{Rp}$. But how can we express this relationship using quaternions?

Firstly, represent the three-dimensional space point using a quaternion:

$$\mathbf{p} = [0, x, y, z]^\top = [0, \mathbf{v}]^\top. \quad (2.31)$$

This is equivalent to associating the three imaginary parts of the quaternion with the three axes in space. Then, the rotated point \mathbf{p}' can be expressed as the following product:

$$\mathbf{p}' = \mathbf{qpq}^{-1}. \quad (2.32)$$

Here, the multiplication is quaternion multiplication, resulting in another quaternion. Finally, extract the imaginary part of \mathbf{p}' to obtain the coordinates of the point after rotation. It can be verified that the real part of the computed result is 0, hence it is a pure imaginary quaternion.

Conversion from Quaternion to Rotation Matrix and Rotation Vector

Any unit quaternion describes a rotation, which can also be described using a rotation matrix or rotation vector. Now, let's examine the relationship between quaternions and rotation vectors, rotation matrices.

Before diving into that, it's worth mentioning that quaternion multiplication can also be expressed as a form of matrix multiplication. Let $\mathbf{q} = [s, \mathbf{v}]^\top$ be a quaternion. Define the following symbols $+$ and \oplus as follows[43]:

$$\mathbf{q}^+ = \begin{bmatrix} s & -\mathbf{v}^\top \\ \mathbf{v} & s\mathbf{I} + \mathbf{v}^\wedge \end{bmatrix}, \quad \mathbf{q}^\oplus = \begin{bmatrix} s & -\mathbf{v}^\top \\ \mathbf{v} & s\mathbf{I} - \mathbf{v}^\wedge \end{bmatrix}, \quad (2.33)$$

where these symbols map quaternions into a 4×4 matrix. Thus, quaternion multiplication can be written in matrix form as follows:

$$\mathbf{q}_1^+ \mathbf{q}_2 = \begin{bmatrix} s_1 & -\mathbf{v}_1^\top \\ \mathbf{v}_1 & s_1\mathbf{I} + \mathbf{v}_1^\wedge \end{bmatrix} \begin{bmatrix} s_2 \\ \mathbf{v}_2 \end{bmatrix} = \begin{bmatrix} -\mathbf{v}_1^\top \mathbf{v}_2 + s_1 s_2 \\ s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1^\wedge \mathbf{v}_2 \end{bmatrix} = \mathbf{q}_1 \mathbf{q}_2. \quad (2.34)$$

Similarly, it can be proven that:

$$\mathbf{q}_1 \mathbf{q}_2 = \mathbf{q}_1^+ \mathbf{q}_2 = \mathbf{q}_2^\oplus \mathbf{q}_1. \quad (2.35)$$

Then, let's consider the problem of rotating a point in space using quaternions. According to the previous discussion, we have:

$$\begin{aligned} \mathbf{p}' &= \mathbf{qpq}^{-1} = \mathbf{q}^+ \mathbf{p}^+ \mathbf{q}^{-1} \\ &= \mathbf{q}^+ \mathbf{q}^{-1 \oplus} \mathbf{p}. \end{aligned} \quad (2.36)$$

Substituting the matrices corresponding to the two symbols, we obtain:

$$\mathbf{q}^+ (\mathbf{q}^{-1})^\oplus = \begin{bmatrix} s & -\mathbf{v}^\top \\ \mathbf{v} & s\mathbf{I} + \mathbf{v}^\wedge \end{bmatrix} \begin{bmatrix} s & \mathbf{v}^\top \\ -\mathbf{v} & s\mathbf{I} + \mathbf{v}^\wedge \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0}^\top & \mathbf{vv}^\top + s^2\mathbf{I} + 2s\mathbf{v}^\wedge + (\mathbf{v}^\wedge)^2 \end{bmatrix}. \quad (2.37)$$

Since both \mathbf{p}' and \mathbf{p} are purely imaginary quaternions, the lower right corner of this matrix actually gives the transformation from quaternion to rotation matrix:

$$\mathbf{R} = \mathbf{v}\mathbf{v}^\top + s^2\mathbf{I} + 2s\mathbf{v}^\wedge + (\mathbf{v}^\wedge)^2. \quad (2.38)$$

To obtain the conversion formula from quaternion to rotation vector, take the trace of both sides of the above equation:

$$\begin{aligned} \text{tr}(\mathbf{R}) &= \text{tr}(\mathbf{v}\mathbf{v}^\top) + 3s^2 + 2s \cdot 0 + \text{tr}((\mathbf{v}^\wedge)^2) \\ &= v_1^2 + v_2^2 + v_3^2 + 3s^2 - 2(v_1^2 + v_2^2 + v_3^2) \\ &= (1 - s^2) + 3s^2 - 2(1 - s^2) \\ &= 4s^2 - 1. \end{aligned} \quad (2.39)$$

Also, from Eq.(2.15), we have:

$$\begin{aligned} \theta &= \arccos\left(\frac{\text{tr}(\mathbf{R}) - 1}{2}\right) \\ &= \arccos(2s^2 - 1). \end{aligned} \quad (2.40)$$

Thus,

$$\cos \theta = 2s^2 - 1 = 2\cos^2 \frac{\theta}{2} - 1, \quad (2.41)$$

so:

$$\theta = 2 \arccos s. \quad (2.42)$$

Regarding the rotation axis, if we substitute \mathbf{q} 's imaginary part for \mathbf{p} in Eq.(2.36), it's easy to see that the vector formed by the imaginary part of \mathbf{q} remains fixed during rotation, constituting the rotation axis. Thus, by normalizing it by its magnitude, we obtain the rotation axis. In summary, the conversion formula from quaternion to rotation vector can be expressed as follows:

$$\begin{cases} \theta = 2 \arccos s \\ [n_x, n_y, n_z]^\top = \mathbf{v}^\top / \sin \frac{\theta}{2} \end{cases}. \quad (2.43)$$

Since quaternions require only four values to represent rotation, most programs choose quaternions as the underlying representation for rotations. They may provide interfaces for matrix operations, such as the previously mentioned \vee or \log operations, or interfaces for quaternions, such as retrieving the four components of a quaternion, and so on. When using these programs, we can simply use these matrix interfaces without concerning ourselves with their underlying storage format.

2.1.4 Lie Group and Lie Algebra

Three-dimensional rotations form the three-dimensional rotation group $\text{SO}(3)$, with its corresponding Lie algebra denoted as $\mathfrak{so}(3)$; three-dimensional transformations form the three-dimensional transformation group $\text{SE}(3)$, with its corresponding Lie algebra denoted as $\mathfrak{se}(3)$.

The mapping from Lie algebra elements to Lie group elements is known as the exponential mapping. For $\mathfrak{so}(3)$ to $\text{SO}(3)$, the exponential mapping is given by:

$$\exp(\phi^\wedge) = \mathbf{R}, \quad (2.44)$$

where the specific computation is provided by the Rodrigues' formula (2.12). The inverse mapping, known as the logarithm mapping, is denoted as:

$$\phi = \log(\mathbf{R})^\vee, \quad (2.45)$$

with specific computation provided by equations (2.15) and (2.16).

We mainly utilize the combination of SO(3) with translation vectors to derive subsequent motion equations, filtering relationships, etc. We omit the introduction of SE(3) and $\mathfrak{se}(3)$.

2.1.5 BCH Linear Approximation on SO(3)

The Baker-Campbell-Hausdorff (BCH) formula [44] provides a relationship between the addition of small quantities in the Lie algebra and the multiplication of small quantities in the Lie group, which is widely used for linearization of various functions. Here, we only present the conclusions.

In SO(3), for a rotation \mathbf{R} (corresponding to the Lie algebra ϕ), left-multiplying it by a small rotation, denoted as $\Delta\mathbf{R}$, with the corresponding Lie algebra $\Delta\phi$, results in $\Delta\mathbf{R} \cdot \mathbf{R}$ on the Lie group. According to the BCH approximation, in the Lie algebra, it is $\mathbf{J}_l^{-1}(\phi)\Delta\phi + \phi$. Thus, we can simply write:

$$\exp(\Delta\phi^\wedge) \exp(\phi^\wedge) = \exp\left((\phi + \mathbf{J}_l^{-1}(\phi)\Delta\phi)^\wedge\right). \quad (2.46)$$

Conversely, if we perform addition in the Lie algebra, adding $\Delta\phi$ to ϕ , it can be approximated as multiplication with left and right Jacobians on the Lie group:

$$\exp((\phi + \Delta\phi)^\wedge) = \exp((\mathbf{J}_l(\phi)\Delta\phi)^\wedge) \exp(\phi^\wedge) = \exp(\phi^\wedge) \exp((\mathbf{J}_r(\phi)\Delta\phi)^\wedge). \quad (2.47)$$

Where the left Jacobian for SO(3) is given by:

$$\mathbf{J}_l(\theta\mathbf{a}) = \frac{\sin\theta}{\theta}\mathbf{I} + (1 - \frac{\sin\theta}{\theta})\mathbf{a}\mathbf{a}^\top + \frac{1 - \cos\theta}{\theta}\mathbf{a}^\wedge \quad (2.48)$$

$$\mathbf{J}_l^{-1}(\theta\mathbf{a}) = \frac{\theta}{2}\cot\frac{\theta}{2}\mathbf{I} + \left(1 - \frac{\theta}{2}\cot\frac{\theta}{2}\right)\mathbf{a}\mathbf{a}^\top - \frac{\theta}{2}\mathbf{a}^\wedge. \quad (2.49)$$

And the right Jacobian for SO(3) is:

$$\mathbf{J}_r(\phi) = \mathbf{J}_l(-\phi). \quad (2.50)$$

Since the Lie algebra ϕ and \mathbf{R} can be easily associated, sometimes we also simply denote $\mathbf{J}_r(\phi)$ as $\mathbf{J}_r(\mathbf{R})$ instead of $\mathbf{J}_r(\text{Log}(\mathbf{R}))$. This can make the formulas look more concise. In many cases, we also omit the part inside the parentheses of $\mathbf{J}_r(\phi)$ and directly write \mathbf{J}_r and \mathbf{J}_l .

All of the above content has been introduced in [1] already. If readers are interested in their detailed derivation process, please check [1], [4] or [6]. This book will directly use the conclusions introduced above.

2.2 Kinematics

Now let's consider a three-dimensional object in motion over time. In this section, we will explore various perspectives on expressing three-dimensional kinematics, which will be correlated with subsequent chapters. Examining three-dimensional kinematics will lead to a series of interesting discussions. Join us as we delve into it.

2.2.1 Kinematics from the Perspective of Lie Groups

Earlier, we discussed how the rotation and translation of an object can be described by \mathbf{R} and \mathbf{t} , respectively (here, we omit the subscript *wb* denoting the coordinate frame). When they vary continuously with time, they become functions of time, $\mathbf{R}(t)$ and $\mathbf{t}(t)$. Obviously, the translational part is trivial, just a function with the codomain \mathbb{R}^3 . Therefore, we focus on the rotational part.

Let's assume that \mathbf{R} varies with time, i.e., $\mathbf{R}(t)$. According to the property of \mathbf{R} being an orthogonal matrix:

$$\mathbf{R}^\top \mathbf{R} = \mathbf{I}, \quad (2.51)$$

it is not difficult to observe:

$$\frac{d}{dt} (\mathbf{R}^\top \mathbf{R}) = \dot{\mathbf{R}}^\top \mathbf{R} + \mathbf{R}^\top \dot{\mathbf{R}} = \mathbf{0}, \quad (2.52)$$

which implies:

$$\mathbf{R}^\top \dot{\mathbf{R}} = -(\mathbf{R}^\top \dot{\mathbf{R}})^\top. \quad (2.53)$$

It can be seen that $\mathbf{R}^\top \dot{\mathbf{R}}$ is a skew-symmetric matrix, and a skew-symmetric matrix can be expressed in vector form using the skew-symmetric symbol \wedge . Let's take $\boldsymbol{\omega}^\wedge \in \mathbb{R}^{3 \times 3} = \mathbf{R}^\top \dot{\mathbf{R}}$, then we can write \mathbf{R} in the form of a differential equation:

$$\dot{\mathbf{R}} = \mathbf{R} \boldsymbol{\omega}^\wedge. \quad (2.54)$$

This equation is also known as the **Poisson equation**[45]. It's worth noting that we could also start from $\mathbf{R} \mathbf{R}^\top = \mathbf{I}$, define $\boldsymbol{\omega}^\wedge = \dot{\mathbf{R}} \mathbf{R}^\top$, and obtain the result $\dot{\mathbf{R}} = \boldsymbol{\omega}^\wedge \mathbf{R}$. These two forms are essentially equivalent, just different in appearance.

If we only consider instantaneous changes, then at a fixed time t , $\boldsymbol{\omega}$ can be considered constant. In physical terms, we call $\boldsymbol{\omega}$ the **instantaneous angular velocity**. Given an initial rotation matrix $\mathbf{R}(t_0)$ at time t_0 , the solution to the above differential equation is:

$$\mathbf{R}(t) = \mathbf{R}(t_0) \exp(\boldsymbol{\omega}^\wedge(t - t_0)). \quad (2.55)$$

If readers are familiar with the knowledge of Lie groups and Lie algebras, it's easy to recognize that Equation (2.55) represents the exponential mapping on $\text{SO}(3)$. Let $\Delta t = t - t_0$, then this equation can also be written as:

$$\mathbf{R}(t) = \mathbf{R}(t_0) \text{Exp}(\boldsymbol{\omega} \Delta t). \quad (2.56)$$

From another perspective, we can also expand $\mathbf{R}(t)$ around time t_0 using Taylor series, and the first-order approximation is:

$$\begin{aligned} \mathbf{R}(t_0 + \Delta t) &\approx \mathbf{R}(t_0) + \dot{\mathbf{R}}(t_0) \Delta t \\ &= \mathbf{R}(t_0) + \mathbf{R}(t_0) \boldsymbol{\omega}^\wedge \Delta t \\ &= \mathbf{R}(t_0) (\mathbf{I} + \boldsymbol{\omega}^\wedge \Delta t). \end{aligned} \quad (2.57)$$

This reveals the approximate form of the exponential mapping:

$$\text{Exp}(\boldsymbol{\omega} \Delta t) = \mathbf{I} + \boldsymbol{\omega}^\wedge \Delta t + \frac{1}{2} (\boldsymbol{\omega}^\wedge \Delta t)^2 + \dots \quad (2.58)$$

By comparing the above equations, we can see that:

1. Equation (2.56) is the discrete-time form of Equation (2.55).
2. Equation (2.57) is the linear approximation of Equation (2.56).

These two sets of equations are very useful in dealing with angular velocities, and we will continue to use them in the subsequent discussion.

2.2.2 Kinematics from the Perspective of Quaternions

Now let's examine how the kinematic equations change if we use quaternions to represent rotations. This is an alternative description of the same problem from a different perspective. Investigating this issue can help us establish connections between different mathematical representations. We know that the rotation of a vector by quaternions should take the form given by Equation (2.36), and quaternions themselves carry the unit constraint $\mathbf{q}\mathbf{q}^* = \mathbf{q}^*\mathbf{q} = \mathbf{1}$. Similar to the case of $\text{SO}(3)$, starting from $\mathbf{q}^*\mathbf{q} = \mathbf{1}$, if we differentiate both sides with respect to time, we get:

$$\dot{\mathbf{q}}^*\mathbf{q} + \mathbf{q}^*\dot{\mathbf{q}} = \mathbf{0}, \quad (2.59)$$

which leads to:

$$\mathbf{q}^*\dot{\mathbf{q}} = -\dot{\mathbf{q}}^*\mathbf{q} = -(\mathbf{q}^*\dot{\mathbf{q}})^*. \quad (2.60)$$

Thus, $\mathbf{q}^*\dot{\mathbf{q}}$ is a pure quaternion (with zero real part). We can denote a pure quaternion as $\boldsymbol{\varpi} = [0, \underbrace{\omega_1, \omega_2, \omega_3}_{\boldsymbol{\omega}}]^\top \in \mathcal{Q}$, so we have:

$$\mathbf{q}^*\dot{\mathbf{q}} = \boldsymbol{\varpi}. \quad (2.61)$$

Multiplying both sides by \mathbf{q} , we get:

$$\dot{\mathbf{q}} = \mathbf{q}\boldsymbol{\varpi}. \quad (2.62)$$

This equation is very similar to Equation (2.54). Analogous to the case of $\text{SO}(3)$, we can also discuss the instantaneous angular velocity, Lie algebra, exponential mapping, and logarithmic mapping near time t . When considering instantaneous changes, we can treat $\boldsymbol{\varpi}$ as a constant value, so the solution to the above differential equation is:

$$\mathbf{q}(t) = \mathbf{q}(t_0) \exp(\boldsymbol{\varpi}\Delta t), \quad (2.63)$$

where we used the quaternion exponential mapping. Let's take a brief pause in the derivation and introduce the quaternion exponential mapping in the usual sense.

For any pure quaternion $\boldsymbol{\varpi} = [0, \boldsymbol{\omega}]^\top \in \mathcal{Q}$, its exponential mapping is defined as:

$$\exp(\boldsymbol{\varpi}) = \sum_{k=0}^{\infty} \frac{1}{k!} \boldsymbol{\varpi}^k. \quad (2.64)$$

Separating its direction and magnitude, let $\boldsymbol{\varpi} = \mathbf{u}\theta$, where θ is the magnitude of $\boldsymbol{\varpi}$ and \mathbf{u} is the unit imaginary quaternion. Since \mathbf{u} is an unit imaginary quaternion, we have:

$$\mathbf{u}^2 = -\mathbf{1}, \quad \mathbf{u}^3 = -\mathbf{u}, \quad (2.65)$$

which is similar to the self-multiplication property of unit imaginary numbers and can be used to simplify higher-order terms. Using this property, we can derive:

$$\begin{aligned} \exp(\mathbf{u}\theta) &= 1 + \mathbf{u}\theta - \frac{1}{2!}\theta^2 - \frac{1}{3!}\theta^3\mathbf{u} + \frac{1}{4!}\theta^4 + \dots \\ &= \underbrace{\left(1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - \dots\right)}_{\cos \theta} + \underbrace{\left(\theta - \frac{1}{3!}\theta^3 + \frac{1}{5!}\theta^5 - \dots\right)\mathbf{u}}_{\sin \theta} \\ &= \cos \theta + \mathbf{u} \sin \theta. \end{aligned} \quad (2.66)$$

This formula is very similar to Euler's formula for complex numbers:

$$\exp(i\theta) = \cos \theta + i \sin \theta, \quad (2.67)$$

and it's indeed its extension to quaternions.

Substituting the pure imaginary ϖ , we obtain:

$$\exp(\varpi) = [\cos \theta, \mathbf{u} \sin \theta]^\top. \quad (2.68)$$

Also, because ϖ is an imaginary quaternion, we have:

$$\|\exp(\varpi)\| = \cos^2 \theta + \sin^2 \theta \|\mathbf{u}\|^2 = 1. \quad (2.69)$$

So, the result of the exponential mapping of an imaginary quaternion is a unit quaternion, which is also a mapping relationship between unit quaternions and pure quaternions. We can also think of the imaginary quaternion ϖ as a quaternion form of the Lie algebra. Therefore, an obvious question arises: what is the relationship between the quaternion form of the Lie algebra and the rotation vector form of the Lie algebra?

2.2.3 Conversion between Lie Algebra of Quaternions and Rotation Vectors

Consider a rotation matrix \mathbf{R} and its rotation vector ϕ . Obviously, their relationship is described by the exponential mapping:

$$\mathbf{R} = \text{Exp}(\phi) = \text{Exp}(\theta \mathbf{n}), \quad (2.70)$$

where \mathbf{n} is the direction of the rotation vector and θ is its magnitude. We also assume that this rotation can be expressed by $\mathbf{q} = \text{Exp}(\varpi)$, where ϖ is a pure imaginary quaternion $[0, \omega]^\top$. Now let's examine the transformation relationship between these two representations.

From Equation (2.43), we know that the quaternion corresponding to \mathbf{R} is:

$$\mathbf{q} = [\cos \frac{\theta}{2}, \mathbf{n} \sin \frac{\theta}{2}], \quad (2.71)$$

By comparing with Equation (2.68), it's easy to see the relationship between ϖ and ϕ :

$$\varpi = [0, \frac{1}{2}\phi]^\top, \quad \text{or } \omega = \frac{1}{2}\phi. \quad (2.72)$$

We miraculously discover that the angular velocity expressed by quaternions is exactly half of the $\text{SO}(3)$ Lie algebra! This is because when using quaternions to rotate a vector, we need to multiply corresponding parts twice. Due to this "half" relationship, the Lie algebra corresponding to quaternions is slightly different from $\mathfrak{so}(3)$. To maintain the continuity of derivation and writing, we use a unified Exp relationship to combine the two definitions. In summary, for a three-dimensional instantaneous angular velocity (or the update quantity of an optimization function) $\omega \in \mathbb{R}^3$, we define its kinematic form on $\text{SO}(3)$ as:

$$\dot{\mathbf{R}} = \mathbf{R}\omega^\wedge \quad (2.73)$$

Its corresponding exponential mapping is:

$$\mathbf{R} = \text{Exp}(\omega) = \exp(\omega^\wedge), \quad (2.74)$$

or, if this quantity is the update amount of a pure imaginary quaternion (typically obtained from solving an optimization function), then the corresponding quaternion should only update half of it. According to the definition in Equation (2.62), the quaternion kinematic equation and exponential mapping can be written as:

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{q}[0, \boldsymbol{\omega}]^\top, \quad (2.75)$$

which can usually be simplified as⁵:

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{q}\boldsymbol{\omega}, \quad (2.76)$$

Here, the coefficient 1/2 is used to unify the definition of angular velocity on SO(3) with quaternion angular velocity, so this equation differs from Equation (2.62). Readers should also note that this equation implies that the three-dimensional vector $\boldsymbol{\omega}$ is first converted to a quaternion before multiplying with \mathbf{q} , rather than directly multiplying \mathbf{q} by $\boldsymbol{\omega}$.

The quaternion exponential mapping can also be written similarly as:

$$\mathbf{q} = \exp\left(\frac{1}{2}[0, \boldsymbol{\omega}]^\top\right) \triangleq \text{Exp}(\boldsymbol{\omega}). \quad (2.77)$$

If $\boldsymbol{\omega}$ is small, then $\cos(\frac{\theta}{2}) \approx 1$, $\mathbf{n} \sin \frac{\theta}{2} \approx \mathbf{n} \frac{\theta}{2}$, and the exponential mapping has a simplified form:

$$\text{Exp}(\boldsymbol{\omega}) \approx [1, \frac{1}{2}\boldsymbol{\omega}], \quad (2.78)$$

So the quaternion update formula can be simplified as⁶:

$$\mathbf{q}\text{Exp}(\boldsymbol{\omega}) \approx \mathbf{q}[1, \frac{1}{2}\boldsymbol{\omega}], \quad (2.79)$$

However, a significant disadvantage of this equation compared to the update equation on SO(3) is that the quaternion on the right-hand side is not a unit quaternion, so after long-term updates, it needs to be re-normalized [6]. This problem does not exist for rotation matrices, as $\text{Exp}(\boldsymbol{\omega})$ is always a rotation matrix.

Thus far, we have introduced the kinematics from the perspectives of SO(3) and quaternions, as well as their conversion relationship. With this relationship, we can use either rotation matrices or quaternions when writing the motion equations of a vehicle or when calculating the Jacobian matrices of optimization problems, just remember the coefficient of 1/2. We can also mix quaternions and rotation matrices, just remember that when updating variables, quaternions only need to be updated by half.

In addition, we can also consider kinematics at the level of the Lie algebra $\mathfrak{so}(3)$. For the translation part, it can be viewed as independent three-dimensional variables or collectively considered in SE(3). In practice, these expressions are interchangeable without essential differences, but there may be differences in the ease of operation. We introduce several other expressions in this section, but only one of them will be detailed in subsequent chapters.

⁵Note that the meaning of $\boldsymbol{\omega}$ has changed here. In the previous equation, it was a three-dimensional vector, while in the next equation, it is a quaternion.

⁶In this equation, there is no need to change the definition of $\boldsymbol{\omega}$, it is still a three-dimensional vector.

2.2.4 Other Kinematic Representations

Kinematics on $\mathfrak{so}(3)$

To give some physical meaning to mathematical symbols, we will use ω to express angular velocity and ϕ to express rotation vectors in the future. The Rodrigues formula tells us that $\mathbf{R} = \text{Exp}(\phi)$. Now we want to examine the derivative of ϕ with respect to time and its relationship with the instantaneous angular velocity ω .

The BCH formula gives the relationship between increments on the Lie group and the Lie algebra. Assuming that at time t to $t + \Delta t$, $\phi(t)$ changes to $\phi(t) + \Delta\phi$ on $\mathfrak{so}(3)$, and at the same time $\text{SO}(3)$ changes from \mathbf{R} to $\mathbf{R} \cdot \Delta\mathbf{R}$, then according to the BCH approximation, we have:

$$\Delta\mathbf{R} = \text{Exp}(\mathbf{J}_r \Delta\phi), \quad (2.80)$$

On the $\text{SO}(3)$ level, according to the definition of angular velocity, we have $\dot{\mathbf{R}} = \mathbf{R}\omega^\wedge$, so:

$$\begin{aligned} \mathbf{R}\omega^\wedge &= \dot{\mathbf{R}} = \lim_{\Delta t \rightarrow 0} \frac{\mathbf{R}(t + \Delta t) - \mathbf{R}(t)}{\Delta t} \\ &\approx \lim_{\Delta t \rightarrow 0} \frac{\mathbf{R}(t)\text{Exp}(\mathbf{J}_r \Delta\phi) - \mathbf{R}(t)}{\Delta t} \\ &\approx \lim_{\Delta t \rightarrow 0} \frac{\mathbf{R}(t)(\text{Exp}(\mathbf{J}_r \Delta\phi) - \mathbf{I})}{\Delta t} = \mathbf{R}(\mathbf{J}_r \dot{\phi})^\wedge, \end{aligned} \quad (2.81)$$

where the last equality requires a Taylor expansion of the Exp function. By comparing the left and right sides, we easily obtain:

$$\omega = \mathbf{J}_r \dot{\phi}, \quad (2.82)$$

or:

$$\dot{\phi} = \mathbf{J}_r^{-1} \omega. \quad (2.83)$$

This shows the relationship between the time derivative on $\mathfrak{so}(3)$ and the instantaneous angular velocity on $\text{SO}(3)$. In principle, we can also use this quantity to derive subsequent filters or optimizers. However, its physical meaning is not as intuitive as ω , so very few people actually choose this method.

Kinematics on $\text{SO}(3) + \mathbf{t}$

We can incorporate linear velocity into consideration. For example, let $\mathbf{v} = \dot{\mathbf{t}}$, then the system kinematic equations can be written as:

$$\dot{\mathbf{R}} = \mathbf{R}\omega^\wedge, \quad \dot{\mathbf{t}} = \mathbf{v}. \quad (2.84)$$

This approach is the simplest and most intuitive, and is widely adopted.

Kinematics on $\text{SE}(3)$

We can also derive kinematics on $\text{SE}(3)$ and make it consistent with the exponential mapping on $\text{SO}(3)$. This requires some modifications to the linear velocity part. Let the transformation matrix be:

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{bmatrix} \in \text{SE}(3), \quad (2.85)$$

then its time derivative is:

$$\dot{\mathbf{T}} = \begin{bmatrix} \dot{\mathbf{R}} & \dot{\mathbf{t}} \\ \mathbf{0}^\top & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{R}\omega^\wedge & \mathbf{v} \\ \mathbf{0}^\top & 0 \end{bmatrix}. \quad (2.86)$$

For instance, to achieve the kinematics on SE(3) in the right-multiplication model, we want to obtain the form $\dot{\mathbf{T}} = \mathbf{T}\xi^\wedge$ ⁷, let $\xi = [\rho, \phi]^\top$, then:

$$\begin{bmatrix} \mathbf{R}\omega^\wedge & \mathbf{v} \\ \mathbf{0}^\top & 0 \end{bmatrix} = \mathbf{T} \begin{bmatrix} \phi^\wedge & \rho \\ \mathbf{0}^\top & 0 \end{bmatrix}. \quad (2.87)$$

It is not difficult to derive:

$$\phi = \omega, \quad \rho = \mathbf{R}^\top \mathbf{v}. \quad (2.88)$$

Therefore, by defining $\xi = [\mathbf{R}^\top \mathbf{v}, \omega]^\top$, we can obtain the kinematics on SE(3) as:

$$\dot{\mathbf{T}} = \mathbf{T}\xi^\wedge. \quad (2.89)$$

Kinematics on $\mathfrak{se}(3)$

Let the Lie algebra be φ . To derive the kinematics of φ , we still use the approach in Section 2.2.1. According to the BCH approximation, when φ increases by $\Delta\varphi$, \mathbf{T} right-multiplies $\Delta\mathbf{T}$. Analogously to the previous approach, we can write:

$$\dot{\mathbf{T}} = \mathbf{T}\xi^\wedge = \lim_{\Delta t \rightarrow 0} \frac{\mathbf{T}(t)\text{Exp}(\mathcal{J}_r\Delta\varphi) - \mathbf{T}(t)}{\Delta t} \quad (2.90)$$

$$= \mathbf{T}\mathcal{J}_r\dot{\varphi}^\wedge. \quad (2.91)$$

Here, some intermediate steps are omitted. Finally, we obtain:

$$\dot{\varphi} = \mathcal{J}_r^{-1}\xi. \quad (2.92)$$

This can also be used to characterize the kinematics on the Lie algebra. However, in these expressions, only the SO(3) + \mathbf{t} representation corresponds to the actual physical meaning, and the other representations require some degree of transformation. In a large number of papers, researchers default to using the simplest kinematic representation, i.e., rotation plus translation. In practice, there is no need to introduce unnecessary complications in theory, so we **default to using kinematics with rotation and translation**, but the rotation representation can freely use either SO(3) or quaternions (corresponding to different update quantities).

2.2.5 Linear Velocity and Acceleration

Now let's consider the transformation relationship of linear velocity and acceleration between different coordinate systems. For simplicity, we consider the transformation of linear velocity and acceleration between two coordinate systems with **only rotational relationship**.

Consider coordinate systems 1 and 2. A certain vector \mathbf{p} has coordinates $\mathbf{p}_1, \mathbf{p}_2$ in the two systems, and it is obvious that they satisfy the relationship $\mathbf{p}_1 = \mathbf{R}_{12}\mathbf{p}_2$, which is a simple geometric relationship.

Now we consider the case where \mathbf{p} varies with time, while the two coordinate systems also undergo rotation. We want to remind the reader that **the velocity vector of \mathbf{p} in the two systems is different**; it is not the expression of the same vector in different coordinate systems. Let's see why.

⁷The \wedge symbol on SE(3) is defined as: $\xi^\wedge = \begin{bmatrix} \phi^\wedge & \rho^\wedge \\ \mathbf{0}^\top & 0 \end{bmatrix} \in \mathbb{R}^{4 \times 4}$, where ϕ is the rotation part and ρ is the translation part.

Taking the time derivative of the above equation, we have:

$$\begin{aligned}\dot{\mathbf{p}}_1 &= \dot{\mathbf{R}}_{12}\mathbf{p}_2 + \mathbf{R}_{12}\dot{\mathbf{p}}_2 \\ &= \mathbf{R}_{12}\boldsymbol{\omega}^\wedge\mathbf{p}_2 + \mathbf{R}_{12}\dot{\mathbf{p}}_2 \\ &= \mathbf{R}_{12}(\boldsymbol{\omega}^\wedge\mathbf{p}_2 + \dot{\mathbf{p}}_2).\end{aligned}\quad (2.93)$$

Traditionally, we denote $\dot{\mathbf{p}}_1 = \mathbf{v}_1$, $\dot{\mathbf{p}}_2 = \mathbf{v}_2$, then we can obtain the transformation equation for the two linear velocities:

$$\mathbf{v}_1 = \mathbf{R}_{12}(\boldsymbol{\omega}^\wedge\mathbf{p}_2 + \mathbf{v}_2). \quad (2.94)$$

We can see that there is actually a relationship between the two velocity vectors and the angular velocity. At this point, we do not speak of **different coordinate expressions of a velocity vector**, but rather **the transformation of velocity vectors in two coordinate systems**.

Continuing to take the time derivative of the above equation, we obtain:

$$\begin{aligned}\dot{\mathbf{v}}_1 &= \dot{\mathbf{R}}_{12}(\boldsymbol{\omega}^\wedge\mathbf{p}_2 + \mathbf{v}_2) + \mathbf{R}_{12}(\dot{\boldsymbol{\omega}}^\wedge\mathbf{p}_2 + \boldsymbol{\omega}^\wedge\dot{\mathbf{p}}_2 + \dot{\mathbf{v}}_2) \\ &= \mathbf{R}_{12}(\boldsymbol{\omega}^\wedge\boldsymbol{\omega}^\wedge\mathbf{p}_2 + \boldsymbol{\omega}^\wedge\mathbf{v}_2 + \dot{\boldsymbol{\omega}}^\wedge\mathbf{p}_2 + \boldsymbol{\omega}^\wedge\dot{\mathbf{p}}_2 + \dot{\mathbf{v}}_2) \\ &= \mathbf{R}_{12}(\dot{\mathbf{v}}_2 + 2\boldsymbol{\omega}^\wedge\mathbf{v}_2 + \dot{\boldsymbol{\omega}}^\wedge\mathbf{p}_2 + \boldsymbol{\omega}^\wedge\boldsymbol{\omega}^\wedge\mathbf{p}_2).\end{aligned}\quad (2.95)$$

Defining $\mathbf{a}_1 = \dot{\mathbf{v}}_1$, $\mathbf{a}_2 = \dot{\mathbf{v}}_2$, the equation can be written as:

$$\mathbf{a}_1 = \mathbf{R}_{12}\left(\underbrace{\mathbf{a}_2}_{\text{acceleration}} + \underbrace{2\boldsymbol{\omega}^\wedge\mathbf{v}_2}_{\text{Coriolis acceleration}} + \underbrace{\dot{\boldsymbol{\omega}}^\wedge\mathbf{p}_2}_{\text{angular acceleration}} + \underbrace{\boldsymbol{\omega}^\wedge\boldsymbol{\omega}^\wedge\mathbf{p}_2}_{\text{centripetal acceleration}}\right). \quad (2.96)$$

This equation gives the transformation between the expressions of acceleration in the two systems. It can be seen that due to the motion relationship between the two systems themselves, the transformation of acceleration is more complex than that of velocity, and it requires considering the angular velocity and angular acceleration between the two systems. Fortunately, these terms have special names to help readers remember. Moreover, in practical processing, since measurement sensors can only measure discrete values, in low-precision applications, we usually choose to ignore the last three terms and only retain the simplest transformation relationship.

In addition, in practical vehicles, we usually take system 1 and system 2 as the world coordinate system and the vehicle coordinate system, respectively. If we consider a moving point in the vehicle coordinate system, it is obvious that the linear velocity of this point in the vehicle system and in the world system are not the same vector, and it should be related to the rotation of the vehicle. These two linear velocities should satisfy the transformation relationship described in this section. However, we don't often talk about a moving point in the vehicle. More often, we discuss the **velocity of the vehicle itself**, which is the velocity of the vehicle body origin in the world system (the velocity of the vehicle origin in the vehicle system is always zero and has no practical meaning). This velocity is defined in the world system and is denoted as \mathbf{v}_w . If left-multiplied by \mathbf{R}_{bw} , this vector can also be transformed into the vehicle coordinate system, denoted as \mathbf{v}_b . We call \mathbf{v}_b the **body velocity**, which essentially means the result of transforming the velocity vector in the world system into the vehicle coordinate system, and it can be measured by various sensors (such as the vehicle speed sensor, rotation sensor on the wheels, etc.). Note that this transformation relationship is different from equation (2.94), one represents the relationship between different vectors, and the other represents the coordinate transformation relationship of the same vector. Please pay attention to their differences.

2.2.6 Perturbation and Jacobian Matrices

When we left-multiply or right-multiply increments on a Lie group (whether represented by rotation matrices or quaternions), there exists a corresponding increment on the Lie algebra. Due to the existence of the Baker-Campbell-Hausdorff formula, there will be a Jacobian matrix between these two increments in the sense of first-order linear approximation. Obviously, this Jacobian matrix will differ depending on the representation method or the definition of increment addition. Below, we discuss some feasible choices and definition methods, and provide some common methods for calculating Jacobians.

If we want to differentiate functions containing rotations or transformations, this derivative can be defined either at the vector level, i.e., $\mathfrak{so}(3)$ and $\mathfrak{se}(3)$, or at the perturbation level, which means left-multiplying or right-multiplying perturbations on the original \mathbf{R} , \mathbf{T} , \mathbf{q} , and then differentiating with respect to the perturbation. In most cases, differentiating with respect to perturbations is a more concise and clear approach. Below, we discuss the differences in formulas when perturbing rotation matrices or quaternions separately.

Example: Rotation of Vectors

Consider a vector \mathbf{a} , and let's rotate it. Rotation can be expressed by either a rotation matrix \mathbf{R} or a quaternion \mathbf{q} . Thus, the rotation of \mathbf{a} can be written as \mathbf{Ra} in the sense of matrix multiplication, or \mathbf{qaq}^* in the sense of quaternion multiplication.

Firstly, the derivation with respect to \mathbf{a} itself is trivial⁸, and there is no need to elaborate⁹:

$$\frac{\partial \mathbf{Ra}}{\partial \mathbf{a}} = \frac{\partial (\mathbf{qaq}^*)}{\partial \mathbf{a}} = \mathbf{R}. \quad (2.97)$$

The derivation with respect to \mathbf{R} or \mathbf{q} depends on the definition method. Generally, we can choose to derive with respect to the four elements of \mathbf{q} itself or the Lie algebra corresponding to \mathbf{R} , but the Jacobian matrix corresponding to the perturbation model will be simpler. Moreover, the perturbation model is divided into left perturbation and right perturbation, and there are different definition methods for \mathbf{R} and \mathbf{q} . Earlier in this book, the right perturbation method was used when introducing angular velocity, so here we also consider right perturbation for \mathbf{R} ¹⁰. Let the perturbation quantity be ϕ , then¹¹:

$$\begin{aligned} \frac{\partial \mathbf{Ra}}{\partial \mathbf{R}} &= \lim_{\phi \rightarrow 0} \frac{\mathbf{R}\text{Exp}(\phi) \mathbf{a} - \mathbf{Ra}}{\phi} \\ &= \lim_{\phi \rightarrow 0} \frac{\mathbf{R}(\mathbf{I} + \phi^\wedge) \mathbf{a} - \mathbf{Ra}}{\phi} = -\mathbf{Ra}^\wedge. \end{aligned} \quad (2.98)$$

Similarly, although it is not impossible to derive with respect to the quaternion itself, it is still relatively cumbersome. Reference [6] provides a method for deriving with respect to \mathbf{q} without detailed derivation. Suppose $\mathbf{q} = [w, \mathbf{v}]$, then the partial derivatives with respect

⁸That is, it can be calculated using standard matrix derivative rules without additional conversion procedures. Readers with a certain level of matrix knowledge should be able to see this on their own.

⁹We still omit the transpose symbol at the denominator to maintain the simplicity of the formula.

¹⁰There is no essential difference between left and right perturbation. However, the expression of velocity or acceleration quantities in different coordinate systems may differ. According to the convention described earlier in this book, we mainly use the *wb* sequence to express the transformation relationship. In this case, the symbols for angular velocity, velocity, etc., are consistent with the measured values. To accommodate this expression method, we use the right perturbation model when deriving. If the reader still cannot understand the reason here, they can reconsider it later in the following text.

¹¹This equation can be denoted as $\frac{\partial \mathbf{Ra}}{\partial \mathbf{R}}$ or $\frac{\partial \mathbf{Ra}}{\partial \phi}$ on the left side.

to the real part and the imaginary part of \mathbf{q} yield:

$$\frac{\partial \mathbf{q} \mathbf{a} \mathbf{q}^*}{\partial \mathbf{q}} = 2 [w\mathbf{a} + \mathbf{v}^\wedge \mathbf{a}, \mathbf{v}^\top \mathbf{a} \mathbf{I}_3 + \mathbf{v} \mathbf{a}^\top - \mathbf{a} \mathbf{v}^\top - w\mathbf{a}^\wedge] \in \mathbb{R}^{3 \times 4}. \quad (2.99)$$

This is evidently too complex. However, we can perturb \mathbf{q} . Let the perturbation quantity be $\boldsymbol{\omega} \in \mathbb{R}^3$, in order to be consistent with $\text{SO}(3)$, we right-multiply \mathbf{q} by $\frac{1}{2}[1, \boldsymbol{\omega}]^\top$, then, since the size of the perturbation on the rotation matrix is still the same, its Jacobian matrix should also be consistent:

$$\frac{\partial \mathbf{R}\mathbf{a}}{\partial \boldsymbol{\omega}} = -\mathbf{R}\mathbf{a}^\wedge. \quad (2.100)$$

This example tells us that in practical operations, whether rotating with \mathbf{q} or \mathbf{R} , **we can use the same Jacobian**. If the perturbation quantity is our optimization variable, then just **update accordingly** when updating the optimization variables, instead of separately deriving Jacobian matrices for the two representation methods.

Example: Composition of Rotations

Now, let's consider the composition of rotations. We want to find the derivative of $\text{Log}(\mathbf{R}_1 \mathbf{R}_2)$ with respect to \mathbf{R}_1 . We cannot directly differentiate $\mathbf{R}_1 \mathbf{R}_2$ with respect to \mathbf{R}_1 or \mathbf{R}_2 because that would involve differentiating a matrix with respect to another matrix, which is not feasible without introducing tensors. Thus, we must introduce the Log operator to ensure we are dealing with vector-to-vector derivatives.

When perturbing \mathbf{R}_1 , we can derive:

$$\begin{aligned} \frac{\partial \text{Log}(\mathbf{R}_1 \mathbf{R}_2)}{\partial \mathbf{R}_1} &= \lim_{\phi \rightarrow 0} \frac{\text{Log}(\mathbf{R}_1 \text{Exp}(\phi) \mathbf{R}_2) - \text{Log}(\mathbf{R}_1 \mathbf{R}_2)}{\phi} \\ &= \lim_{\phi \rightarrow 0} \frac{\text{Log}(\mathbf{R}_1 \mathbf{R}_2 \text{Exp}(\mathbf{R}_2^\top \phi)) - \text{Log}(\mathbf{R}_1 \mathbf{R}_2)}{\phi} \\ &= \mathbf{J}_r^{-1}(\text{Log}(\mathbf{R}_1 \mathbf{R}_2)) \mathbf{R}_2^\top. \end{aligned} \quad (2.101)$$

Here, the second line uses the adjoint property of $\text{SO}(3)$:

$$\mathbf{R}^\top \text{Exp}(\phi) \mathbf{R} = \text{Exp}(\mathbf{R}^\top \phi), \quad (2.102)$$

and the third line uses the first-order approximation of the Baker-Campbell-Hausdorff formula:

$$\text{Log}(\mathbf{R}_1 \mathbf{R}_2 \text{Exp}(\mathbf{R}_2^\top \phi)) = \text{Log}(\mathbf{R}_1 \mathbf{R}_2) + \mathbf{J}_r^{-1}(\mathbf{R}_1 \mathbf{R}_2) \text{Log}(\text{Exp}(\mathbf{R}_2^\top \phi)). \quad (2.103)$$

Similarly, when perturbing \mathbf{R}_2 , we can obtain:

$$\frac{\partial \text{Log}(\mathbf{R}_1 \mathbf{R}_2)}{\partial \mathbf{R}_2} = \mathbf{J}_r^{-1}(\text{Log}(\mathbf{R}_1 \mathbf{R}_2)). \quad (2.104)$$

Equations (2.101) and (2.104) serve as the basis for many complex function derivatives. It's essential for readers to grasp them. In practical applications, it's common to encounter compositions involving rotation matrices and other matrices or vectors. Many composite formulas can be derived using the above two equations.

2.3 Kinematics Example: Circular Motion

Below we demonstrate the differences in handling angular velocity using quaternions and rotation matrices through some practical examples.

When driving a car, if we maintain a constant speed and fix the steering wheel at a certain angle, the car should trace out a circular path. Now consider: how can we simulate this in a program?

Clearly, such a vehicle should have a fixed angular velocity. Assuming **forward-left-up** as the coordinate system, the angular velocity vector ω of the vehicle should point towards the Z direction. The previously mentioned **constant speed** implies that in the vehicle's coordinate system, the velocity vector should be fixed pointing forward $\mathbf{v}_b = [v_x, 0, 0]^\top$. Of course, its velocity in the world frame is not solely along the X -axis because it is also turning simultaneously. Now let's implement a program to simulate this vehicle's motion. We'll use both rotation matrices and quaternions to handle the vehicle's rotation.

Listing 2.1: src/ch2/motion.cc

```

1 #include <gflags/gflags.h>
2 #include <glog/logging.h>
3
4 #include "common/eigen_types.h"
5 #include "common/math_utils.h"
6 #include "tools/ui/pangolin_window.h"
7
8 /// This program demonstrates a vehicle undergoing circular motion
9 /// Angular velocity and linear velocity of the vehicle can be set in flags
10
11 DEFINE_double(angular_velocity, 10.0, "Angular velocity in degrees per second");
12 DEFINE_double(linear_velocity, 5.0, "Linear velocity of the vehicle in m/s");
13 DEFINE_bool(use_quaternion, false, "Whether to use quaternion calculations");
14
15 int main(int argc, char** argv) {
16     google::InitGoogleLogging(argv[0]);
17     FLAGS_stderrthreshold = google::INFO;
18     FLAGS_colorlogtosterr = true;
19     google::ParseCommandLineFlags(&argc, &argv, true);
20
21     /// Visualization
22     sad::ui::PangolinWindow ui;
23     if (ui.Init() == false) {
24         return -1;
25     }
26
27     double angular_velocity_rad = FLAGS_angular_velocity * sad::math::kDEG2RAD; // 
28     // Angular velocity in radians
29     SE3 pose; // Pose represented by TWB
30     Vec3d omega(0, 0, angular_velocity_rad); // Angular velocity vector
31     Vec3d v_body(FLAGS_linear_velocity, 0, 0); // Body frame velocity
32     const double dt = 0.05; // Time for each update
33
34     while (ui.ShouldQuit() == false) {
35         // Update position
36         Vec3d v_world = pose.so3() * v_body;
37         pose.translation() += v_world * dt;
38
39         // Update rotation
40         if (FLAGS_use_quaternion) {
41             Quatd q = pose.unit_quaternion() * Quatd(1, 0.5 * omega[0] * dt, 0.5 * omega[1]
42             * dt, 0.5 * omega[2] * dt);
43             q.normalize();
44             pose.so3() = S03(q);
45         } else {
46             pose.so3() = pose.so3() * S03::exp(omega * dt);
47         }
48
49         LOG(INFO) << "pose: " << pose.translation().transpose();
50         ui.UpdateNavState(sad::NavStated(0, pose, v_world));
51     }
52 }
```

```

50     usleep(dt * 1e6);
51 }
52 ui.Quit();
53 return 0;
54 }
```

Since this is the first program appearing in this book, let's provide a more complete version of it. Subsequent programs will only display the core code.

Throughout this book, we use GLog for managing logs and GFlags for managing program parameters. This program accepts user-specified angular velocity and linear velocity magnitudes, as well as the choice between using rotation matrices or quaternions for processing. Here's what we do:

1. Firstly, we convert the user-provided angular velocity to radians and linear velocity to v_{body} in the vehicle's body frame. We set the simulation time interval to 0.05 seconds.
2. During each update, we calculate the velocity in the world frame. For this, we need to know the orientation of the vehicle, so we extract the pose variable's pose and right-multiply it by the body velocity.
3. Then we update the vehicle's state. If the user specifies quaternion representation, we use Equation (2.79); otherwise, we update the self-attitude using Equation (2.56).
4. Finally, we pass the calculated pose to the UI for display and wait for a time interval.

To visualize the effect of this program in real-time, we provide a UI interface for readers. By updating the current pose and velocity in the UI interface, they will be displayed in real-time in a 3D window, as shown in Figure 2-2. After compiling the code in this chapter, readers can execute the program using:

Listing 2.2: Terminal Input:

```
./bin/motion
```

To change parameters or calculation methods, simply fill in the GFlags:

Listing 2.3: Terminal Input:

```
bin/motion --use_quaternion=true --angular_velocity=15
```

Most programs in this book can be executed in a similar manner. Readers can familiarize themselves with the code style of this book using this section's program. Through this experiment, we observe the vehicle tracing out a complete circular path. Its velocity in the world frame resembles trigonometric functions, while the velocity in the body frame remains fixed along the X axis. Whether using quaternions or rotation matrices, there's no essential difference in handling kinematics. Readers can utilize this section's program to demonstrate common free fall or parabolic motion. These are left as exercises for the readers.

2.4 Filters and Optimization

Here we review the basic principles of filters and their connection to optimization methods. We'll start with state estimation.

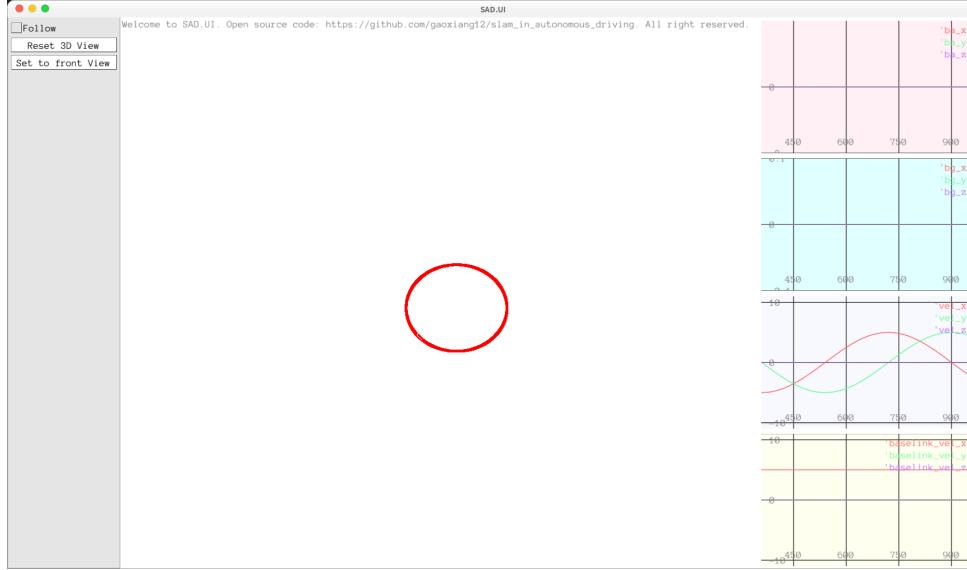


Figure 2-2: Kinematic simulation of a vehicle undergoing circular motion

2.4.1 State Estimation and Least Squares

SLAM problems, localization problems, or mapping problems can all be summarized as state estimation problems. A typical discrete-time state estimation problem consists of a set of motion equations and a set of observation equations:

$$\begin{cases} \mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k, & k = 1, \dots, N \\ \mathbf{z}_k = \mathbf{h}(\mathbf{x}_k) + \mathbf{v}_k, \end{cases} \quad (2.105)$$

where \mathbf{f} is called the motion equation, \mathbf{h} is called the observation equation, $\mathbf{w}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_k)$, $\mathbf{v}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_k)$ are Gaussian-distributed random noises. If \mathbf{f} and \mathbf{h} are assumed to be linear functions, we obtain the state estimation problem of a Linear Gaussian (LG) system:

$$\begin{cases} \mathbf{x}_k = \mathbf{A}_k \mathbf{x}_{k-1} + \mathbf{u}_k + \mathbf{w}_k \\ \mathbf{z}_k = \mathbf{C}_k \mathbf{x}_k + \mathbf{v}_k \end{cases}. \quad (2.106)$$

Here \mathbf{A}_k and \mathbf{C}_k are the system's transition matrix and observation matrix, respectively. LG systems represent the simplest form of state estimation problems, and their unbiased optimal estimation is provided by the **Kalman Filter** (KF) [4, 46].

2.4.2 Kalman Filter

The Kalman filter describes how to recursively estimate the state from one time step to the next. It consists of two steps: **prediction** and **update**. The prediction step propagates the motion equation, while the update step corrects the result from the previous step. Let \mathbf{x}_{k-1} , \mathbf{P}_{k-1} denote the state estimate and its covariance matrix at time $k-1$, where \mathbf{x}_{k-1} is the mean and \mathbf{P}_{k-1} is the estimated covariance matrix.

1. Prediction:

$$\mathbf{x}_{k,\text{pred}} = \mathbf{A}_k \mathbf{x}_{k-1} + \mathbf{u}_k, \quad \mathbf{P}_{k,\text{pred}} = \mathbf{A}_k \mathbf{P}_{k-1} \mathbf{A}_k^\top + \mathbf{R}_k. \quad (2.107)$$

2. Update: First, calculate \mathbf{K} , also known as the **Kalman gain**.

$$\mathbf{K}_k = \mathbf{P}_{k,\text{pred}} \mathbf{C}_k^\top \left(\mathbf{C}_k \mathbf{P}_{k,\text{pred}} \mathbf{C}_k^\top + \mathbf{Q}_k \right)^{-1}. \quad (2.108)$$

Then compute the posterior distribution.

$$\begin{aligned} \mathbf{x}_k &= \mathbf{x}_{k,\text{pred}} + \mathbf{K}_k (\mathbf{z}_k - \mathbf{C}_k \mathbf{x}_{k,\text{pred}}), \\ \mathbf{P}_k &= (\mathbf{I} - \mathbf{K}_k \mathbf{C}_k) \mathbf{P}_{k,\text{pred}}. \end{aligned} \quad (2.109)$$

Here, the subscript "pred" denotes the predicted result. Different books may use various symbols to express the difference between it and the final estimate, such as $\hat{\mathbf{x}}$, \mathbf{x}^* , $\check{\mathbf{x}}$, and so on. In this book, we use subscript notation to distinguish predicted variables.

We won't delve into the derivation process of the linear Kalman filter. However, we would like to remind readers that in linear systems, various methods (Bayesian filters, Kalman filters, least squares, gain optimization, etc.) will all reach the same conclusion, so the Kalman filter can be derived from various methods, such as:

1. Derivation from the perspective of gain optimization, i.e., assuming the optimal estimate takes the form of $\mathbf{x}_{k,\text{pred}} + \mathbf{K}_k (\mathbf{z}_k - \mathbf{C}_k \mathbf{x}_{k,\text{pred}})$ and then finding the optimal \mathbf{K}_k . This derivation method is the simplest and is the preferred method for most similar materials.
2. Derivation from the Bayesian filter, which requires the use of linear transformations and marginalization of Gaussian distributions. This is also the preferred method in [7] and is our approach in [1].
3. Derivation from Maximum A Posteriori (MAP) estimation, which only requires basic linear algebra.
4. Derivation from batch MAP solution, using Cholesky decomposition to distinguish between forward and backward processes, and deriving the Kalman filter from the forward process. This is the preferred method in [4], which has the advantage of showing the connection between the Kalman filter and the Rauch-Tung-Stribel Smoother method (as well as the connection with batch MAP), but the downside is that the derivation process is more complex and requires a lot of space to write it.

Different from traditional Kalman filters, this book uniformly employs Lie group and Lie algebra methods to handle Kalman filters. Since we need to consider motion equations, the state variable \mathbf{x} not only includes position and orientation but also includes other variables such as velocity and sensor biases. Thus, such a high-dimensional \mathbf{x} lies on a high-dimensional manifold \mathcal{M} , known as the **Kalman Filter on Manifold** [47]. In the next chapter, we will see that this manifold-based approach is more concise than methods based on Euler angles or raw quaternion components.

2.4.3 Nonlinear Systems

In nonlinear systems, the preferred approach is to linearize \mathbf{f} and \mathbf{h} (**linearization**). Linearization essentially involves finding a **Taylor expansion** of a function around a fixed point and retaining only the first-order coefficients. Linearization is a widely used theory in the subsequent discussion. If linearization is performed on a general vector function $\mathbf{f}(\mathbf{x})$ at point \mathbf{x}_0 , the result should be:

$$\mathbf{f}(\mathbf{x}_0 + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}_0) + \mathbf{J}\Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^\top \mathbf{H}\Delta\mathbf{x} + O(\Delta\mathbf{x}^2), \quad (2.110)$$

where \mathbf{J} is the **Jacobian matrix**, and \mathbf{H} is the **Hessian matrix**, which are the two most important matrices in linearization. If only the first-order term is retained, $\mathbf{f}(\mathbf{x})$ can be approximated as:

$$\mathbf{f}(\mathbf{x}_0 + \Delta\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + \mathbf{J}\Delta\mathbf{x}. \quad (2.111)$$

We can linearize the motion equations and observation equations of nonlinear systems, and then apply the conclusions of the Kalman filter to nonlinear systems, resulting in the **Extended Kalman Filter** (EKF). Without discussing the definition of \mathbf{x} and the detailed forms of each matrix, a general EKF can be described as follows.

First, linearize the motion equations around the previous state to obtain:

$$\mathbf{x}_k \approx \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{F}_k \Delta\mathbf{x}_k + \mathbf{w}_k, \quad (2.112)$$

where \mathbf{F} is the Jacobian matrix relative to the previous state. This matrix is mainly used to compute the predicted covariance. As for the predicted mean value, it can be obtained by substituting \mathbf{x}_{k-1} into \mathbf{f} . This forms the prediction process of the EKF:

$$\mathbf{x}_{k,\text{pred}} = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k), \quad \mathbf{P}_{k,\text{pred}} = \mathbf{F}_k \mathbf{P}_{k-1} \mathbf{F}_k^\top + \mathbf{R}_k. \quad (2.113)$$

It should be noted that this actually assumes that a **Gaussian distributed state variable remains Gaussian distributed after passing through a nonlinear function**. This is actually an approximation and may differ significantly from the actual situation. For the observation equation, linearize it around $\mathbf{x}_{k,\text{pred}}$ to obtain:

$$\mathbf{z}_k \approx \mathbf{h}(\mathbf{x}_{k,\text{pred}}) + \mathbf{H}_k (\mathbf{x}_k - \mathbf{x}_{k,\text{pred}}) + \mathbf{n}_k, \quad (2.114)$$

then substitute it into the Kalman filter's gain formula and update equation to obtain the update process of the EKF:

$$\mathbf{K}_k = \mathbf{P}_{k,\text{pred}} \mathbf{H}_k^\top (\mathbf{H}_k \mathbf{P}_{k,\text{pred}} \mathbf{H}_k^\top + \mathbf{Q}_k)^{-1}, \quad (2.115)$$

$$\mathbf{x}_k = \mathbf{x}_{k,\text{pred}} + \mathbf{K}_k (\mathbf{z}_k - \mathbf{H}_k \mathbf{x}_{k,\text{pred}}), \quad (2.116)$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{C}_k) \mathbf{P}_{k,\text{pred}}. \quad (2.117)$$

Comparing the KF and EKF, we find that they are basically the same in terms of formulas, except that the coefficients matrices of the EKF are not fixed and can change with the linearization point.

In this way, we have quickly reviewed the formulas of KF and EKF. However, the discussion here does not elaborate on how to calculate each matrix when \mathbf{x} contains variables such as displacement, rotation, velocity, etc. In particular, if the rotation in \mathbf{x} is represented in the form of \mathbf{R} , we actually cannot directly write expressions like $\mathbf{x}_k - \mathbf{x}_{k-1}$ or $\mathbf{x}_k - \mathbf{x}_{k,\text{pred}}$, and should use left and right perturbation models to handle these terms. After introducing Lie groups and Lie algebras, how should the EKF be modified is the problem we will discuss in Chapter 3.

2.4.4 Graph Optimization

On the other hand, both the motion equations and observation equations can be viewed as residuals between a state variable \mathbf{x} and the kinematic inputs or observations. This is a perspective of **batch least squares**:

$$\mathbf{e}_{\text{motion}} = \mathbf{x}_k - \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}) \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_k), \quad (2.118)$$

$$\mathbf{e}_{\text{obs}} = \mathbf{z}_k - \mathbf{h}(\mathbf{x}_k) \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_k). \quad (2.119)$$

The optimal state estimation in the filter can be seen as a least squares problem with respect to the error terms:

$$\mathbf{x}^* = \arg \min_{\mathbf{x}} \sum_k (\mathbf{e}_k^\top \Omega_k^{-1} \mathbf{e}_k). \quad (2.120)$$

Here, \mathbf{e}_k represents the k -th error term, and Ω_k is the covariance matrix of this error. The error \mathbf{e}_k can be replaced by the motion error or observation error mentioned above, but the least squares algorithm does not deliberately distinguish between kinematic errors or observation errors. For a least squares problem composed of a general error function \mathbf{e}_k , we can use **iterative optimization** methods for solving. The overall idea is as follows:

1. First, start from some initial value of \mathbf{x} , for example \mathbf{x}_0 .
2. Suppose the i -th iteration value is \mathbf{x}_i . Then, linearize the error function at \mathbf{x}_i :

$$\mathbf{e}_k(\mathbf{x}_i + \Delta\mathbf{x}) \approx \mathbf{e}_k(\mathbf{x}_i) + \mathbf{J}_{k,i} \Delta\mathbf{x}_i, \quad (2.121)$$

where the linearization matrix is $\mathbf{J}_{k,i}$.

3. Use methods like Gauss-Newton or similar to solve for the increment $\Delta\mathbf{x}_i$. For example, the linear equation solved by Gauss-Newton is:

$$\sum_k (\mathbf{J}_{k,i} \Omega_k^{-1} \mathbf{J}_{k,i}^\top) \Delta\mathbf{x}_i = -\sum_k (\mathbf{J}_{k,i} \Omega_k^{-1} \mathbf{e}_k). \quad (2.122)$$

4. Update \mathbf{x}_i to obtain the next iteration value: $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i$.
5. Check if the algorithm has converged. If it has, exit; if not, proceed to the next iteration.

Optimization methods are intricately linked with filtering methods. They yield the same results in linear systems, but generally not in nonlinear systems. The main reasons are as follows:

1. Optimization methods involve an iterative process, while the EKF does not.
2. The iterative process continuously computes Jacobian matrices at new linearization points \mathbf{x}_i , whereas the EKF's Jacobian matrix is only computed once at the prediction position.
3. The EKF also distinguishes $\mathbf{x}_{k,\text{pred}}$, treating the prediction process and observation process separately, while optimization methods do not have $\mathbf{x}_{k,\text{pred}}$ and treat state variables uniformly.

An important question is, if we ignore the third point above and regard the Kalman filter as a nonlinear optimization problem, how many optimization variables and error functions should the Kalman filter have? The answer is: two optimization variables and three error functions. The two optimization variables refer to \mathbf{x}_{k-1} and \mathbf{x}_k , while the three error functions are:

1. The prior error from the $k-1$ time step state \mathbf{x}_{k-1} , which follows its prior Gaussian distribution. If we assume $\mathbf{x}_{k-1} \sim \mathcal{N}(\bar{\mathbf{x}}_{k-1}, \mathbf{P}_{k-1})$, then a **prior error** is generated:

$$\mathbf{e}_{\text{prior}} = \mathbf{x}_{k-1} - \bar{\mathbf{x}}_{k-1} \sim \mathcal{N}(0, \mathbf{P}_{k-1}). \quad (2.123)$$

2. The **motion error** from $k-1$ to k .
3. The **observation error** at time step k .

The latter two are already listed in Equation (2.118). Thus, the Kalman filter and the optimization problem are equivalent (see Figure 2-3). However, their actual solution processes differ. The EKF does not update \mathbf{x}_{k-1} but only calculates the change of \mathbf{x}_k , while the optimizer treats all states equally. On the other hand, the EKF also updates the covariance matrix \mathbf{P}_k , while a regular optimizer only computes the mean part \mathbf{x}_k . If we want to obtain \mathbf{P}_k , we also need to perform **marginalization** on the optimization problem.

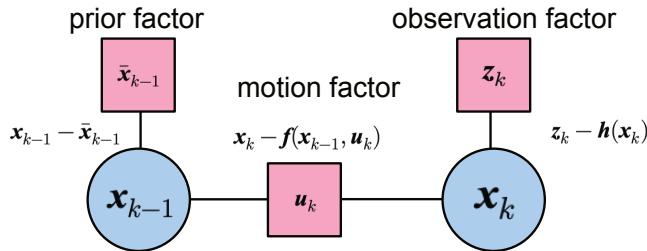


Figure 2-3: Kalman Filter and Graph Optimization Models

In the field of SLAM, optimization problems are described using graph models, corresponding to **graph optimization** or **factor graph**. Factor graph models can further introduce methods from **probabilistic graphical models** for solving. This book does not deliberately distinguish between the concepts of graph optimization and factor graph, as they usually do not have much difference in practical operation. However, comparing and discussing Kalman filters with graph optimization methods is one of the focuses of this book. We will implement classic EKF and graph optimization methods to solve problems involving inertial navigation, GPS, and laser point clouds. We will also extend EKF to Iterated Extended Kalman Filter (IEKF) to handle cases where there are nearest neighbor issues in the observation model (observation models with nearest neighbor issues may change in equation number and form during the iteration process, rather than simply linearizing at different points).

2.5 Summary

This section introduced various common coordinate systems and kinematic theories to the readers, focusing on two methods for handling kinematics: quaternions and $\text{SO}(3)$. We discussed their similarities and differences. We also reviewed the basic equations of KF and EKF, and discussed some differences between them and graph optimization.

The content of this section is primarily a review. In the next section, we will introduce ESKF in detail and provide implementations as well as animated demonstrations.

Exercises

1. Calculate

$$\frac{\partial \mathbf{R}^{-1} \mathbf{p}}{\partial \mathbf{R}}$$

using both left and right perturbation models.

2. Calculate

$$\frac{\partial \mathbf{R}_1 \mathbf{R}_2^{-1}}{\partial \mathbf{R}_2}$$

using both left and right perturbation models.

3. Modify the experiment in Section 2.3 to include parabolic motion with rotation. The object should rotate along the Z-axis while having an initial horizontal linear velocity and being subjected to gravity acceleration in the $-Z$ direction. Design a program and complete the animated demonstration.

Chapter 3

Inertial Navigation and Integrated Navigation

This chapter introduces the most fundamental techniques of inertial navigation and integrated navigation to the readers. In practice, if you work in the field of traditional navigation, your basic task is to refine details at the level of integrated navigation. However, this book aims to introduce the fusion positioning knowledge of traditional inertial navigation and laser positioning at the beginning of the book.

This section will cover the basics of IMU and satellite positioning. We will discuss the **measurement model** and **noise model** of IMU, demonstrating its integration effect. We will find that estimating system state solely based on IMU is not realistic; they tend to diverge quickly (in terms of position) if you use cheap MEMS IMUs. Then we will implement a simple integrated navigation scheme using an error state Kalman filter. It is consistent with the principles of traditional integrated navigation but uses a manifold notation manner and does not introduce complex compensation parameters. You can think of it as an extremely simple integrated navigation scheme. They can achieve integrated navigation functions and can flexibly integrate with other data sources in subsequent chapters. We hope that through this approach, readers can clearly see the differences between traditional and modern theories, providing inspiration for readers in various research directions.

$$\delta x_{\text{pred}} = F \delta x$$

$$P_{\text{pred}} = F P F^T + Q$$

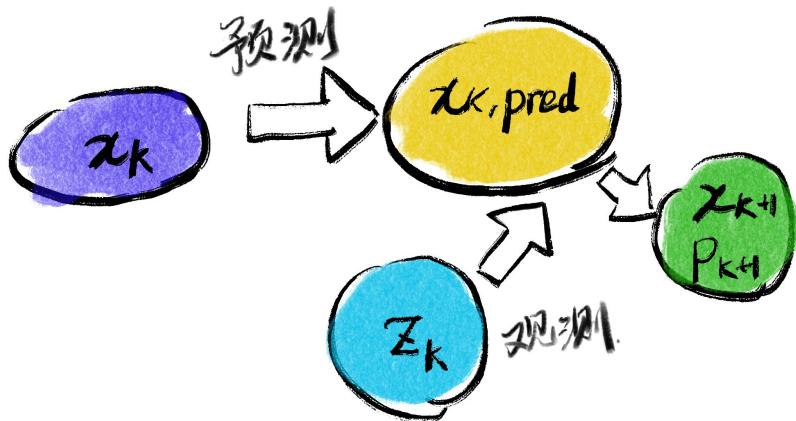
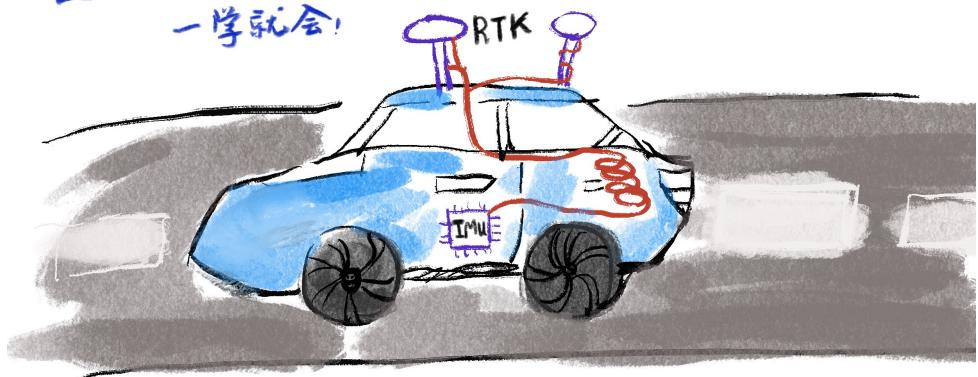
ESKF 又简单，又好用
一学就会！

$$K = P_{\text{pred}} H^T (H P_{\text{pred}} H^T + R)^{-1}$$

$$\delta x = K(z - h(x_{\text{pred}}))$$

$$x = x_{\text{pred}} + \delta x$$

$$P = (I - KH) P_{\text{pred}}$$



3.1 Kinematics of IMU Systems

3.1.1 Kinematics

The **Inertial Measurement Unit** (IMU) has become very common. IMUs can be found in most electronic devices: vehicles, smartphones, watches, helmets, and even soccer balls. They are small in size and, once installed in a device, can provide effective local motion estimation, enabling various interesting functionalities. In autonomous driving, inertial navigation devices are also fundamental positioning devices. The positioning provided by inertial navigation is basically independent of the external environment and other sensor data, exhibiting high versatility and reliability.

A typical six-axis IMU consists of a **gyroscope** and an **accelerometer**. Although they measure the inertia of objects, the means of implementation are diverse, ranging from low-cost MEMS (Micro-electromechanical systems) inertial navigation to expensive fiber-optic gyroscopes (Figure 3-2). Their goal is to accurately measure the inertia of objects. The goal of this book is not to introduce the types and working principles of IMUs themselves¹, but to examine their mathematical model properties from the perspective of fusion positioning and state estimation, and further introduce their applications in laser and vision systems.

IMUs are typically installed in a moving system. We infer the state of the object itself by measuring the inertia of the moving carrier. These inertia-related physical quantities are usually not directly position and rotation but rather their differentials. The gyroscope of an IMU can measure the object's **angular velocity**, while the accelerometer measures the object's **acceleration**. Internally, they can calculate angular velocity and acceleration based on other physical quantities such as force or time. However, from an external perspective, we only need to be concerned with whether their measurements of angular velocity and acceleration are accurate and the relationship between these quantities and the vehicle's position and orientation.

Based on the kinematics introduced earlier, we can simply write down the kinematic equations in continuous time²:

$$\dot{\mathbf{R}} = \mathbf{R}\boldsymbol{\omega}^\wedge, \quad \text{or } \dot{\mathbf{q}} = \frac{1}{2}\mathbf{q}\boldsymbol{\omega}, \quad (3.1a)$$

$$\dot{\mathbf{p}} = \mathbf{v}, \quad (3.1b)$$

$$\dot{\mathbf{v}} = \mathbf{a}. \quad (3.1c)$$

The rotational part can be represented either by a rotation matrix, as seen in Equation (2.73), or by quaternions, as seen in Equation (2.75). These physical quantities, when annotated with subscripts, should be denoted as \mathbf{R}_{wb} and \mathbf{p}_{wb} . Since \mathbf{p}_{wb} corresponds to the vehicle's world coordinates, after differentiation, it becomes the velocity and acceleration of the vehicle in the world coordinate system, denoted as \mathbf{v}_w and \mathbf{a}_w . This notation is intuitive, so subsequent text will omit the subscripts related to coordinate systems. Other materials may define variables such as $w\mathbf{v}_{wb}$ to distinguish between world-frame velocity and body-frame velocity, but this book uniformly uses quantities in the world frame, with special cases mentioned separately.

¹Readers interested in how IMUs measure angular velocity and acceleration can refer to books that focus more on the manufacturing and measurement principles of IMUs, such as [8, 9].

²The mathematical symbols in this book are kept simple. We try to avoid adding various subscripts and superscripts, while maintaining consistency throughout the text. However, most of the materials still tend to write out complete symbols with subscripts and superscripts [48]. This may make the formulae appear more complex, but readers should be able to discern the writing habits of different books.

The above equations assume that the world frame is fixed, akin to outer space or virtual space. When we do not consider the Earth's rotation, we can simply regard the surface the vehicle travels on as a fixed world coordinate system. In this case, the measured values of IMU, $\tilde{\omega}$ and $\tilde{\mathbf{a}}$, are the vehicle's own angular velocity and acceleration in the body frame³:

$$\tilde{\mathbf{a}} = \mathbf{R}^\top \mathbf{a}, \quad (3.2a)$$

$$\tilde{\omega} = \omega. \quad (3.2b)$$

Note that \mathbf{R}^\top with the subscript denotes \mathbf{R}_{bw} , which transforms quantities from the world frame to the body frame.

However, real vehicles and robots operate on the surface of the Earth. These systems are influenced by gravity, so we should include gravity in the system equations. In most IMU systems, we can ignore the disturbance caused by the Earth's rotation⁴, thus writing the IMU measurements as:

$$\tilde{\mathbf{a}} = \mathbf{R}^\top (\mathbf{a} - \mathbf{g}), \quad (3.3a)$$

$$\tilde{\omega} = \omega. \quad (3.3b)$$

Here, \mathbf{g} represents the gravity of the Earth. Of course, if measuring the object's acceleration in a zero-gravity environment, the gravity term would not appear.

Note that the sign of \mathbf{g} is related to the definition of the coordinate system. In our coordinate system, both the body frame and the world frame have the Z -axis pointing upwards, so \mathbf{g} typically takes the value $(0, 0, -9.8)^\top$. However, in some books, the Z -axis can be defined as pointing towards the center of the Earth, in which case the value of \mathbf{g} itself or the sign here might be reversed. According to the coordinate system definition in this book, the term in the measurement equation should be $\mathbf{a} - \mathbf{g}$.

To aid understanding, readers can also imagine a horizontally placed IMU (see Figure 3-1). If the IMU is stationary, since the measurement of object acceleration is actually obtained by measuring the force applied, the IMU should experience a supporting force in the opposite direction. Therefore, it should measure a gravity in the $-\mathbf{g}$ direction. If the IMU is flipped upside down, \mathbf{R}^\top changes, and a positive gravity \mathbf{g} would be measured. On the other hand, if the IMU is in free fall, the sensors themselves would not detect any external force influence, so $\mathbf{a} - \mathbf{g} = \mathbf{0}$, and the accelerometer should output a zero measurement.

Note that Equation (3.3) is written under the assumption of **no noise influence**. If one wants to design a simulation system, equations without noise models can be used. However, actual IMU measurements typically contain noise, so we need to consider the influence of noise.

3.1.2 Explanation of IMU Measurements

Here, we provide some explanations for the IMU measurement equations mentioned earlier, which might be areas where many students encounter issues in engineering practice.

In theory, according to Newton's second law, the force acting on an object is directly proportional to its acceleration, with the coefficient being the object's mass. For an object located in outer space or in free fall, unaffected by external forces, this is indeed the case.

³We need a notation to distinguish between **state variables** and **measurement values**. They can refer to the same physical quantity. However, state variables are estimable and variable, while measurement values are the readings of instruments and are constant. Later text generally uses variables with a tilde to express measurement values, while those without denote state variables.

⁴In some high-precision systems, IMUs can measure the Earth's rotation, but in platforms such as vehicles and drones, we usually choose to ignore these physical quantities.

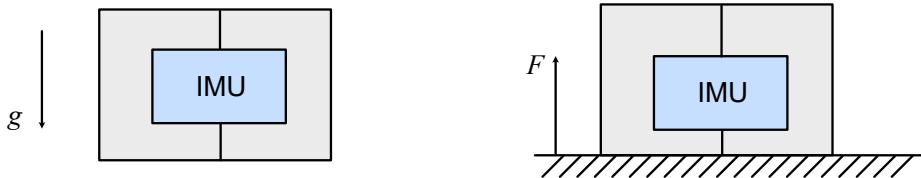


Figure 3-1: Illustration of IMU measurements: during free fall, the IMU does not detect any readings; when horizontally placed, the IMU measures gravity in the opposite direction through the supporting force.

We can also imagine an IMU using springs to measure force. In outer space or during free fall, the springs should be in a relaxed state, and no force should be detected. However, most of the time, we are concerned with real moving objects such as vehicles and robots, which mostly operate on the Earth's surface. Due to the Earth's gravitational force, these objects are naturally subjected to an external force \mathbf{g} . If an object is stationary on the Earth's surface, although the net force acting on it is zero, the IMU still naturally detects a supporting force in the opposite direction. At this time, although the object is not accelerating, the IMU can still read the opposite gravity. Hence, we have a $-\mathbf{g}$ term in the measurement equation. If we consider an IMU in space, we would remove this gravity term from the measurement equation. Alternatively, if the gravity at a particular location differs from elsewhere, we should adjust the value of \mathbf{g} . However, in some materials, gravity is not included in the measurement equation, and real-world IMUs may remove gravity when outputting data. Readers should be aware of such cases.

Furthermore, if the IMU is not placed at the center of the vehicle, when the vehicle rotates and moves, the IMU should also measure the centrifugal force, Coriolis force, and angular acceleration caused by the vehicle's rotation, reflected in the accelerometer readings. Some vehicles also experience various mechanical vibrations, such as suspension systems, moving parts of the vehicle (brushes, rollers, mechanical arms, etc.), which can also affect IMU readings. Therefore, the complete equation should include these terms. However, if all these small terms are included in the subsequent state estimation equations, the equations will become excessively long, which is not conducive to teaching. In practice, these readings themselves are small, and by ensuring that the IMU is installed as close to the vehicle's center as possible, we can avoid the problems caused by misalignment between the IMU and the vehicle's coordinate system. For these reasons, we will continue to use this simplified IMU measurement model in the subsequent discussions.

3.1.3 Noise Model in IMU Measurement Equations

In most systems, we consider the noise in an Inertial Measurement Unit (IMU) to consist of two parts: **measurement noise** and **bias**. Why do we do this? Due to various reasons, even when a vehicle is stationary, the output of angular velocity and acceleration from an IMU may not form a white noise with a mean of $\mathbf{0}$, but rather have some **offset**. This offset is caused by the electromechanical measuring devices within the IMU; some IMUs have small offsets while others may have larger ones. Additionally, this offset is also influenced by factors such as temperature and changes over time. Mathematically, we model it and consider **bias** as a state variable of the system, which undergoes random changes over time. However, readers should understand that this is a **mathematical model**, not the **essence**

of the system. We do not derive the variation relationship of bias from the mechanical or physical characteristics of the IMU, nor do we physically describe the relationship between IMU bias and temperature, even if such a relationship objectively exists. We simply **assume** that the mathematical model is such, and then observe if it significantly differs from the actual readings of the IMU device⁵. In most systems, such modeling relationships are sufficient. However, if not, we can also add various compensation parameters as needed to accurately describe the variation of bias.

Let the measurement noise of the gyroscope and accelerometer be denoted as η_g, η_a , and let the biases be denoted as $\mathbf{b}_g, \mathbf{b}_a$, where the subscript g represents the gyroscope and a represents the accelerometer. Then, these parameters manifest in the measurement equations as follows:

$$\tilde{\mathbf{a}} = \mathbf{R}^\top(\mathbf{a} - \mathbf{g}) + \mathbf{b}_a + \eta_a, \quad (3.4a)$$

$$\tilde{\omega} = \omega + \mathbf{b}_g + \eta_g. \quad (3.4b)$$

In continuous time, we consider the IMU measurement noise to be a zero-mean white Gaussian process with variances $\text{Cov}(\eta_g)$ and $\text{Cov}(\eta_a)$ ⁶. Additionally, we regard the biases as Wiener processes, also known as Brownian motion or random walk. These are common and classical stochastic processes.

A zero-mean white Gaussian process random variable $\mathbf{w}(t)$ with covariance Σ can be expressed as:

$$\mathbf{w}(t) \sim \mathcal{GP}(\mathbf{0}, \Sigma\delta(t - t')), \quad (3.5)$$

where Σ is termed the **energy spectral density matrix**, and δ denotes the Dirac delta function. The presence of the Dirac delta function enables us to easily derive the IMU measurement noise after discrete-time sampling from the continuous-time Gaussian process. For detailed derivations, please refer to the appendix of [49], and we will also discuss this in the subsequent sections when introducing the discrete-time model.

On the other hand, a typical random walk process for a bias \mathbf{b} can be modeled as:

$$\dot{\mathbf{b}}(t) = \eta_b(t), \quad (3.6)$$

where $\eta_b(t)$ is also a Gaussian process. Therefore, the random walks for \mathbf{b}_a and \mathbf{b}_g can be modeled as follows:

$$\dot{\mathbf{b}}_a(t) = \eta_{ba}(t) \sim \mathcal{GP}(\mathbf{0}, \text{Cov}(\mathbf{b}_a)\delta(t - t')), \quad (3.7a)$$

$$\dot{\mathbf{b}}_g(t) = \eta_{bg}(t) \sim \mathcal{GP}(\mathbf{0}, \text{Cov}(\mathbf{b}_g)\delta(t - t')). \quad (3.7b)$$

If readers are not familiar with stochastic processes, we can also provide an intuitive understanding. Since the covariance of a Gaussian process increases with time, the measurement values of an IMU itself become less accurate as the sampling time increases. Therefore, a higher sampling frequency IMU tends to have higher precision. Meanwhile, the bias part is described by Brownian motion, exhibiting a random walk behavior. In practice, we can think of an IMU's bias starting from some initial value and randomly moving irregularly around it. The larger the magnitude of this movement, the more unstable the bias is. Therefore, a better quality IMU should maintain its bias near the initial value without significant deviation.

Random walk is essentially a stochastic process whose derivative is a Gaussian process. From the perspective of an IMU, since we are concerned with measuring angular velocity and

⁵Therefore, readers should not assume that there objectively exists a physically measurable quantity termed bias inside the IMU, and then this quantity is superimposed on the measurement values.

⁶For basic information on Gaussian processes, readers can refer to Section 2.3 in [4].

acceleration, the bias part appears as a random walk. However, from a higher-level system perspective, angular velocity is the derivative of angle, and acceleration is the derivative of velocity. So, the IMU's measurement noise can also be interpreted as a **random walk of angles** and **random walk of velocities**. Therefore, when encountering the term "random walk," it's important not to only associate it with the bias part, but rather to consider the problem more holistically.

Please note that Gaussian processes and Brownian motion processes mentioned here are **mathematical models** of IMU measurement data. Mathematical models do not necessarily correspond exactly to the real world. Sometimes, mathematical models are a **simplification** of reality, which facilitate subsequent algorithmic computations. We should understand this concept. Later on, when we perform linear approximations and retain various first-order terms in many systems, it's based on this **simplification** mindset. The real measurement noise and bias of IMUs are influenced by many factors such as vehicle vibration, temperature, IMU's own forces, calibration, installation errors, and so on. Modeling them as two stochastic processes is more for the convenience of state estimation algorithm computation, rather than being a perfect, accurate modeling approach. This **simplify-then-compensate** approach is very common in practice.

3.1.4 Discrete-Time Noise Model for IMU

Although the noise equations for IMUs in continuous time are relatively complex, they simplify significantly in discrete time. In practice, IMUs sample the inertia of moving objects at fixed time intervals, so the data we obtain is always discrete. The complete derivation of the discrete-time model is cumbersome; interested readers can refer to [49]. Here, we only present the conclusions. Because IMU sensors sample at a fixed frequency, let's assume the sampling interval is Δt . Then, for the noise, the discrete measurement noises of the gyroscope and accelerometer can be simplified as⁷:

$$\boldsymbol{\eta}_g(k) \sim \mathcal{N}(0, \frac{1}{\Delta t} \text{Cov}(\boldsymbol{\eta}_g)), \quad (3.8a)$$

$$\boldsymbol{\eta}_a(k) \sim \mathcal{N}(0, \frac{1}{\Delta t} \text{Cov}(\boldsymbol{\eta}_a)). \quad (3.8b)$$

As for the bias part, it can be written as:

$$\mathbf{b}_g(k+1) - \mathbf{b}_g(k) \sim \mathcal{N}(\mathbf{0}, \Delta t \text{Cov}(\mathbf{b}_g)), \quad (3.9a)$$

$$\mathbf{b}_a(k+1) - \mathbf{b}_a(k) \sim \mathcal{N}(\mathbf{0}, \Delta t \text{Cov}(\mathbf{b}_a)). \quad (3.9b)$$

Therefore, in discrete-time systems (which is what we usually deal with), both noises are very easy to handle. In many system implementations, even the **covariance matrices** are not used to express the IMU measurement noise and bias random walk. Instead, they are simply represented as **diagonal matrices**, effectively ignoring the correlation between different axes. In programming, parameters such as σ_g, σ_a are often used to express the **standard deviations** of IMU noise, and parameters σ_{bg}, σ_{ba} are used to express the **standard deviations** of bias drift. In this case, the standard deviations of noise in discrete time should be written as:

$$\sigma_g(k) = \frac{1}{\sqrt{\Delta t}} \sigma_g, \quad \sigma_a(k) = \frac{1}{\sqrt{\Delta t}} \sigma_a, \quad (3.10a)$$

$$\sigma_{bg}(k) = \sqrt{\Delta t} \sigma_{bg}, \quad \sigma_{ba}(k) = \sqrt{\Delta t} \sigma_{ba}. \quad (3.10b)$$

⁷Some small terms from [49] are neglected.

When discussing filters and preintegration later on, we will use these symbols to configure the noise conditions of the IMU. Physically, in discrete time, the noise is directly added to the measured physical quantities, making it easy to determine their physical units. The bias in discrete time itself is added to the measured physical quantities, so they share the same units [50].

$$\sigma_g(k) \rightarrow \frac{\text{rad}}{s}, \quad \sigma_a(k) \rightarrow \frac{m}{s^2}, \quad \sigma_{bg}(k) \rightarrow \frac{\text{rad}}{s}, \quad \sigma_{ba}(k) \rightarrow \frac{m}{s^2}. \quad (3.11)$$

Continuous-time variances need to be multiplied or divided by a square root time unit to obtain discrete variances, so their physical units become:

$$\sigma_g \rightarrow \frac{\text{rad}}{\sqrt{s}}, \quad \sigma_a \rightarrow \frac{m}{s\sqrt{s}}, \quad \sigma_{bg} \rightarrow \frac{\text{rad}}{s\sqrt{s}}, \quad \sigma_{ba} \rightarrow \frac{m}{s^2\sqrt{s}}. \quad (3.12)$$

Note that units can be converted between similar units, such as radians to degrees, and seconds to minutes or hours, etc. Some materials may use the unit $\frac{1}{\Delta t}$ as Hz, so the physical units of the above variables can also be denoted as:

$$\sigma_g \rightarrow \frac{\text{rad}}{s\sqrt{\text{Hz}}}, \quad \sigma_a \rightarrow \frac{m}{s^2\sqrt{\text{Hz}}}, \quad \sigma_{bg} \rightarrow \frac{\text{rad}}{s^2\sqrt{\text{Hz}}}, \quad \sigma_{ba} \rightarrow \frac{m}{s^3\sqrt{\text{Hz}}}. \quad (3.13)$$

3.1.5 IMUs in Reality



Figure 3-2: Various IMU Products in Reality

Figure 3-2 displays examples of typical IMU products. Readers can directly purchase various IMU-related products in most markets, including standalone IMU sensors and integrated products. With the miniaturization of IMUs, many products are also integrated with IMUs internally at the factory. The most common example is our everyday smartphones, which commonly integrate low-cost MEMS IMU devices. In the field of robotics, sensors commonly used in applications such as LiDAR, cameras, etc., are also increasingly integrating ready-made IMUs. During the period of writing this book, many solid-state LiDARs (e.g., DJI Livox series, Ouster OS2), monocular and stereo cameras (e.g., Zed 2, MYNT Eye S) have already provided built-in IMUs as sources of inertial navigation data.

IMU products generally provide their own datasheets to illustrate their data accuracy, stability, and other metrics at the time of manufacture. These metrics can also be used to guide us in adjusting the weights of state estimation algorithms. Figure 3-3 shows a typical parameter configuration of an IMU (ADIS 16488). It can be seen that the main parameters relevant to our subsequent algorithms are as follows:

Table 1.

Parameter	Test Conditions/Comments	Min	Typ	Max	Unit
GYROSCOPES					
Dynamic Range		±450		±480	"/sec
Sensitivity	x_GYRO_OUT and x_GYRO_LOW (32-bit)		3.052×10^{-7}		"/sec/LSB
Repeatability ¹	-40°C ≤ T _A ≤ +70°C			±1	%
Sensitivity Temperature Coefficient	-40°C ≤ T _A ≤ +70°C, 1 σ		±35		ppm/"C
Misalignment	Axis-to-axis		±0.05		Degrees
	Axis-to-frame (package)		±1.0		Degrees
Nonlinearity	Best-fit straight line, FS = 450°/sec		0.01		% of FS
Bias Repeatability ^{1,2}	-40°C ≤ T _A ≤ +70°C, 1 σ		±0.2		"/sec
In-Run Bias Stability	1 σ		6.25		"/hr
Angular Random Walk	1 σ		0.3		"/hr
Bias Temperature Coefficient	-40°C ≤ T _A ≤ +70°C, 1 σ		±0.0025		"/sec/"C
Linear Acceleration Effect on Bias	Any axis, 1 σ (CONFIG[7] = 1)		0.009		"/sec/g
Output Noise	No filtering		0.16		"/sec rms
Rate Noise Density	f = 25 Hz, no filtering		0.0066		"/sec/Hz rms
3 dB Bandwidth		330			Hz
Sensor Resonant Frequency		18			kHz
ACCELEROMETERS					
Dynamic Range	Each axis	±18			g
Sensitivity	x_ACCL_OUT and x_ACCL_LOW (32-bit)		1.221×10^{-8}		g/LSB
Repeatability ¹	-40°C ≤ T _A ≤ +70°C			±0.5	%
Sensitivity Temperature Coefficient	-40°C ≤ T _A ≤ +70°C, 1 σ		±25		ppm/"C
Misalignment	Axis-to-axis		±0.035		Degrees
	Axis-to-frame (package)		±1.0		Degrees
Nonlinearity	Best-fit straight line, ±10 g		0.1		% of FS
	Best-fit straight line, ±18 g		0.5		% of FS
Bias Repeatability ^{1,2}	-40°C ≤ T _A ≤ +70°C, 1 σ		±16		mg
In-Run Bias Stability	1 σ		0.1		mg
Velocity Random Walk	1 σ		0.029		m/sec./hr
Bias Temperature Coefficient	-40°C ≤ T _A ≤ +85°C		±0.1		mg/"C
Output Noise	No filtering		1.5		mg rms
Noise Density	f = 25 Hz, no filtering		0.067		mg/Hz rms
3 dB Bandwidth		330			Hz
Sensor Resonant Frequency		5.5			kHz

Figure 3-3: Datasheet of Real IMUs

1. Measurement noise, which in the entire motion model is regarded as **angular random walk** and **velocity random walk**, corresponding to σ_g and σ_a in the **continuous-time** noise model. We can also simply refer to them as **gyroscope white noise** and **accelerometer white noise**. It can be observed that the specifications of this IMU are $0.66^\circ/\sqrt{\text{hour}}$ and $0.11\text{m/s}/\sqrt{\text{hour}}$. Intuitively, this can be understood as follows: assuming we have correctly identified the bias, if we integrate this IMU, the standard deviation of its integration error per hour should be 0.66° and 0.11m/s .
2. Bias instability variance, which is also known as σ_{bg} and σ_{ba} in the noise model. However, this quantity is usually not directly corresponding in the datasheet and is difficult to measure in practice. The datasheet often provides bias repeatability and bias stability during operation instead. When we first turn on the IMU, we can estimate the bias of the IMU in a static state. The variability of each bias change is described by bias repeatability. On the other hand, if other objective conditions remain unchanged, this boot-up bias will also change to some extent during operation. Its magnitude is described by **operational bias stability**, which in this datasheet is $14.5^\circ/\text{hour}$. Intuitively, under ideal conditions, we can assume that the bias of the IMU will remain within this range near the initial bias. However, in reality, IMUs often do not operate under constant temperature conditions, and their bias changes need to be estimated in real time. Overall, we can refer to this specification to set the magnitude of bias instability.⁸

⁸For detailed definitions of these two values, please refer to "GJB 585A-1998 Inertial Technology Terminology".

3.2 Trajectory Estimation Using IMU

Previously, we introduced how angular velocity and acceleration are measured in a motion system. This approach is a simulation-based perspective or a description of a known system. In other words, we can first assume what motion the object undergoes and then consider what kind of IMU measurements should be generated under this motion. However, the reality is the opposite. We can read the readings of IMU sensors, but we must infer the motion of the system based on the readings of IMU and other sensors, rather than directly obtaining various velocity and acceleration information of the system.

When there are many sensors, we integrate various sensor data for fusion-based positioning or SLAM, which is the main content of this book. Later, we will discuss the fusion of IMU with RTK, LiDAR, and other systems. In this section, we will first examine how to infer the motion state of the system when only IMU data is available. We will find that this approach is feasible, but a system with only IMU requires double integration of IMU readings, and the existence of measurement errors and biases will cause the state variables to drift quickly.

3.2.1 Short-Term Trajectory Estimation Using IMU Data

We have introduced the kinematic model of the IMU system itself in Section 3.1, while the measurement model of the IMU is introduced in Equation (3.4). Therefore, by directly substituting the measurement model into the kinematic equation and ignoring the influence of measurement noise, we can obtain the integration model in continuous time:

$$\dot{\mathbf{R}} = \mathbf{R}(\tilde{\boldsymbol{\omega}} - \mathbf{b}_g)^\wedge, \quad \text{or } \dot{\mathbf{q}} = \mathbf{q} \left[0, \frac{1}{2} (\tilde{\boldsymbol{\omega}} - \mathbf{b}_g) \right], \quad (3.14a)$$

$$\dot{\mathbf{p}} = \mathbf{v}, \quad (3.14b)$$

$$\dot{\mathbf{v}} = \mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a) + \mathbf{g}. \quad (3.14c)$$

Sometimes we also refer to $\mathbf{p}, \mathbf{v}, \mathbf{q}$ as the PVQ state. This equation can be integrated from time t to $t + \Delta t$ to derive the state at the next moment:

$$\mathbf{R}(t + \Delta t) = \mathbf{R}(t) \text{Exp}((\tilde{\boldsymbol{\omega}} - \mathbf{b}_g)\Delta t), \quad \text{or } \mathbf{q}(t + \Delta t) = \mathbf{q}(t) \left[1, \frac{1}{2} (\tilde{\boldsymbol{\omega}} - \mathbf{b}_g) \Delta t \right], \quad (3.15a)$$

$$\mathbf{p}(t + \Delta t) = \mathbf{p}(t) + \mathbf{v}\Delta t + \frac{1}{2} (\mathbf{R}(t)(\tilde{\mathbf{a}} - \mathbf{b}_a)) \Delta t^2 + \frac{1}{2} \mathbf{g}\Delta t^2, \quad (3.15b)$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{R}(t)(\tilde{\mathbf{a}} - \mathbf{b}_a)\Delta t + \mathbf{g}\Delta t. \quad (3.15c)$$

With this equation, we can use the state at one moment plus the IMU data at the next moment to infer the state at the next moment. This approach is generally referred to as **propagation**. From the perspective of numerical integration, it corresponds to the Euler method in numerical integration (Figure 3-4).

Different numerical integration methods vary, with the Euler method being the simplest. The rotation and translation of an object itself are continuous, while the IMU samples at fixed time intervals. During the sampling interval Δt , there are several approaches to handling the angular velocity and acceleration within this small time frame. The Euler method adopts the simplest approach: it assumes that during the time from t to $t + \Delta t$, the object's angular velocity is entirely equal to $\boldsymbol{\omega}(t)$, and the acceleration is equal to $\mathbf{a}(t)$. This is equivalent to

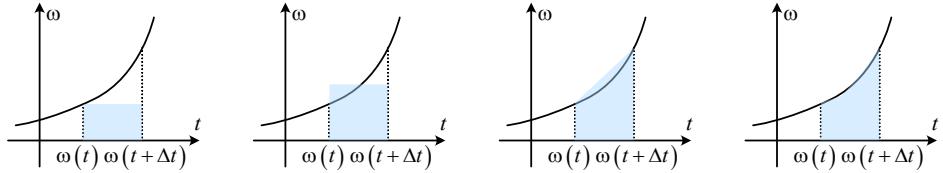


Figure 3-4: Schematic diagram of different integration methods. From left to right: start-point integration, mid-point integration, trapezoidal integration, true integration.

using the starting point of the interval as the function value for the integration rectangle in numerical integration. Other methods, such as the midpoint method, trapezoidal rule, and higher-order interpolation methods (most notably the Runge-Kutta method [51]), can also be employed. These methods are not overly complex in practice—they simply replace the measurements in the above equations with interpolated values $\tilde{\omega}, \tilde{\mathbf{a}}$ for the recursion. Theoretically, the midpoint and trapezoidal methods should be more accurate than the simplest Euler integration, while interpolation methods introduce additional computational overhead. Whether the improvement in accuracy justifies the increased computation depends on the specific application.

The above equations can also be accumulated further, such as from time i to time j . We only need to aggregate the IMU readings in between:

$$\mathbf{R}_j = \mathbf{R}_i \prod_{k=i}^{j-1} \text{Exp}((\tilde{\omega}_k - \mathbf{b}_{g,k}) \Delta t) \quad \text{or} \quad \mathbf{q}_j = \mathbf{q}_i \prod_{k=i}^{j-1} \left[1, \frac{1}{2} (\tilde{\omega}_k - \mathbf{b}_{g,k}) \Delta t \right], \quad (3.16a)$$

$$\mathbf{p}_j = \mathbf{p}_k + \sum_{k=i}^{j-1} \left[\mathbf{v}_k \Delta t + \frac{1}{2} \mathbf{g} \Delta t^2 \right] + \frac{1}{2} \sum_{k=i}^{j-1} \mathbf{R}_k (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,k}) \Delta t^2, \quad (3.16b)$$

$$\mathbf{v}_j = \mathbf{v}_i + \sum_{k=i}^{j-1} [\mathbf{R}_k (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,k}) \Delta t + \mathbf{g} \Delta t]. \quad (3.16c)$$

Note that we have not yet considered the influence of noise. In Chapter 4, we will further examine the impact of measurement noise and bias noise on IMU integration. Here, we focus only on its recursive form. Observe that the above equations are essentially **incremental**: each $\mathbf{R}_k, \mathbf{v}_k$ can serve as the computation result for the next time step. Therefore, in code implementation, when an IMU frame arrives, we can use the result from the previous frame to compute the recursion for the next frame, without waiting for all data to arrive before performing the accumulation as per the formula. For rotation, there is no fundamental difference between using quaternions or rotation matrices, though the rotation matrix representation is more concise mathematically and does not require frequent normalization. In this chapter, we retain both notations for readers to compare. However, in subsequent chapters, we will primarily use the rotation matrix representation.

3.2.2 Code Experiment for IMU Recursion

Below, we conduct an experiment to demonstrate how to use IMU data for trajectory propagation. Without external observations, we can only use Equation (3.16) to perform double

integration and obtain the position and attitude information of the moving object. This integration typically diverges rapidly, making IMU unsuitable for standalone dead reckoning. Our experiment will illustrate this.

Listing 3.1: ch3/imu_integration.h

```

1 class IMUIntegration {
2     public:
3         IMUIntegration(const Vec3d& gravity, const Vec3d& init_bg, const Vec3d& init_ba)
4             : gravity_(gravity), bg_(init_bg), ba_(init_ba) {}
5
6     text
7     // Add IMU reading
8     void AddIMU(const IMU& imu) {
9         double dt = imu.timestamp_ - timestamp_;
10        if (dt > 0 && dt < 0.1) {
11            // Assume IMU time interval is between 0 and 0.1
12            p_ = p_ + v_ * dt + 0.5 * gravity_ * dt * dt + 0.5 * (R_ * (imu.acce_ - ba_)) *
13                dt * dt;
14            v_ = v_ + R_ * (imu.acce_ - ba_) * dt + gravity_ * dt;
15            R_ = R_ * Sophus::SO3d::exp((imu.gyro_ - bg_) * dt);
16        }
17
18        // Update time
19        timestamp_ = imu.timestamp_;
20    }
21
22    /// Compose NavState
23    NavStated GetNavState() const { return NavStated(timestamp_, R_, p_, v_, bg_, ba_); }
24
25    SO3 GetR() const { return R_; }
26    Vec3d GetV() const { return v_; }
27    Vec3d GetP() const { return p_; }
28
29    private:
30        // Accumulated quantities
31        SO3 R_;
32        Vec3d v_ = Vec3d::Zero();
33        Vec3d p_ = Vec3d::Zero();
34
35        double timestamp_ = 0.0;
36
37        // Biases, set externally
38        Vec3d bg_ = Vec3d::Zero();
39        Vec3d ba_ = Vec3d::Zero();
40
41        Vec3d gravity_ = Vec3d(0, 0, -9.8); // Gravity
42    };

```

This function implements a simple IMU integrator that continuously reads IMU data and provides its integration results. We have prepared some sensor data for readers in data/ch3/ (which will also be used in later chapters). Readers can select any trajectory segment and run the following program to observe the IMU integration results.

Listing 3.2: ch3/run_imu_integration.cc

```

1 sad::TxtIO io(FLAGS_imu_txt_path);
2
3 // In this experiment, we assume biases are known
4 Vec3d gravity(0, 0, -9.8); // Gravity direction
5 Vec3d init_bg(0.000224886, -7.61038e-05, -0.000742259);
6 Vec3d init_ba(-0.165205, 0.0926887, 0.0058049);
7
8 IMUIntegration imu_integ(gravity, init_bg, init_ba);
9
10 sad::ui::PangolinWindow ui;
11 ui.Init();
12
13 /// Record results
14 auto save_result = [](std::ofstream& fout, double timestamp, const Sophus::SO3d& R,
15                      const Vec3d& v,

```

```

15 const Vec3d& p) {
16     auto save_vec3 = [] (std::ofstream& fout, const Vec3d& v) { fout << v[0] << " " << v
17         [1] << " " << v[2] << " "; };
18     auto save_quat = [] (std::ofstream& fout, const Quatd& q) {
19         fout << q.w() << " " << q.x() << " " << q.y() << " " << q.z() << " ";
20     };
21
22     text
23     fout << std::setprecision(18) << timestamp << " " << std::setprecision(9);
24     save_vec3(fout, p);
25     save_quat(fout, R.unit_quaternion());
26     save_vec3(fout, v);
27     fout << std::endl;
28
29     std::ofstream fout("./data/ch3/state.txt");
30     io.SetIMUProcessFunc([&imu_integ, &save_result, &fout, &ui](const sad::IMU& imu) {
31         imu_integ.AddIMU(imu);
32         save_result(fout, imu.timestamp_, imu_integ.GetR(), imu_integ.GetV(), imu_integ.GetP()
33             ());
34         ui.UpdateNavState(imu_integ.GetNavState());
35         usleep(1e2);
36     }).Go();
37
38 ui.Quit();

```

This program stores the integration results in ch3/state.txt. Note that this book frequently uses C++ lambda functions for flexible function calls. Here, TxtIO reads and parses sensor data from text files, then executes callbacks for various sensors as predefined. Since programs in different chapters process this sensor data differently, the callback portion is implemented using lambda functions.

Now, execute this program:

Listing 3.3: Terminal input:

```
bin/run_imu_integration
```

It will display the vehicle's real-time position in the UI. However, this program will quickly diverge, and the vehicle will disappear off the screen edge. After the program ends, we can run the plotting script to visualize the trajectory:

Listing 3.4: Terminal input:

```
python3 scripts/plot_ch3_state.py data/ch3/state.txt
```

The vehicle motion in the UI is shown in Figure 3-5, and the trajectory plotting result is shown in Figure 3-6. We observe that the attitude, expressed in quaternions, remains relatively stable overall. The data source is an onboard IMU, with the q_y , q_z components of the quaternion attitude remaining near zero. However, the displacement quickly diverges. Due to the lack of external observations, the velocity state soon becomes uncontrollable, far exceeding the vehicle's actual speed (this is a low-speed vehicle under 25 km/h), causing the position estimate to diverge to an extremely large value. Readers can also try other datasets, all of which will diverge rapidly.

Later, we will discuss how to fuse IMU data with other sensor data to obtain more accurate state estimates. In traditional navigation, the primary fusion method for IMU is integration with satellite navigation, forming what is known as a GINS (GNSS-INS) system.

3.3 Satellite Navigation

Satellite navigation (Global Navigation Satellite System, GNSS) is another primary source of positioning information for outdoor vehicles. While the internal principles of satellite

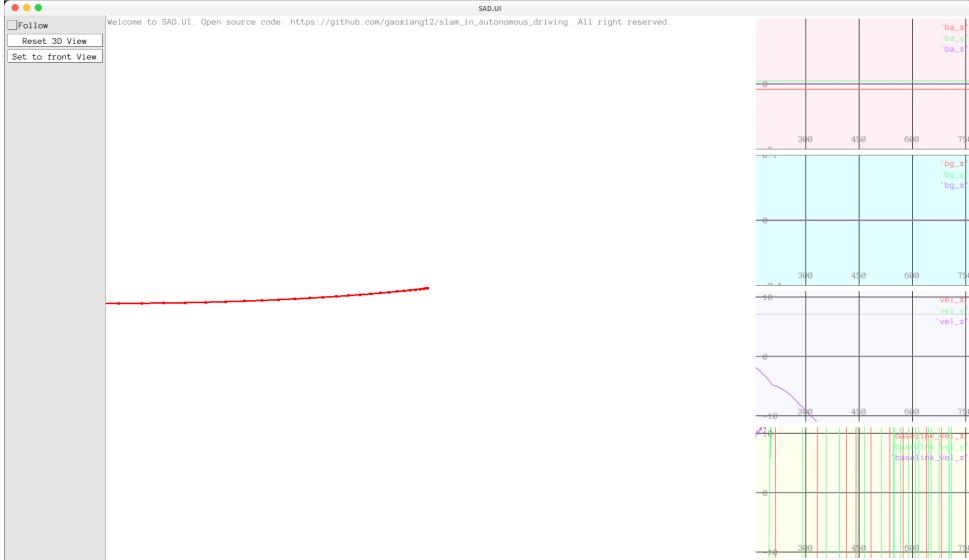


Figure 3-5: IMU integration results displayed in the UI

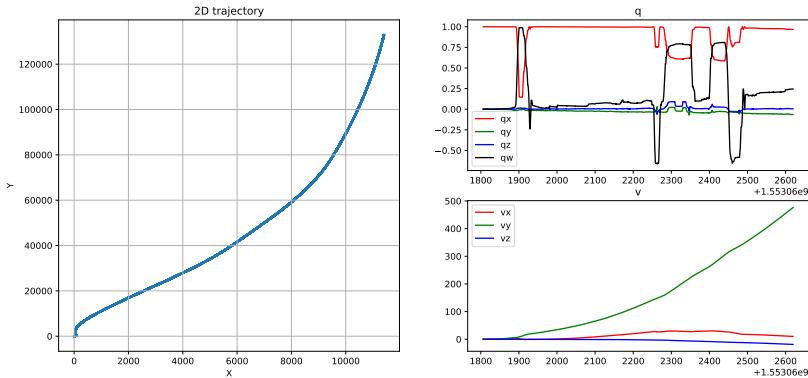


Figure 3-6: Results of IMU integration

navigation are highly complex, its output is remarkably simple. That said, if we dissect practical systems down to their fundamental principles, we quickly become overwhelmed by engineering details. Even seemingly simple systems like microcontrollers can appear extremely intricate when explained at the circuit level. Therefore, we will not introduce GNSS systems starting from rocket or satellite launches but instead approach the topic from a more practical perspective: What information can satellite navigation provide to an autonomous vehicle?

The answer to this question is straightforward. During normal operation, satellite navigation can provide all the positioning information a vehicle needs, including its position, attitude, velocity, and other physical quantities. Readers might then ask: Can autonomous driving be achieved using satellite positioning alone? To answer this, we must consider several factors:

1. Does the positioning accuracy of GNSS meet the requirements?
2. Is the positioning frequency of GNSS sufficient for downstream applications?
3. How reliable is GNSS positioning? Can it be used in all weather conditions and locations?

In reality, GNSS encompasses many subtypes, and there is no uniform answer to these questions. Some GNSS methods offer high accuracy but require the object to remain stationary for a period (typically over ten minutes). Others provide better dynamic positioning but require the prior setup of one or more base stations. Almost all GNSS systems face the challenge of being "weather-dependent"—their stability is influenced by the environment, structures, obstructions, and even the day's weather. This makes satellite navigation in the autonomous driving industry often accurate enough but difficult to control in terms of reliability.

Classification and Providers of GNSS Broadly speaking, GNSS systems determine their position by measuring the distance to satellites orbiting the Earth, primarily through time-of-flight calculations. A satellite signal carries a transmission timestamp when emitted, and the GNSS receiver records its arrival time. By comparing these timestamps, the distance to each satellite can be estimated. The main differences among GNSS systems and measurement methods lie in how they reduce timing errors. From this perspective, GNSS can essentially be viewed as a high-precision timekeeping system.



Figure 3-7: Various GNSS/RTK receivers.

Currently, satellite signals received worldwide primarily come from four systems:

1. GPS (Global Positioning System, USA)
2. BDS (Beidou Navigation Satellite System, China)
3. GLONASS (Russia)
4. GALILEO (European Union)

Each system has deployed 20 to 30 satellites in space. Systems that directly use these satellites for positioning are typically called standalone GNSS⁹. Most ground-based GNSS receivers can access signals from multiple satellite systems, choosing either a single source or combining multiple sources. Standalone GNSS typically provides positioning accuracy within a few meters. Most mobile devices use standalone GNSS for navigation (Figure 3-7).

The final accuracy of satellite positioning is affected by many factors, from the satellite's electromagnetic signals to transmission delays and receiver clock errors. Various correction

⁹GNSS systems are continuously evolving. Due to historical reasons, the term "GPS" was commonly used in the past, but most literature now distinguishes between GPS and GNSS.

techniques have been developed to mitigate these errors, such as PPP (Precise Point Positioning) and RTK (Real-Time Kinematic), which continue to improve positioning accuracy. These technologies are extensive, but this book focuses on the user perspective rather than the development side.

By 2023, GNSS positioning has evolved from 10-meter accuracy to real-time centimeter-level precision, becoming widely accessible. In China, providers like Qianxun, Hezhong, and Xunteng offer extensive and affordable satellite positioning services, increasingly integrated into consumer devices like smartphones and vehicles.

For autonomous vehicles, the most commonly used satellite positioning technologies include:

Standalone GNSS: Traditional meter-level satellite positioning, cost-effective and widely used in phones and car systems. While sufficient for road-level navigation¹⁰, its accuracy struggles to distinguish between parallel roads (e.g., highways vs. service roads).

RTK Positioning: To address signal transmission errors, differential positioning was developed, where a ground-based reference station with known coordinates corrects the vehicle's GNSS signals. RTK, based on carrier-phase differential technology, uses one or more base stations to provide real-time corrected positioning.

Many companies now offer RTK services for autonomous driving, deploying extensive base station networks and using 4G/5G to deliver real-time vehicle positions. In favorable conditions, RTK alone can enable autonomous driving. Some providers combine RTK with inertial navigation for enhanced reliability.

3.3.1 Practical RTK Installation and Data Reception

As the saying goes, "seeing is believing." Below we present some actual RTK data (the same applies to GNSS data) used in autonomous driving applications.

RTK receivers are typically installed on vehicle rooftops in disc-shaped configurations (often affectionately called "mushroom heads"). A single mushroom head can provide precise satellite positioning. When two receivers are configured in a **dual-antenna** setup, the vehicle's real-time heading can be calculated based on the positional difference between the two antennas.

Figure 3-8 demonstrates a practical dual-antenna RTK installation on a vehicle. With the vehicle body removed, we can clearly observe the installation positions of various sensors. This vehicle features two RTK mushroom heads - one at the front and another at the rear - aligned along the vehicle's longitudinal axis. Other vehicles may employ horizontal or lateral dual-antenna configurations.

In dual-antenna systems, we generally follow these conventions:

1. One antenna is designated as the **primary** antenna, representing the vehicle's position.
2. The **secondary** antenna's position is subtracted from the primary's to obtain the directional vector between them, from which the vehicle's heading angle (primarily yaw) can be derived.
3. For left-right installations, the left antenna is typically designated as primary.
4. For front-rear installations, the rear antenna is usually primary.

However, these are merely conventional designations - the two antennas are fundamentally identical and their roles could be reversed.

Several key considerations for dual-antenna systems include:

¹⁰ Autonomous driving typically requires lane-level navigation, which identifies the vehicle's specific lane, offering greater stability than road-level navigation.



Figure 3-8: A dual-antenna RTK installation configuration from Baidu Apollo’s development kit.

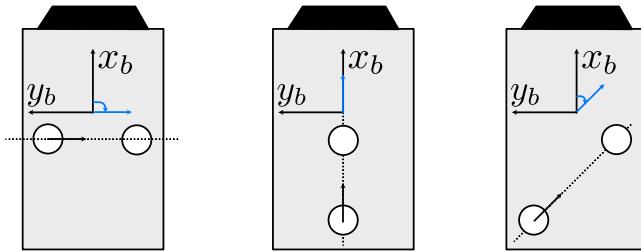


Figure 3-9: Various dual-antenna RTK installation configurations: left-right, front-rear, and diagonal.

- To accurately measure vehicle angles, the two antennas should be placed as far apart as possible¹¹.
- The distance between dual antennas is called the RTK **baseline**¹².
- As a sensor system, the RTK’s installation position on the vehicle can be described by extrinsic parameters represented by a transformation matrix.
- Since dual antennas are typically coplanar, the rotational extrinsic parameters can be described by a single angle called the **installation yaw angle** (antenna angle), while the translational parameters are called **installation offset** (antenna position). These parameters are usually determined during structural design.

Satellite positioning systems universally output object positions in latitude and longitude coordinates. This output format is tied to Earth’s fixed coordinate systems, unlike LiDAR

¹¹Both antennas’ positions are subject to measurement noise. Greater separation reduces the noise’s impact on angle calculations.

¹²The term “baseline” is widely used across different sensors (e.g., RTK baseline, stereo camera baseline). Readers need not concern themselves with potential connections between these usages.

or visual positioning systems where coordinate frames can be arbitrarily defined. Additionally, dual-antenna RTK angle measurements follow conventional definitions. We will first introduce common global coordinate systems before processing actual RTK data.

3.3.2 Common World Coordinate Systems

Various world coordinate systems are widely used in the physical world. Below we briefly introduce their definitions.

Geographic Coordinate System

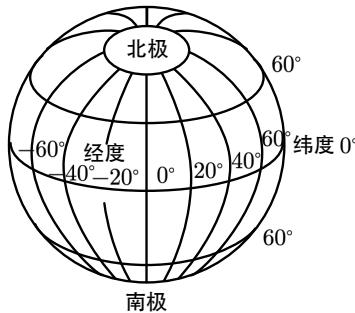


Figure 3-10: Schematic diagram of latitude-longitude coordinate system

The most common coordinate system on Earth is the **latitude-longitude** coordinate system, also known as the **Geographic Coordinate System** (see Figure 3-10). Combined with altitude, it forms the latitude-longitude-altitude (LLA) coordinate system. Latitude and longitude divide the Earth's surface uniformly: longitude ranges 180 degrees east and west from the prime meridian, while latitude ranges 90 degrees north and south from the equator. Both values are expressed in degrees or radians. Altitude can be either elevation above sea level or geodetic height relative to a reference surface.

While intuitive and globally applicable (being the default for many mapping systems), latitude-longitude coordinates become cumbersome for autonomous driving applications at city-scale or smaller areas. Building-level coordinates typically require 8-9 decimal places, and the nonlinear conversion to metric units (where 1 degree longitude corresponds to 0 meters at the poles but over 100 km at the equator) motivates the use of local coordinate systems.

UTM Coordinate System

The Universal Transverse Mercator (UTM) system projects the WGS84 ellipsoid Earth onto a transverse cylinder, divided into 60 longitudinal and 20 latitudinal zones with numeric and alphabetic labels respectively. While roughly uniform in angular distribution, metric distances vary due to Earth's curvature, with significant distortion excluding polar regions beyond 80° latitude.

Within each zone:

- Easting coordinates reference a central meridian (assigned $x=500,000m$), increasing eastward
- Northing coordinates measure distance from the equator

- The resulting **ENU** (East-North-Up) system follows right-hand convention
- Alternative **NED** (North-East-Down) configurations also exist

UTM advantages include metric compatibility with sensors, though zone boundaries require special handling. Note:

1. A 0.9996 scale factor corrects projection distortion
2. ENU/NED systems have opposite Z-axis orientations affecting angle definitions
3. Maximum easting span is 667 km per zone

This metric local frame facilitates autonomous vehicle operations while maintaining global reference.

3.3.3 Display of RTK Measurements

Many RTK and integrated navigation manufacturers can output coordinates in user-selected coordinate systems, with the most basic being the latitude-longitude coordinates measured by the RTK receiver. Below we demonstrate how to convert RTK's latitude-longitude coordinates to metric UTM coordinates while using a dual-antenna configuration to determine the vehicle's heading angle. Here we disregard the vehicle's pitch and roll, treating them as zero. Thus, although the vehicle outputs **4-DOF coordinates**, under the assumption of zero pitch and roll, the RTK output can also be considered as a 6-DOF pose transformation, i.e., an SE(3) pose.

We continue using the data from the previous section. This data consists of IMU, RTK, and wheel speed measurements located in the ‘data/ch3/’ directory. Readers can open these files with a text editor, with the format as follows:

Listing 3.5: Example data file

```

1 GNSS 1571900872.47168827 30.0011840411666668 117.97859182983332 305.98748779296875
2 330.04779999999995 1
3 ODOM 1571900872.50085688 0 0
4 IMU 1571900872.56527948 -0.000740019602845583204 -0.000471238898038460995
5 6.98131700797720067e-06 0.36251916166666645 -0.060801229999999908
6 9.8213599750000002

```

Each line in the file represents a measurement, with the prefix ”GNSS”, ”IMU”, or ”ODOM” indicating the record type. For GNSS measurements, each line contains: timestamp, latitude, longitude, altitude, heading angle, and heading validity flag. For IMU or wheel speed measurements, the readings from the accelerometer, gyroscope, and wheel speed sensors are provided. The GNSS positioning here is provided by the Qianxun FindCM solution, with a nominal accuracy of 2 cm in fixed solution mode.

Since the algorithm for converting latitude-longitude to UTM coordinates is complex and not the focus of this book, we use an open-source conversion method from the ‘third-party/utm_convert‘ directory¹³. We’ve added a wrapper function to conveniently compute the SE(3) pose corresponding to GNSS measurements. The conversion code is as follows:

Listing 3.6: ch3/utm_convert.cc

```

1 bool LatLon2UTM(const Vec2d& latlon, UTMCoordinate& utm_coor) {
2     long zone = 0;
3     char char_north = 0;
4     long ret = Convert_Geodetic_To_UTM(latlon[0] * math::kDEG2RAD, latlon[1] * math::
      kDEG2RAD, &zone, &char_north,

```

¹³See: <https://github.com/hobu/mgrs>

```

5   &utm_coor.xy_[0], &utm_coor.xy_[1]);
6   utm_coor.zone_ = (int)zone;
7   utm_coor.north_ = char_north == 'N';
8
9   return ret == 0;
10}
11
12 bool ConvertGps2UTM(GNSS& gps_msg, const Vec2d& antenna_pos, const double&
13   antenna_angle, const Vec3d& map_origin) {
14   /// Convert latitude-longitude-altitude to UTM
15   UTMCoordinate utm_rtk;
16   if (!LatLon2UTM(gps_msg.lat_lon_alt_.head<2>(), utm_rtk)) {
17     return false;
18   }
19   utm_rtk.z_ = gps_msg.lat_lon_alt_[2];
20
21   /// Convert GPS heading to radians
22   double heading = 0;
23   if (gps_msg.heading_valid_) {
24     heading = (90 - gps_msg.heading_) * math::kDEG2RAD; // Convert from NED to ENU
25   }
26
27   /// Transform from TWG to TWB
28   SE3 TBG(SO3::rotZ(antenna_angle * math::kDEG2RAD), Vec3d(antenna_pos[0], antenna_pos
29   [1], 0));
30   SE3 TGB = TBG.inverse();
31
32   /// Subtract map origin if specified
33   double x = utm_rtk.xy_[0] - map_origin[0];
34   double y = utm_rtk.xy_[1] - map_origin[1];
35   double z = utm_rtk.z_ - map_origin[2];
36   SE3 TWG(SO3::rotZ(heading), Vec3d(x, y, z));
37   SE3 TWB = TWG * TGB;
38
39   gps_msg.utm_valid_ = true;
40   gps_msg.utm_xy_[0] = TWB.translation().x();
41   gps_msg.utm_xy_[1] = TWB.translation().y();
42   gps_msg.utm_z_ = TWB.translation().z();
43
44   if (gps_msg.heading_valid_) {
45     // Construct pose with rotation
46     gps_msg.utm_pose_ = TWB;
47   } else {
48     // Construct SE3 with only translation
49     // Note: When installation offset exists, the actual vehicle pose cannot be
50     // derived
51     gps_msg.utm_pose_ = SE3(SO3(), TWB.translation());
52   }
53
54   return true;
55}

```

The latitude-longitude to UTM conversion is handled by the library function, while we need to transform the UTM coordinates from the GNSS frame to the vehicle's observed pose, considering the RTK's extrinsic parameters. The dual-antenna RTK installation used in this section is shown in Figure 3-11, where the blue axes x_B, y_B, O_B represent the vehicle body frame, and the red axes x_G, y_G, O_G represent the GNSS receiver frame.

Mathematically, we can treat the RTK's UTM coordinate reading as \mathbf{T}_{WG} , where W represents the world frame and G represents the GNSS receiver frame. To facilitate subsequent fusion localization, we transform it to \mathbf{T}_{WB} , where B is the vehicle body frame. Thus, the extrinsic parameters between the GNSS receiver and the vehicle can be described by either \mathbf{T}_{GB} or \mathbf{T}_{BG} .

In the calibration parameters, we specify the installation offset \mathbf{a}_t as the vector from O_B to O_G expressed in the B frame, which essentially corresponds to the translational component of \mathbf{T}_{BG} . Simultaneously, the installation angle a_θ is defined as the rotation angle from the x -axis of the B frame to the x -axis of the G frame. Substituting \mathbf{a}_t and a_θ into \mathbf{T}_{BG} , we obtain:

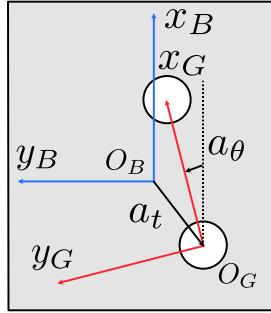


Figure 3-11: RTK installation configuration used in this book’s example. Subscript G denotes the GNSS receiver frame, while B denotes the vehicle body frame. The primary antenna is located at the right rear of the vehicle, and the secondary antenna at the front left.

$$\mathbf{T}_{BG} = \begin{bmatrix} \mathbf{R}_Z(a_\theta) & \mathbf{a}_t \\ \mathbf{0}^\top & 1 \end{bmatrix}, \quad (3.17)$$

where \mathbf{R}_Z represents the rotation matrix about the Z -axis.

It’s important to note that this definition of **installation angle** and **installation offset** is intuitive but not unique. The specific definition should prioritize operational convenience. These parameters could alternatively be defined in the opposite direction, as long as calibration personnel can measure them easily. We typically align mathematical notation with practical conventions rather than forcing real-world operations to conform to mathematical definitions. While calibration personnel might struggle to directly measure \mathbf{R}_{BG} or \mathbf{t}_{BG} , they can easily measure the connection between two points or the angle between two lines.

The transformation matrix \mathbf{T}_{WB} from the vehicle body frame to the world frame can be derived from RTK measurements and its extrinsic parameters:

$$\mathbf{T}_{WB} = \mathbf{T}_{WG}\mathbf{T}_{GB}. \quad (3.18)$$

Expanding the rotation and translation components yields:

$$\mathbf{R}_{WB} = \mathbf{R}_{WG}\mathbf{R}_{GB}, \quad \mathbf{t}_{WB} = \mathbf{R}_{WG}\mathbf{t}_{GB} + \mathbf{t}_{WG}. \quad (3.19)$$

An important consideration: even with known RTK extrinsic parameters \mathbf{t}_{GB} , determining the vehicle coordinates \mathbf{t}_{WB} requires knowledge of the RTK orientation \mathbf{R}_{WG} . In a single-antenna configuration with non-zero installation offset \mathbf{t}_{GB} , **the vehicle’s world coordinates cannot be precisely determined when the vehicle orientation is unknown**. However, in state estimation algorithms where the vehicle attitude \mathbf{R}_{WB} is estimated, we can use $\mathbf{R}_{WB}\mathbf{R}_{BG}$ as the current \mathbf{R}_{WG} .

Additionally, the conversion program handles the transformation between **ENU** (East-North-Up) and **NED** (North-East-Down) coordinate systems. Different RTK manufacturers may output angles according to their predefined schemes, potentially causing inconsistencies in angle definitions. This book uses UTM coordinates with east and north as XY axes (ENU frame), while the RTK manufacturer outputs NED coordinates. The former uses east as zero degrees, while the latter uses north as zero degrees with opposite rotation direction. Therefore, an azimuth angle h in the NED frame converts to angle h' in the ENU frame as:

$$h' = \pi/2 - h. \quad (3.20)$$

The code above implements this angle conversion.

We then write a program to convert GNSS readings from the data file into poses and write them to an output file. Readers can use Python scripts to plot the entire GNSS trajectory. Simultaneously, we display the RTK poses in real-time graphical interfaces, allowing immediate visualization of current position and orientation.

3.3.4 RTK Measurement Visualization

Many RTK and integrated navigation manufacturers can output coordinates in user-selected coordinate systems, with the most basic output being the latitude-longitude coordinates measured by the RTK receiver. Below we demonstrate how to convert RTK's latitude-longitude coordinates to metric UTM coordinates while using a dual-antenna configuration to determine the vehicle's heading angle.

Listing 3.7: ch3/process_gnss.cc

```

1 DEFINE_string(txt_path, "./data/ch3/10.txt", "Input data file path");
2
3 // Parameters specific to the provided dataset
4 DEFINE_double(antenna_angle, 12.06, "RTK antenna installation angle (degrees)");
5 DEFINE_double(antenna_pox_x, -0.17, "RTK antenna installation offset X");
6 DEFINE_double(antenna_pox_y, -0.20, "RTK antenna installation offset Y");
7
8 /**
9 * This program demonstrates GNSS data processing
10 * We convert raw GNSS readings into 6-DOF poses for subsequent processing
11 * Requires UTM conversion, RTK antenna extrinsics, and coordinate transformation
12 */
13
14 * Results are saved to file and visualized using Python scripts
15 */
16
17 int main(int argc, char** argv) {
18     sad::TxtIO io(fLS::FLAGS_txt_path);
19
20     std::ofstream fout("./data/ch3/gnss_output.txt");
21     Vec2d antenna_pos(FLAGS_antenna_pox_x, FLAGS_antenna_pox_y);
22
23     auto save_result = [] (std::ofstream& fout, double timestamp, const SE3& pose) {
24         auto save_vec3 = [] (std::ofstream& fout, const Vec3d& v) {
25             fout << v[0] << " " << v[1] << " " << v[2] << " ";
26         };
27         auto save_quat = [] (std::ofstream& fout, const Quatd& q) {
28             fout << q.w() << " " << q.x() << " " << q.y() << " " << q.z() << " ";
29         };
30
31         fout << std::setprecision(18) << timestamp << " " << std::setprecision(9);
32         save_vec3(fout, pose.translation());
33         save_quat(fout, pose.unit_quaternion());
34         fout << std::endl;
35     };
36
37     std::shared_ptr<sad::ui::PangolinWindow> ui = nullptr;
38     if (FLAGS_with_ui) {
39         ui = std::make_shared<sad::ui::PangolinWindow>();
40         ui->Init();
41     }
42
43     bool first_gnss_set = false;
44     Vec3d origin = Vec3d::Zero();
45     io.SetGNSSProcessFunc([](const sad::GNSS& gnss) {
46         sad::GNSS gnss_out = gnss;
47         if (sad::ConvertGps2UTM(gnss_out, antenna_pos, FLAGS_antenna_angle)) {
48             if (first_gnss_set == false) {
49                 origin = gnss_out.utm_pose_.translation();
50                 first_gnss_set = true;
51             }
52             gnss_out.utm_pose_.translation() -= origin;
53         }
54         save_result(fout, gnss_out.unix_time_, gnss_out.utm_pose_);
55     });

```

```

54     ui->UpdateNavState(
55     sad::NavStated(gnss_out.unix_time_, gnss_out.utm_pose_.so3(),
56     gnss_out.utm_pose_.translation()));
57
58     usleep(1e4);
59 }
60 }).Go();
61
62 if (ui) {
63     while (!ui->ShouldQuit()) {
64         usleep(1e5);
65     }
66     ui->Quit();
67 }
68
69 return 0;
70 }
```

The program converts RTK readings to UTM poses, removes the origin offset, writes to a text file, and displays in UI. To run:

Listing 3.8: Terminal command:

```
bin/process_gnss --txt_path ./data/ch3/10.txt
```

The converted 6-DOF coordinates are saved in data/ch3/gnss_output.txt. Visualization scripts:

Listing 3.9: Terminal command:

```
python3 scripts/plot_ch3_gnss_2d.py ./data/ch3/gnss_output.txt
```

Figure 3-12 shows 2D and 3D trajectory plots. Observations:

- RTK provides excellent horizontal trajectory accuracy
- Vertical measurements show noticeable noise and errors
- The example shows good GNSS conditions; other datasets include challenging segments

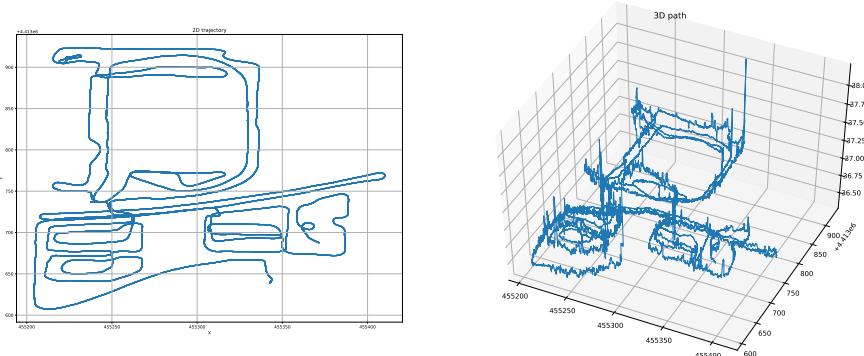


Figure 3-12: 2D and 3D views of GNSS trajectory

The real-time display (Figure 3-13) shows:

- Proper vehicle orientation (X-forward, Y-left, Z-up) when RTK heading is valid

- Noticeable instability in angle measurements compared to position
- Position jitter during heading outages due to incomplete pose solution

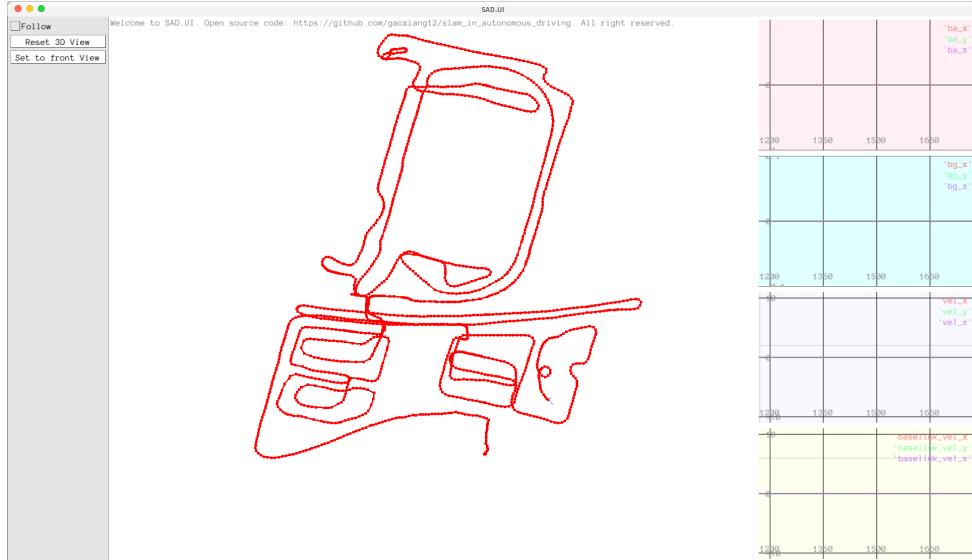


Figure 3-13: Real-time GNSS trajectory visualization

Note: This section only uses RTK-provided position and heading. Some RTK/INS systems output additional IMU data (angular/linear velocity), but we focus on fundamental principles here.

3.4 Implementing Integrated Navigation Using Error-State Kalman Filter

The RTK device provides us with a somewhat unstable pose observation source. We can directly treat it as an observation input for a positioning filter. In this section, we will combine RTK with IMU using an extended Kalman filter to form a traditional integrated navigation algorithm for subsequent algorithm comparisons. Strictly speaking, what we present to readers is the **Error-State Kalman Filter** (ESKF). The ESKF finds extensive applications ranging from GINS integrated navigation to visual SLAM[52–54] and extrinsic calibration[55, 56].

Why do we need ESKF? What's the difference between ESKF and EKF? We'll start from the motivation level. This progressive approach from simple to complex can also help readers better understand the development process of various algorithms.

3.4.1 Mathematical Derivation of ESKF

We have previously introduced the observation model of IMU devices. Now we need to treat IMU as the motion model and GNSS observations as the measurement model to derive the complete filter. This is not particularly difficult.

We define the state variables as:

$$\mathbf{x} = [\mathbf{p}, \mathbf{v}, \mathbf{R}, \mathbf{b}_g, \mathbf{b}_a, \mathbf{g}]^\top, \quad (3.21)$$

where all variables default to subscript $(\cdot)_{WB}$, with \mathbf{p} representing translation, \mathbf{v} velocity, \mathbf{R} rotation, \mathbf{b}_g , \mathbf{b}_a biases, and \mathbf{g} gravity. According to the kinematic equations (3.1) we introduced earlier and substituting IMU measurements, the continuous-time motion equations for the state variables are:

$$\dot{\mathbf{p}} = \mathbf{v}, \quad (3.22a)$$

$$\dot{\mathbf{v}} = \mathbf{R}(\ddot{\mathbf{a}} - \mathbf{b}_a - \boldsymbol{\eta}_a) + \mathbf{g}, \quad (3.22b)$$

$$\dot{\mathbf{R}} = \mathbf{R}(\ddot{\boldsymbol{\omega}} - \mathbf{b}_g - \boldsymbol{\eta}_g)^{\wedge}, \quad (3.22c)$$

$$\dot{\mathbf{b}}_g = \boldsymbol{\eta}_{bg}, \quad (3.22d)$$

$$\dot{\mathbf{b}}_a = \boldsymbol{\eta}_{ba}, \quad (3.22e)$$

$$\dot{\mathbf{g}} = \mathbf{0}. \quad (3.22f)$$

To predict the covariance in the EKF prediction step, we need to linearize these equations. Theoretically, the discrete-time linearized form is:

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k) + \mathbf{F}\mathbf{dx} + \mathbf{w}, \quad (3.23)$$

where $\mathbf{F} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}|_{\mathbf{x}_k}$ is the coefficient matrix. However, we encounter a practical problem: on one hand, \mathbf{F} needs to compute the derivative of the rotation matrix \mathbf{R} relative to some perturbation, and without introducing tensors, we cannot express the derivative of a matrix with respect to a vector. Traditional algorithms often compromise by using Euler angles or four scalars of quaternions as state variables[57], but this approach cannot elegantly utilize methods on manifolds. On the other hand, if we consider integrating inertial navigation systems with satellite navigation systems, the translation variables in \mathbf{x} should use a global coordinate system. This makes the numerical values in \mathbf{x} very large, potentially exceeding the effective digit range of floating-point numbers in some scenarios, leading to common computational failures like "large number swallowing small number" in numerical calculations[58].

This leads us to consider: can we avoid directly using \mathbf{x} and \mathbf{P} to express the mean and covariance of the state when deriving the motion and observation equations? Can we use the **update quantity** from the original Kalman filter to derive these two equations? Recall the observation part of the Kalman filter:

$$\mathbf{x}_k = \mathbf{x}_{k,\text{pred}} + \mathbf{K}_k \underbrace{(\mathbf{z}_k - \mathbf{H}_k \mathbf{x}_{k,\text{pred}})}_{\text{update quantity}}. \quad (3.24)$$

In the context of manifolds, the update quantity on the right side should be a vector in the tangent space, and the addition in the middle should be the generalized addition of the manifold and the exponential map of the tangent space. However, we can also treat the update quantity (or error state) as the state variable of the filter to derive the motion and observation models. This leads to the **Error-State Kalman Filter**. Furthermore, not just for translation and rotation, we express all states in terms of error states, which is the typical approach of ESKF.

The ESKF is widely used in many traditional and modern systems, serving both as a filter for integrated navigation and for implementing complex systems like LIO and VIO[59–61]. Compared to traditional KF, the advantages of ESKF can be summarized as follows[62]:

1. In handling rotation, ESKF state variables can use minimal parameter representation, i.e., using three-dimensional variables to express rotation increments. These variables lie in the tangent space, which is a vector space. Traditional KF requires quaternions (4D) or higher-dimensional variables (rotation matrices, 9D) to express the state, or must use representations with singularities (Euler angles).

2. ESKF always operates near the origin, far from singular points, making it more numerically stable and avoiding issues where linearization approximations become inadequate due to operating too far from the working point.
3. The ESKF state quantities are small, making their second-order variables relatively negligible. Most Jacobian matrices become very simple under small quantities, sometimes even replaceable by identity matrices.
4. The kinematics of error states are also smaller compared to the original state variables (kinematics of small quantities), allowing us to return the update part to the original state variables.

In ESKF, we typically refer to the original state variables as **nominal state variables** and the state variables in ESKF as **error state variables**. The sum of nominal and error state variables is called the **true value**. By handling noise in the error state variables, we can consider the equations for nominal state variables as noise-free. This approach may seem complex initially, but separating the noise makes the nominal state equations more concise. The filter itself primarily needs to consider how the error state evolves, is observed, and finally filtered, with minimal relation to the nominal state.

The ESKF workflow is as follows: When IMU measurement data arrives, we integrate it into the nominal state variables. Since this approach doesn't account for noise, the results naturally drift quickly, so we treat the error component as error variables. During motion, the nominal state propagates with IMU data, while the error state grows due to Gaussian noise. At this point, the mean and covariance of ESKF's error state describe the specific magnitude of error state expansion (treated as a Gaussian distribution)¹⁴. Additionally, ESKF's update process relies on sensor observations beyond IMU. During updates, we use sensor data to update the **posterior** mean and covariance of the error state. We can then **merge** this error into the nominal state variables and reset ESKF to zero, completing one predict-update cycle.

Let's derive the two processes of ESKF. We define the true state of ESKF as: $\mathbf{x}_t = [\mathbf{p}_t, \mathbf{v}_t, \mathbf{R}_t, \mathbf{b}_{at}, \mathbf{b}_{gt}, \mathbf{g}_t]^\top$. The subscript t denotes true, i.e., the true state. This state changes over time and can be denoted as $\mathbf{x}_t(t)$. In continuous time, we record IMU readings as $\tilde{\omega}$, $\tilde{\mathbf{a}}$, allowing us to write the relationship between state variable derivatives and observations:

$$\dot{\mathbf{p}}_t = \mathbf{v}_t, \quad (3.25a)$$

$$\dot{\mathbf{v}}_t = \mathbf{R}_t(\tilde{\mathbf{a}} - \mathbf{b}_{at} - \boldsymbol{\eta}_a) + \mathbf{g}_t, \quad (3.25b)$$

$$\dot{\mathbf{R}}_t = \mathbf{R}_t (\tilde{\boldsymbol{\omega}} - \mathbf{b}_{gt} - \boldsymbol{\eta}_g)^\wedge, \quad (3.25c)$$

$$\dot{\mathbf{b}}_{gt} = \boldsymbol{\eta}_{bg}, \quad (3.25d)$$

$$\dot{\mathbf{b}}_{at} = \boldsymbol{\eta}_{ba}, \quad (3.25e)$$

$$\dot{\mathbf{g}}_t = \mathbf{0}. \quad (3.25f)$$

This is consistent with Equation (3.22) mentioned earlier. Note that including gravity \mathbf{g} mainly facilitates determining the IMU's initial attitude. If we don't express gravity in the state equation, we must determine the IMU's initial orientation $\mathbf{R}(0)$ beforehand to perform subsequent calculations. In this case, the IMU's attitude is described relative to the initial horizontal plane. By explicitly expressing gravity, we can set the IMU's initial attitude as the identity matrix $\mathbf{R} = \mathbf{I}$, treating the gravity direction as a measure of the IMU's current

¹⁴However, as we'll see later, since all noise terms are zero-mean white noise, the mean of the error state remains zero in the motion equations, only the covariance increases.

attitude relative to the horizontal plane. Both methods are feasible, but separately expressing gravity makes initial attitude representation simpler and increases some linearity[63].

If we organize observations and noise into a vector, we can express the above in matrix form. However, such a matrix would contain many zero entries without significant simplification over the current form, so we'll continue using these expanded equations. Next, we derive the error state equations. First, define the error state variables as:

$$\mathbf{p}_t = \mathbf{p} + \delta\mathbf{p}, \quad (3.26a)$$

$$\mathbf{v}_t = \mathbf{v} + \delta\mathbf{v}, \quad (3.26b)$$

$$\mathbf{R}_t = \mathbf{R}\delta\mathbf{R} \quad \text{or} \quad \mathbf{q}_t = \mathbf{q}\delta\mathbf{q}, \quad (3.26c)$$

$$\mathbf{b}_{gt} = \mathbf{b}_g + \delta\mathbf{b}_g, \quad (3.26d)$$

$$\mathbf{b}_{at} = \mathbf{b}_a + \delta\mathbf{b}_a, \quad (3.26e)$$

$$\mathbf{g}_t = \mathbf{g} + \delta\mathbf{g}. \quad (3.26f)$$

The terms without subscripts are the **nominal state variables** in Equation (3.25). The kinematic equations for nominal state variables are identical to the true values, except they **ignore noise** (since noise is considered in the error state equations). For the rotation part, $\delta\mathbf{R}$ can be represented by its Lie algebra $\text{Exp}(\delta\theta)$, requiring Equation (c) to be expressed in exponential form.

For error state Equations (a,d,e,f), taking time derivatives on both sides easily yields corresponding derivative expressions:

$$\dot{\delta\mathbf{p}} = \delta\mathbf{v}, \quad (3.27a)$$

$$\dot{\delta\mathbf{b}}_g = \boldsymbol{\eta}_{bg}, \quad (3.27b)$$

$$\dot{\delta\mathbf{b}}_a = \boldsymbol{\eta}_{ba}, \quad (3.27c)$$

$$\dot{\delta\mathbf{g}} = \mathbf{0}. \quad (3.27d)$$

Equations (b) and (c) are slightly more complex due to their relationship with $\delta\mathbf{R}$, and we provide their separate derivation below.

Error State Rotation Term

Taking the time derivative of both sides of Equation (3.26)(c) yields:

$$\begin{aligned} \dot{\mathbf{R}}_t &= \dot{\mathbf{R}}\text{Exp}(\delta\theta) + \mathbf{R}\dot{\text{Exp}}(\delta\theta), \\ &\stackrel{3.25(c)}{=} \mathbf{R}_t (\tilde{\boldsymbol{\omega}} - \mathbf{b}_{gt} - \boldsymbol{\eta}_g)^{\wedge}. \end{aligned} \quad (3.28)$$

Note that the term $\dot{\text{Exp}}(\delta\theta)$ on the right side satisfies:

$$\dot{\text{Exp}}(\delta\theta) = \text{Exp}(\delta\theta)\delta\dot{\theta}^{\wedge}. \quad (3.29)$$

Thus, the first expression in (3.28) can be written as:

$$\dot{\mathbf{R}}\text{Exp}(\delta\theta) + \mathbf{R}\dot{\text{Exp}}(\delta\theta) = \mathbf{R}(\tilde{\boldsymbol{\omega}} - \mathbf{b}_g)^{\wedge}\text{Exp}(\delta\theta) + \mathbf{R}\text{Exp}(\delta\theta)\delta\dot{\theta}^{\wedge}. \quad (3.30)$$

While the second expression becomes:

$$\mathbf{R}_t (\tilde{\boldsymbol{\omega}} - \mathbf{b}_{gt} - \boldsymbol{\eta}_g)^{\wedge} = \mathbf{R}\text{Exp}(\delta\theta) (\tilde{\boldsymbol{\omega}} - \mathbf{b}_{gt} - \boldsymbol{\eta}_g)^{\wedge}. \quad (3.31)$$

By comparing these two expressions, moving $\dot{\theta}^\wedge$ to one side, canceling \mathbf{R} on both sides, and organizing similar terms, we obtain:

$$\text{Exp}(\delta\theta)\delta\dot{\theta}^\wedge = \text{Exp}(\delta\theta)(\tilde{\omega} - \mathbf{b}_{gt} - \boldsymbol{\eta}_g)^\wedge - (\tilde{\omega} - \mathbf{b}_g)^\wedge \text{Exp}(\delta\theta). \quad (3.32)$$

Noting that $\text{Exp}(\delta\theta)$ itself is an $\text{SO}(3)$ matrix, we utilize the adjoint property of $\text{SO}(3)$:

$$\boldsymbol{\phi}^\wedge \mathbf{R} = \mathbf{R}(\mathbf{R}^\top \boldsymbol{\phi})^\wedge \quad (3.33)$$

to interchange $\text{Exp}(\delta\theta)$ above:

$$\begin{aligned} \text{Exp}(\delta\theta)\delta\dot{\theta}^\wedge &= \text{Exp}(\delta\theta)(\tilde{\omega} - \mathbf{b}_{gt} - \boldsymbol{\eta}_g)^\wedge - \text{Exp}(\delta\theta)(\text{Exp}(-\delta\theta)(\tilde{\omega} - \mathbf{b}_g))^\wedge \\ &= \text{Exp}(\delta\theta)[(\tilde{\omega} - \mathbf{b}_{gt} - \boldsymbol{\eta}_g)^\wedge - (\text{Exp}(-\delta\theta)(\tilde{\omega} - \mathbf{b}_g))^\wedge] \\ &\approx \text{Exp}(\delta\theta)[(\tilde{\omega} - \mathbf{b}_{gt} - \boldsymbol{\eta}_g)^\wedge - ((\mathbf{I} - \delta\theta^\wedge)(\tilde{\omega} - \mathbf{b}_g))^\wedge] \quad (3.34) \\ &= \text{Exp}(\delta\theta)[\mathbf{b}_g - \mathbf{b}_{gt} - \boldsymbol{\eta}_g + \delta\theta^\wedge \tilde{\omega} - \delta\theta^\wedge \mathbf{b}_g]^\wedge \\ &= \text{Exp}(\delta\theta)[(-\tilde{\omega} + \mathbf{b}_g)^\wedge \delta\theta - \delta\mathbf{b}_g - \boldsymbol{\eta}_g]^\wedge. \end{aligned}$$

After canceling the coefficients on the left side, we get:

$$\delta\dot{\theta} \approx -(\tilde{\omega} - \mathbf{b}_g)^\wedge \delta\theta - \delta\mathbf{b}_g - \boldsymbol{\eta}_g. \quad (3.35)$$

The \approx in this equation comes from the expansion of $\text{Exp}(-\delta\theta)$. If we ignore second-order small quantities of $\delta\theta$, the equation can also be written with an equality sign.

Velocity Term of the Error State

Next, we consider the error form of Equation (3.26)(b). Similarly, by taking the time derivative of both sides, we can obtain the expression for $\delta\dot{\mathbf{v}}$. The left-hand side of the equation is:

$$\begin{aligned} \dot{\mathbf{v}}_t &= \mathbf{R}_t(\tilde{\mathbf{a}} - \mathbf{b}_{at} - \boldsymbol{\eta}_a) + \mathbf{g}_t \\ &= \mathbf{R}\text{Exp}(\delta\theta)(\tilde{\mathbf{a}} - \mathbf{b}_a - \delta\mathbf{b}_a - \boldsymbol{\eta}_a) + \mathbf{g} + \delta\mathbf{g} \\ &\approx \mathbf{R}(\mathbf{I} + \delta\theta^\wedge)(\tilde{\mathbf{a}} - \mathbf{b}_a - \delta\mathbf{b}_a - \boldsymbol{\eta}_a) + \mathbf{g} + \delta\mathbf{g} \quad (3.36) \\ &\approx \mathbf{R}\tilde{\mathbf{a}} - \mathbf{R}\mathbf{b}_a - \mathbf{R}\delta\mathbf{b}_a - \mathbf{R}\boldsymbol{\eta}_a + \mathbf{R}\delta\theta^\wedge \tilde{\mathbf{a}} - \mathbf{R}\delta\theta^\wedge \mathbf{b}_a + \mathbf{g} + \delta\mathbf{g} \\ &= \mathbf{R}\tilde{\mathbf{a}} - \mathbf{R}\mathbf{b}_a - \mathbf{R}\delta\mathbf{b}_a - \mathbf{R}\boldsymbol{\eta}_a - \mathbf{R}\tilde{\mathbf{a}}^\wedge \delta\theta + \mathbf{R}\mathbf{b}_a^\wedge \delta\theta + \mathbf{g} + \delta\mathbf{g}. \end{aligned}$$

When moving from the third line to the fourth line, it is necessary to neglect the second-order small quantities resulting from the multiplication of $\delta\theta^\wedge$ with $\delta\mathbf{b}_a$ and $\boldsymbol{\eta}_a$. The transition from the fourth line to the fifth line utilizes the property that the cross-product symbol changes sign when the order is swapped. On the other hand, the right-hand side of the equation is:

$$\dot{\mathbf{v}} + \delta\dot{\mathbf{v}} = \mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a) + \mathbf{g} + \delta\dot{\mathbf{v}}. \quad (3.37)$$

Since the two expressions above are equal, we obtain:

$$\delta\dot{\mathbf{v}} = -\mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a)^\wedge \delta\theta - \mathbf{R}\delta\mathbf{b}_a - \mathbf{R}\boldsymbol{\eta}_a + \delta\mathbf{g}. \quad (3.38)$$

Thus, we have derived the kinematic model for $\delta\mathbf{v}$. It is worth noting that since $\boldsymbol{\eta}_a$ is a zero-mean white noise, multiplying it by any rotation matrix still results in a zero-mean white noise. Moreover, because $\mathbf{R}^\top \mathbf{R} = \mathbf{I}$, it is straightforward to show that its covariance

matrix remains unchanged (left as an exercise). Therefore, the above equation can also be simplified as:

$$\delta\dot{\mathbf{v}} = -\mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a)^\wedge \delta\boldsymbol{\theta} - \mathbf{R}\delta\mathbf{b}_a - \boldsymbol{\eta}_a + \delta\mathbf{g}. \quad (3.39)$$

At this point, we can summarize the kinematic equations for the error variables as follows:

$$\delta\dot{\mathbf{p}} = \delta\mathbf{v}, \quad (3.40a)$$

$$\delta\dot{\mathbf{v}} = -\mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a)^\wedge \delta\boldsymbol{\theta} - \mathbf{R}\delta\mathbf{b}_a - \boldsymbol{\eta}_a + \delta\mathbf{g}, \quad (3.40b)$$

$$\delta\dot{\boldsymbol{\theta}} = -(\tilde{\boldsymbol{\omega}} - \mathbf{b}_g)^\wedge \delta\boldsymbol{\theta} - \delta\mathbf{b}_g - \boldsymbol{\eta}_g, \quad (3.40c)$$

$$\delta\dot{\mathbf{b}}_g = \boldsymbol{\eta}_{bg}, \quad (3.40d)$$

$$\delta\dot{\mathbf{b}}_a = \boldsymbol{\eta}_{ba}, \quad (3.40e)$$

$$\delta\dot{\mathbf{g}} = \mathbf{0}. \quad (3.40f)$$

3.4.2 Discrete-Time ESKF Kinematic Equations

Deriving the discrete-time state equations from the continuous-time state equations is straightforward. Let us directly present them. The discrete-time kinematic equations for the **nominal state variables** can be written as:

$$\mathbf{p}(t + \Delta t) = \mathbf{p}(t) + \mathbf{v}\Delta t + \frac{1}{2}(\mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a))\Delta t^2 + \frac{1}{2}\mathbf{g}\Delta t^2, \quad (3.41a)$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a)\Delta t + \mathbf{g}\Delta t, \quad (3.41b)$$

$$\mathbf{R}(t + \Delta t) = \mathbf{R}(t)\text{Exp}((\tilde{\boldsymbol{\omega}} - \mathbf{b}_g)\Delta t), \quad (3.41c)$$

$$\mathbf{b}_g(t + \Delta t) = \mathbf{b}_g(t), \quad (3.41d)$$

$$\mathbf{b}_a(t + \Delta t) = \mathbf{b}_a(t), \quad (3.41e)$$

$$\mathbf{g}(t + \Delta t) = \mathbf{g}(t). \quad (3.41f)$$

This equation is obtained by simply adding the bias terms and gravity term to Equation (3.15). Note that the third line is essentially the integration formula for angular velocity. The discrete form of the **error state** is very similar to that of the **nominal state**, but special attention must be paid to the angular velocity part:

$$\delta\mathbf{p}(t + \Delta t) = \delta\mathbf{p} + \delta\mathbf{v}\Delta t, \quad (3.42a)$$

$$\delta\mathbf{v}(t + \Delta t) = \delta\mathbf{v} + (-\mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a)^\wedge \delta\boldsymbol{\theta} - \mathbf{R}\delta\mathbf{b}_a + \delta\mathbf{g})\Delta t - \boldsymbol{\eta}_v, \quad (3.42b)$$

$$\delta\boldsymbol{\theta}(t + \Delta t) = \text{Exp}(-(\tilde{\boldsymbol{\omega}} - \mathbf{b}_g)\Delta t)\delta\boldsymbol{\theta} - \delta\mathbf{b}_g\Delta t - \boldsymbol{\eta}_\theta, \quad (3.42c)$$

$$\delta\mathbf{b}_g(t + \Delta t) = \delta\mathbf{b}_g + \boldsymbol{\eta}_{bg}, \quad (3.42d)$$

$$\delta\mathbf{b}_a(t + \Delta t) = \delta\mathbf{b}_a + \boldsymbol{\eta}_{ba}, \quad (3.42e)$$

$$\delta\mathbf{g}(t + \Delta t) = \delta\mathbf{g}. \quad (3.42f)$$

Notes:

1. For simplicity, we omit the (t) in parentheses on the right-hand side of the equations.
2. For the integration of the rotation part, we can treat Equation (3.40)(c) as a differential equation for $\delta\boldsymbol{\theta}$ and solve it. The solution process is similar to integrating angular velocity.

3. The noise terms do not participate in the recursion and must be treated separately as part of the noise. Continuous-time noise terms can be regarded as the power spectral density of a random process, while discrete-time noise variables are the random variables we commonly encounter. The standard deviations of these noise random variables can be written as follows:

$$\sigma(\boldsymbol{\eta}_v) = \Delta t \sigma_a(k), \quad \sigma(\boldsymbol{\eta}_\theta) = \Delta t \sigma_g(k), \quad \sigma(\boldsymbol{\eta}_{bg}) = \sqrt{\Delta t} \sigma_{bg}, \quad \sigma(\boldsymbol{\eta}_{ba}) = \sqrt{\Delta t} \sigma_{ba}, \quad (3.43)$$

where the Δt in the first two equations arises from the integration relationship, and the last two equations are consistent with (3.10).

At this point, we have described the process of IMU propagation in the ESKF, corresponding to the state equation in the Kalman filter. To ensure the convergence of the filter, external observations are needed to correct the Kalman filter, which is the so-called integrated navigation. Of course, there are many methods for integrated navigation, ranging from traditional EKF to the ESKF introduced in this section, as well as the pre-integration and graph optimization techniques to be discussed in later chapters, all of which can be applied to integrated navigation [64]. In this section, we take the fusion of GNSS observations as an example to demonstrate how to integrate these observation data into the ESKF to form a convergent Kalman filter. In the application chapters at the end of this book, we will also introduce filter solutions that fuse LiDAR point clouds or LiDAR positioning data.

3.4.3 Motion Process of ESKF

Based on the above discussion, we can formulate the motion process of ESKF. The discrete-time motion equation for the error state variable $\delta \mathbf{x}$ has been given in Eq. (3.42), which can be expressed as:

$$\delta \mathbf{x}_{k+1} = \mathbf{f}(\delta \mathbf{x}_k) + \mathbf{w}, \quad \mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}), \quad (3.44)$$

where \mathbf{w} is the noise term. According to the previous definition, the noise covariance matrix \mathbf{Q} should be:

$$\mathbf{Q} = \text{diag}(\mathbf{0}_3, \text{Cov}(\boldsymbol{\eta}_v), \text{Cov}(\boldsymbol{\eta}_\theta), \text{Cov}(\boldsymbol{\eta}_{bg}), \text{Cov}(\boldsymbol{\eta}_{ba}), \mathbf{0}_3), \quad (3.45)$$

where the zero matrices at the beginning and end are due to the fact that the position and gravity error equations themselves contain no noise.

To maintain consistency with EKF notation, we compute the linearized form of the motion equation:

$$\delta \mathbf{x}(t + \Delta t) = \underbrace{\mathbf{f}(\delta \mathbf{x}(t))}_{=0} + \mathbf{F} \delta \mathbf{x} + \mathbf{w}, \quad (3.46)$$

where \mathbf{F} is the state transition Jacobian matrix. Since Eq. (3.42) is already explicitly linearized, we can directly extract the coefficient matrix (note the variable order):

$$\mathbf{F} = \begin{bmatrix} \mathbf{I} & \mathbf{I}\Delta t & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & -\mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a)^\wedge \Delta t & \mathbf{0} & -\mathbf{R}\Delta t & \mathbf{I}\Delta t \\ \mathbf{0} & \mathbf{0} & \text{Exp}(-(\tilde{\boldsymbol{\omega}} - \boldsymbol{\omega}_g)\Delta t) & -\mathbf{I}\Delta t & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix}. \quad (3.47)$$

On this basis, we implement the prediction process of ESKF. The prediction process includes:

- Prediction of the nominal state (IMU integration)
- Prediction of the error state

$$\delta\mathbf{x}_{\text{pred}} = \mathbf{F}\delta\mathbf{x}, \quad (3.48\text{a})$$

$$\mathbf{P}_{\text{pred}} = \mathbf{F}\mathbf{P}\mathbf{F}^{\top} + \mathbf{Q}. \quad (3.48\text{b})$$

However, since the error state of ESKF is reset to $\delta\mathbf{x} = \mathbf{0}$ after each update, the mean part of the motion equation (i.e., Eq. (3.48)(a)) is not particularly meaningful. The covariance part describes the distribution of the entire error estimate. Intuitively, the noise covariance \mathbf{Q} added in the motion equation can be interpreted as an **increase** in uncertainty during the prediction process.

3.4.4 Update Process of ESKF

The previous section described the motion process of ESKF. Now we consider the update process. Assuming an abstract sensor can generate observations of the state variables with an observation equation \mathbf{h} , we can write:

$$\mathbf{z} = \mathbf{h}(\mathbf{x}) + \mathbf{v}, \quad \mathbf{v} \sim \mathcal{N}(\mathbf{0}, \mathbf{V}), \quad (3.49)$$

where \mathbf{z} is the observation data, \mathbf{v} is the observation noise, and \mathbf{V} is the noise covariance matrix¹⁵.

In traditional EKF, we can directly linearize the observation equation to compute the Jacobian matrix of the observation equation with respect to the state variables, then update the Kalman filter. In ESKF, since we have estimates of both the nominal state \mathbf{x} and the error state $\delta\mathbf{x}$, and we want to update the error state, we need to compute the Jacobian matrix of the observation equation with respect to the error state:

$$\mathbf{H} = \left. \frac{\partial \mathbf{h}}{\partial \delta\mathbf{x}} \right|_{\mathbf{x}_{\text{pred}}}, \quad (3.50)$$

We then compute the Kalman gain and perform the error state update:

$$\mathbf{K} = \mathbf{P}_{\text{pred}} \mathbf{H}^{\top} (\mathbf{H} \mathbf{P}_{\text{pred}} \mathbf{H}^{\top} + \mathbf{V})^{-1}, \quad (3.51\text{a})$$

$$\delta\mathbf{x} = \mathbf{K}(\mathbf{z} - \mathbf{h}(\mathbf{x}_{\text{pred}})), \quad (3.51\text{b})$$

$$\mathbf{x} = \mathbf{x}_{\text{pred}} + \delta\mathbf{x}, \quad (3.51\text{c})$$

$$\mathbf{P} = (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P}_{\text{pred}}. \quad (3.51\text{d})$$

Here \mathbf{K} is the Kalman gain, \mathbf{P}_{pred} is the predicted covariance matrix, and \mathbf{P} is the updated covariance matrix.

Most observation data are measurements of the nominal state¹⁶. In this case, \mathbf{H} can be computed using the chain rule:

$$\mathbf{H} = \frac{\partial \mathbf{h}}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \delta\mathbf{x}}, \quad (3.52)$$

¹⁵We use a different symbol here since \mathbf{R} is already used in the state variables.

¹⁶However, in some cases we can directly derive observations of the error state, which would simplify the subsequent derivation. Both GNSS and LiDAR observations discussed later will use direct observation of error states.

The first term requires linearization of the observation equation. For the second term, based on our definition of state variables, we obtain:

$$\frac{\partial \mathbf{x}}{\partial \delta \mathbf{x}} = \text{diag} \left(\mathbf{I}_3, \mathbf{I}_3, \frac{\partial \text{Log}(\mathbf{R}(\text{Exp}(\delta \boldsymbol{\theta})))}{\partial \delta \boldsymbol{\theta}}, \mathbf{I}_3, \mathbf{I}_3, \mathbf{I}_3 \right). \quad (3.53)$$

While other terms are straightforward, the rotation part requires special attention since $\delta \boldsymbol{\theta}$ is defined as right multiplication of \mathbf{R} . Using the right BCH formula:

$$\frac{\partial \text{Log}(\mathbf{R}(\text{Exp}(\delta \boldsymbol{\theta})))}{\partial \delta \boldsymbol{\theta}} = \mathbf{J}_r^{-1}(\mathbf{R}). \quad (3.54)$$

Finally, we could add subscript k to each variable to indicate the time step, but this is unnecessary as the above equations already clearly express their meanings. These formulas can also be derived using quaternion representation, with similar forms but more complex details (see [6]). This book presents only the derivation using SO(3) and its Lie algebra.

3.4.5 Post-Processing of ESKF Error State

After completing the prediction and update processes, we have corrected the estimation of the error state. The next step is to incorporate the error state into the nominal state and then reset the ESKF. The incorporation can be simply expressed as:

$$\mathbf{p}_{k+1} = \mathbf{p}_k + \delta \mathbf{p}_k, \quad (3.55a)$$

$$\mathbf{v}_{k+1} = \mathbf{v}_k + \delta \mathbf{v}_k, \quad (3.55b)$$

$$\mathbf{R}_{k+1} = \mathbf{R}_k \text{Exp}(\delta \boldsymbol{\theta}_k), \quad (3.55c)$$

$$\mathbf{b}_{g,k+1} = \mathbf{b}_{g,k} + \delta \mathbf{b}_{g,k}, \quad (3.55d)$$

$$\mathbf{b}_{a,k+1} = \mathbf{b}_{a,k} + \delta \mathbf{b}_{a,k}, \quad (3.55e)$$

$$\mathbf{g}_{k+1} = \mathbf{g}_k + \delta \mathbf{g}_k. \quad (3.55f)$$

Some literature defines this computation as a generalized state variable addition operation:

$$\mathbf{x}_{k+1} = \mathbf{x}_k \oplus \delta \mathbf{x}_k, \quad (3.56)$$

This notation can simplify the overall expression but sacrifices some readability. When formulas contain too many generalized addition/subtraction operations (especially different definitions like \oplus , \ominus , \ominus , \boxplus , etc.), it becomes difficult for readers to quickly recognize their specific meanings. Therefore, this book prefers to explicitly write out each state component using standard addition rather than generalized addition symbols.

The ESKF reset consists of two parts: mean reset and covariance reset. The mean part can be simply implemented as:

$$\delta \mathbf{x} = \mathbf{0}. \quad (3.57)$$

Since the mean has been reset, the covariance we previously described was in the tangent space of \mathbf{x}_k , but now needs to describe the covariance in \mathbf{x}_{k+1} . This reset introduces some subtle differences, primarily affecting the rotation component. In fact, before resetting, the Kalman filter characterized a Gaussian distribution $\mathcal{N}(\delta \mathbf{x}, \mathbf{P})$ in the tangent space of \mathbf{x}_{pred} , while after resetting, it should characterize $\mathcal{N}(0, \mathbf{P}_{\text{reset}})$ at $\mathbf{x}_{\text{pred}} + \delta \mathbf{x}$. This is identical for vector states, but for rotation variables, the zero point of their tangent space has changed, requiring mathematical distinction.

Let the nominal rotation estimate before reset be \mathbf{R}_k , the error state be $\delta\boldsymbol{\theta}$, and the Kalman filter's incremental result be $\delta\boldsymbol{\theta}_k^{17}$. After reset, the nominal rotation becomes $\mathbf{R}_k \text{Exp}(\delta\boldsymbol{\theta}_k) = \mathbf{R}^+$, and the error state becomes $\delta\boldsymbol{\theta}^+$. Since the error state is reset, clearly $\delta\boldsymbol{\theta}^+ = \mathbf{0}$. However, what we care about is not their direct values but the linearized relationship between $\delta\boldsymbol{\theta}^+$ and $\delta\boldsymbol{\theta}$. Writing out the actual reset process:

$$\mathbf{R}^+ \text{Exp}(\delta\boldsymbol{\theta}^+) = \mathbf{R}_k \text{Exp}(\delta\boldsymbol{\theta}_k) \text{Exp}(\delta\boldsymbol{\theta}^+) = \mathbf{R}_k \text{Exp}(\delta\boldsymbol{\theta}). \quad (3.58)$$

We can obtain:

$$\text{Exp}(\delta\boldsymbol{\theta}^+) = \text{Exp}(-\delta\boldsymbol{\theta}_k) \text{Exp}(\delta\boldsymbol{\theta}), \quad (3.59)$$

Here $\delta\boldsymbol{\theta}$ is small. Using the linearized BCH formula, we get:

$$\delta\boldsymbol{\theta}^+ = -\delta\boldsymbol{\theta}_k + \delta\boldsymbol{\theta} - \frac{1}{2}\delta\boldsymbol{\theta}_k^\wedge \delta\boldsymbol{\theta} + o((\delta\boldsymbol{\theta})^2). \quad (3.60)$$

Thus:

$$\frac{\partial \delta\boldsymbol{\theta}^+}{\partial \delta\boldsymbol{\theta}} \approx \mathbf{I} - \frac{1}{2}\delta\boldsymbol{\theta}_k^\wedge. \quad (3.61)$$

This shows that the error states before and after reset differ by a small Jacobian matrix in rotation, which we denote as $\mathbf{J}_{\boldsymbol{\theta}} = \mathbf{I} - \frac{1}{2}\delta\boldsymbol{\theta}_k^\wedge$. Extending this small Jacobian to the full state dimension while keeping other parts as identity matrices gives the complete Jacobian:

$$\mathbf{J}_k = \text{diag}(\mathbf{I}_3, \mathbf{I}_3, \mathbf{J}_{\boldsymbol{\theta}}, \mathbf{I}_3, \mathbf{I}_3, \mathbf{I}_3), \quad (3.62)$$

Therefore, while resetting the error state mean to zero, their covariance matrix should also undergo linear transformation:

$$\mathbf{P}_{\text{reset}} = \mathbf{J}_k \mathbf{P} \mathbf{J}_k^\top. \quad (3.63)$$

However, since $\delta\boldsymbol{\theta}_k$ is not large, \mathbf{J}_k remains very close to the identity matrix. Therefore, most materials do not handle this term but directly use the previously estimated \mathbf{P} matrix as the starting point for the next time step. Nevertheless, this book still introduces this concept and will further discuss it in Chapter 8. The practical significance of this issue is performing a **tangent space projection**, i.e., projecting a Gaussian distribution from one tangent space to another. In ESKF, the difference is not significant, but the later Iterated Extended Kalman Filter (IESKF) will involve multiple tangent space transformations during the observation process. In comparison, ESKF only has a single transformation during reset, making its principle simpler.

3.5 Implementation of ESKF-based Integrated Navigation

Below we implement an ESKF that fuses IMU and GNSS observations. The code in this section will output results to text files for subsequent visualization. During the ESKF implementation, we will encounter some practical details that will be discussed here.

¹⁷Note that here $\delta\boldsymbol{\theta}_k$ is known, while $\delta\boldsymbol{\theta}$ is a random variable.

3.5.1 Implementation of the ESKF Filter

First, we define the ESKF class. Its member variables should include the nominal state, error state, covariance matrix, and various sensor noise parameters.

Listing 3.10: src/ch3/eskf.hpp

```

1 template <typename S = double>
2 class ESKF {
3     /// Type definitions
4     using SO3 = Sophus::SO3<S>;           // Rotation type
5     using Vect = Eigen::Matrix<S, 3, 1>;      // Vector type
6     using Vec18T = Eigen::Matrix<S, 18, 1>;    // 18-dimensional vector
7     using Mat3T = Eigen::Matrix<S, 3, 3>;      // 3x3 matrix
8     using MotionNoiseT = Eigen::Matrix<S, 18, 18>; // Motion noise type
9     using OdomNoiseT = Eigen::Matrix<S, 3, 3>;    // Odometry noise type
10    using GnssNoiseT = Eigen::Matrix<S, 6, 6>;    // GNSS noise type
11    using Mat18T = Eigen::Matrix<S, 18, 18>;    // 18-dimensional covariance
12    using NavStateT = NavState<S>;            // Complete nominal state type
13
14    /// Other constructors and member functions omitted
15    private:
16    /// Member variables
17    double current_time_ = 0.0; // Current timestamp
18
19    /// Nominal state
20    Vect p_ = Vect::Zero();
21    Vect v_ = Vect::Zero();
22    SO3 R_;
23    Vect bg_ = Vect::Zero();
24    Vect ba_ = Vect::Zero();
25    Vect g_{0, 0, -9.8};
26
27    /// Error state
28    Vec18T dx_ = Vec18T::Zero();
29
30    /// Covariance matrix
31    Mat18T cov_ = Mat18T::Identity();
32
33    /// Noise matrices
34    MotionNoiseT Q_ = MotionNoiseT::Zero();
35    OdomNoiseT odom_noise_ = OdomNoiseT::Zero();
36    GnssNoiseT gnss_noise_ = GnssNoiseT::Zero();
37
38    /// Flags
39    bool first_gnss_ = true; // Whether it's the first GNSS data
40
41    /// Configuration options
42    Options options_;
43 };
44
45 using ESKFD = ESKF<double>;
46 using ESKFF = ESKF<float>;

```

According to previous derivations, the nominal state includes position, velocity, rotation, biases, and gravity, while the error state corresponds to their vector forms. The error state vector should be 18-dimensional (3×6), with a corresponding 18×18 covariance matrix. We allow users to choose between single-precision or double-precision ESKF implementations, which have minor performance differences. The floating-point precision can be specified through template parameters, making our ESKF class a template class. Some ESKF implementations also allow users to define custom variable dimensions and ordering, which would make the class more complex when templated, as seen in [65]. This book only defines float and double ESKF classes, with internal types specified using 'using' declarations.

3.5.2 Implementing the Prediction Process

Next, we implement the ‘Predict’ function, which propagates the current state based on IMU measurements. The prediction process involves updating the nominal state variables and propagating the covariance matrix. The implementation is as follows:

Listing 3.11: src/ch3/eskf.hpp

```

1 template <typename S>
2 bool ESKF<S>::Predict(const IMU& imu) {
3     assert(imu.timestamp_ >= current_time_);
4
5     double dt = imu.timestamp_ - current_time_;
6     if (dt > (5 * options_.imu_dt_) || dt < 0) {
7         // Invalid time interval, possibly the first IMU measurement with no prior
8         // information
9         LOG(INFO) << "skip this imu because dt_ = " << dt;
10        current_time_ = imu.timestamp_;
11        return false;
12    }
13
14    // Propagate the nominal state
15    VecT new_p = p_ + v_ * dt + 0.5 * (R_ * (imu.acce_ - ba_)) * dt * dt + 0.5 * g_ * dt
16    * dt;
17    VecT new_v = v_ + R_ * (imu.acce_ - ba_) * dt + g_ * dt;
18    SO3 new_R = R_ * SO3::exp((imu.gyro_ - bg_) * dt);
19
20    R_ = new_R;
21    v_ = new_v;
22    p_ = new_p;
23    // Other state dimensions remain unchanged
24
25    // Propagate the error state
26    // Compute the Jacobian matrix F for the motion process (see Eq. 3.47)
27    // F is a sparse matrix, but we use matrix form for clarity (scattered computation
28    // is also possible for efficiency)
29    Mat18T F = Mat18T::Identity();                                // Main diagonal
30    F.template block<3, 3>(0, 0) = Mat3T::Identity() * dt;          // p
31    F.template block<3, 3>(0, 1) = -R_.matrix() * SO3::hat(imu.acce_ - ba_) * dt; // v
32    F.template block<3, 3>(0, 2) = -R_.matrix() * dt;                // v
33    F.template block<3, 3>(1, 0) = -R_.matrix() * dt;                // v
34    F.template block<3, 3>(1, 1) = SO3::exp(-(imu.gyro_ - bg_) * dt).matrix(); // g
35    F.template block<3, 3>(1, 2) = -Mat3T::Identity() * dt;           // g
36    F.template block<3, 3>(2, 0) = -Mat3T::Identity() * dt;           // theta
37    F.template block<3, 3>(2, 1) = -Mat3T::Identity() * dt;           // theta
38    F.template block<3, 3>(2, 2) = -Mat3T::Identity() * dt;           // theta
39
40    // Mean and covariance prediction
41    dx_ = F * dx_; // This line is technically unnecessary since dx_ is reset to zero
42    // after updates, but F is needed for covariance propagation
43    cov_ = F * cov_.eval() * F.transpose() + Q_;
44    current_time_ = imu.timestamp_;
45    return true;
46}

```

Here, we explicitly construct the full \mathbf{F} matrix and update the covariance matrix using matrix operations. Alternatively, one could compute sparse matrix blocks separately for each state variable to improve computational efficiency.

The prediction process essentially propagates the nominal state using IMU measurements. Simultaneously, it incorporates motion noise into the covariance matrix, causing it to **expand** intuitively. The error state update can be skipped here because the observation step resets the error state to zero, making $\mathbf{F}\delta\mathbf{x}$ naturally zero regardless of \mathbf{F} ’s values.

The motion process is triggered by IMU measurements and can be called at high frequencies, providing high-frequency predicted pose updates.

3.5.3 Implementing RTK Observation Process

Next, we consider how to implement the GNSS observation equation. Here we assume RTK can provide 6-DoF observations, including both position and orientation. Note that single-antenna solutions cannot be processed this way - they should use 3-DoF observation information.

The abstract form of the observation equation is $\mathbf{y} = \mathbf{h}(\mathbf{x})$. In Section 3.4.4, we introduced the general observation model. For GNSS observations converted to vehicle UTM coordinates, they can be directly treated as observations of current \mathbf{R} and \mathbf{p} . Let \mathbf{R}_{gnss} , \mathbf{p}_{gnss} denote observations at a certain time, from which we derive the observation equation and Kalman gain. Key points include:

1. In dual-antenna setups, vehicle orientation is determined by two GNSS receivers. However, there may be cases where one GNSS is valid while the other fails - making position observations valid but heading invalid. Here we only use observations where both position and orientation are valid.
2. GNSS observations of \mathbf{R} can be directly expressed as observations of error state $\delta\theta$, simplifying the linearization process by avoiding chain rule derivation.
3. Since UTM coordinates from GNSS are typically large, we subtract the first valid GNSS observation as origin to maintain numerical precision. This keeps ESKF and visualization coordinates near zero, avoiding potential issues with excessive significant digits in plotting/visualization software.

Let's elaborate point 2. The GNSS rotation observation equation is:

$$\mathbf{R}_{\text{gnss}} = \mathbf{R}\text{Exp}(\delta\theta), \quad (3.64)$$

where \mathbf{R} is the nominal state and $\delta\theta$ the error state. Since \mathbf{R} is fixed during observation, we can treat \mathbf{R}_{gnss} as direct observation of $\delta\theta$. Transforming the equation:

$$\mathbf{z}_{\delta\theta} = \mathbf{h}(\delta\theta) = \text{Log}(\mathbf{R}^\top \mathbf{R}_{\text{gnss}}). \quad (3.65)$$

Here $\mathbf{z}_{\delta\theta}$ directly observes $\delta\theta$, so its Jacobian is identity:

$$\frac{\partial \mathbf{z}_{\delta\theta}}{\partial \delta\theta} = \mathbf{I}. \quad (3.66)$$

This avoids nominal-to-error state conversion. Note the ESKF innovation term $\mathbf{z} - \mathbf{h}(\mathbf{x})$ should then use manifold form:

$$\mathbf{z} - \mathbf{h}(\mathbf{x}) = [\mathbf{p}_{\text{gnss}} - \mathbf{p}, \text{Log}(\mathbf{R}^\top \mathbf{R}_{\text{gnss}})]^\top. \quad (3.67)$$

Since $\delta\theta$ remains zero after prediction, $\mathbf{h}(\mathbf{x})$'s rotation part is treated as zero while translation follows standard definition. This 6D innovation updates system state:

$$\mathbf{x} = \mathbf{x}_{\text{pred}} + \mathbf{K}(\mathbf{z} - \mathbf{h}(\mathbf{x})). \quad (3.68)$$

Note the rotation update should use manifold methods.

The translation part is straightforward:

$$\mathbf{p}_{\text{gnss}} = \mathbf{p} + \delta\mathbf{p}. \quad (3.69)$$

Thus its Jacobian is identity:

$$\frac{\partial \mathbf{p}_{\text{gnss}}}{\partial \delta \mathbf{p}} = \mathbf{I}_{3 \times 3}. \quad (3.70)$$

In the ESKF class, we define SE(3) observations (reused later) and convert GNSS readings to SE(3) observations:

Listing 3.12: src/ch3/eskf.hpp

```

1 template <typename S>
2 bool ESKF<S>::ObserveGps(const GNSS& gnss) {
3     /// GNSS observation correction
4     assert(gnss.unix_time_ >= current_time_);
5
6     if (first_gnss_) {
7         R_ = gnss.utm_pose_.so3();
8         p_ = gnss.utm_pose_.translation();
9         first_gnss_ = false;
10        current_time_ = gnss.unix_time_;
11        return true;
12    }
13
14    assert(gnss.heading_valid_);
15    ObserveSE3(gnss.utm_pose_, options_.gnss_pos_noise_, options_.gnss_ang_noise_);
16    current_time_ = gnss.unix_time_;
17
18    return true;
19}
20
21 template <typename S>
22 bool ESKF<S>::ObserveSE3(const SE3& pose, double trans_noise, double ang_noise) {
23     /// SE(3) observation with both rotation and translation
24     /// Observe p and R, H is 6x18 (zeros elsewhere)
25     Eigen::Matrix<S, 6, 18> H = Eigen::Matrix<S, 6, 18>::Zero();
26     H.template block<3, 3>(0, 0) = Mat3T::Identity(); // Position part
27     H.template block<3, 3>(3, 6) = Mat3T::Identity(); // Rotation part (Eq.3.66)
28
29     // Kalman gain and update
30     Vec6d noise_vec;
31     noise_vec << trans_noise, trans_noise, trans_noise, ang_noise, ang_noise, ang_noise;
32
33     Mat6d V = noise_vec.asDiagonal();
34     Eigen::Matrix<S, 18, 6> K = cov_* H.transpose() * (H * cov_* H.transpose() + V).
35         inverse();
36
37     // State and covariance update
38     Vec6d innov = Vec6d::Zero();
39     innov.template head<3>() = (pose.translation() - p_); // Translation
40     innov.template tail<3>() = (R_.inverse() * pose.so3()).log(); // Rotation (Eq.3.67)
41
42     dx_ = K * innov;
43     cov_ = (Mat18T::Identity() - K * H) * cov_;
44
45     UpdateAndReset();
46     return true;
47 }
48
49 void UpdateAndReset() {
50     p_ += dx_.template block<3, 1>(0, 0);
51     v_ += dx_.template block<3, 1>(3, 0);
52     R_ = R_ * SO3::exp(dx_.template block<3, 1>(6, 0));
53
54     if (options_.update_bias_gyro_) {
55         bg_ += dx_.template block<3, 1>(9, 0);
56     }
57
58     if (options_.update_bias_acce_) {
59         ba_ += dx_.template block<3, 1>(12, 0);
60     }
61
62     g_ += dx_.template block<3, 1>(15, 0);
63
64     ProjectCov();

```

```

64     dx_.setZero();
65 }

```

Readers can verify consistency between code implementation and mathematical derivation. Visually, RTK readings primarily affect error states through Kalman gain during observation. Some ESKF implementations allow tuning Kalman gain to adjust RTK's influence on state updates.

3.5.4 Initialization of the ESKF System

Finally, we bring the entire ESKF into operation. Here we encounter several implementation details. For instance, the ESKF requires knowledge of initial conditions such as IMU biases, initial gravity direction, and the RTK processing function needs to wait for valid measurements to determine the initial nominal state. In traditional integrated navigation systems, the most common approach is the **static initialization** method.

Static initialization involves placing the IMU stationary for a period of time. During this static period, since the object undergoes no motion, we can reasonably assume:

- The gyroscope measurements reflect only bias
- The accelerometer measurements reflect bias plus gravity

We implement a static initialization procedure to estimate these parameters:

1. Keep the IMU stationary for a predetermined duration (set to 10 seconds in the program). Stationarity is verified when wheel speeds (if available) from both wheels are below a threshold. In absence of wheel speed measurements, we can directly assume the vehicle is stationary to estimate the relevant variables.
2. Compute the mean values of gyroscope and accelerometer readings during the static period, denoted as $\bar{\mathbf{d}}_{\text{gyr}}$, $\bar{\mathbf{d}}_{\text{acc}}$;
3. Since no rotation occurs, the gyroscope bias can be estimated as $\mathbf{b}_g = \bar{\mathbf{d}}_{\text{gyr}}$.
4. The accelerometer measurement model is:

$$\tilde{\mathbf{a}} = \mathbf{R}^\top (\mathbf{a} - \mathbf{g}) + \mathbf{b}_a + \boldsymbol{\eta}_a. \quad (3.71)$$

When the actual acceleration is zero and the rotation is considered as $\mathbf{R} = \mathbf{I}$ ¹⁸, the accelerometer actually measures $\mathbf{b}_a - \mathbf{g}$, where \mathbf{b}_a is small and the magnitude of \mathbf{g} can be considered constant. Under these assumptions, we take the vector with direction $-\bar{\mathbf{d}}_{\text{acc}}$ and magnitude 9.8 as the **gravity vector**. This step determines the gravity direction.

5. Now remove gravity from the accelerometer readings during this period and recalculate $\bar{\mathbf{d}}_{\text{acc}}$;
6. Take $\mathbf{b}_a = \bar{\mathbf{d}}_{\text{acc}}$.
7. Simultaneously, assuming constant biases, estimate the measurement variances of the gyroscope and accelerometer. These variances can be used as noise parameters for the ESKF.

Here is the translation of the provided content while preserving the LaTeX format:

¹⁸Note that in our system, we estimate the initial gravity direction, so the vehicle attitude can be treated as \mathbf{I} while gravity may not necessarily point vertically along the $-Z$ axis. Some references assume fixed gravity direction with uncertain initial state, which makes the derivation slightly more complicated.

3.5.5 Implementation of Static Initialization

Below is the implementation code for static initialization:

Listing 3.13: src/ch3/static_imu_init.cc

```

1 class StaticIMUInit {
2 public:
3     struct Options {
4         double init_time_seconds_ = 10.0;           // Static duration
5         int init_imu_queue_max_size_ = 2000;        // Max size of initialization IMU queue
6         int static_odom_pulse_ = 5;                  // Wheel speed noise when static
7         double max_static_gyro_var = 0.2;           // Gyro measurement variance when static
8         double max_static_acce_var = 0.05;          // Accelerometer measurement variance when
9             static
10        double gravity_norm_ = 9.81;                // Gravity magnitude
11        bool use_speed_for_static_checking_ = true; // Whether to use odom for static
12            checking (some datasets don't have odom)
13    };
14
15    /// Constructor
16    StaticIMUInit(Options options) : options_(options) {}
17
18    /// Add IMU data
19    bool AddIMU(const IMU& imu);
20    /// Add wheel speed data
21    bool AddOdom(const Odom& odom);
22
23    /// Check if initialization succeeded
24    bool InitSuccess() const { return init_success_; }
25
26    /// Get Cov, bias, gravity
27    Vec3d GetCovGyro() const { return cov_gyro_; }
28    Vec3d GetCovAcce() const { return cov_acce_; }
29    Vec3d GetInitBg() const { return init_bg_; }
30    Vec3d GetInitBa() const { return init_ba_; }
31    Vec3d GetGravity() const { return gravity_; }
32
33 private:
34    /// Attempt system initialization
35    bool TryInit();
36
37    Options options_;                                // Configuration options
38    bool init_success_ = false;                      // Whether initialization succeeded
39    Vec3d cov_gyro_ = Vec3d::Zero();                // Gyro measurement noise covariance (evaluated
40        during init)
41    Vec3d cov_acce_ = Vec3d::Zero();                // Accelerometer measurement noise covariance (
42        evaluated during init)
43    Vec3d init_bg_ = Vec3d::Zero();                 // Initial gyro bias
44    Vec3d init_ba_ = Vec3d::Zero();                 // Initial accelerometer bias
45    Vec3d gravity_ = Vec3d::Zero();                 // Gravity
46    bool is_static_ = false;                        // Flag indicating if vehicle is static
47    std::deque<IMU> init_imu_deque_;              // Data for initialization
48    double current_time_ = 0.0;                     // Current time
49    double init_start_time_ = 0.0;                  // Initial static time
50
51    bool StaticIMUInit::TryInit() {
52        if (init_imu_deque_.size() < 10) {
53            return false;
54        }
55
56        // Compute mean and variance
57        Vec3d mean_gyro, mean_acce;
58        math::ComputeMeanAndCovDiag(init_imu_deque_, mean_gyro, cov_gyro_, [] (const IMU& imu)
59            { return imu.gyro_; });
60        math::ComputeMeanAndCovDiag(init_imu_deque_, mean_acce, cov_acce_, [this] (const IMU&
61            imu) { return imu.acce_; });
62
63        // Use accelerometer mean direction with 9.8 magnitude as gravity
64        gravity_ = -mean_acce / mean_acce.norm() * options_.gravity_norm_;
65
66        // Recompute accelerometer covariance
67        math::ComputeMeanAndCovDiag(init_imu_deque_, mean_acce, cov_acce_,

```

```

63 [this](const IMU& imu) { return imu.acce_ + gravity_; });
64
65 // Check IMU noise
66 if (cov_gyro_.norm() > options_.max_static_gyro_var) {
67     LOG(ERROR) << "Gyro measurement noise too large" << cov_gyro_.norm() << " > "
68         options_.max_static_gyro_var;
69     return false;
70 }
71 if (cov_acce_.norm() > options_.max_static_acce_var) {
72     LOG(ERROR) << "Accelerometer measurement noise too large" << cov_acce_.norm() << " >
73         " << options_.max_static_acce_var;
74     return false;
75 }
76 // Estimate measurement noise and bias
77 init_bg_ = mean_gyro;
78 init_ba_ = mean_acce;
79
80 LOG(INFO) << "IMU initialization successful, init time= " << current_time_ -
81     init_start_time_ << ", bg = " << init_bg_.transpose()
82 << ", ba = " << init_ba_.transpose() << ", gyro sq = " << cov_gyro_.transpose()
83 << ", acce sq = " << cov_acce_.transpose() << ", grav = " << gravity_.transpose()
84 << ", norm: " << gravity_.norm();
85 LOG(INFO) << "mean gyro: " << mean_gyro.transpose() << " acce: " << mean_acce.
86     transpose();
87 init_success_ = true;
88 return true;
89 }
```

This primarily calls the mean and covariance computation functions from the math library:

Listing 3.14: src/common/math_utils.h

```

1 /**
2 * Compute mean and diagonal covariance for data in a container
3 * @tparam C Container type
4 * @tparam D Result type
5 * @tparam Getter Data accessor function that takes container element type and
6 *                 returns D type
7 */
8 template <typename C, typename D, typename Getter>
9 void ComputeMeanAndCovDiag(const C& data, D& mean, D& cov_diag, Getter&& getter) {
10     size_t len = data.size();
11     assert(len > 1);
12     mean = std::accumulate(data.begin(), data.end(), D::Zero().eval(),
13     [&getter](const D& sum, const auto& data) -> D { return sum + getter(data); }) /
14     len;
15     cov_diag = std::accumulate(data.begin(), data.end(), D::Zero().eval(),
16     [&mean, &getter](const D& sum, const auto& data) -> D {
17         return sum + (getter(data) - mean).cwiseAbs2().eval();
18     }) / (len - 1);
19 }
```

As long as the data fields are stored in Eigen format, this function can compute the mean and diagonal covariance for specified fields in any container. It uses lambda functions to access user-specified fields and template types for good compatibility, allowing it to work with various storage forms (std::vector, std::deque, etc.) and different field types. In this example, we call this function on the gyroscope and accelerometer readings from the IMU data queue to estimate their means and variances. This information will be used in the ESKF initialization process.

3.5.6 Running the ESKF

Now we proceed to run the ESKF. We read recorded sensor data from text files, process this information through callback functions, and then output the ESKF's post-update states to

result files. Several logical relationships need to be handled:

1. First, the static initialization method requires a period of IMU readings to estimate biases and gravity direction. During this period, the ESKF will not process RTK data. If initialization succeeds, we pass the initial biases and noise parameters to the ESKF.
 2. On the other hand, the ESKF also requires the first valid RTK measurement to determine the map origin and initial nominal state values. This is because the vehicle's initial world position and attitude may not be at the origin. If IMU initialization is complete but the first RTK hasn't been acquired, the ESKF won't process subsequent IMU readings.
 3. When the ESKF operates normally, we write both the predicted nominal state and post-observation nominal state to files and send them to the graphical interface.

Listing 3.15: src/ch4/run_ekf_gins.cc

```

50    }
51
52    /// Remove origin offset
53    if (!first_gnss_set) {
54        origin = gnss_convert.utm_pose_.translation();
55        first_gnss_set = true;
56    }
57    gnss_convert.utm_pose_.translation() -= origin;
58
59    // Require valid RTK heading for EKF integration
60    eskf.ObserveGps(gnss_convert);
61
62    auto state = eskf.GetNominalState();
63    ui->UpdateNavState(state);
64    save_result(fout, state);
65
66    gnss_initiated = true;
67}
68.SetOdomProcessFunc([&imu_init](const sad::Odom& odom) {
69    /// Odom processing function (used only for initialization in this chapter)
70    imu_init.AddOdom(odom);
71})
72.Go();

```

Now please compile and run this program. You can specify the text file to run using gflags:

Listing 3.16: Terminal input:

```
bin/run_eskf_gins --txt_path ./data/ch3/10.txt
```

The program will display real-time filter states as shown in Figure 3-14. On the right panel, we can also observe the ESKF’s real-time estimates of IMU biases and vehicle velocities in both world and body frames. Readers should observe the following phenomena:

1. First, the ESKF incorporates prior information in each prediction and update, resulting in relatively smooth trajectories when good RTK and IMU data are available. Readers can zoom in on the trajectory in the UI to examine individual data points.
2. The body-frame velocity is primarily along the positive X-axis, consistent with actual forward vehicle motion. World-frame velocity has both X and Y components.
3. During periods of poor RTK quality, the ESKF lacks observations and the position estimates diverge rapidly. When RTK recovers, the position converges back to the RTK trajectory.
4. This implementation unconditionally trusts RTK when heading is valid. However, actual RTK data may contain jitter and outliers, causing corresponding jitter in ESKF output trajectories.

After program execution, ESKF state variables are saved to data/ch3/gins.txt. We can visualize the 2D trajectory using a plotting script, as shown in Figure 3-15:

Listing 3.17: Terminal command:

```
python3 scripts/plot_ch3_state.py ./data/ch3/gins.txt
```

Comparing with Figure 3-12, we see generally similar shapes but observe divergence in areas where RTK angles were invalid. In practical systems, when RTK position is valid but heading is invalid, we could use ESKF-estimated angles to compute vehicle position. We leave this as an exercise.

Some experimental observations about ESKF:

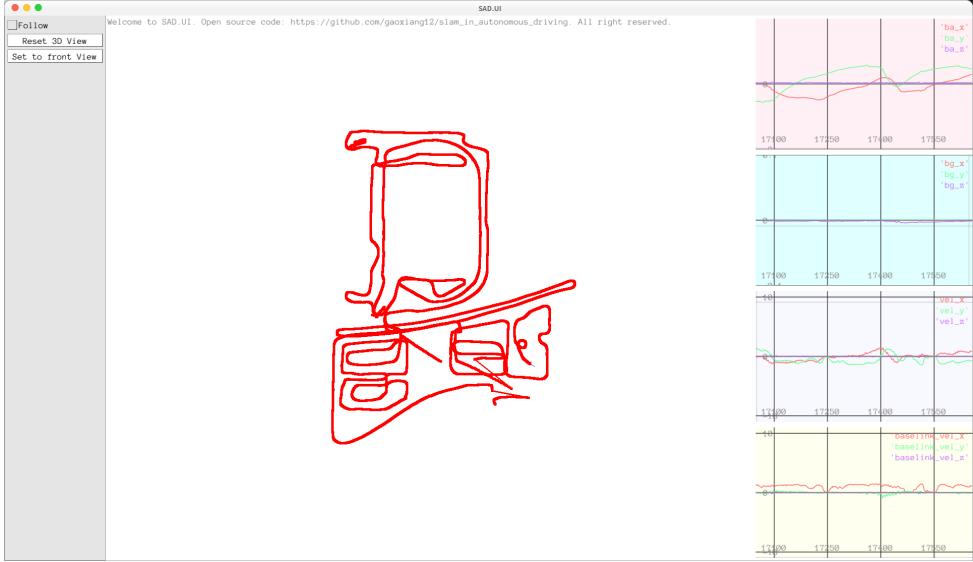


Figure 3-14: Real-time results of ESKF integrated navigation

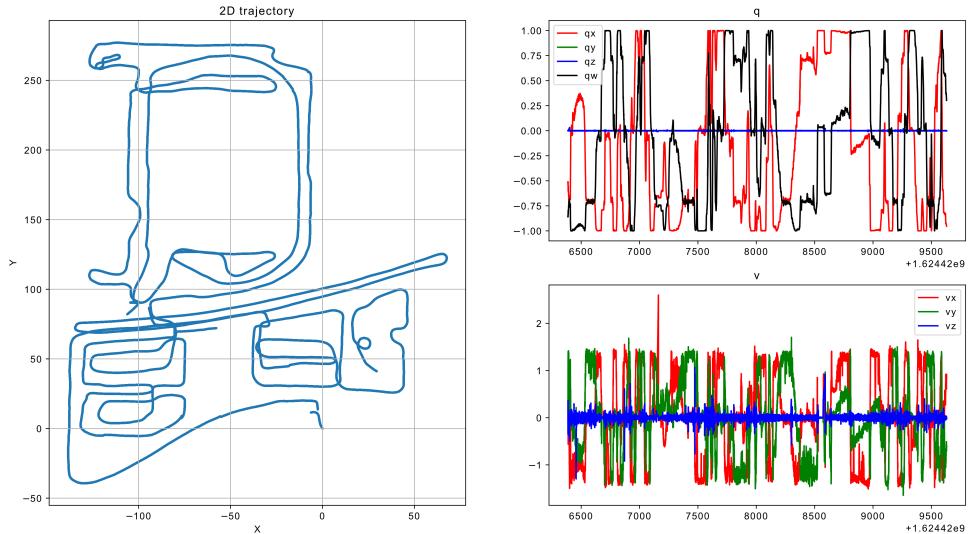


Figure 3-15: ESKF estimated state variables. Left: 2D trajectory; Top-right: Quaternion states; Bottom-right: Velocity states in world frame

1. This ESKF implementation demonstrates IMU-GNSS fusion, with results comparable to GNSS observations. When ESKF runs properly, we can output IMU-predicted poses at higher frequencies. The ESKF framework can incorporate other observation sources - observation models need not be uniform, only linearizable. For example, we'll later integrate LiDAR observations using similar theoretical foundations but different noise characteristics (loose coupling). Alternatively, directly incorporating LiDAR point cloud residuals or visual feature reprojection errors would constitute tight coupling.

2. Maintaining ESKF convergence requires continuous GNSS observations. Prolonged GNSS outages cause IMU predictions to diverge like standalone IMU integration. GNSS observations essentially fix velocity and bias states, making them observable [66]. We won't rigorously discuss estimator observability despite its theoretical importance.
3. We haven't handled GNSS outliers. Practical GNSS may lose signal or report plausible but erroneous positions. While ESKF's predict-update cycle can smooth IMU-GNSS integration, uncorrected outliers can severely corrupt filter states. Innovation checks or position residuals could reject outliers, but distinguishing RTK errors from filter errors remains challenging. Later graph optimization methods will better handle outlier rejection.
4. Readers familiar with graph optimization may compare ESKF's motion/observation models with optimization approaches. They share similarities but differ in implementation. ESKF updates essentially marginalize past observations into priors for the current state - the mean \mathbf{x} and covariance \mathbf{P} become prior information for the next step, smoothing the filtering process. Conventional graph optimization lacks such explicit priors - handling this difference becomes algorithmically crucial. The next chapter will revisit this from a graph optimization perspective for deeper understanding.

3.5.7 Velocity Observations

We observe that in the GINS system, when RTK observations are unavailable for extended periods, the ESKF degenerates into pure IMU integration mode, where position estimates rapidly diverge. This divergence primarily stems from the lack of velocity observations. Are there methods to constrain velocity divergence? The most common approach incorporates vehicle speed sensors. Velocity measurements primarily come from **motor RPM** or **wheel encoders**. Most wheeled robots carry encoders to measure their own speed and infer local motion. Sometimes, we can also use underlying measurements like motor RPM or throttle to determine robot speed. These velocity observations, when combined with IMU, enable local **dead reckoning**. Alternatively, they can be directly integrated into ESKF as velocity observations. Below we derive the mathematical formulation and code implementation.

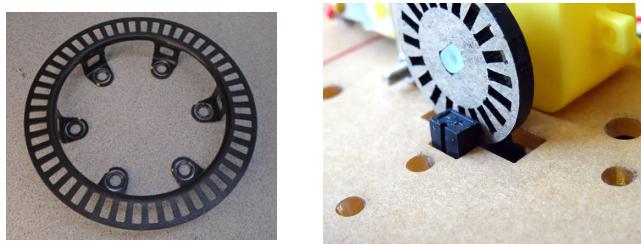


Figure 3-16: A wheel encoder and its installation example

Figure 3-16 illustrates a typical wheel encoder. These encoders usually feature a disk shape with regularly spaced holes along the perimeter. During wheel rotation, an infrared or laser transmitter-receiver pair mounted on the chassis detects these holes. The receiver outputs alternating signals (blocked/unblocked) corresponding to wheel rotation. This periodic output reflects the wheel's angular displacement, which combined with wheel radius and mounting position, can determine the vehicle's relative ground motion.

The characteristics of wheel encoders can be summarized as:

1. A single wheel encoder only outputs angular displacement. By sampling this displacement at fixed intervals, we effectively obtain velocity measurements.
2. For fixed-angle wheel installations (non-steering), encoder readings can compute vehicle displacement. With three-wheeled chassis (common in cleaning robots and small platforms), differential measurements between left/right wheels can additionally compute body rotation - though IMU inherently measures rotation.
3. Wheels only measure **forward-direction** velocity/displacement, unable to detect lateral or vertical motion. For large vehicles on uneven terrain, encoder results may significantly deviate from actual motion.
4. In practice, wheels are prone to slippage. During slippage, wheels spin without actual vehicle movement, causing encoder outputs to misrepresent true motion. Incorporating additional observations can improve localization accuracy.

In this book, we treat wheel speed observations as measurements of the **forward velocity (X-axis) in the body frame**. Specifically, let the scalar wheel speed measurement over a time interval be v_{wheel} , representing the vehicle's forward speed. Using the **front-left-up coordinate convention** (X forward, Y left, Z up), the vector form of wheel speed observation is:

$$\mathbf{v}_{\text{wheel}} = [v_{\text{wheel}}, 0, 0]^T. \quad (3.72)$$

The corresponding observation model is:

$$\mathbf{v}_{\text{wheel}} = \mathbf{R}^T \mathbf{v}, \quad (3.73)$$

where \mathbf{R} , \mathbf{v} are the current vehicle states. While wheel encoders don't physically measure \mathbf{R} , common practice transforms $\mathbf{v}_{\text{wheel}}$ to world coordinates using the estimated \mathbf{R} :

$$\mathbf{R}\mathbf{v}_{\text{wheel}} = \mathbf{v}. \quad (3.74)$$

We formulate this as abstract observation model $\mathbf{h}(\mathbf{x})$. Its Jacobian w.r.t. nominal state \mathbf{x} is:

$$\frac{\partial \mathbf{h}(\mathbf{x})}{\partial \mathbf{x}} = [\mathbf{0}_{3 \times 3}, \mathbf{I}_{3 \times 3}, \mathbf{0}_{3 \times 12}]. \quad (3.75)$$

The Jacobian shows wheel speed only observes \mathbf{v} without affecting other states. More strictly, $\mathbf{v}_{\text{wheel}}$ lacks Y/Z-axis measurements. However, combined with IMU, observed \mathbf{v} prevents rapid divergence. The implementation:

Listing 3.18: src/ch3/eskf.hpp

```

1 template <typename S>
2 bool ESKF<S>::ObserveWheelSpeed(const Odom& odom) {
3     assert(odom.timestamp_ >= current_time_);
4     // Odom correction and Jacobian
5     // 3D wheel speed observation, H is 3x18 (mostly zeros)
6     Eigen::Matrix<S, 3, 18> H = Eigen::Matrix<S, 3, 18>::Zero();
7     H.template block<3, 3>(0, 0) = Mat3T::Identity();
8
9     // Kalman gain
10    Eigen::Matrix<S, 18, 3> K = cov_* H.transpose() * (H * cov_* H.transpose() +
11        odom_noise_).inverse();
12
13    // Convert pulse to velocity
14    double velo_l = options_.wheel_radius_* odom.left_pulse_ / options_.circle_pulse_* 2
15        * M_PI / options_.odom_span_;
14    double velo_r = options_.wheel_radius_* odom.right_pulse_ / options_.circle_pulse_* 2
15        * M_PI / options_.odom_span_;
15    double average_vel = 0.5 * (velo_l + velo_r);

```

```

16
17 VecT vel_odom(average_vel, 0.0, 0.0);
18 VecT vel_world = R_ * vel_odom;
19
20 dx_ = K * (vel_world - v_);
21
22 // Update covariance
23 cov_ = (Mat18T::Identity() - K * H) * cov_;
24
25 UpdateAndReset();
26 return true;
27 }
```

The implementation converts wheel encoder pulses p (measured over interval t) to linear velocity using wheel radius r and pulses per revolution n :

$$v_{\text{wheel}} = \frac{2\pi r p}{nt}. \quad (3.76)$$

Under ideal conditions, this provides velocity observations. We average left/right wheels for robustness. While simplified (ignoring vehicle kinematics), this demonstrates ESKF integration. The error state $\delta\mathbf{x}$ updates the nominal state.

To run ESKF with wheel speed:

Listing 3.19: Terminal command:

```
bin/run_eskf_gins --with_odom=true
```

Plot results as before. The real-time UI (Fig. 3-17) shows improved trajectory stability during RTK outages compared to Fig. 3-14, demonstrating wheel speed’s benefits.

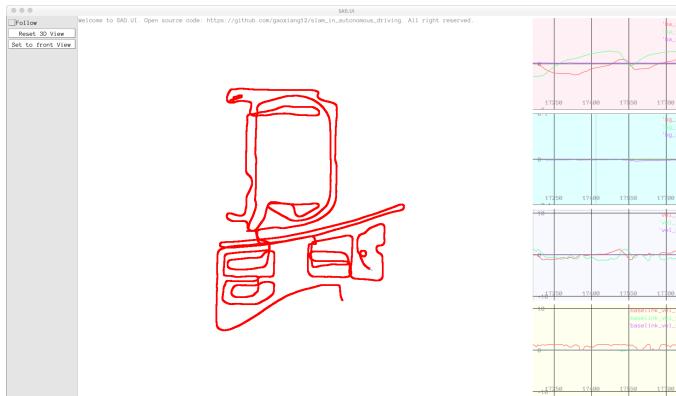


Figure 3-17: Real-time trajectory with wheel speed observations

3.6 Summary

This chapter introduced the fundamental principles of inertial navigation systems, presenting a traditional integrated navigation solution that combines IMU, GNSS, and wheel odometry through an error-state Kalman filter framework. This approach proves effective for vehicle localization. We demonstrated its operation through code implementation and graphical interfaces. However, the current implementation doesn’t incorporate visual or LiDAR sensors, limiting our observation to vehicle position and attitude data without scene structure representation. Subsequent chapters on LiDAR will build upon these results by integrating 2D/3D LiDAR data for mapping and localization.

Exercises

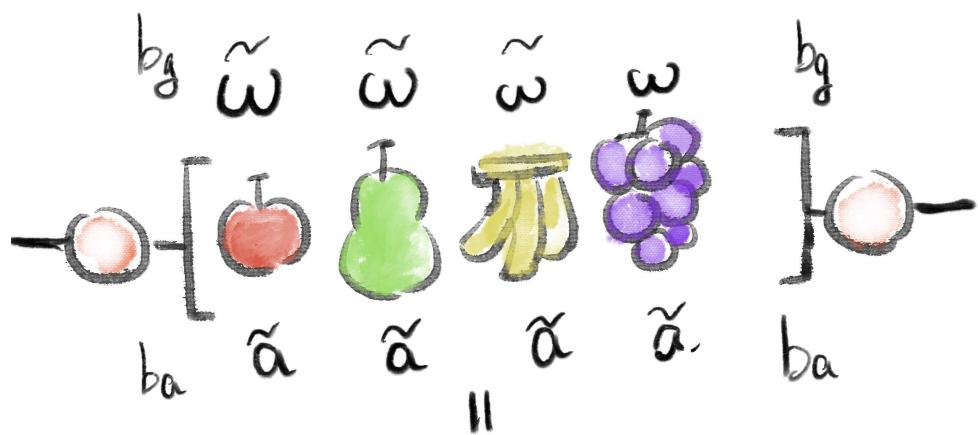
1. Prove that any zero-mean white noise random variable remains zero-mean white noise with unchanged covariance matrix when multiplied by an arbitrary rotation matrix coefficient.
2. Derive the ESKF state transition equations using quaternion representation.
3. In rotational component processing, left-multiplication and right-multiplication Lie algebras are fundamentally equivalent. This book adopts right-multiplication convention - derive the ESKF state equations using left-multiplication Lie algebra as error variables. Does left-multiplication offer simpler formulation?
4. By slightly modifying the ESKF's motion and observation equations, variants like UKF and IEKF can be obtained. Implement UKF or IEKF-based integrated navigation using the provided codebase.
5. Simplify the ESKF prediction process by avoiding matrix-form \mathbf{F} and instead writing separate equations for each variable's motion process. Evaluate potential efficiency gains. This approach is sometimes called **Sparse ESKF**.
6. Attempt to simplify matrix computations in ESKF by eliminating unnecessary calculations while preserving essential operations. Assess efficiency improvements.
7. Analyze the ESKF's innovation vector. What are typical magnitudes? How to determine appropriate outlier detection thresholds?
8. Design a checking mechanism to filter anomalous RTK observations in ESKF.
9. Develop a monitoring system that triggers alerts when ESKF operates without RTK observations for extended periods, automatically resetting using the most recent valid RTK observation.

Chapter 4

Pre-integration

In Chapter 3 , we introduced the observation model of IMU data and basic filter methods. In ESKF, we integrate the IMU data between two GNSS observations as the prediction process of ESKF. This approach treats IMU data as a **one-time** usage: integrating them into the current estimate and then updating the estimate with observation data. Obviously, this method is related to the current state estimate. However, if the state variables change, can we reuse these IMU data? From a physical perspective, IMU reflects the **angular change** and **velocity change** of the vehicle between two moments. If we want the IMU computation to be independent of **the current state estimate**, how should we handle it algorithmically? This is the topic we will discuss in this chapter.

This chapter introduces a very common IMU data processing method: **Pre-integration** [67]. Unlike traditional kinematic integration of IMU, pre-integration can accumulate IMU measurement data over a period of time to establish pre-integrated measurements while ensuring that the measurements are independent of state variables. To use an analogy, ESKF is like **eating dishes one bite at a time**, while pre-integration is **first picking pieces of food from the pot into a bowl and then eating all the food in the bowl in one go**. As for how big the bowl should be and how many times to pick food before eating it all at once, the form is relatively flexible. Whether in LIO or VIO systems, pre-integration has become a standard method in many tightly coupled IMU systems [68–70], but its principle is more complex compared to the traditional ESKF prediction process. Below, we will derive its basic principles and then implement a pre-integration system to solve the same problem as in the previous chapter. The content of this chapter serves as preparatory knowledge for many subsequent chapters, so readers must master it thoroughly.



$\Delta R, \Delta V, \Delta P$

预和分可以一次积累多个数据
且与状态无关。

4.1 Pre-integration of IMU States

4.1.1 Definition of Pre-integration

We still start from the kinematic model of IMU. In an IMU system, we consider five variables: rotation \mathbf{R} , translation \mathbf{p} , angular velocity $\boldsymbol{\omega}$, linear velocity \mathbf{v} , and acceleration \mathbf{a} . According to the kinematics introduced in Section 2, the kinematic relationships of these variables can be written as [71]:

$$\dot{\mathbf{R}} = \mathbf{R}\boldsymbol{\omega}^\wedge, \quad (4.1a)$$

$$\dot{\mathbf{p}} = \mathbf{v}, \quad (4.1b)$$

$$\dot{\mathbf{v}} = \mathbf{a}. \quad (4.1c)$$

Within the time interval t to $t + \Delta t$, applying Euler integration to the above equations yields:

$$\mathbf{R}(t + \Delta t) = \mathbf{R}(t)\text{Exp}(\boldsymbol{\omega}(t)\Delta t), \quad (4.2a)$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{a}(t)\Delta t, \quad (4.2b)$$

$$\mathbf{p}(t + \Delta t) = \mathbf{p}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2. \quad (4.2c)$$

Here, the angular velocity and acceleration can be measured by the IMU but are affected by noise and gravity. Let the measurements be $\tilde{\boldsymbol{\omega}}$ and $\tilde{\mathbf{a}}$, then:

$$\tilde{\boldsymbol{\omega}}(t) = \boldsymbol{\omega}(t) + \mathbf{b}_g(t) + \boldsymbol{\eta}_g(t), \quad (4.3a)$$

$$\tilde{\mathbf{a}}(t) = \mathbf{R}^\top(\mathbf{a}(t) - \mathbf{g}) + \mathbf{b}_a(t) + \boldsymbol{\eta}_a(t), \quad (4.3b)$$

where $\mathbf{b}_g, \mathbf{b}_a$ are the biases of the gyroscope and accelerometer, and $\boldsymbol{\eta}_a, \boldsymbol{\eta}_g$ are Gaussian measurement noises. Substituting these into the above equations, we obtain the relationship between the measurements and the state variables:

$$\mathbf{R}(t + \Delta t) = \mathbf{R}(t)\text{Exp}((\tilde{\boldsymbol{\omega}} - \mathbf{b}_g(t) - \boldsymbol{\eta}_{gd}(t))\Delta t), \quad (4.4a)$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{g}\Delta t + \mathbf{R}(t)(\tilde{\mathbf{a}} - \mathbf{b}_a(t) - \boldsymbol{\eta}_{ad}(t))\Delta t, \quad (4.4b)$$

$$\mathbf{p}(t + \Delta t) = \mathbf{p}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{g}\Delta t^2 + \frac{1}{2}\mathbf{R}(t)(\tilde{\mathbf{a}} - \mathbf{b}_a(t) - \boldsymbol{\eta}_{ad}(t))\Delta t^2, \quad (4.4c)$$

where $\boldsymbol{\eta}_{gd}, \boldsymbol{\eta}_{ad}$ are the discretized random walk noises [49]:

$$\text{Cov}(\boldsymbol{\eta}_{gd}(t)) = \frac{1}{\Delta t}\text{Cov}(\boldsymbol{\eta}_g(t)), \quad (4.5a)$$

$$\text{Cov}(\boldsymbol{\eta}_{ad}(t)) = \frac{1}{\Delta t}\text{Cov}(\boldsymbol{\eta}_a(t)). \quad (4.5b)$$

The above process has already been described in the IMU measurement and noise equations. Of course, we could directly use these constraints to construct graph optimization and solve IMU-related problems. However, these equations describe an extremely short time interval, involving only a single IMU measurement. In other words, the IMU measurement frequency is too high. We do not want the optimization process to be invoked with every IMU measurement, as that would waste computational resources. Instead, we prefer to process these IMU measurements collectively.

4.2 Pre-integration of IMU States

4.2.1 Definition of Pre-integration

Now we introduce how to perform IMU pre-integration between keyframes. Let us assume that IMU data between discrete times i and j are accumulated, a process that can last several seconds. This accumulated observation is called **pre-integration** [72]. Of course, if we use different forms of kinematics (such as those introduced in Section 2), the resulting pre-integration forms will also differ [42]. This book primarily uses the $\text{SO}(3) + \mathbf{t}$ approach to derive pre-integration. Thus, during the interval from i to j , we can accumulate the variables in Equation (4.4) to obtain:

$$\mathbf{R}_j = \mathbf{R}_i \prod_{k=i}^{j-1} (\text{Exp}((\tilde{\boldsymbol{\omega}}_k - \mathbf{b}_{g,k} - \boldsymbol{\eta}_{gd,k}) \Delta t)), \quad (4.6a)$$

$$\mathbf{v}_j = \mathbf{v}_i + \mathbf{g} \Delta t_{ij} + \sum_{k=i}^{j-1} \mathbf{R}_k (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,k} - \boldsymbol{\eta}_{ad,k}) \Delta t, \quad (4.6b)$$

$$\mathbf{p}_j = \mathbf{p}_i + \sum_{k=i}^{j-1} \mathbf{v}_k \Delta t + \frac{1}{2} \sum_{k=i}^{j-1} \mathbf{g} \Delta t^2 + \frac{1}{2} \sum_{k=i}^{j-1} \mathbf{R}_k (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,k} - \boldsymbol{\eta}_{ad,k}) \Delta t^2, \quad (4.6c)$$

where $\Delta t_{ij} = \sum_{k=i}^{j-1} \Delta t$ is the accumulated time. Given the state at time i and all measurements, this equation can be used to infer the state at time j . Of course, this is merely the accumulated form of Equation (4.4), with no fundamental difference. This is the traditional **direct integration**, identical to the prediction process in ESKF.

The drawback of direct integration is that it describes a process dependent on the state variables. If we optimize the state at time i , the states at times $i+1, i+2, \dots, j-1$ will also change, requiring this integration to be recalculated [73], which is highly inconvenient. To address this, we slightly modify the above equations, striving to place IMU readings on one side and state variables on the other. Thus, we define relative motion quantities as follows:

$$\Delta \mathbf{R}_{ij} \doteq \mathbf{R}_i^\top \mathbf{R}_j = \prod_{k=i}^{j-1} \text{Exp}((\tilde{\boldsymbol{\omega}}_k - \mathbf{b}_{g,k} - \boldsymbol{\eta}_{gd,k}) \Delta t), \quad (4.7a)$$

$$\Delta \mathbf{v}_{ij} \doteq \mathbf{R}_i^\top (\mathbf{v}_j - \mathbf{v}_i - \mathbf{g} \Delta t_{ij}) = \sum_{k=i}^{j-1} \Delta \mathbf{R}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,k} - \boldsymbol{\eta}_{ad,k}) \Delta t, \quad (4.7b)$$

$$\Delta \mathbf{p}_{ij} \doteq \mathbf{R}_i^\top (\mathbf{p}_j - \mathbf{p}_i - \mathbf{v}_i \Delta t_{ij} - \mathbf{g} \Delta t_{ij}^2), \quad (4.7c)$$

$$= \sum_{k=i}^{j-1} \left[\Delta \mathbf{v}_{ik} \Delta t + \frac{1}{2} \Delta \mathbf{R}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,k} - \boldsymbol{\eta}_{ad,k}) \Delta t^2 \right]. \quad (4.7d)$$

This modification essentially computes a certain "difference" from i to j . Although written in the form of $\mathbf{p}, \mathbf{v}, \mathbf{R}$, they are not direct physical quantities of displacement, velocity, or rotation but artificially defined variables. This definition has some interesting computational properties:

1. Consider starting from time i , where all three quantities are zero. At time $i+1$, we compute $\Delta \mathbf{R}_{i,i+1}, \Delta \mathbf{v}_{i,i+1}$, and $\Delta \mathbf{p}_{i,i+1}$. At time $i+2$, since these equations are in cumulative product or sum forms, we only need to add the measurements at time $i+2$

to the results from times i and $i+1$. This brings great convenience at the computational level. Furthermore, we will find that this property is very convenient for subsequent calculations of various Jacobian matrices.

2. From the rightmost side of the equal signs, all the above computations are independent of the values of \mathbf{R} , \mathbf{v} , \mathbf{p} . Even if their estimates change, the IMU integration quantities do not need to be recalculated.
3. However, if the biases $\mathbf{b}_{a,k}$ or $\mathbf{b}_{g,k}$ change, the above equations theoretically still need to be recalculated. Nevertheless, we can also adjust our pre-integration quantities through the idea of "**correction**" rather than "**recalculation**."
4. Note that pre-integration quantities have no direct physical meaning. Although symbols like $\Delta\mathbf{v}$, $\Delta\mathbf{p}$ are used, they do not represent deviations between two velocities or positions. They are simply defined as such. Of course, dimensionally, they should correspond to angles, velocities, and displacements.
5. Similarly, since pre-integration quantities are not direct physical quantities, the noise of this "measurement model" must also be derived from the original IMU noise.

From these issues, we will introduce how to construct the pre-integration measurement model, noise model, and how to **efficiently** compute its Jacobians with respect to various state variables.

4.2.2 Pre-integration Measurement Model

As seen from the previous discussion, pre-integration inherently involves IMU bias terms, thus inevitably depending on the current bias estimates. To handle this dependency, we make some practical adjustments to the pre-integration definition:

1. We first assume that the bias at time i is **fixed** and remains constant throughout the pre-integration computation.
2. We construct a first-order linearized model of the pre-integration with respect to the bias terms, i.e., discarding higher-order terms of the bias.
3. When the bias estimate changes, we use this linear model to **correct** the pre-integration.

First, we fix the bias estimate at time i to analyze the noise in pre-integration. Both graph optimization and filter techniques require knowing the noise level of a measurement. Let us start with rotation, as it is relatively simpler. Using the BCH expansion, we can make the following approximation:

$$\begin{aligned} \Delta\mathbf{R}_{ij} &= \prod_{k=i}^{j-1} \underbrace{\text{Exp}((\tilde{\boldsymbol{\omega}}_k - \mathbf{b}_{g,i} - \boldsymbol{\eta}_{gd,k}) \Delta t)}_{\text{Using BCH: } \approx \text{Exp}((\tilde{\boldsymbol{\omega}}_k - \mathbf{b}_{g,i}) \Delta t) \text{Exp}(-\mathbf{J}_{r,k} \boldsymbol{\eta}_{gd,k} \Delta t)}, \\ &\approx \prod_{k=i}^{j-1} [\text{Exp}((\tilde{\boldsymbol{\omega}}_k - \mathbf{b}_{g,i}) \Delta t) \text{Exp}(-\mathbf{J}_{r,k} \boldsymbol{\eta}_{gd,k} \Delta t)]. \end{aligned} \quad (4.8)$$

In this equation, we aim to separate the noise term to define the **pre-integration measurement** $\Delta\tilde{\mathbf{R}}_{ij}$. Similar to previous IMU measurements, the measurement is denoted with a $\tilde{(\cdot)}$ symbol:

$$\Delta\tilde{\mathbf{R}}_{ij} = \prod_{k=i}^{j-1} \text{Exp}((\tilde{\boldsymbol{\omega}}_k - \mathbf{b}_{g,i}) \Delta t). \quad (4.9)$$

Note that this model can also be used to define $\Delta\tilde{\mathbf{R}}_{kj}$, $\forall k \in (i, j)$. Based on this clever definition, the above equation can be rewritten as:

$$\begin{aligned}\Delta\mathbf{R}_{ij} &= \underbrace{\text{Exp}((\tilde{\boldsymbol{\omega}}_i - \mathbf{b}_{g,i})\Delta t)}_{\Delta\tilde{\mathbf{R}}_{i,i+1}} \text{Exp}(-\mathbf{J}_{r,i}\boldsymbol{\eta}_{gd,i}\Delta t) \underbrace{\text{Exp}((\tilde{\boldsymbol{\omega}}_{i+1} - \mathbf{b}_{g,i})\Delta t)}_{\Delta\tilde{\mathbf{R}}_{i+1,i+2}} \text{Exp}(-\mathbf{J}_{r,i+1}\boldsymbol{\eta}_{gd,i}\Delta t) \dots, \\ &= \Delta\tilde{\mathbf{R}}_{i,i+1} \underbrace{\text{Exp}(-\mathbf{J}_{r,i}\boldsymbol{\eta}_{gd,i}\Delta t) \Delta\tilde{\mathbf{R}}_{i+1,i+2}}_{=\Delta\tilde{\mathbf{R}}_{i+1,i+2}\text{Exp}(-\Delta\tilde{\mathbf{R}}_{i+1,i+2}^\top \mathbf{J}_{r,i}\boldsymbol{\eta}_{gd,i}\Delta t)} \text{Exp}(-\mathbf{J}_{r,i+1}\boldsymbol{\eta}_{gd,i}\Delta t) \dots, \\ &= \Delta\tilde{\mathbf{R}}_{i,i+2}\text{Exp}(-\Delta\tilde{\mathbf{R}}_{i+1,i+2}^\top \mathbf{J}_{r,i}\boldsymbol{\eta}_{gd,i}\Delta t)\text{Exp}(-\mathbf{J}_{r,i+1}\boldsymbol{\eta}_{gd,i}\Delta t) \Delta\tilde{\mathbf{R}}_{i+2,i+3} \dots\end{aligned}\quad (4.10)$$

By continuously using the adjoint formula to move the observations to the left and the noise terms to the right, and merging the $\Delta\tilde{\mathbf{R}}$ terms within the noise, we obtain:

$$\begin{aligned}\Delta\mathbf{R}_{ij} &= \Delta\tilde{\mathbf{R}}_{ij} \prod_{k=i}^{j-1} \text{Exp}(-\Delta\tilde{\mathbf{R}}_{k+1,j}^\top \mathbf{J}_{r,k}\boldsymbol{\eta}_{gd,k}\Delta t), \\ &\doteq \Delta\tilde{\mathbf{R}}_{ij}\text{Exp}(-\delta\phi_{ij}).\end{aligned}\quad (4.11)$$

For convenience, we collectively define the right-hand side as a noise term. Later, we will discuss the magnitude of this noise term.

Next, consider the velocity part. The form of Equation (4.7) remains unchanged, but now we can happily substitute the previously defined $\Delta\tilde{\mathbf{R}}_{ij}$:

$$\begin{aligned}\Delta\mathbf{v}_{ij} &= \sum_{k=i}^{j-1} \Delta\mathbf{R}_{ik}(\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i} - \boldsymbol{\eta}_{ad,k})\Delta t, \\ &= \sum_{k=i}^{j-1} \Delta\tilde{\mathbf{R}}_{ik} \underbrace{\text{Exp}(-\delta\phi_{ik})}_{\approx \mathbf{I} - \delta\phi_{ik}^\wedge} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i} - \boldsymbol{\eta}_{ad,k})\Delta t, \\ &= \sum_{k=i}^{j-1} \Delta\tilde{\mathbf{R}}_{ik}(\mathbf{I} - \delta\phi_{ik}^\wedge)(\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i} - \boldsymbol{\eta}_{ad,k})\Delta t.\end{aligned}\quad (4.12)$$

We discard the second-order noise terms and define the **pre-integrated velocity measurement** as:

$$\Delta\tilde{\mathbf{v}}_{ij} = \sum_{k=i}^{j-1} \Delta\tilde{\mathbf{R}}_{ik}(\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})\Delta t. \quad (4.13)$$

Thus, the equation can be simplified to:

$$\begin{aligned}\Delta\mathbf{v}_{ij} &= \sum_{k=i}^{j-1} \left[\underbrace{\Delta\tilde{\mathbf{R}}_{ik}(\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})\Delta t + \Delta\tilde{\mathbf{R}}_{ik}(\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \delta\phi_{ik}\Delta t - \Delta\tilde{\mathbf{R}}_{ik}\boldsymbol{\eta}_{ad,k}\Delta t}_{\text{Accumulate this term}} \right], \\ &= \Delta\tilde{\mathbf{v}}_{ij} + \sum_{k=i}^{j-1} [\Delta\tilde{\mathbf{R}}_{ik}(\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \delta\phi_{ik}\Delta t - \Delta\tilde{\mathbf{R}}_{ik}\boldsymbol{\eta}_{ad,k}\Delta t], \\ &= \Delta\tilde{\mathbf{v}}_{ij} - \delta\mathbf{v}_{ij}.\end{aligned}\quad (4.14)$$

Similarly, $\delta\mathbf{v}_{ij}$ is also a defined noise term, whose magnitude we will analyze later.

Finally, we can define similar operations for the translation component. Substituting Equations (4.11) and (4.14) into the translation definition yields:

$$\begin{aligned}
\Delta \mathbf{p}_{ij} &= \sum_{k=i}^{j-1} \left[\Delta \mathbf{v}_{ik} \Delta t + \frac{1}{2} \Delta \mathbf{R}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i} - \boldsymbol{\eta}_{ad,k}) \Delta t^2 \right], \\
&= \sum_{k=i}^{j-1} \left[(\Delta \tilde{\mathbf{v}}_{ik} - \delta \mathbf{v}_{ik}) \Delta t + \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} \underbrace{\text{Exp}(-\delta \phi_{ik})}_{\mathbf{I} - \delta \phi_{ik}^\wedge} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i} - \boldsymbol{\eta}_{ad,k}) \Delta t^2 \right], \\
&\approx \sum_{k=i}^{j-1} \left[(\Delta \tilde{\mathbf{v}}_{ik} - \delta \mathbf{v}_{ik}) \Delta t + \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} (\mathbf{I} - \delta \phi_{ik}^\wedge) (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i}) \Delta t^2 - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} \boldsymbol{\eta}_{ad,k} \Delta t^2 \right], \\
&\approx \sum_{k=i}^{j-1} \left[\Delta \tilde{\mathbf{v}}_{ik} \Delta t + \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i}) \Delta t^2 - \delta \mathbf{v}_{ik} \Delta t + \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \delta \phi_{ik} \Delta t^2 - \right. \\
&\quad \left. \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} \boldsymbol{\eta}_{ad,k} \Delta t^2 \right]. \tag{4.15}
\end{aligned}$$

In the derivation from the third to fourth line, we discarded second-order noise terms. As before, we define the **pre-integrated position measurement** as:

$$\Delta \tilde{\mathbf{p}}_{ij} = \sum_{k=i}^{j-1} \left[(\Delta \tilde{\mathbf{v}}_{ik} \Delta t) + \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i}) \Delta t^2 \right]. \tag{4.16}$$

The previous equation can then be written as:

$$\begin{aligned}
\Delta \mathbf{p}_{ij} &= \Delta \tilde{\mathbf{p}}_{ij} + \sum_{k=i}^{j-1} \left[-\delta \mathbf{v}_{ik} \Delta t + \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \delta \phi_{ik} \Delta t^2 - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} \boldsymbol{\eta}_{ad,k} \Delta t^2 \right], \\
&\doteq \Delta \tilde{\mathbf{p}}_{ij} - \delta \mathbf{p}_{ij}. \tag{4.17}
\end{aligned}$$

Thus, Equations (4.11), (4.14), and (4.17) collectively define the three pre-integration measurements and their corresponding noise terms. Substituting these back into the original definition (4.7), we can concisely express:

$$\Delta \tilde{\mathbf{R}}_{ij} = \mathbf{R}_i^\top \mathbf{R}_j \text{Exp}(\delta \phi_{ij}), \tag{4.18a}$$

$$\Delta \tilde{\mathbf{v}}_{ij} = \mathbf{R}_i^\top (\mathbf{v}_j - \mathbf{v}_i - \mathbf{g} \Delta t_{ij}) + \delta \mathbf{v}_{ij}, \tag{4.18b}$$

$$\Delta \tilde{\mathbf{p}}_{ij} = \mathbf{R}_i^\top \left(\mathbf{p}_j - \mathbf{p}_i - \mathbf{v}_i \Delta t_{ij} - \frac{1}{2} \mathbf{g} \Delta t_{ij}^2 \right) + \delta \mathbf{p}_{ij}. \tag{4.18c}$$

This formulation summarizes our previous discussion and highlights several key advantages of pre-integration:

1. The left-hand side represents observable quantities obtainable through sensor data integration, while the right-hand side shows predicted values derived from state variables plus (or multiplied by) random noise terms.
2. The definition of left-hand variables is particularly suitable for implementation. $\Delta \tilde{\mathbf{R}}_{ik}$ can be obtained from IMU readings, $\Delta \tilde{\mathbf{v}}_{ik}$ can be calculated using IMU readings at time k and $\Delta \tilde{\mathbf{R}}_{ik}$, and $\Delta \tilde{\mathbf{p}}_{ik}$ can be derived from these two. Moreover, knowing the

pre-integration measurements at time k makes it straightforward to compute those at $k + 1$ using new sensor readings, thanks to the cumulative nature of the definitions.

3. From the right-hand perspective, it's easy to predict measurement values based on state variables at times i and j , enabling error formulation for least-squares optimization. The remaining questions are: Does the pre-integration noise follow zero-mean Gaussian distribution? If so, what is its covariance? And how does it relate to the intrinsic IMU noise?

We will address these questions next.

Next, let's consider the velocity component. Similar to the rotation part, the velocity component can also be expressed as a linear combination of Gaussian noise variables:

$$\delta \mathbf{v}_{ij} \approx \sum_{k=i}^{j-1} [-\Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \delta \phi_{ik} \Delta t + \Delta \tilde{\mathbf{R}}_{ik} \boldsymbol{\eta}_{ad,k} \Delta t]. \quad (4.19)$$

It can also be written in cumulative form:

$$\begin{aligned} \delta \mathbf{v}_{ij} &= \sum_{k=i}^{j-1} [-\Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \delta \phi_{ik} \Delta t + \Delta \tilde{\mathbf{R}}_{ik} \boldsymbol{\eta}_{ad,k} \Delta t], \\ &= \sum_{k=i}^{j-2} [-\Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \delta \phi_{ik} \Delta t + \Delta \tilde{\mathbf{R}}_{ik} \boldsymbol{\eta}_{ad,k} \Delta t] \\ &\quad - \Delta \tilde{\mathbf{R}}_{i,j-1} (\tilde{\mathbf{a}}_{j-1} - \mathbf{b}_{a,i})^\wedge \delta \phi_{i,j-1} \Delta t + \Delta \tilde{\mathbf{R}}_{i,j-1} \boldsymbol{\eta}_{ad,j-1} \Delta t, \\ &= \delta \mathbf{v}_{i,j-1} - \Delta \tilde{\mathbf{R}}_{i,j-1} (\tilde{\mathbf{a}}_{j-1} - \mathbf{b}_{a,i})^\wedge \delta \phi_{i,j-1} \Delta t + \Delta \tilde{\mathbf{R}}_{i,j-1} \boldsymbol{\eta}_{ad,j-1} \Delta t. \end{aligned} \quad (4.20)$$

Thus, the covariance of $\delta \mathbf{v}_{ij}$ can also be determined based on the cumulative coefficients.

The same treatment can be applied to the translation component. We directly present the cumulative form of the translation noise:

$$\begin{aligned} \delta \mathbf{p}_{ij} &= \sum_{k=i}^{j-1} \left[\delta \mathbf{v}_{ik} \Delta t - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \delta \phi_{ik} \Delta t^2 + \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} \boldsymbol{\eta}_{ad,k} \Delta t^2 \right], \\ &= \sum_{k=i}^{j-2} \left[\delta \mathbf{v}_{ik} \Delta t - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \delta \phi_{ik} \Delta t^2 + \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} \boldsymbol{\eta}_{ad,k} \Delta t^2 \right] \\ &\quad + \delta \mathbf{v}_{i,j-1} \Delta t - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{i,j-1} (\tilde{\mathbf{a}}_{j-1} - \mathbf{b}_{a,i})^\wedge \delta \phi_{i,j-1} \Delta t^2 + \frac{1}{2} \Delta \tilde{\mathbf{R}}_{i,j-1} \boldsymbol{\eta}_{ad,j-1} \Delta t^2, \\ &= \delta \mathbf{p}_{i,j-1} + \delta \mathbf{v}_{i,j-1} \Delta t - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{i,j-1} (\tilde{\mathbf{a}}_{j-1} - \mathbf{b}_{a,i})^\wedge \delta \phi_{i,j-1} \Delta t^2 + \frac{1}{2} \Delta \tilde{\mathbf{R}}_{i,j-1} \boldsymbol{\eta}_{ad,j-1} \Delta t^2. \end{aligned} \quad (4.21)$$

Thus, we have derived how to propagate the noise terms from time $j - 1$ to time j . For readers who prefer matrix form, we can easily organize this into matrix notation. For convenience, let's combine these three noise terms into one:

$$\boldsymbol{\eta}_{ik} = \begin{bmatrix} \delta \phi_{ik} \\ \delta \mathbf{v}_{ik} \\ \delta \mathbf{p}_{ik} \end{bmatrix}, \quad (4.22)$$

and define the IMU bias noise as:

$$\boldsymbol{\eta}_{d,j} = \begin{bmatrix} \boldsymbol{\eta}_{gd,j} \\ \boldsymbol{\eta}_{ad,j} \end{bmatrix}, \quad (4.23)$$

then the recursive formula from $\boldsymbol{\eta}_{i,j-1}$ to $\boldsymbol{\eta}_{i,j}$ can be written as:

$$\boldsymbol{\eta}_{ij} = \mathbf{A}_{j-1}\boldsymbol{\eta}_{i,j-1} + \mathbf{B}_{j-1}\boldsymbol{\eta}_{d,j-1}, \quad (4.24)$$

where the coefficient matrices \mathbf{A}_{j-1} and \mathbf{B}_{j-1} are:

$$\mathbf{A}_{j-1} = \begin{bmatrix} \Delta\tilde{\mathbf{R}}_{j-1,j}^\top & \mathbf{0} & \mathbf{0} \\ -\Delta\tilde{\mathbf{R}}_{i,j-1}(\tilde{\mathbf{a}}_{j-1} - \mathbf{b}_{a,i})^\wedge\Delta t & \mathbf{I} & \mathbf{0} \\ -\frac{1}{2}\Delta\tilde{\mathbf{R}}_{i,j-1}(\tilde{\mathbf{a}}_{j-1} - \mathbf{b}_{a,i})^\wedge\Delta t^2 & \Delta t\mathbf{I} & \mathbf{I} \end{bmatrix}, \quad \mathbf{B}_{j-1} = \begin{bmatrix} \mathbf{J}_{r,j-1}\Delta t & \mathbf{0} \\ \mathbf{0} & \Delta\tilde{\mathbf{R}}_{i,j-1}\Delta t \\ \mathbf{0} & \frac{1}{2}\Delta\tilde{\mathbf{R}}_{i,j-1}\Delta t^2 \end{bmatrix}. \quad (4.25)$$

The matrix form more clearly shows the cumulative recursive relationship between the noise terms. If we record the noise in covariance form, then with each additional IMU observation, the noise should exhibit a gradually increasing relationship:

$$\boldsymbol{\Sigma}_{i,k+1} = \mathbf{A}_{k+1}\boldsymbol{\Sigma}_{i,k}\mathbf{A}_{k+1}^\top + \mathbf{B}_{k+1}\text{Cov}(\boldsymbol{\eta}_{d,k})\mathbf{B}_{k+1}^\top, \quad (4.26)$$

Here, the \mathbf{A}_{k+1} matrix is close to the identity matrix \mathbf{I} , so it can be viewed as accumulating the noise. The gyroscope noise enters the rotation measurements through the \mathbf{B} matrix, while the accelerometer noise mainly affects the velocity and translation estimates. This cumulative relationship is easily implemented in programs. Later, we will see their implementation in the experimental chapters. Note that if the order of residual terms in the pre-integration definition changes, we also need to adjust the system matrix rows and columns here to maintain consistency.

4.2.3 Bias Update

Previous discussions assumed constant IMU biases at time i for computational convenience. However, in practical graph optimization, we frequently update state variables (optimization variables). Theoretically, if IMU biases change, pre-integration should be recomputed since each step depends on the biases at time i . However, we can adopt an approximate approach: **assuming pre-integration measurements vary linearly with bias**¹, then correcting the original measurements. Specifically, treating pre-integration measurements as functions of $\mathbf{b}_{g,i}$, $\mathbf{b}_{a,i}$, when biases update by $\delta\mathbf{b}_{g,i}$, $\delta\mathbf{b}_{a,i}$, the measurements should be corrected as:

$$\begin{aligned} \Delta\tilde{\mathbf{R}}_{ij}(\mathbf{b}_{g,i} + \delta\mathbf{b}_{g,i}) &= \Delta\tilde{\mathbf{R}}_{ij}(\mathbf{b}_{g,i})\text{Exp}\left(\frac{\partial\Delta\tilde{\mathbf{R}}_{ij}}{\partial\mathbf{b}_{g,i}}\delta\mathbf{b}_{g,i}\right), \\ \Delta\tilde{\mathbf{v}}_{ij}(\mathbf{b}_{g,i} + \delta\mathbf{b}_{g,i}, \mathbf{b}_{a,i} + \delta\mathbf{b}_{a,i}) &= \Delta\tilde{\mathbf{v}}_{ij}(\mathbf{b}_{g,i}, \mathbf{b}_{a,i}) + \frac{\partial\Delta\tilde{\mathbf{v}}_{ij}}{\partial\mathbf{b}_{g,i}}\delta\mathbf{b}_{g,i} + \frac{\partial\Delta\tilde{\mathbf{v}}_{ij}}{\partial\mathbf{b}_{a,i}}\delta\mathbf{b}_{a,i}, \\ \Delta\tilde{\mathbf{p}}_{ij}(\mathbf{b}_{g,i} + \delta\mathbf{b}_{g,i}, \mathbf{b}_{a,i} + \delta\mathbf{b}_{a,i}) &= \Delta\tilde{\mathbf{p}}_{ij}(\mathbf{b}_{g,i}, \mathbf{b}_{a,i}) + \frac{\partial\Delta\tilde{\mathbf{p}}_{ij}}{\partial\mathbf{b}_{g,i}}\delta\mathbf{b}_{g,i} + \frac{\partial\Delta\tilde{\mathbf{p}}_{ij}}{\partial\mathbf{b}_{a,i}}\delta\mathbf{b}_{a,i}. \end{aligned} \quad (4.27)$$

The problem reduces to computing these partial derivatives (Jacobians). This process resembles our earlier noise variable linearization.

For rotation:

$$\begin{aligned} \Delta\tilde{\mathbf{R}}_{ij}(\mathbf{b}_{g,i} + \delta\mathbf{b}_{g,i}) &= \prod_{k=i}^{j-1} \text{Exp}((\tilde{\omega}_k - (\mathbf{b}_{g,i} + \delta\mathbf{b}_{g,i}))\Delta t) \\ &\approx \Delta\tilde{\mathbf{R}}_{ij}\text{Exp}\left(-\sum_{k=i}^{j-1} \Delta\tilde{\mathbf{R}}_{k+1,j}^\top \mathbf{J}_{r,k} \Delta t \delta\mathbf{b}_{g,i}\right). \end{aligned} \quad (4.28)$$

¹While not strictly linear, we can always linearize complex functions by retaining first-order terms.

The Jacobian is:

$$\frac{\partial \Delta \tilde{\mathbf{R}}_{ij}}{\partial \mathbf{b}_{g,i}} = - \sum_{k=i}^{j-1} \Delta \tilde{\mathbf{R}}_{k+1,j}^\top \mathbf{J}_{r,k} \Delta t. \quad (4.29)$$

The recursive form is:

$$\frac{\partial \Delta \tilde{\mathbf{R}}_{ij}}{\partial \mathbf{b}_{g,i}} = \Delta \tilde{\mathbf{R}}_{j-1,j}^\top \frac{\partial \Delta \tilde{\mathbf{R}}_{i,j-1}}{\partial \mathbf{b}_{g,i}} - \mathbf{J}_{r,k} \Delta t. \quad (4.30)$$

For velocity:

$$\begin{aligned} \Delta \tilde{\mathbf{v}}_{ij}(\mathbf{b}_i + \delta \mathbf{b}_i) &\approx \Delta \tilde{\mathbf{v}}_{ij} - \sum_{k=i}^{j-1} \Delta \tilde{\mathbf{R}}_{ik} \Delta t \delta \mathbf{b}_{a,i} \\ &\quad - \sum_{k=i}^{j-1} \Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \frac{\partial \Delta \tilde{\mathbf{R}}_{ik}}{\partial \mathbf{b}_{g,i}} \Delta t \delta \mathbf{b}_{g,i}. \end{aligned} \quad (4.31)$$

For position:

$$\begin{aligned} \Delta \tilde{\mathbf{p}}_{ij}(\mathbf{b}_i + \delta \mathbf{b}_i) &\approx \Delta \tilde{\mathbf{p}}_{ij} + \sum_{k=i}^{j-1} \left[\frac{\partial \Delta \mathbf{v}_{ik}}{\partial \mathbf{b}_{a,i}} \Delta t - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} \Delta t^2 \right] \delta \mathbf{b}_{a,i} \\ &\quad + \sum_{k=i}^{j-1} \left[\frac{\partial \Delta \mathbf{v}_{ik}}{\partial \mathbf{b}_{g,i}} \Delta t - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \frac{\partial \Delta \tilde{\mathbf{R}}_{ik}}{\partial \mathbf{b}_{g,i}} \Delta t^2 \right] \delta \mathbf{b}_{g,i}. \end{aligned} \quad (4.32)$$

The complete Jacobians are:

$$\frac{\partial \Delta \tilde{\mathbf{R}}_{ij}}{\partial \mathbf{b}_{g,i}} = - \sum_{k=i}^{j-1} [\Delta \tilde{\mathbf{R}}_{k+1,j}^\top \mathbf{J}_{r,k} \Delta t], \quad (4.33a)$$

$$\frac{\partial \Delta \tilde{\mathbf{v}}_{ij}}{\partial \mathbf{b}_{a,i}} = - \sum_{k=i}^{j-1} \Delta \tilde{\mathbf{R}}_{ik} \Delta t, \quad (4.33b)$$

$$\frac{\partial \Delta \tilde{\mathbf{v}}_{ij}}{\partial \mathbf{b}_{g,i}} = - \sum_{k=i}^{j-1} \Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \frac{\partial \Delta \tilde{\mathbf{R}}_{ik}}{\partial \mathbf{b}_{g,i}} \Delta t, \quad (4.33c)$$

$$\frac{\partial \Delta \tilde{\mathbf{p}}_{ij}}{\partial \mathbf{b}_{a,i}} = \sum_{k=i}^{j-1} \left[\frac{\partial \Delta \tilde{\mathbf{v}}_{ik}}{\partial \mathbf{b}_{a,i}} \Delta t - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} \Delta t^2 \right], \quad (4.33d)$$

$$\frac{\partial \Delta \tilde{\mathbf{p}}_{ij}}{\partial \mathbf{b}_{g,i}} = \sum_{k=i}^{j-1} \left[\frac{\partial \Delta \tilde{\mathbf{v}}_{ik}}{\partial \mathbf{b}_{g,i}} \Delta t - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{ik} (\tilde{\mathbf{a}}_k - \mathbf{b}_{a,i})^\wedge \frac{\partial \Delta \tilde{\mathbf{R}}_{ik}}{\partial \mathbf{b}_{g,i}} \Delta t^2 \right]. \quad (4.33e)$$

The recursive forms are:

$$\frac{\partial \Delta \tilde{\mathbf{R}}_{ij}}{\partial \mathbf{b}_{g,i}} = \Delta \tilde{\mathbf{R}}_{j-1,j}^\top \frac{\partial \Delta \tilde{\mathbf{R}}_{i,j-1}}{\partial \mathbf{b}_{g,i}} - \mathbf{J}_{r,k} \Delta t, \quad (4.34a)$$

$$\frac{\partial \Delta \tilde{\mathbf{v}}_{ij}}{\partial \mathbf{b}_{a,i}} = \frac{\partial \Delta \tilde{\mathbf{v}}_{i,j-1}}{\partial \mathbf{b}_{a,i}} - \Delta \tilde{\mathbf{R}}_{i,j-1} \Delta t, \quad (4.34b)$$

$$\frac{\partial \Delta \tilde{\mathbf{v}}_{ij}}{\partial \mathbf{b}_{g,i}} = \frac{\partial \Delta \tilde{\mathbf{v}}_{i,j-1}}{\partial \mathbf{b}_{g,i}} - \Delta \tilde{\mathbf{R}}_{i,j-1} (\tilde{\mathbf{a}}_{j-1} - \mathbf{b}_{a,i})^\wedge \frac{\partial \Delta \tilde{\mathbf{R}}_{i,j-1}}{\partial \mathbf{b}_{g,i}} \Delta t, \quad (4.34c)$$

$$\frac{\partial \Delta \tilde{\mathbf{p}}_{ij}}{\partial \mathbf{b}_{a,i}} = \frac{\partial \Delta \tilde{\mathbf{p}}_{i,j-1}}{\partial \mathbf{b}_{a,i}} + \frac{\partial \Delta \tilde{\mathbf{v}}_{i,j-1}}{\partial \mathbf{b}_{a,i}} \Delta t - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{i,j-1} \Delta t^2, \quad (4.34d)$$

$$\frac{\partial \Delta \tilde{\mathbf{p}}_{ij}}{\partial \mathbf{b}_{g,i}} = \frac{\partial \Delta \tilde{\mathbf{p}}_{i,j-1}}{\partial \mathbf{b}_{g,i}} + \frac{\partial \Delta \tilde{\mathbf{v}}_{i,j-1}}{\partial \mathbf{b}_{g,i}} \Delta t - \frac{1}{2} \Delta \tilde{\mathbf{R}}_{i,j-1} (\tilde{\mathbf{a}}_{j-1} - \mathbf{b}_{a,i})^\wedge \frac{\partial \Delta \tilde{\mathbf{R}}_{i,j-1}}{\partial \mathbf{b}_{g,i}} \Delta t^2. \quad (4.34e)$$

4.2.4 Preintegration Model Reduced to Graph Optimization

Now that we have defined the measurement model of preintegration, derived its noise model and covariance matrix, and explained how preintegration should be updated with bias updates. In fact, we can already use the preintegration observation as a Factor or Edge in graph optimization. Below we explain how to use such edges and derive their Jacobian matrices with respect to state variables.

In IMU-related applications, the state at each time is typically modeled as variables containing rotation, translation, linear velocity, and IMU biases, forming the state variable set \mathcal{X} :

$$\mathbf{x}_k = [\mathbf{R}, \mathbf{p}, \mathbf{v}, \mathbf{b}_a, \mathbf{b}_g]_k \in \mathcal{X}, \quad (4.35)$$

while the preintegration model establishes a constraint between keyframe i and keyframe j . The observation model of preintegration itself has been introduced in (4.7). We can use the state variables at time i and j along with the preintegration measurements to compute the difference, resulting in the residual definition formula. It should be noted that the specific computation of residuals varies across different literature, and some papers and implementations may not be consistent. The actual definition of residuals is quite flexible, and the corresponding Jacobian matrices also differ, with some forms being relatively simpler. A common approach is to directly use the definition for derivation or to use the state at time i and preintegration to predict the state at time j , then compute the difference with the estimated state at j to obtain the residual (left as an exercise; note that changing the residual definition may alter the corresponding noise covariance). Following the approach in [67], we define the residuals as:

$$\mathbf{r}_{\Delta \mathbf{R}_{ij}} = \text{Log}(\Delta \tilde{\mathbf{R}}_{ij}^\top (\mathbf{R}_i^\top \mathbf{R}_j)), \quad (4.36a)$$

$$\mathbf{r}_{\Delta \mathbf{v}_{ij}} = \mathbf{R}_i^\top (\mathbf{v}_j - \mathbf{v}_i - \mathbf{g} \Delta t_{ij}) - \Delta \tilde{\mathbf{v}}_{ij}, \quad (4.36b)$$

$$\mathbf{r}_{\Delta \mathbf{p}_{ij}} = \mathbf{R}_i^\top \left(\mathbf{p}_j - \mathbf{p}_i - \mathbf{v}_i \Delta t_{ij} - \frac{1}{2} \mathbf{g} \Delta t_{ij}^2 \right) - \Delta \tilde{\mathbf{p}}_{ij}. \quad (4.36c)$$

Typically, we write \mathbf{r} as a unified 9-dimensional residual variable. It appears to connect the rotation, translation, and linear velocity of two timesteps, but since the preintegration observation internally contains IMU biases, it is also related to the two biases at time i . During optimization, if the biases at time i are updated, the preintegration measurements should also change linearly, affecting the residual values. Thus, although the residual term

does not explicitly contain $\mathbf{b}_{a,i}, \mathbf{b}_{g,i}$, they are clearly related to the residual. Therefore, if we treat the preintegration residual as a single entity, its connection to state vertices should be as shown in Figure 4-1 . In addition to the preintegration factor itself, the random walk of the IMU must also be constrained, so there will be a constraint factor between the biases at different IMU timesteps.

Please note that earlier we discussed the linear form of preintegration measurements with respect to IMU biases, so in the preintegration residual definition, we could also substitute the linear approximation listed earlier to obtain the Jacobian of the preintegration factor with respect to IMU biases. Since this would complicate the expression, we do not explicitly expand it here.

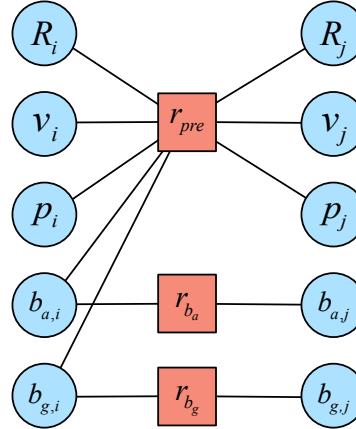


Figure 4-1: Graph optimization form of the preintegration factor

Apart from placing all state variables in the same vertex, we can also choose a "disassembled form," where rotation, translation, linear velocity, and the two biases are each constructed as separate vertices, and then the Jacobians between these vertices are computed. If this approach is adopted, the number of Jacobian matrices will increase, but the dimensionality of individual Jacobians can be reduced (single Jacobians are typically 3×3 , while the Jacobian of preintegration observations with respect to state variables becomes 9×15 , with many zero blocks). Moreover, if we need to implement a tightly coupled visual or LiDAR system, since visual and LiDAR observation constraints are usually only related to \mathbf{R}, \mathbf{t} , some zero blocks in the Jacobian matrices can be avoided. In summary, both approaches have their merits. The code implementation in this book adopts the disassembled approach, where each state variable is written as a separate vertex to constrain them individually.

4.2.5 Jacobians of Preintegration

Finally, we discuss the Jacobian matrices of preintegration with respect to the state variables. Since the preintegration measurements already summarize the IMU readings over a short period, the derivation of the residuals' Jacobians relative to the state variables becomes straightforward. Below, we derive them.

First, consider rotation. Rotation depends on \mathbf{R}_i , \mathbf{R}_j , and $\mathbf{b}_{g,i}$. We derive it using the

right perturbation on $\text{SO}(3)$:

$$\begin{aligned}\mathbf{r}_{\Delta \mathbf{R}_{ij}}(\mathbf{R}_i \text{Exp}(\phi_i)) &= \text{Log}(\Delta \tilde{\mathbf{R}}_{ij}^\top ((\mathbf{R}_i \text{Exp}(\phi_i))^\top \mathbf{R}_j)), \\ &= \text{Log}(\Delta \tilde{\mathbf{R}}_{ij}^\top \text{Exp}(-\phi_i) \mathbf{R}_i^\top \mathbf{R}_j), \\ &= \text{Log}(\Delta \tilde{\mathbf{R}}_{ij}^\top \mathbf{R}_i^\top \mathbf{R}_j \text{Exp}(-\mathbf{R}_j^\top \mathbf{R}_i \phi_i)), \\ &= \mathbf{r}_{\Delta \mathbf{R}_{ij}} - \mathbf{J}_r^{-1}(\mathbf{r}_{\Delta \mathbf{R}_{ij}}) \mathbf{R}_j^\top \mathbf{R}_i \phi_i.\end{aligned}\quad (4.37)$$

The derivative with respect to ϕ_j is:

$$\begin{aligned}\mathbf{r}_{\Delta \mathbf{R}_{ij}}(\mathbf{R}_j \text{Exp}(\phi_j)) &= \text{Log}(\Delta \tilde{\mathbf{R}}_{ij}^\top \mathbf{R}_i^\top \mathbf{R}_j \text{Exp}(\phi_j)), \\ &= \mathbf{r}_{\Delta \mathbf{R}_{ij}} + \mathbf{J}_r^{-1}(\mathbf{r}_{\Delta \mathbf{R}_{ij}}) \phi_j.\end{aligned}\quad (4.38)$$

These derivations closely resemble those in pose graphs. However, dealing with the bias terms is slightly more involved. Note that during optimization, the bias terms are continuously updated, and each update uses (4.27) to correct the preintegration measurements. Since this process is iterative, we always have an initial measurement and a corrected measurement, which must be taken into account during the derivation.

4.2.6 Jacobians of Preintegration

Assume the initial bias for optimization is $\mathbf{b}_{g,i}$. At a certain iteration step, the current estimated bias correction is $\delta \mathbf{b}_{g,i}$, and the corrected preintegrated rotation measurement is $\Delta \tilde{\mathbf{R}}'_{ij} = \Delta \tilde{\mathbf{R}}_{ij}(\mathbf{b}_{g,i} + \delta \mathbf{b}_{g,i})$, with the residual being $\mathbf{r}'_{\Delta \mathbf{R}_{ij}}$. To compute the derivative, we further add $\tilde{\delta} \mathbf{b}_{g,i}$ to these two terms, yielding:

$$\begin{aligned}\mathbf{r}_{\Delta \mathbf{R}_{ij}}(\mathbf{b}_{g,i} + \delta \mathbf{b}_{g,i} + \tilde{\delta} \mathbf{b}_{g,i}) &= \text{Log}\left(\left(\Delta \tilde{\mathbf{R}}_{ij} \text{Exp}\left(\frac{\partial \Delta \tilde{\mathbf{R}}_{ij}}{\partial \mathbf{b}_{g,i}}(\delta \mathbf{b}_{g,i} + \tilde{\delta} \mathbf{b}_{g,i})\right)\right)^\top \mathbf{R}_i^\top \mathbf{R}_j\right), \\ &\stackrel{\text{BCH}}{\approx} \text{Log}\left(\left(\underbrace{\Delta \tilde{\mathbf{R}}_{ij} \text{Exp}\left(\frac{\partial \Delta \tilde{\mathbf{R}}_{ij}}{\partial \mathbf{b}_{g,i}} \delta \mathbf{b}_{g,i}\right)}_{\Delta \tilde{\mathbf{R}}'_{ij}} \text{Exp}\left(\mathbf{J}_{r,b} \frac{\partial \Delta \tilde{\mathbf{R}}_{ij}}{\partial \mathbf{b}_{g,i}} \tilde{\delta} \mathbf{b}_{g,i}\right)\right)^\top \mathbf{R}_i^\top \mathbf{R}_j\right), \\ &= \text{Log}\left(\text{Exp}\left(-\mathbf{J}_{r,b} \frac{\partial \Delta \tilde{\mathbf{R}}_{ij}}{\partial \mathbf{b}_{g,i}} \tilde{\delta} \mathbf{b}_{g,i}\right) \underbrace{(\Delta \tilde{\mathbf{R}}'_{ij})^\top \mathbf{R}_i^\top \mathbf{R}_j}_{\text{Exp}(\mathbf{r}'_{\Delta \mathbf{R}_{ij}})}\right), \\ &= \text{Log}\left(\text{Exp}(\mathbf{r}'_{\Delta \mathbf{R}_{ij}}) \text{Exp}\left(-\text{Exp}(\mathbf{r}'_{\Delta \mathbf{R}_{ij}})^\top \mathbf{J}_{r,b} \frac{\partial \Delta \tilde{\mathbf{R}}_{ij}}{\partial \mathbf{b}_{g,i}} \tilde{\delta} \mathbf{b}_{g,i}\right)\right), \\ &\approx \mathbf{r}'_{\Delta \mathbf{R}_{ij}} - \mathbf{J}_r^{-1}(\mathbf{r}'_{\Delta \mathbf{R}_{ij}}) \text{Exp}\left(\mathbf{r}'_{\Delta \mathbf{R}_{ij}}\right)^\top \mathbf{J}_{r,b} \frac{\partial \Delta \tilde{\mathbf{R}}_{ij}}{\partial \mathbf{b}_{g,i}} \tilde{\delta} \mathbf{b}_{g,i}.\end{aligned}\quad (4.39)$$

Thus, we finally obtain:

$$\frac{\partial \mathbf{r}_{\Delta \mathbf{R}_{ij}}}{\partial \mathbf{b}_{g,i}} = -\mathbf{J}_r^{-1}(\mathbf{r}'_{\Delta \mathbf{R}_{ij}}) \text{Exp}\left(\mathbf{r}'_{\Delta \mathbf{R}_{ij}}\right)^\top \mathbf{J}_{r,b} \frac{\partial \Delta \tilde{\mathbf{R}}_{ij}}{\partial \mathbf{b}_{g,i}}. \quad (4.40)$$

Next, consider the Jacobians for the velocity term. The velocity term is simpler, as it has a linear relationship with \mathbf{v}_i and \mathbf{v}_j , yielding:

$$\frac{\partial \mathbf{r}_{\Delta \mathbf{v}_{ij}}}{\partial \mathbf{v}_i} = -\mathbf{R}_i^\top, \quad (4.41a)$$

$$\frac{\partial \mathbf{r}_{\Delta \mathbf{v}_{ij}}}{\partial \mathbf{v}_j} = \mathbf{R}_i^\top. \quad (4.41b)$$

For the rotation part, a first-order Taylor expansion suffices:

$$\begin{aligned} \mathbf{r}_{\Delta \mathbf{v}_{ij}}(\mathbf{R}_i \text{Exp}(\delta \phi_i)) &= (\mathbf{R}_i \text{Exp}(\delta \phi_i))^\top (\mathbf{v}_j - \mathbf{v}_i - \mathbf{g} \Delta t_{ij}) - \Delta \tilde{\mathbf{v}}_{ij}, \\ &= (\mathbf{I} - \delta \phi_i^\wedge) \mathbf{R}_i^\top (\mathbf{v}_j - \mathbf{v}_i - \mathbf{g} \Delta t_{ij}) - \Delta \tilde{\mathbf{v}}_{ij}, \\ &= \mathbf{r}_{\Delta \mathbf{v}_{ij}}(\mathbf{R}_i) + (\mathbf{R}_i^\top (\mathbf{v}_j - \mathbf{v}_i - \mathbf{g} \Delta t_{ij}))^\wedge \delta \phi_i. \end{aligned} \quad (4.42)$$

The Jacobians of the velocity residual with respect to $\mathbf{b}_{g,i}$ and $\mathbf{b}_{a,i}$ depend only on $\Delta \tilde{\mathbf{v}}_{ij}$. Since the velocity residual term differs from it only by a negative sign, we simply need to add a negative sign to (4.33).

Finally, consider the translation part. The translation term depends linearly on \mathbf{p}_i , \mathbf{p}_j , \mathbf{v}_i , \mathbf{R}_i , and the two biases, making the Jacobians straightforward to derive:

$$\frac{\partial \mathbf{r}_{\Delta \mathbf{p}_{ij}}}{\partial \mathbf{p}_i} = -\mathbf{R}_i^\top, \quad (4.43a)$$

$$\frac{\partial \mathbf{r}_{\Delta \mathbf{p}_{ij}}}{\partial \mathbf{p}_j} = \mathbf{R}_i^\top, \quad (4.43b)$$

$$\frac{\partial \mathbf{r}_{\Delta \mathbf{p}_{ij}}}{\partial \mathbf{v}_i} = -\mathbf{R}_i^\top \Delta t_{ij}, \quad (4.43c)$$

$$\frac{\partial \mathbf{r}_{\Delta \mathbf{p}_{ij}}}{\partial \phi_i} = \left(\mathbf{R}_i^\top \left(\mathbf{p}_j - \mathbf{p}_i - \mathbf{v}_i \Delta t_{ij} - \frac{1}{2} \mathbf{g} \Delta t_{ij}^2 \right) \right)^\wedge. \quad (4.43d)$$

The residual for the biases simply requires adding a negative sign to Equation (4.33).

At this point, we have derived the derivative forms of the preintegrated measurements with respect to all state variables. If desired, we could write the preintegrated observations as a column vector, the state variables as another column vector, and combine all these Jacobian matrices accordingly. To save space, this book presents the preintegration matrices in their decomposed form.

4.2.7 Summary

Finally, we summarize the practical implementation process of the aforementioned preintegration in real-world applications.

In a system composed of keyframes, we can initiate the preintegration process starting from any keyframe at any given time and terminate it at any desired moment. Subsequently, we can extract the preintegrated measurements, noise terms, and various accumulated Jacobians to constrain the states between two keyframes. Based on previous discussions, once preintegration begins, whenever a new IMU measurement arrives, our program should perform the following tasks:

1. Compute the three **preintegrated measurements** using Equation (4.7) based on the previous data: $\Delta \tilde{\mathbf{R}}_{ij}$, $\Delta \tilde{\mathbf{v}}_{ij}$, $\Delta \tilde{\mathbf{p}}_{ij}$;

2. Calculate the **covariance matrices** of the three noise terms, which will serve as the information matrices for subsequent graph optimization;
3. Compute the **Jacobian matrices** of the preintegrated measurements with respect to the biases - five in total;

Upon completion of the preintegration calculations, these results can be extracted and applied during the optimization process.

4.3 Practice: Implementation of Preintegration

4.3.1 Implementing the Preintegration Class

Following the derivations from the previous section, we now implement the preintegration program. The implementation mainly consists of two parts: the computation of preintegration itself, and its integration into graph optimization. The former is relatively simple as it only involves IMU measurements, while the latter depends on the specific graph optimization framework - in this book we'll implement it using the g2o framework.

First, let's implement the preintegration structure itself. A preintegration class should store the following data:

- Preintegrated measurements $\Delta\tilde{\mathbf{R}}_{ij}, \Delta\tilde{\mathbf{p}}_{ij}, \Delta\tilde{\mathbf{v}}_{ij}$;
- IMU biases at the start of preintegration $\mathbf{b}_g, \mathbf{b}_a$;
- Measurement noise covariance $\Sigma_{i,k+1}$ during integration, as specified by Equation (4.26);
- Jacobian matrices of integrated quantities with respect to IMU biases (see Equation (4.34));
- Total integration time Δt_{ij} .

The above are all essential information. Additionally, we may choose to store IMU measurements in the preintegration class (though this is optional since they're already integrated). The IMU measurement noise and bias random walk noise can also be included as configuration parameters. Here's the basic implementation of such a preintegration class:

Listing 4.1: src/ch4/imu_preintegration.h

```

1 class IMUPreintegration {
2     public:
3         /// Other functions omitted
4         struct Options {
5             Options() {}
6             Vec3d init_bg_ = Vec3d::Zero(); // Initial bias
7             Vec3d init_ba_ = Vec3d::Zero(); // Initial bias
8             double noise_gyro_ = 1e-2; // Gyro noise (std dev)
9             double noise_acce_ = 1e-1; // Accelerometer noise (std dev)
10        };
11
12    public:
13        double dt_ = 0; // Total preintegration time
14        Mat9d cov_ = Mat9d::Zero(); // Accumulated noise matrix
15        Mat6d noise_gyro_acce_ = Mat6d::Zero(); // Measurement noise matrix
16
17        // Biases
18        Vec3d bg_ = Vec3d::Zero(); // Initial bias
19        Vec3d ba_ = Vec3d::Zero(); // Initial bias
20

```

```

21 // Preintegrated measurements
22 SO3 dR_;
23 Vec3d dv_ = Vec3d::Zero();
24 Vec3d dp_ = Vec3d::Zero();
25
26 // Jacobian matrices
27 Mat3d dR_dbg_ = Mat3d::Zero();
28 Mat3d dV_dbg_ = Mat3d::Zero();
29 Mat3d dV_dba_ = Mat3d::Zero();
30 Mat3d dP_dbg_ = Mat3d::Zero();
31 Mat3d dP_dba_ = Mat3d::Zero();
32 };

```

The variables maintained by this class correspond to those introduced earlier. Note that the IMU bias-related noise terms are not directly part of the preintegration class - we'll move them to the optimization class. This class mainly handles the preintegration of IMU data and provides the integrated measurements and noise values.

4.4 Practice: Implementation of Preintegration

4.4.1 Implementing the Single IMU Integration

The implementation of single IMU integration is as follows:

Listing 4.2: src/ch4 imu_preintegration.cc

```

1 void IMUPreintegration::Integrate(const IMU &imu, double dt) {
2     // Remove bias from measurements
3     Vec3d gyr = imu.gyro_ - bg_; // Gyroscope
4     Vec3d acc = imu.acce_ - ba_; // Accelerometer
5
6     // Update dv, dp, see (4.7)
7     dp_ = dp_ + dv_ * dt + 0.5f * dR_.matrix() * acc * dt * dt;
8     dv_ = dv_ + dR_ * acc * dt;
9
10    // dR update deferred as current dR needed for A, B matrices
11
12    // Jacobian coefficients for motion equation, matrices A,B, see (4.29)
13    // Other terms handled later
14    Eigen::Matrix<double, 9, 9> A;
15    A.setIdentity();
16    Eigen::Matrix<double, 9, 6> B;
17    B.setZero();
18
19    Mat3d acc_hat = SO3::hat(acc);
20    double dt2 = dt * dt;
21
22    // NOTE: Top-left blocks of A, B differ slightly from formula
23    A.block<3, 3>(3, 0) = -dR_.matrix() * dt * acc_hat;
24    A.block<3, 3>(6, 0) = -0.5f * dR_.matrix() * acc_hat * dt2;
25    A.block<3, 3>(6, 3) = dt * Mat3d::Identity();
26
27    B.block<3, 3>(3, 3) = dR_.matrix() * dt;
28    B.block<3, 3>(6, 3) = 0.5f * dR_.matrix() * dt2;
29
30    // Update Jacobians, see (4.39)
31    dP_dba_ = dP_dba_ + dV_dba_ * dt - 0.5f * dR_.matrix() * dt2;
32        // (4.39d)
33    dP_dbg_ = dP_dbg_ + dV_dbg_ * dt - 0.5f * dR_.matrix() * dt2 * acc_hat * dR_dbg_;
34        // (4.39e)
35    dV_dba_ = dV_dba_ - dR_.matrix() * dt;
36        // (4.39b)
37    dV_dbg_ = dV_dbg_ - dR_.matrix() * dt * acc_hat * dR_dbg_;
38        // (4.39c)
39
40    // Rotation part
41    Vec3d omega = gyr * dt;           // Rotation amount
42    Mat3d rightJ = SO3::jr(omega);   // Right Jacobian

```

```

39 SO3 deltaR = SO3::exp(omega); // After exp
40 dR_ = dR_ * deltaR; // (4.7a)
41
42 A.block<3, 3>(0, 0) = deltaR.matrix().transpose();
43 B.block<3, 3>(0, 0) = rightJ * dt;
44
45 // Update noise term
46 cov_ = A * cov_ * A.transpose() + B * noise_gyro_acce_ * B.transpose();
47
48 // Update dR_dbg
49 dR_dbg_ = deltaR.matrix().transpose() * dR_dbg_ - rightJ * dt; // (4.39a)
50
51 // Increment integration time
52 dt_ += dt;
53

```

We've added equation numbers in the code comments to help readers locate corresponding formulas. Overall, it updates internal member variables in the following order:

1. Updates position and velocity measurements;
 2. Updates noise matrix for motion model;
 3. Updates Jacobians of measurements with respect to biases;
 4. Updates rotation measurements;
 5. Updates integration time.

This completes one IMU data operation. Note that without optimization, preintegration and direct integration produce identical results - both integrate IMU data. After preintegration, we can predict from initial to final state similar to ESKF. The prediction function is straightforward:

Listing 4.3: src/ch4 imu_preintegration.cc

```
1 NavStated IMUPreintegration::Predict(const sad::NavStated &start, const Vec3d &grav) {
2     SO3 Rj = start.R_ * dR_;
3     Vec3d vj = start.R_ * dv_ + start.v_ + grav * dt_;
4     Vec3d pj = start.R_ * dp_ + start.p_ + start.v_ * dt_ + 0.5f * grav * dt_ * dt_;
5
6     auto state = NavStated(start.timestamp_ + dt_, Rj, pj, vj);
7     state.bg_ = bg_;
8     state.ba_ = ba_;
9     return state;
10 }
```

Unlike ESKF, preintegration can predict multiple IMU measurements and predict forward from any starting time, while ESKF typically only predicts from current state to next timestamp for single IMU measurement.

Next, we write a test program to verify whether there is a significant difference between preintegration and direct integration when a constant angular velocity and acceleration are present in a single direction. This method is very effective for identifying obvious errors in the code. Since we implemented the ESKF in the previous chapter, we can also compare the prediction process of ESKF with preintegration. If the initial states are the same, the results should be completely consistent.

Listing 4.4: src/ch4/test preintegration.cc

```

1 TEST(PREINTEGRATION_TEST, ROTATION_TEST) {
2     // Test the case of preintegration under constant angular velocity
3     double imu_time_span = 0.01;           // IMU measurement interval
4     Vec3d constant_omega(0, 0, M_PI);    // Angular velocity of 180 deg/s, rotating for 1
5         second equals 180 degrees

```

```

5   Vec3d gravity(0, 0, -9.8);           // Z is upward, gravity is in the negative
6   direction
7
8   sad::NavStated start_status(0), end_status(1.0);
9   sad::IMUPreintegration pre_integ;
10
11  // Compare with direct integration
12  Sophus::SO3d R;
13  Vec3d t = Vec3d::Zero();
14  Vec3d v = Vec3d::Zero();
15
16  for (int i = 1; i <= 100; ++i) {
17      double time = imu_time_span * i;
18      Vec3d acce = -gravity; // Accelerometer should measure an upward force
19      pre_integ.Integrate(sad::IMU(time, constant_omega, acce), imu_time_span);
20
21      sad::NavStated this_status = pre_integ.Predict(start_status, gravity);
22
23      t = t + v * imu_time_span + 0.5 * gravity * imu_time_span * imu_time_span +
24          0.5 * (R * acce) * imu_time_span * imu_time_span;
25      v = v + gravity * imu_time_span + (R * acce) * imu_time_span;
26      R = R * Sophus::SO3d::exp(constant_omega * imu_time_span);
27
28      // Verify that under simple conditions, direct integration and preintegration
29      // yield the same result
30      EXPECT_NEAR(t[0], this_status.p_[0], 1e-2);
31      EXPECT_NEAR(t[1], this_status.p_[1], 1e-2);
32      EXPECT_NEAR(t[2], this_status.p_[2], 1e-2);
33
34      EXPECT_NEAR(v[0], this_status.v_[0], 1e-2);
35      EXPECT_NEAR(v[1], this_status.v_[1], 1e-2);
36      EXPECT_NEAR(v[2], this_status.v_[2], 1e-2);
37
38      EXPECT_NEAR(R.unit_quaternion().x(), this_status.R_.unit_quaternion().x(), 1e-4);
39      EXPECT_NEAR(R.unit_quaternion().y(), this_status.R_.unit_quaternion().y(), 1e-4);
40      EXPECT_NEAR(R.unit_quaternion().z(), this_status.R_.unit_quaternion().z(), 1e-4);
41      EXPECT_NEAR(R.unit_quaternion().w(), this_status.R_.unit_quaternion().w(), 1e-4);
42  }
43
44  end_status = pre_integ.Predict(start_status);
45
46  LOG(INFO) << "preinteg result: ";
47  LOG(INFO) << "end rotation: \n" << end_status.R_.matrix();
48  LOG(INFO) << "end trans: \n" << end_status.p_.transpose();
49  LOG(INFO) << "end v: \n" << end_status.v_.transpose();
50
51  LOG(INFO) << "direct integ result: ";
52  LOG(INFO) << "end rotation: \n" << R.matrix();
53  LOG(INFO) << "end trans: \n" << t.transpose();
54  LOG(INFO) << "end v: \n" << v.transpose();
55  SUCCEED();
}

```

This code uses the gtest framework to verify whether there is a significant difference between IMU integration and preintegration under a fixed angular velocity measurement along the Z axis. The same file also contains tests for constant acceleration and comparisons with ESKF. Since the code is very similar, we do not list it here. Readers can run this code to check whether the IMU operations in the preintegration module are implemented correctly.

4.4.2 Graph Optimization Vertices for Pre-integration

Next, we will implement the graph optimization related to pre-integration. Compared to the filter framework, the graph optimization framework is slightly more complex but offers greater flexibility in usage. We will introduce the variables and classes related to graph optimization one by one.

First, our 15-dimensional or 18-dimensional state variables should correspond to the vertices in graph optimization. They use generalized addition to implement operations on

matrix manifolds and tangent spaces. This book adopts a **bulk** form, so each state is divided into four types of vertices: pose, velocity, gyroscope bias, and accelerometer bias. The latter three are essentially variables in \mathbb{R}^3 and can be directly implemented using inheritance.

Listing 4.5: src/common/g2o_types.h

```

1 class VertexPose : public g2o::BaseVertex<6, SE3> {
2     public:
3         EIGEN_MAKE_ALIGNED_OPERATOR_NEW
4         VertexPose() {}
5
6         virtual void oplusImpl(const double* update_) {
7             _estimate.so3() = _estimate.so3() * S03::exp(Eigen::Map<const Vec3d>(&update_[0]))
8                 ; // Rotation part
9             _estimate.translation() += Eigen::Map<const Vec3d>(&update_[3]);
10                // Translation part
11            updateCache();
12        }
13
14     /**
15      * Velocity vertex, simply Vec3d
16     */
17     class VertexVelocity : public g2o::BaseVertex<3, Vec3d> {
18         public:
19             VertexVelocity() {}
20             virtual void oplusImpl(const double* update_) {
21                 Vec3d uv;
22                 uv << update_[0], update_[1], update_[2];
23                 setEstimate(estimate() + uv);
24             }
25
26     /**
27      * Gyroscope bias vertex, also Vec3d, inherits from velocity vertex
28     */
29     class VertexGyroBias : public VertexVelocity {
30         public:
31             VertexGyroBias() {}
32
33
34     /**
35      * Accelerometer bias vertex, Vec3d, also inherits from velocity vertex
36     */
37     class VertexAccBias : public VertexVelocity {
38         public:
39             VertexAccBias() {}
40

```

We have only listed the key implementation parts, omitting some default constructors. We combine rotation and translation in the same VertexPose vertex. Special attention should be paid to the variable ordering here. Inside VertexPose, rotation comes first followed by translation, so the Jacobian matrix ordering must correspond accordingly.

4.4.3 Graph Optimization Edges for Pre-integration Scheme

Next, we will formulate the prediction and update equations from the previous chapter's GINS system into graph optimization form. Let's summarize the optimization-related edges as follows:

1. Pre-integration edges, which constrain the 15-dimensional state from the previous timestamp to the rotation, translation, and velocity at the next timestamp;
2. Bias random walk edges (two types), connecting bias states between two timestamps;
3. GNSS observation edges. Since we use 6-DOF observations, they associate with the pose at a single timestamp;

4. Prior information, characterizing the state distribution at the previous timestamp and associating with the 15-dimensional state at that time;
5. Wheel odometer observation edges, associating with the velocity vertex at the previous timestamp.

We will implement these components sequentially. First is the most complex pre-integration edge. Its error function and Jacobian function are as follows:

Listing 4.6: src/ch4/g2o_types.cc

```

1 class EdgeInertial : public g2o::BaseMultiEdge<9, Vec9d> {
2     public:
3         EIGEN_MAKE_ALIGNED_OPERATOR_NEW
4
5         /**
6          * The constructor needs to specify the pre-integration class object
7          * @param preinteg Pointer to the pre-integration object
8          * @param gravity Gravity vector
9          * @param weight Weight
10         */
11        EdgeInertial(std::shared_ptr<IMUPreintegration> preinteg, const Vec3d& gravity,
12                     double weight = 1.0);
13
14        void computeError() override;
15        void linearizeOplus() override;
16    private:
17        const double dt_;
18        std::shared_ptr<IMUPreintegration> preint_ = nullptr;
19        Vec3d grav_;
20    };
21
22        EdgeInertial::EdgeInertial(std::shared_ptr<IMUPreintegration> preinteg, const Vec3d&
23                                     gravity, double weight)
24 : preint_(preinteg), dt_(preinteg->dt_) {
25     resize(6); // 6 connected vertices
26     grav_ = gravity;
27     setInformation(preinteg->cov_.inverse() * weight);
28 }
29
30 void EdgeInertial::computeError() {
31     auto* p1 = dynamic_cast<const VertexPose*>(_vertices[0]);
32     auto* v1 = dynamic_cast<const VertexVelocity*>(_vertices[1]);
33     auto* bg1 = dynamic_cast<const VertexGyroBias*>(_vertices[2]);
34     auto* ba1 = dynamic_cast<const VertexAccBias*>(_vertices[3]);
35     auto* p2 = dynamic_cast<const VertexPose*>(_vertices[4]);
36     auto* v2 = dynamic_cast<const VertexVelocity*>(_vertices[5]);
37
38     Vec3d bg = bg1->estimate();
39     Vec3d ba = ba1->estimate();
40
41     const SO3 dR = preint_->GetDeltaRotation(bg);
42     const Vec3d dv = preint_->GetDeltaVelocity(bg, ba);
43     const Vec3d dp = preint_->GetDeltaPosition(bg, ba);
44
45     /// Pre-integration error terms (4.41)
46     const Vec3d er = (dR.inverse() * p1->estimate().so3().inverse() * p2->estimate().so3()
47                       .log());
48     Mat3d RiT = p1->estimate().so3().inverse().matrix();
49     const Vec3d ev = RiT * (v2->estimate() - v1->estimate() - grav_ * dt_) - dv;
50     const Vec3d ep = RiT * (p2->estimate().translation() - p1->estimate().translation()
51                            - v1->estimate() * dt_ -
52                            grav_ * dt_ * dt_ / 2) -
53     dp;
54     _error << er, ev, ep;
55 }
56
57 void EdgeInertial::linearizeOplus() {
58     auto* p1 = dynamic_cast<const VertexPose*>(_vertices[0]);
59     auto* v1 = dynamic_cast<const VertexVelocity*>(_vertices[1]);
60     auto* bg1 = dynamic_cast<const VertexGyroBias*>(_vertices[2]);
61     auto* ba1 = dynamic_cast<const VertexAccBias*>(_vertices[3]);

```

```

58 auto* p2 = dynamic_cast<const VertexPose*>(_vertices[4]);
59 auto* v2 = dynamic_cast<const VertexVelocity*>(_vertices[5]);
60
61 Vec3d bg = bg1->estimate();
62 Vec3d ba = ba1->estimate();
63 Vec3d dbg = bg - preint_->bg_;
64
65 // Intermediate symbols
66 const SO3 R1 = p1->estimate().so3();
67 const SO3 R1T = R1.inverse();
68 const SO3 R2 = p2->estimate().so3();
69
70 auto dR_dbg = preint_->dR_dbg_;
71 auto dv_dbg = preint_->dv_dbg_;
72 auto dp_dbg = preint_->dp_dbg_;
73 auto dv_dba = preint_->dv_dba_;
74 auto dp_dba = preint_->dp_dba_;
75
76 // Estimates
77 Vec3d vi = v1->estimate();
78 Vec3d vj = v2->estimate();
79 Vec3d pi = p1->estimate().translation();
80 Vec3d pj = p2->estimate().translation();
81
82 const SO3 dR = preint_->GetDeltaRotation(bg);
83 const SO3 eR = SO3(dR).inverse() * R1T * R2;
84 const Vec3d er = eR.log();
85 const Mat3d invJr = SO3::jr_inv(eR);
86
87 /// Jacobian matrices
88 /// Note there are 3 indices: vertex index, own error row, and vertex internal
89 /// variable column
90 /// Variable order: pose1(R1,p1), v1, bg1, ba1, pose2(R2,p2), v2
91 /// Residual order: eR, ev, ep, with residual order as rows and variable order as
92 /// columns
93
94 // | R1 | p1 | v1 | bg1 | ba1 | R2 | p2 | v2 |
95 // vert | 0 | 1 | 2 | 3 | 4 | 5 |
96 // col | 0 | 3 | 0 | 0 | 0 | 0 | 3 | 0 |
97 // row
98 // eR 0 |
99 // ev 3 |
100 // ep 6 |
101
102 /// Residual w.r.t R1, 9x3
103 _jacobian0plus[0].setZero();
104 // dr/dR1, 4.42
105 _jacobian0plus[0].block<3, 3>(0, 0) = -invJr * (R2.inverse() * R1).matrix();
106 // dv/dR1, 4.47
107 _jacobian0plus[0].block<3, 3>(3, 0) = SO3::hat(R1T * (vj - vi - grav_ * dt_));
108 // dp/dR1, 4.48d
109 _jacobian0plus[0].block<3, 3>(6, 0) = SO3::hat(R1T * (pj - pi - v1->estimate() * dt_
110 - 0.5 * grav_ * dt_ * dt_));
111
112 /// Residual w.r.t p1, 9x3
113 // dp/dp1, 4.48a
114 _jacobian0plus[0].block<3, 3>(6, 3) = -R1T.matrix();
115
116 /// Residual w.r.t v1, 9x3
117 _jacobian0plus[1].setZero();
118 // dv/dv1, 4.46a
119 _jacobian0plus[1].block<3, 3>(3, 0) = -R1T.matrix();
120 // dp/dv1, 4.48c
121 _jacobian0plus[1].block<3, 3>(6, 0) = -R1T.matrix() * dt_;
122
123 /// Residual w.r.t bg1
124 _jacobian0plus[2].setZero();
125 // dr/dbg1, 4.45
126 _jacobian0plus[2].block<3, 3>(0, 0) = -invJr * eR.inverse().matrix() * SO3::jr((
127     dr_dbg * dbg).eval()) * dr_dbg;
128 // dv/dbg1
129 _jacobian0plus[2].block<3, 3>(3, 0) = -dv_dbg;
130 // dp/dbg1
131 _jacobian0plus[2].block<3, 3>(6, 0) = -dp_dbg;

```

```

129 //////////////////////////////////////////////////////////////////
130     /// Residual w.r.t ba1
131     _jacobian0plus[3].setZero();
132     // dv/dba1
133     _jacobian0plus[3].block<3, 3>(3, 0) = -dv_db;
134     // dp/dba1
135     _jacobian0plus[3].block<3, 3>(6, 0) = -dp_db;
136 //////////////////////////////////////////////////////////////////
137     /// Residual w.r.t pose2
138     _jacobian0plus[4].setZero();
139     // dr/dr2, 4.43
140     _jacobian0plus[4].block<3, 3>(0, 0) = invJr;
141     // dp/dp2, 4.48b
142     _jacobian0plus[4].block<3, 3>(6, 3) = R1T.matrix();
143 //////////////////////////////////////////////////////////////////
144     /// Residual w.r.t v2
145     _jacobian0plus[5].setZero();
146     // dv/dv2, 4.46b
147     _jacobian0plus[5].block<3, 3>(3, 0) = R1T.matrix(); // OK
}

```

We have also provided corresponding formulas in the comments for readers to compare. During implementation, we must be careful with the order of Jacobian matrices here. They actually have three indices: **which vertex**, the **row of the error term**, and the **column of the vertex's internal variables**. For example, if we want to compute the Jacobian block of the pre-integration $\Delta\tilde{\mathbf{p}}_{ij}$ with respect to the translation part of the second pose, i.e., \mathbf{p}_j , then the corresponding Jacobian block should be located at the 4th vertex, 6th row, and 3rd column. Here, the 4th vertex means the pose vertex at time j is the 4th vertex of the pre-integration edge, the 6th row means $\delta\tilde{\mathbf{p}}_{ij}$ is the 6th row in the pre-integration observation, and the 3rd column means \mathbf{p}_j is the 3rd column in VertexPose. Regardless of which optimization framework we use, when defining Jacobian matrices ourselves, we will encounter such matrix block ordering issues when dealing with connections between multiple vertices. This is an error-prone area.

Now let's define the edges for biases, GNSS, prior states, and odometry. The two bias edges are essentially identical, so we'll only show one:

Listing 4.7: src/common/g2o_types.h

```

1 class EdgeGyroRW : public g2o::BaseBinaryEdge<3, Vec3d, VertexGyroBias, VertexGyroBias> {
2     public:
3     void computeError() {
4         const VertexGyroBias* VG1 = static_cast<const VertexGyroBias*>(_vertices[0]);
5         const VertexGyroBias* VG2 = static_cast<const VertexGyroBias*>(_vertices[1]);
6         _error = VG2->estimate() - VG1->estimate();
7     }
8
9     virtual void linearizeOplus() {
10        _jacobian0plusXi = -Mat3d::Identity();
11        _jacobian0plusXj.setIdentity();
12    }
13};
14
15 /**
16 * Prior for previous frame's IMU pvq bias
17 * info is specified externally, given by marginalization in time window
18 *
19 * Vertex order: pose, v, bg, ba
20 * Residual order: R, p, v, bg, ba, 15-dimensional
21 */
22 class EdgePriorPoseNavState : public g2o::BaseMultiEdge<15, Vec15d> {
23     public:
24     void computeError();
25     virtual void linearizeOplus();
26     NavStated state_;
27 };
28
29 void EdgePriorPoseNavState::computeError() {

```

```

30 auto* vp = dynamic_cast<const VertexPose*>(_vertices[0]);
31 auto* vv = dynamic_cast<const VertexVelocity*>(_vertices[1]);
32 auto* vg = dynamic_cast<const VertexGyroBias*>(_vertices[2]);
33 auto* va = dynamic_cast<const VertexAccBias*>(_vertices[3]);
34
35 const Vec3d er = SO3(state_.R_.matrix().transpose() * vp->estimate().so3().matrix())
36     .log();
37 const Vec3d ep = vp->estimate().translation() - state_.p_;
38 const Vec3d ev = vv->estimate() - state_.v_;
39 const Vec3d ebg = vg->estimate() - state_.bg_;
40 const Vec3d eba = va->estimate() - state_.ba_;
41
42 _error << er, ep, ev, ebg, eba;
43 }
44 void EdgePriorPoseNavState::linearize0plus() {
45     const auto* vp = dynamic_cast<const VertexPose*>(_vertices[0]);
46     const Vec3d er = SO3(state_.R_.matrix().transpose() * vp->estimate().so3().matrix())
47         .log();
48     /// Note there are 3 indices: vertex index, error row, and vertex internal variable
49     /// column
50     _jacobian0plus[0].setZero();
51     _jacobian0plus[0].block<3, 3>(0, 0) = SO3::jr_inv(er); // dr/dr
52     _jacobian0plus[0].block<3, 3>(3, 3) = Mat3d::Identity(); // dp/dp
53     _jacobian0plus[1].setZero();
54     _jacobian0plus[1].block<3, 3>(6, 0) = Mat3d::Identity(); // dv/dv
55     _jacobian0plus[2].setZero();
56     _jacobian0plus[2].block<3, 3>(9, 0) = Mat3d::Identity(); // dbg/dbg
57     _jacobian0plus[3].setZero();
58     _jacobian0plus[3].block<3, 3>(12, 0) = Mat3d::Identity(); // dba/dba
59 }
60 class EdgeGNSS : public g2o::BaseUnaryEdge<6, SE3, VertexPose> {
61 public:
62     void computeError() override {
63         VertexPose* v = (VertexPose*)_vertices[0];
64         _error.head<3>() = (_measurement.so3().inverse() * v->estimate().so3().log());
65         _error.tail<3>() = v->estimate().translation() - _measurement.translation();
66     };
67     void linearize0plus() override {
68         VertexPose* v = (VertexPose*)_vertices[0];
69         // jacobian 6x6
70         _jacobian0plusXi.setZero();
71         _jacobian0plusXi.block<3, 3>(0, 0) = (_measurement.so3().inverse() * v->estimate()
72             .so3()).jr_inv(); // dR/dR
73         _jacobian0plusXi.block<3, 3>(3, 3) = Mat3d::Identity();
74     }
75 }
```

Most of the Jacobian matrices here are quite straightforward, and readers should be able to derive them themselves.

4.4.4 Implementation of GINS Based on Pre-integration and Graph Optimization

Finally, we utilize the graph optimization edges defined earlier to implement a GNSS/INS fusion positioning system similar to the ESKF. Readers can also use this experiment to gain deeper insights into the similarities and differences between graph optimization and filter-based approaches. This graph optimization-based GINS system follows essentially the same logic as the ESKF, still requiring static IMU initialization to determine the initial IMU biases and gravity direction. We encapsulate these logical processes in a separate class while focusing on how this graph optimization model is constructed. The basic workflow is as follows:

1. We obtain initial biases and gravity direction through an external static IMU initial-

ization algorithm, then use the first GNSS measurement with attitude information to determine the initial position and orientation. When both IMU and GNSS data become available, we begin prediction and optimization.

2. When IMU data arrives, we use the pre-integrator to accumulate IMU integration information.
3. When odometry data arrives, we record it as the most recent velocity observation and retain its readings.
4. When GNSS data arrives, we construct a graph optimization problem between **the previous timestamp's** GNSS and **the current timestamp's** GNSS. The nodes and edges of this problem are defined as:
 - Nodes: Pose, velocity, and both bias terms (gyro and accelerometer) from the previous and current timestamps - totaling 8 vertices.
 - Edges: Pre-integration observation edge between two timestamps, GNSS observation edges for both timestamps, prior edge for the previous timestamp, two bias random walk edges, and velocity observation edge - making 7 edges in total.
5. We use the predicted values from IMU pre-integration as initial values for optimization. Alternatively, GNSS observations could also be used for initialization. While these two initialization methods yield different starting points, they produce similar results in our current implementation. Readers are encouraged to experiment with both approaches.

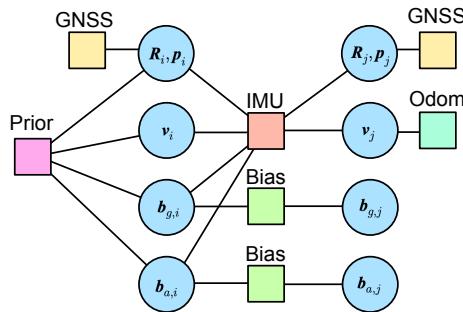


Figure 4-2: Actual graph optimization structure used in the GINS case study

The complete graph optimization structure is shown in Figure 4-2 . We implement GINS as a class that handles IMU, Odom and GNSS observations:

Listing 4.8: src/ch4/gins_pre_integ.cc

```

1 void GInsPreInteg::AddImu(const IMU& imu) {
2   if (first_gnss_received_ && first_imu_received_) {
3     pre_integ_->Integrate(imu, imu.timestamp_ - last_imu_.timestamp_);
4   }
5
6   first_imu_received_ = true;
7   last_imu_ = imu;
8   current_time_ = imu.timestamp_;
9 }
10
11 void GInsPreInteg::AddOdom(const sad::Odom& odom) {

```

```

12     last_odom_ = odom;
13     last_odom_set_ = true;
14 }
15
16 void GInsPreInteg::AddGnss(const GNSS& gnss) {
17     this_frame_ = std::make_shared<NavStated>(current_time_);
18     this_gnss_ = gnss;
19
20     if (!first_gnss_received_) {
21         if (!gnss.heading_valid_) {
22             // First GNSS must have valid heading
23             return;
24         }
25
26         // First GNSS measurement sets initial pose
27         this_frame_->timestamp_ = gnss.unix_time_;
28         this_frame_->p_ = gnss.utm_pose_.translation();
29         this_frame_->R_ = gnss.utm_pose_.so3();
30         this_frame_->v_.setZero();
31         this_frame_->bg_ = options_.preinteg_options_.init_bg_;
32         this_frame_->ba_ = options_.preinteg_options_.init_ba_;
33
34         pre_integ_ = std::make_shared<IMUPreintegration>(options_.preinteg_options_);
35
36         last_frame_ = this_frame_;
37         last_gnss_ = this_gnss_;
38         first_gnss_received_ = true;
39         current_time_ = gnss.unix_time_;
40         return;
41     }
42
43     current_time_ = gnss.unix_time_;
44     *this_frame_ = pre_integ_->Predict(*last_frame_, options_.gravity_);
45
46     Optimize();
47
48     last_frame_ = this_frame_;
49     last_gnss_ = this_gnss_;
50 }
```

The processing functions mainly handle workflow logic, with the key content being the Optimize function:

Listing 4.9: src/ch4/gins_pre_integ.cc

```

1 void GInsPreInteg::Optimize() {
2     if (pre_integ_->dt_ < 1e-3) {
3         // No integration available
4         return;
5     }
6
7     LOG(INFO) << "calling optimization";
8
9     using BlockSolverType = g2o::BlockSolverX;
10    using LinearSolverType = g2o::LinearSolverEigen<BlockSolverType::PoseMatrixType>;
11
12    auto* solver = new g2o::OptimizationAlgorithmLevenberg(
13        g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>()));
14    g2o::SparseOptimizer optimizer;
15    optimizer.setAlgorithm(solver);
16
17    // Previous timestamp vertices: pose, v, bg
18    auto v0_pose = new VertexPose();
19    v0_pose->setId(0);
20    v0_pose->setEstimate(last_frame_->GetSE3());
21    optimizer.addVertex(v0_pose);
22
23    auto v0_vel = new VertexVelocity();
24    v0_vel->setId(1);
25    v0_vel->setEstimate(last_frame_->v_);
26    optimizer.addVertex(v0_vel);
27
28    auto v0_bg = new VertexGyroBias();
```

```

29     v0_bg->setId(2);
30     v0_bg->setEstimate(last_frame_->bg_);
31     optimizer.addVertex(v0_bg);
32
33     auto v0_ba = new VertexAccBias();
34     v0_ba->setId(3);
35     v0_ba->setEstimate(last_frame_->ba_);
36     optimizer.addVertex(v0_ba);
37
38     // Current timestamp vertices: pose, v, bg, ba
39     auto v1_pose = new VertexPose();
40     v1_pose->setId(4);
41     v1_pose->setEstimate(this_frame_->GetSE3());
42     optimizer.addVertex(v1_pose);
43
44     auto v1_vel = new VertexVelocity();
45     v1_vel->setId(5);
46     v1_vel->setEstimate(this_frame_->v_);
47     optimizer.addVertex(v1_vel);
48
49     auto v1_bg = new VertexGyroBias();
50     v1_bg->setId(6);
51     v1_bg->setEstimate(this_frame_->bg_);
52     optimizer.addVertex(v1_bg);
53
54     auto v1_ba = new VertexAccBias();
55     v1_ba->setId(7);
56     v1_ba->setEstimate(this_frame_->ba_);
57     optimizer.addVertex(v1_ba);
58
59     // Pre-integration edge
60     auto edge_inertial = new EdgeInertial(pre_integ_, options_.gravity_);
61     edge_inertial->setVertex(0, v0_pose);
62     edge_inertial->setVertex(1, v0_vel);
63     edge_inertial->setVertex(2, v0_bg);
64     edge_inertial->setVertex(3, v0_ba);
65     edge_inertial->setVertex(4, v1_pose);
66     edge_inertial->setVertex(5, v1_vel);
67     auto* rk = new g2o::RobustKernelHuber();
68     rk->setDelta(200.0);
69     edge_inertial->setRobustKernel(rk);
70     optimizer.addEdge(edge_inertial);
71
72     // Bias random walk edges
73     auto* edge_gyro_rw = new EdgeGyroRW();
74     edge_gyro_rw->setVertex(0, v0_bg);
75     edge_gyro_rw->setVertex(1, v1_bg);
76     edge_gyro_rw->setInformation(options_.bg_rw_info_);
77     optimizer.addEdge(edge_gyro_rw);
78
79     auto* edge_acc_rw = new EdgeAccRW();
80     edge_acc_rw->setVertex(0, v0_ba);
81     edge_acc_rw->setVertex(1, v1_ba);
82     edge_acc_rw->setInformation(options_.ba_rw_info_);
83     optimizer.addEdge(edge_acc_rw);
84
85     // Previous timestamp prior
86     auto* edge_prior = new EdgePriorPoseNavState(*last_frame_, prior_info_);
87     edge_prior->setVertex(0, v0_pose);
88     edge_prior->setVertex(1, v0_vel);
89     edge_prior->setVertex(2, v0_bg);
90     edge_prior->setVertex(3, v0_ba);
91     optimizer.addEdge(edge_prior);
92
93     // GNSS edges
94     auto edge_gnss0 = new EdgeGNSS(v0_pose, last_gnss_.utm_pose_);
95     edge_gnss0->setInformation(options_.gnss_info_);
96     optimizer.addEdge(edge_gnss0);
97
98     auto edge_gnss1 = new EdgeGNSS(v1_pose, this_gnss_.utm_pose_);
99     edge_gnss1->setInformation(options_.gnss_info_);
100    optimizer.addEdge(edge_gnss1);
101
102    // Odom edge

```

```

103     EdgeEncoder3D* edge_odom = nullptr;
104     Vec3d vel_world = Vec3d::Zero();
105     Vec3d vel_odom = Vec3d::Zero();
106     if (last_odom_set_) {
107         // velocity obs
108         double velo_l =
109             options_.wheel_radius_ * last_odom_.left_pulse_ / options_.circle_pulse_ * 2 *
110             M_PI / options_.odom_span_;
111         double velo_r =
112             options_.wheel_radius_ * last_odom_.right_pulse_ / options_.circle_pulse_ * 2 *
113             M_PI / options_.odom_span_;
114         double average_vel = 0.5 * (velo_l + velo_r);
115         vel_odom = Vec3d(average_vel, 0.0, 0.0);
116         vel_world = this_frame_->R_ * vel_odom;
117
118         edge_odom = new EdgeEncoder3D(v1_vel, vel_world);
119         edge_odom->setInformation(options_.odom_info_);
120         optimizer.addEdge(edge_odom);
121     }
122
123     optimizer.setVerbose(options_.verbose_);
124     optimizer.initializeOptimization();
125     optimizer.optimize(20);
126
127     // Some print functions omitted
128
129     // Reset integrator
130     options_.preinteg_options_.init_bg_ = this_frame_->bg_;
131     options_.preinteg_options_.init_ba_ = this_frame_->ba_;
132     pre_integ_ = std::make_shared<IMUPreintegration>(options_.preinteg_options_);
133 }
```

We have omitted some print information and result retrieval steps. From the code, we can see how the entire optimization model is constructed and solved. This modular vertex approach involves more vertex types and quantities, making implementation slightly more complex. Readers are encouraged to compile and run this program using gflags to specify the input file. The test program source code is similar to the previous chapter and won't be listed again. Execute in terminal:

Listing 4.10: Terminal command:

```
bin/run_gins_pre_integ --txt_path ./data/ch3/10.txt
```

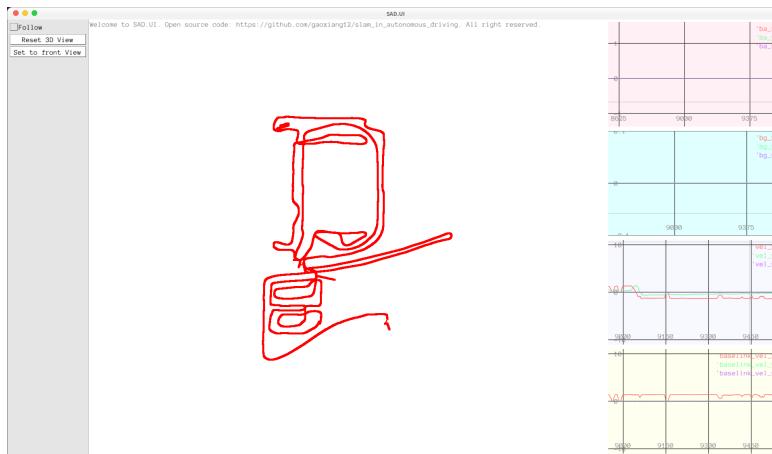


Figure 4-3: GINS results based on pre-integration graph optimization

The program will similarly output real-time results and state variable text files. The real-

time results are shown in Figure 4-3 , and trajectory results can be plotted using the script from the previous chapter:

Listing 4.11: Terminal command:

```
! python3 scripts/plot_ch3_state.py ./data/ch4/gins_preintg.txt
```

The state diagram is shown in Figure4-4. Overall they are similar to the previous chapter, with velocity states within expected ranges. However, since this chapter's GINS doesn't directly optimize at the Odom level, pure IMU prediction will still diverge locally when GNSS observations are unavailable. Below we discuss the program's approach and its differences from ESKF:

1. Compared to ESKF, the pre-integration based graph optimization can accumulate IMU readings. The accumulation duration or number of iterations can be manually selected, while ESKF by default can only iterate once and predict based on single-timestamp IMU data.
2. The pre-integration edge (or called IMU factor/pre-integration factor in factor graph terminology²) is a very flexible factor. All six connected vertices can change. To prevent arbitrary state changes, pre-integration factors typically need to be used with other factors. In our case, GNSS factors at both ends constrain pose changes, Odom factors constrain velocity changes, and two bias factors constrain bias variations without limiting absolute bias values.
3. Prior factors make the estimation smoother. Strictly speaking, prior factor covariance matrices should be handled through marginalization. Since this chapter focuses on pre-integration principles, we've set fixed information matrices for prior factors to simplify implementation. In Chapter 8 , we'll discuss prior factor information matrix settings and implementation. Readers can try removing this factor to observe its impact on trajectory estimation.
4. Graph optimization conveniently allows setting kernel functions and examining each factor's error contribution, helping identify which parts dominate the optimization. For example, we can analyze normal vs abnormal RTK observation residuals to determine GNSS measurement reliability. Later we'll also introduce how to control optimization flow for more robust results. Readers can enable debug output to examine this information.
5. Due to additional computations, graph optimization is noticeably more time-consuming than filter-based approaches. However, with significant increases in computing power for smart vehicles, graph optimization can now be effectively used in some real-time applications.

4.5 Summary

Finally, let's summarize the content of this chapter.

²This book doesn't distinguish between graph optimization and factor graph concepts, as they are essentially the same in practice. Sometimes we refer to them as **optimization edges**, other times as **optimization factors**. Conceptually, **factor** is more intuitive than **edge**, so we often discuss various factors while implementing them as edges.

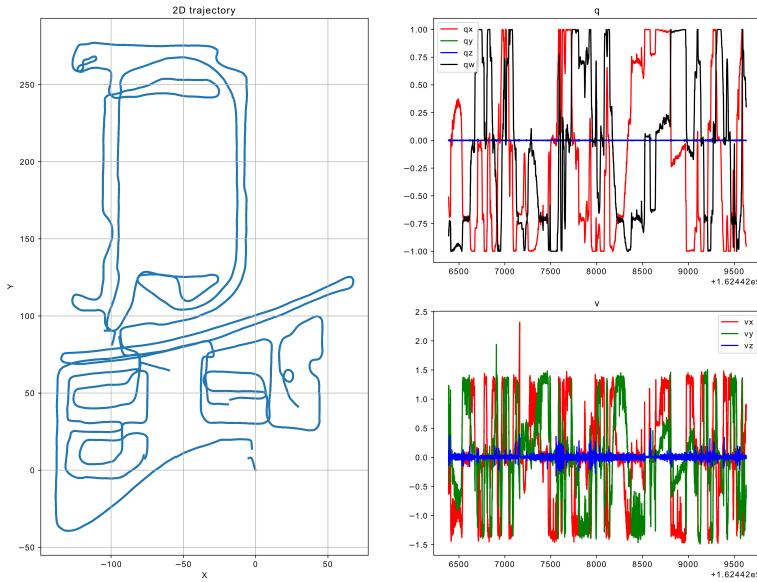


Figure 4-4: GINS results based on pre-integration graph optimization

This chapter introduced the fundamental principles of IMU preintegration, including its measurement model, noise model, Jacobian derivation, and handling of biases. Readers can flexibly apply preintegration in practice:

- If optimization is not considered, preintegration is completely equivalent to direct integration; it can be used to predict future states.
- When used in optimization, preintegration conveniently models the relative motion between two frames. If the IMU bias is fixed, the preintegration model can be greatly simplified. If bias is considered, the preintegrated measurements must be updated accordingly.
- The preintegration model can be easily fused with other graph-based optimization models and optimized within the same problem. It also allows flexible configuration of integration time, number of optimized frames, etc., offering more freedom compared to filter-based approaches.

Readers are encouraged to compare the content of this chapter with the previous one to understand how two different methods address the same problem. This should be insightful for researchers from various fields.

Exercises

1. Use numerical differentiation tools to verify the correctness of the Jacobian matrices in preintegration.
2. Based on the g2o version, implement a Ceres-based version of preintegration DR.
3. Derive a simplified version of the preintegration model assuming no bias drift. How does it compare to the ESKF approach?

4. According to the alternative definition in Section 4.2.4, define the preintegration residual, derive its form and Jacobians with respect to each state variable, and discuss whether it offers computational advantages.
5. Simplify the computation of some Jacobians in preintegration by caching intermediate results to avoid redundant calculations.
6. In a GINS system based on preintegration, consider how to prevent divergence in position estimates due to prolonged absence of RTK observations. Implement a method to trigger optimization using odometry.

Part II

Laser Localization and Mapping

Chapter 5

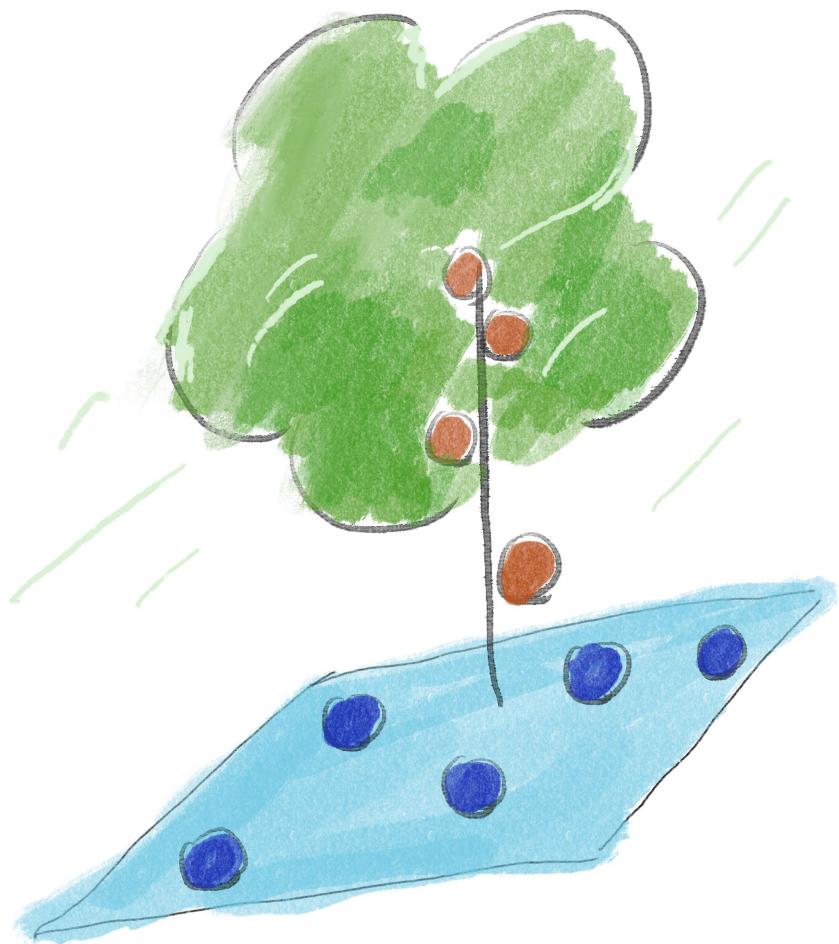
Basic Point Cloud Processing

From this section onwards, we will spend some time introducing the laser SLAM system. Laser sensors are one of the most important sensors in autonomous driving and robotics applications. We can use devices such as lasers, inertial navigation, and satellite navigation to build a complete high-precision mapping and localization application. However, not all researchers are familiar with these sensors. Laser sensors themselves can be categorized into various types, such as single-line, multi-line, mechanical, and solid-state, each with vastly different processing methods.

In this chapter, we will start with basic point cloud processing algorithms and gradually introduce readers to a complete laser SLAM system, including both 2D and 3D aspects. Compared to inertial navigation data or image data, laser data is relatively simple: lasers only detect the three-dimensional structure of objects and do not involve the kinematics of objects or complex projection processes. Mathematically, a \mathbb{R}^3 space can well describe the measurement data from a laser.

However, at the computational level, we still encounter some issues. The most fundamental of these is **how to define spatial adjacency**. This problem is referred to as the **Nearest Neighbour (NN)** problem. We will find that the **nearest neighbour** between points is the foundation of many algorithms, yet this seemingly simple task can be approached in many different ways computationally. We could use the simplest arrays to represent point clouds, but that would fail to capture the relationships between points. To facilitate adjacency calculations, we also explore more complex **tree structures**, which are more efficient than traditional methods in handling nearest neighbour problems. Many point cloud registration algorithms require calculating the error between a point and its surrounding points based on some metric. This metric could be the Euclidean distance between points, the **point-to-line** or **point-to-plane** distance, or some **statistically meaningful** metric. Different metric selection methods lead to different algorithms, but they all share common theoretical foundations. For example, pre-dividing space according to some **criterion** and then building an **indexed** data structure to facilitate the search for point-to-point adjacency relationships. The methods of division here are diverse, ranging from simple grid-based approaches to planes, spheres, interfaces, etc., giving rise to numerous algorithms. We will introduce the important ones to the readers.

In this chapter, we will first cover the basic algorithms for point clouds, including how to represent point clouds, how to describe the mathematical model of a laser sensor, how to find the neighbouring points of a given point, how to fit simple geometric shapes to a set of points, and so on. The next two chapters will build upon this content to develop 2D and 3D registration methods.



点云的相邻关系是许多算法的基础。

5.1 Mathematical Models of Laser Sensors and Point Clouds

5.1.1 Mathematical Model of Laser Sensors



Figure 5-1: Various radar models used in autonomous driving. From left to right: Velodyne HDL-64, DJI Livox, Hesai, RoboSense, and LeiShen.

Lidar is the most important sensor in autonomous driving. It provides high-precision distance measurement information but remains quite expensive. To this day, there is still intense debate over whether Lidar should be used in autonomous vehicles. Autonomous driving employs various models of Lidar, some of which are listed in Figure 5-1. Broadly speaking, Lidar used in autonomous driving can be divided into two types: **mechanical spinning Lidar** and **solid-state Lidar**¹.

1. **Mechanical spinning Lidar** can be viewed as a column of laser probes rotating at a fixed frequency. Each probe can quickly measure the distance to external objects. Each full rotation of the probes completes one scan of the surrounding environment. Lidar can be further categorized by the number of lines, common configurations include single-line, 4-line, 8-line, 16-line, 32-line, 64-line, 80-line, and 128-line. The higher the number of lines, the more points are captured per scan, and the richer the information. However, even after multiple price reductions, high-line-count spinning Lidar (32-line and above) remains a very expensive sensor, often costing several times the price of the vehicle itself².
2. **Solid-state Lidar** is a rapidly developing new type of Lidar in recent years. Unlike mechanical Lidar, solid-state Lidar does not perform 360-degree scans; it can only detect 3D information within a field of view of about 120 degrees. They are very similar to RGB-D cameras (the two are also similar in principle). Most solid-state Lidar systems have a field of view ranging from 60 to 120 degrees but are cheaper and can achieve image-like scanning. In terms of equivalent line count, solid-state Lidar can even achieve effects comparable to 200 lines or more, though solid-state Lidar does not necessarily scan along horizontal lines—some have unique scanning patterns.

Spinning Lidar and solid-state Lidar each have their pros and cons. The 360-degree scanning capability of spinning Lidar is highly advantageous for localization and mapping,

¹Hereafter, we will consistently use the term **radar** to refer to laser ranging sensors, though this may cause some ambiguity in expression. The **radar** mentioned here is a transliteration of the English term "Lidar," which stands for Light Detection and Ranging. In the context of autonomous driving, Lidar is often abbreviated as radar. However, in other fields, radar refers to Radio Detection and Ranging, or Radar for short. In Chinese, both Lidar and Radar can be referred to as radar, while in English, they are distinct terms. Notably, autonomous driving also uses millimeter-wave radar, often called Radar. In this book, radar uniformly refers to Lidar, not millimeter-wave radar.

²Based on 2021 prices.

as the panoramic view ensures that the entire road segment can be mapped in a single pass, and point cloud localization is less susceptible to occlusion. When using solid-state Lidar, multiple units are often combined to achieve a similar panoramic effect. However, spinning Lidar has clear disadvantages in terms of cost and lifespan, and in the near term, it still does not meet automotive-grade or consumer-level price requirements. In contrast, solid-state Lidar can easily achieve costs in the range of several thousand yuan, meeting safety and lifespan requirements at the expense of some field of view (which can be compensated for by using multiple units), and has gained many supporters in recent years. Some high-end vehicle models have already begun equipping solid-state Lidar as part of their perception systems.

The debate over whether autonomous driving should use Lidar has never ceased. Some believe L4 autonomous driving cannot do without Lidar, while others vehemently argue that Lidar is a burden on vehicles. There are also moderates who are indifferent to the type of sensor, focusing only on functionality and cost. For L4, the early development of autonomous driving primarily relied on mechanical spinning Lidar, so current technical solutions exhibit some degree of path dependence on spinning Lidar. Path dependence refers to the phenomenon where early algorithms and research were tailored to an initial solution, often without regard to cost. Without causing significant issues, there is little incentive to change this solution over time. However, viewed years later, this solution may not necessarily be the best in practice.

This book does not intend to engage in debates about sensors but focuses solely on introducing their principles and algorithms. Compared to visual and IMU sensors, the measurement of a single laser probe is extremely simple: it merely measures the distance to a point in space, denoted as r . Of course, the laser probe itself can be mounted on the vehicle at a certain tilt angle, allowing the spatial position of the endpoint to be determined. This model is called the RAE (Range, Azimuth, Elevation) model, as shown in Figure 5-2.

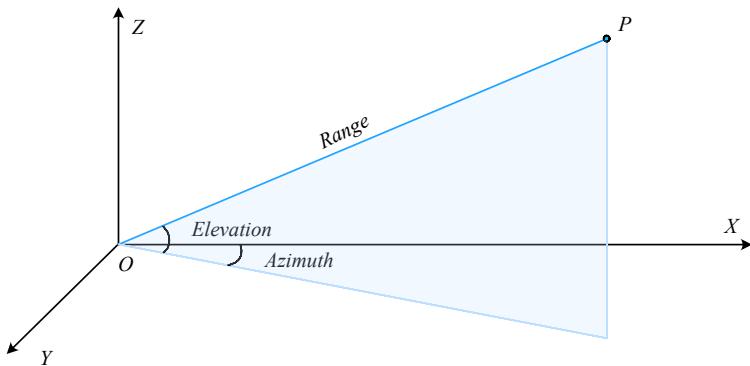


Figure 5-2: RAE model for single-point measurement

Let the distance be $r = \text{Range}$, the azimuth angle be $A = \text{Azimuth}$, and the elevation angle be $E = \text{Elevation}$. These angles are similar to Euler angles. Based on geometric relationships, the position of \mathbf{P} in the radar reference frame can easily be derived:

$$\mathbf{P} = [r \cos E \cos A, r \cos E \sin A, r \sin E]^\top. \quad (5.1)$$

Conversely, r, A, E can be calculated from $\mathbf{P} = [x, y, z]^\top$:

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2}, \\ A &= \arctan(y/x), \\ E &= \arcsin(z/r). \end{aligned} \tag{5.2}$$

These are simply conversions between Euclidean and polar coordinates, with no substantive difference. Spinning Lidar can be viewed as multiple RAE probe models fixed in elevation angle and rotating synchronously in azimuth angle. When the probes complete one full rotation, we obtain endpoints with azimuth angles ranging from 0 to 360 degrees. Typically, this set of points is called a **scan** because it closely resembles the scanning process of a scanner or television. Most Lidar systems can scan the surrounding environment at frequencies of 10 Hz or higher. If such a probe has only one line, it is a single-line Lidar, usually mounted horizontally. Multi-line Lidar can be seen as composed of multiple single-line Lidar units arranged vertically in a column, rotating at a fixed frequency. After one full scan, the multi-line Lidar produces a point cloud that fully reflects the three-dimensional structure within a certain distance. For a single probe, E can be considered fixed, A changes at a constant speed, and only r is derived from actual measurements. This property can be leveraged to compress Lidar data—recording time and distance is simpler than recording Cartesian coordinates (i.e., the XYZ data of the point cloud).

In addition to distance, Lidar can also carry additional data. For example, multi-line Lidar can record the timestamp, reflectivity, and line number (which line the point belongs to) for each point. This information, particularly reflectivity, can be used in various practical scenarios, such as extracting road markings based on reflectivity or identifying the ground using line numbers. Basic point cloud algorithms, such as nearest neighbor and fitting algorithms, rely solely on Cartesian coordinate position data. Unless otherwise specified, the point clouds discussed hereafter refer to those containing only positional information. However, in visualization, point clouds may be rendered based on height or reflectivity.

5.1.2 Representation of Point Clouds

Point cloud, in its most literal sense, refers to a set of points scattered in space. The term “**cloud**” inherently discards the relationships between points. These points are merely Cartesian coordinates in Euclidean space and carry no additional information. Do these points form a mesh? Do certain points constitute a triangle? The point cloud structure does not include such extra details.

Point clouds are the most fundamental way to represent 3D structures and are the primary data format output by most laser sensors. If we wish to extract additional information from a point cloud, we must employ supplementary data structures to express these relationships. For instance, many algorithms require answering questions like: Which point is the nearest to a given point? What shape do this point and its neighboring points collectively form? To achieve such functionality, additional data structures must be introduced—this is the focus of this section.

The content of this chapter overlaps with many books on 3D point clouds. Readers may also refer to similar textbooks, such as [74?]. As a book aimed at SLAM practitioners, we emphasize point cloud processing for laser sensors rather than discussing generic point cloud processing broadly. We will introduce some usage of the PCL library [75], and for key algorithms, we also provide handwritten implementations along with performance comparisons to their PCL counterparts.

The simplest and most primitive way to represent a point cloud is as an array: just store the points in an array. This can be easily implemented using ‘`std::vector`’ in C++. Of course,

as mentioned earlier, point clouds can also carry additional information, such as reflectivity, the line number (for multi-line Lidar), or RGB color data. Point clouds from RGB-D cameras may also store the row and column indices of each point in the image. Therefore, a more convenient approach is to define a point cloud structure and use a templated container to store it. If you need to store other information, such as the global position and orientation of the entire point cloud, you can define your own point cloud data structure.

We illustrate the basics of reading, writing, and visualizing point clouds through a program example, giving readers an intuitive understanding. We have prepared two files for readers: a single-scan point cloud and a map point cloud file, located at ‘`data/ch5/map_example.pcd`’ and ‘`scan_example.pcd`’, respectively. Readers can directly view these files using the ‘`pcl_viewer`’ command:

Listing 5.1: Terminal input:

```
1 pcl_viewer ./data/ch5/map_example.pcd
```

We will also introduce other representation methods later. The example map and scan data are shown in Figure 5-3. The left point cloud depicts a small “L”-shaped scene consisting of a central building and surrounding greenery, while the right shows data from a single scan. By stitching multiple scans together, we can reconstruct the full map, whereas a single scan clearly contains far fewer points.

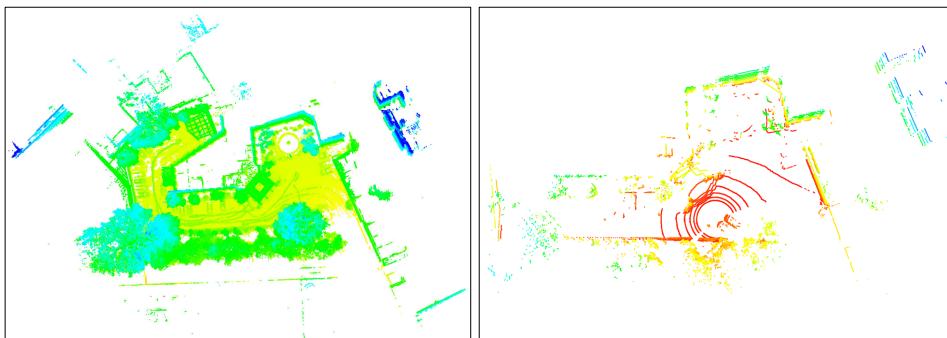


Figure 5-3: Example data of a point cloud map and a single scan.

Below, we demonstrate how to read and visualize a point cloud, serving as a basic tutorial for PCL.

Listing 5.2: `src/ch5/point_cloud_load_and_vis.cc`

```
1 int main(int argc, char** argv) {
2     // .. Omitted parameter checking
3     // Load the point cloud
4     PointCloudType::Ptr cloud(new PointCloudType);
5     pcl::io::loadPCDFile(FLAGS_pcd_path, *cloud);
6
7     if (cloud->empty()) {
8         LOG(ERROR) << "cannot load cloud file";
9         return -1;
10    }
11
12    LOG(INFO) << "cloud points: " << cloud->size();
13
14    // Visualize
15    pcl::visualization::PCLVisualizer viewer("cloud viewer");
16    pcl::visualization::PointCloudColorHandlerGenericField<PointType> handle(cloud, "z")
17        ; // Color by height
viewer.addPointCloud<PointType>(cloud, handle);
```

```

18     viewer.spin();
19
20     return 0;
21 }
```

Since data I/O and visualization are handled by existing PCL modules, we only need to call its functions. After compilation, readers can also use the program to view point clouds:

Listing 5.3: Terminal input:

```
bin/point_cloud_load_and_vis --pcd_path ./data/ch5/map_example.pcd
```

Most programs supporting 3D display can render point clouds effectively. Later, we will continue using 3D graphical interfaces to showcase point cloud registration results.

5.1.3 Packet Representation

In addition to raw point clouds, there are several other representation methods. Raw point clouds store the coordinates of all points, which consumes significant space. Some representations are more suitable for storage and transmission, while others provide advantageous properties for algorithmic processing. Below, we introduce a few commonly used methods.

In LiDAR sensors, parameters such as the rotation rate of the radar and the elevation angles of each probe relative to the sensor center are known during the design or operation of the sensor. These are referred to as the **intrinsic parameters** of the LiDAR. Therefore, these parameters can be stored in a fixed configuration file, while only the runtime-varying measurements need to be recorded in the actual data. For LiDAR, the varying components typically include the detected distance and reflectivity of objects. Leveraging this property can significantly reduce the data transmission volume between the sensor and the computer. Similarly, when storing raw point cloud data, this approach can be used to save disk space instead of storing the full point cloud. This is the concept behind **data packets** (Packets).

Most LiDAR manufacturers define their own packet formats, which vary in implementation. These packets can be transmitted and received via various communication protocols (usually network protocols such as UDP). Taking the Velodyne HDL-64S3 as an example (Figure 5-4), let's examine how hardware manufacturers compress point cloud data.

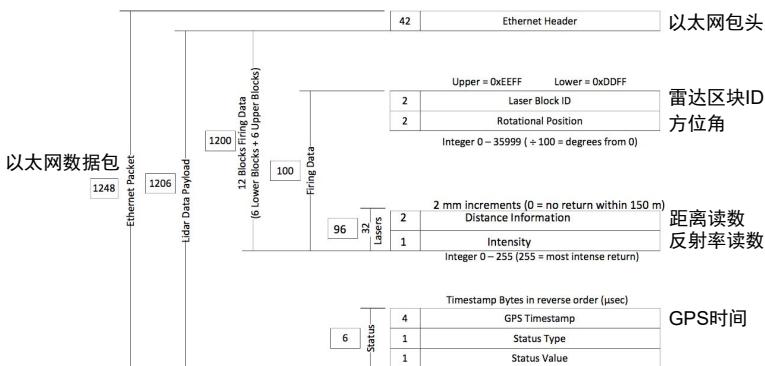


Figure 5-4: Packet format definition for the Velodyne HDL-64S3

In the Velodyne HDL-64S3, the distance and reflectivity measurements are stored in 3 bytes, while readings from the same column share an azimuth angle and block ID. As shown, 32 readings occupy 100 bytes in total. If stored in PCL format, the positional coordinates

and reflectivity of 32 scan lines would require $32 \times (4 + 4 + 4 + 1) = 416$ bytes. Clearly, storing packets is far more efficient than storing raw point clouds.

However, in many algorithmic implementations, direct access to the coordinates of each point is often preferred. Therefore, packets are typically used as compressed data in modules such as driver software, raw sensor data handling, or map compression. Due to the variety of LiDAR brands and their differing driver implementations, manufacturers usually provide SDKs to handle the conversion between packets and point clouds. Here, we will not delve into the specifics of how each LiDAR compresses and decompresses its data.

5.1.4 Bird's-Eye View and Range Image

Bird's-eye view (BEV) is another commonly used representation for maps, especially in outdoor environments. If we wish to express LiDAR point clouds in a grid-based format and apply grid-based algorithms for path planning, obstacle avoidance, or 2D annotations, it becomes necessary to represent the point cloud map from a top-down perspective. Of course, converting 3D information to 2D typically involves discarding some data—in this case, the height information of the point cloud.

Since vehicles are usually horizontally positioned, the sensors mounted on them are also typically level. Under this assumption, converting an outdoor point cloud map to a bird's-eye view is straightforward, as the x, y, z coordinates of the point cloud correspond directly to horizontal and vertical axes. For sensors with tilted or rotating configurations, additional ground or world coordinate system information is required. Below, we convert the point cloud from Figure 5-3 into a bird's-eye view. The BEV is implemented using OpenCV as a 2D image. To map point cloud coordinates to image coordinates, we define a resolution r , which determines how many meters each pixel represents. Additionally, we ensure the image center aligns with the point cloud center, while the image dimensions depend on the x, y range of the point cloud.

Let the point cloud center be $\mathbf{c} = [c_x, c_y]^\top$ and the image center be I_x, I_y . A point with coordinates (x, y, z) should map to image coordinates (u, v) as follows:

$$\begin{cases} u &= (x - c_x)/r + I_x, \\ v &= (y - c_y)/r + I_y. \end{cases} \quad (5.3)$$

The z -coordinate can then be represented using different colors to indicate height variations. The implementation of this process is shown below:

Listing 5.4: src/ch5/pcd_to_bird_eye.cc

```

1 DEFINE_string(pcd_path, "./data/ch5/map_example.pcd", "Point cloud file path");
2 DEFINE_double(image_resolution, 0.1, "BEV resolution");
3 DEFINE_double(min_z, 0.2, "Minimum height for BEV");
4 DEFINE_double(max_z, 2.5, "Maximum height for BEV");
5
6 void GenerateBEVImage(PointCloudType::Ptr cloud) {
7     // Compute point cloud boundaries
8     auto minmax_x = std::minmax_element(cloud->points.begin(), cloud->points.end(),
9     [] (const PointType& p1, const PointType& p2) { return p1.x < p2.x; });
10    auto minmax_y = std::minmax_element(cloud->points.begin(), cloud->points.end(),
11     [] (const PointType& p1, const PointType& p2) { return p1.y < p2.y; });
12    double min_x = minmax_x.first->x;
13    double max_x = minmax_x.second->x;
14    double min_y = minmax_y.first->y;
15    double max_y = minmax_y.second->y;
16
17    const double inv_r = 1.0 / FLAGS_image_resolution;
18
19    const int image_rows = int((max_y - min_y) * inv_r);
20    const int image_cols = int((max_x - min_x) * inv_r);
21

```

```

22 float x_center = 0.5 * (max_x + min_x);
23 float y_center = 0.5 * (max_y + min_y);
24 float x_center_image = image_cols / 2;
25 float y_center_image = image_rows / 2;
26
27 // Generate image
28 cv::Mat image(image_rows, image_cols, CV_8UC3, cv::Scalar(255, 255, 255));
29
30 for (const auto& pt : cloud->points) {
31     int x = int((pt.x - x_center) * inv_r + x_center_image);
32     int y = int((pt.y - y_center) * inv_r + y_center_image);
33     if (x < 0 || x >= image_cols || y < 0 || y >= image_rows || pt.z < FLAGS_min_z || 
34         pt.z > FLAGS_max_z) {
35         continue;
36     }
37     image.at<cv::Vec3b>(y, x) = cv::Vec3b(227, 143, 79);
38 }
39
40 cv::imwrite("./bev.png", image);
41 }
```

Now run:

Listing 5.5: Terminal input:

```
bin/pcd_to_bird_eye --pcd_path ./data/ch5/map_example.pcd
```

We assume the point cloud is horizontally aligned and map the X and Y axes to the image. The initial part of the code determines the image boundaries, and then each point's position in the image is calculated based on the specified resolution. Obstacles within a certain height range (here, 0.2 to 2.5 meters, adjustable based on vehicle height) are considered valid and projected onto the BEV, resulting in the output shown in Figure 5-6.



Figure 5-5: Converting a point cloud to a bird's-eye view

Many path-planning algorithms, such as A* and D* [76], operate on grid maps, while perception or obstacle-avoidance algorithms can also use grid maps as input [77]. As seen, most obstacle information is preserved when converting point clouds to grid maps, though dynamic objects may leave smearing effects. We will discuss the probabilistic mechanism of grid maps in the 2D LiDAR SLAM chapter, which effectively mitigates the impact of dynamic objects.

Some readers might wonder: If point clouds can be projected into a bird’s-eye view, can they also be projected from other angles? While BEV is the most intuitive choice for top-down observation, similar logic can be applied to generate front or side views, which are computationally equivalent. However, another useful representation for algorithms is the **range image**.

The concept of a range image aligns with that of RGB-D cameras. To ensure consistency between depth and color images, RGB-D cameras project point clouds into the color camera’s frame. Similarly, can LiDAR point clouds be projected into a virtual camera? The answer is yes. However, since LiDAR point clouds cover a full 360-degree field of view, the resulting image is panoramic. Here, the horizontal axis represents the LiDAR’s azimuth angle, while the vertical axis corresponds to the elevation angle. Alternatively, if the elevation angles for each scan line are known, the **line number** can serve as the vertical axis. Both methods produce what is known as a range image.

As before, we provide an example to illustrate the appearance of a range image. While generating range images directly from raw LiDAR data (e.g., using block IDs or azimuth angles from packets) is more efficient, this approach depends on the specific LiDAR model. Our example uses post-processed point clouds, requiring only an additional azimuth angle calculation step.

5.1.5 Bird’s-Eye View and Range Image

Bird’s-eye view (BEV) is another commonly used representation for maps, especially in outdoor environments. If we wish to express LiDAR point clouds in a grid-based format and apply grid-based algorithms for path planning, obstacle avoidance, or 2D annotations, it becomes necessary to represent the point cloud map from a top-down perspective. Of course, converting 3D information to 2D typically involves discarding some data—in this case, the height information of the point cloud.

Since vehicles are usually horizontally positioned, the sensors mounted on them are also typically level. Under this assumption, converting an outdoor point cloud map to a bird’s-eye view is straightforward, as the x, y, z coordinates of the point cloud correspond directly to horizontal and vertical axes. For sensors with tilted or rotating configurations, additional ground or world coordinate system information is required. Below, we convert the point cloud from Figure 5-3 into a bird’s-eye view. The BEV is implemented using OpenCV as a 2D image. To map point cloud coordinates to image coordinates, we define a resolution r , which determines how many meters each pixel represents. Additionally, we ensure the image center aligns with the point cloud center, while the image dimensions depend on the x, y range of the point cloud.

Let the point cloud center be $\mathbf{c} = [c_x, c_y]^\top$ and the image center be I_x, I_y . A point with coordinates (x, y, z) should map to image coordinates (u, v) as follows:

$$\begin{cases} u &= (x - c_x)/r + I_x, \\ v &= (y - c_y)/r + I_y. \end{cases} \quad (5.4)$$

The z -coordinate can then be represented using different colors to indicate height variations. The implementation of this process is shown below:

Listing 5.6: src/ch5/pcd_to_bird_eye.cc

```

1 DEFINE_string(pcd_path, "./data/ch5/map_example.pcd", "Point cloud file path");
2 DEFINE_double(image_resolution, 0.1, "BEV resolution");
3 DEFINE_double(min_z, 0.2, "Minimum height for BEV");
4 DEFINE_double(max_z, 2.5, "Maximum height for BEV");
5
6 void GenerateBEVImage(PointCloudType::Ptr cloud) {

```

```

7 // Compute point cloud boundaries
8 auto minmax_x = std::minmax_element(cloud->points.begin(), cloud->points.end(),
9     [](<const PointType& p1, <const PointType& p2) { return p1.x < p2.x; });
10 auto minmax_y = std::minmax_element(cloud->points.begin(), cloud->points.end(),
11     [](<const PointType& p1, <const PointType& p2) { return p1.y < p2.y; });
12 double min_x = minmax_x.first->x;
13 double max_x = minmax_x.second->x;
14 double min_y = minmax_y.first->y;
15 double max_y = minmax_y.second->y;
16
17 const double inv_r = 1.0 / FLAGS_image_resolution;
18
19 const int image_rows = int((max_y - min_y) * inv_r);
20 const int image_cols = int((max_x - min_x) * inv_r);
21
22 float x_center = 0.5 * (max_x + min_x);
23 float y_center = 0.5 * (max_y + min_y);
24 float x_center_image = image_cols / 2;
25 float y_center_image = image_rows / 2;
26
27 // Generate image
28 cv::Mat image(image_rows, image_cols, CV_8UC3, cv::Scalar(255, 255, 255));
29
30 for (const auto& pt : cloud->points) {
31     int x = int((pt.x - x_center) * inv_r + x_center_image);
32     int y = int((pt.y - y_center) * inv_r + y_center_image);
33     if (x < 0 || x >= image_cols || y < 0 || y >= image_rows || pt.z < FLAGS_min_z || 
34         pt.z > FLAGS_max_z) {
35         continue;
36     }
37     image.at<cv::Vec3b>(y, x) = cv::Vec3b(227, 143, 79);
38 }
39
40 cv::imwrite("./bev.png", image);
41 }
```

Now run:

Listing 5.7: Terminal input:

```
bin/pcd_to_bird_eye --pcd_path ./data/ch5/map_example.pcd
```

We assume the point cloud is horizontally aligned and map the X and Y axes to the image. The initial part of the code determines the image boundaries, and then each point's position in the image is calculated based on the specified resolution. Obstacles within a certain height range (here, 0.2 to 2.5 meters, adjustable based on vehicle height) are considered valid and projected onto the BEV, resulting in the output shown in Figure 5-6.

Many path-planning algorithms, such as A* and D* [76], operate on grid maps, while perception or obstacle-avoidance algorithms can also use grid maps as input [77]. As seen, most obstacle information is preserved when converting point clouds to grid maps, though dynamic objects may leave smearing effects. We will discuss the probabilistic mechanism of grid maps in the 2D LiDAR SLAM chapter, which effectively mitigates the impact of dynamic objects.

Some readers might wonder: If point clouds can be projected into a bird's-eye view, can they also be projected from other angles? While BEV is the most intuitive choice for top-down observation, similar logic can be applied to generate front or side views, which are computationally equivalent. However, another useful representation for algorithms is the **range image**.

The concept of a range image aligns with that of RGB-D cameras. To ensure consistency between depth and color images, RGB-D cameras project point clouds into the color camera's frame. Similarly, can LiDAR point clouds be projected into a virtual camera? The answer is yes. However, since LiDAR point clouds cover a full 360-degree field of view, the resulting



Figure 5-6: Converting a point cloud to a bird's-eye view

image is panoramic. Here, the horizontal axis represents the LiDAR's azimuth angle, while the vertical axis corresponds to the elevation angle. Alternatively, if the elevation angles for each scan line are known, the **line number** can serve as the vertical axis. Both methods produce what is known as a range image.

As before, we provide an example to illustrate the appearance of a range image. While generating range images directly from raw LiDAR data (e.g., using block IDs or azimuth angles from packets) is more efficient, this approach depends on the specific LiDAR model. Our example uses post-processed point clouds, requiring only an additional azimuth angle calculation step.

Listing 5.8: src/ch5/scan_to_range_image.cc

```

1 DEFINE_string(pcd_path, "./data/ch5/scan_example.pcd", "Point cloud file path");
2 DEFINE_double(azimuth_resolution_deg, 0.3, "Azimuth resolution (degrees)");
3 DEFINE_int32(elevation_rows, 16, "Number of rows for elevation");
4 DEFINE_double(elevation_range, 15.0, "Elevation range"); // VLP-16 has ±15 degree
range
5 DEFINE_double(lidar_height, 1.128, "LiDAR mounting height");
6
7 void GenerateRangeImage(PointCloudType::Ptr cloud) {
8     int image_cols = int(360 / FLAGS_azimuth_resolution_deg); // 360 degrees
// horizontally divided by resolution
9     int image_rows = FLAGS_elevation_rows; // Fixed number of rows
LOG(INFO) << "range image: " << image_rows << "x" << image_cols;
10
11 // Create HSV image for better distance visualization
12 cv::Mat image(image_rows, image_cols, CV_8UC3, cv::Scalar(0, 0, 0));
13
14 double ele_resolution = FLAGS_elevation_range * 2 / FLAGS_elevation_rows; // Elevation resolution
15
16 for (const auto& pt : cloud->points) {
17     double azimuth = atan2(pt.y, pt.x) * 180 / M_PI;
18     double range = sqrt(pt.x * pt.x + pt.y * pt.y);
19     double elevation = atan2((pt.z - FLAGS_lidar_height), range) * 180 / M_PI;
20
21     // Normalize azimuth to 0~360
22     if (azimuth < 0) {
23         azimuth += 360;
24     }
25
26     int x = int(azimuth / FLAGS_azimuth_resolution_deg); // Column
27     int y = int((elevation + FLAGS_elevation_range) / ele_resolution + 0.5); // Row
28 }
```

```

29     if (x >= 0 && x < image.cols && y >= 0 && y < image.rows) {
30         image.at<cv::Vec3b>(y, x) = cv::Vec3b(uchar(range / 100 * 255.0), 255, 127);
31     }
32 }
33
34 // Flip along Y-axis to make Z-up correspond to image-up
35 cv::Mat image_flipped;
36 cv::flip(image, image_flipped, 0);
37
38 // Convert HSV to RGB
39 cv::Mat image_rgb;
40 cv::cvtColor(image_flipped, image_rgb, cv::COLOR_HSV2BGR);
41 cv::imwrite("./range_image.png", image_rgb);
42
43 }
```

Several details require attention in this program, such as degree-radian conversion and image flipping along the Y-axis. We use HSV color space to display range information, making distance variations more visually apparent. To convert a single scan to a range image, run:

Listing 5.9: Terminal input:

```
bin/scan_to_range_image
```

Readers can adjust parameters in gflags to achieve different conversion effects. The converted image is saved as range_image.png in the current directory. With default parameters, since the LiDAR has 16 scan lines, the output will be a long rectangular image of size 1200×16 , as shown in Figure 5-7. The image height can be adjusted by changing the number of elevation rows. This representation resembles camera images but without perspective projection. Some algorithms discussed later will use range images to extract vertical features for localization. Readers can also try identifying ground regions and prominent pole-like objects in this image.

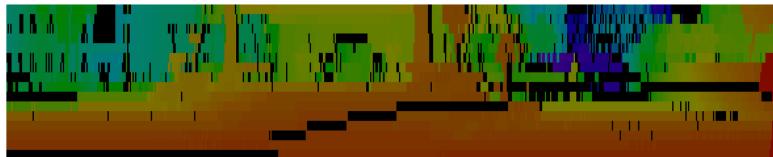


Figure 5-7: Converting point cloud to range image. The original range image is hard to display, so we scaled it here.

5.1.6 Alternative Representations

In addition to the methods previously discussed, some research has applied RGB-D 3D reconstruction techniques to LiDAR point clouds, aiming to achieve better reconstruction quality (and visual effects). For instance, works like [78, 79] employ Surfel maps to restore local surface reconstruction effects based on point clouds, while [80] demonstrates surface reconstruction using LiDAR data. However, in autonomous driving scenarios, LiDAR's primary functions remain localization and obstacle detection, and these algorithms have yet to become mainstream. We mention these approaches for reference, and interested readers may explore the cited papers.

Of course, regardless of the representation method, the fundamental measurement data from the LiDAR remains unchanged—it neither becomes artificially denser nor more information-rich. So why use representations like range images or bird's-eye views? The key observation is that while these methods do not alter the raw measurements, they fundamentally change the

neighborhood relationships between points. In raw point clouds, the adjacency between points is not explicitly encoded. In contrast, bird’s-eye views and range images inherently encode pixel-to-pixel spatial relationships, enabling image-based processing.

For example, a vertical pillar in 3D space is distributed along the Z -axis, but in a range image, it appears as a feature along the Y -axis, while in a bird’s-eye view, it collapses to a single point. This shift in distribution affects the performance of clustering or feature extraction algorithms. Consequently, some algorithms prefer to extract features from range images or bird’s-eye views first, then compute their corresponding 3D positions.

5.2 Nearest Neighbor Problem

The nearest neighbor problem is one of the most fundamental issues in point cloud processing and a step that will be repeatedly invoked in many matching algorithms. The problem can be described very simply: Given a point cloud $\mathcal{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ containing n points, we ask, which point is closest to a given point \mathbf{x}_m ? Further, what are the k nearest points to it? Or which points lie within a fixed range r of it? The former is called k -nearest neighbor search (kNN), while the latter is called range search.

Although this problem appears simple, solving it is not trivial, and there are many different approaches. We are particularly concerned with the efficiency of solving the nearest neighbor problem because this algorithm is typically invoked thousands or even millions of times. A small increase in computation time per call can lead to significant efficiency differences in the overall matching algorithm. Below, we introduce nearest neighbor methods in order of increasing complexity, which also aligns with the historical development of algorithmic techniques.

The algorithms in this chapter will emphasize parallelization, as point cloud algorithms require nearest neighbor searches on a large number of points, making parallel performance a critical consideration.

5.2.1 Brute-Force Nearest Neighbor Search

The brute-force nearest neighbor search (BF search) is the simplest and most intuitive method, requiring no auxiliary data structures³ [82]. If we search for the nearest neighbor (NN) of a point, we call it **brute-force nearest neighbor search**; if we search for the k nearest neighbors, we call it brute-force k -nearest neighbor search. Overall, this is a straightforward but computationally intensive approach.

Brute-Force Nearest Neighbor Search Given a point cloud \mathcal{X} and a query point \mathbf{x}_m , compute the distance between \mathbf{x}_m and every point in \mathcal{X} , then return the minimum distance.

Similarly, the brute-force k -nearest neighbor search can be defined as:

Brute-Force k -Nearest Neighbor (BF kNN)

1. For a given point cloud \mathcal{X} and query point \mathbf{x}_m , compute the distance between \mathbf{x}_m and every point in \mathcal{X} .

³Sometimes referred to as linear search [81].

2. Sort the results from step 1.
3. Select the k nearest points.
4. Repeat steps 1–3 for all \mathbf{x}_m .

Alternatively, we can maintain only the top k results during computation, comparing each new distance with the existing ones to save storage space.

It is clear that the brute-force method requires traversing the entire point cloud each time. For a point cloud of size n , its complexity is $O(n)$. When dealing with the matching problem between two point clouds (assuming both have n points), the complexity of brute-force nearest neighbor search becomes $O(n^2)$. This is obviously a time-consuming method, and BF kNN also requires an additional sorting step. However, the computation for each point in BF search is very simple and does not rely on complex data structures, making it highly parallelizable. A GPU-accelerated BF search or BF kNN may outperform some more sophisticated algorithms [83, 84]. In most engineering applications, it is also unnecessary to search the entire target point cloud \mathcal{X} —instead, searches can be restricted to a predefined local region. Therefore, BF search remains highly practical in many real-world applications.

To facilitate comparison, we will use a pair of example point clouds (see `data/ch5/first.pcd` and `second.pcd`) as data sources in this section, applying various methods to compute their nearest neighbors. Considering that readers may not have access to GPUs, we provide both single-threaded and multi-threaded CPU implementations of nearest neighbor search. Subsequent algorithms will also compare the efficiency of single-threaded versus multi-threaded implementations. For simpler methods (such as the BF method in this section), we will also compare handwritten implementations with those provided by the PCL library.

Brute-Force Nearest Neighbor Implementation

The BF (Brute-Force) matching implementation is straightforward. Using C++17’s parallel mechanisms, it’s easy to extend the single-threaded version to a multi-threaded one. We utilize STL algorithms to implement both single-threaded and multi-threaded BF matching, first defining a brute-force nearest neighbor search for a single point and then extending it to compute nearest neighbors for multiple points.

Listing 5.10: `src/ch5/bfnn.cc`

```

1 int bfnn_point(CloudPtr cloud, const Vec3f& point) {
2     return std::min_element(cloud->points.begin(), cloud->points.end(),
3     [&point](const PointType& pt1, const PointType& pt2) -> bool {
4         return (pt1.getVector3fMap() - point).squaredNorm() <
5             (pt2.getVector3fMap() - point).squaredNorm();
6     }) - cloud->points.begin();
7 }
8
9 void bfnn_cloud_mt(CloudPtr cloud1, CloudPtr cloud2, std::vector<std::pair<size_t,
10 size_t>>&
11 matches) {
12     // Generate indices first
13     std::vector<size_t> index(cloud1->size());
14     std::for_each(index.begin(), index.end(), [idx = 0](size_t& i) mutable { i = idx++;
15 });
16
17     // Parallel for_each
18     matches.resize(index.size());
19     std::for_each(std::execution::par_unseq, index.begin(), index.end(), [&](auto idx) {
20         matches[idx].second = idx;
21         matches[idx].first = bfnn_point(cloud1, ToVec3f(cloud2->points[idx]));
22     });
23 }
```

In the single-point nearest neighbor search, we take a point as input, compute the distance to every point in the cloud, and then retrieve the minimum. This is achieved using ‘`std::min_element`’ and a lambda function. The parallel version uses ‘`std::execution`’ to concurrently invoke the single-point nearest neighbor algorithm for each point.

Next, let’s examine the test program. This section uses gtest to benchmark various nearest neighbor methods for comparison:

Listing 5.11: src/ch5/test_nn.cc

```

1 TEST(CH5_TEST, BFNN) {
2     sad::CloudPtr first(new sad::PointCloudType), second(new sad::PointCloudType);
3     pcl::io::loadPCDFile(FLAGS_first_scan_path, *first);
4     pcl::io::loadPCDFile(FLAGS_second_scan_path, *second);
5
6     if (first->empty() || second->empty()) {
7         LOG(ERROR) << "cannot load cloud";
8         FAIL();
9     }
10
11    // Apply voxel grid downsampling to 0.05
12    sad::VoxelGrid(first);
13    sad::VoxelGrid(second);
14
15    // Benchmark single-threaded and multi-threaded brute-force matching
16    sad::evaluate_and_call(
17        [&first, &second]{
18            std::vector<std::pair<size_t, size_t>> matches;
19            sad::bfnn_cloud(first, second, matches);
20        },
21        "Brute-Force Matching (Single-threaded)", 5);
22    sad::evaluate_and_call(
23        [&first, &second]{
24            std::vector<std::pair<size_t, size_t>> matches;
25            sad::bfnn_cloud_mt(first, second, matches);
26        },
27        "Brute-Force Matching (Multi-threaded)", 5);
28
29    SUCCEED();
30 }
```

Here, the ‘`evaluate_and_call`’ function is used. This function invokes a specified method a fixed number of times and measures its execution time:

Listing 5.12: src/common/sys_utils.h

```

1 /**
2 * Measure code execution time
3 * @param FuncT
4 * @param func Function to be called
5 * @param func_name Function name
6 * @param times Number of calls
7 */
8 template <typename FuncT>
9 void evaluate_and_call(FuncT func, const std::string &func_name = "", int times = 10)
10 {
11     double total_time = 0;
12     for (int i = 0; i < times; ++i) {
13         auto t1 = std::chrono::high_resolution_clock::now();
14         func();
15         auto t2 = std::chrono::high_resolution_clock::now();
16         total_time += std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1).
17             count() * 1000;
18     }
19
20     LOG(INFO) << "Method " << func_name << " average call time/iterations: " <<
21         total_time / times << "/" << times << " ms.";
22 }
```

We call both the single-threaded and multi-threaded versions five times each and compute their average execution times.

Listing 5.13: Terminal output

```

1 bin/test_nn --gtest_filter=CH5_TEST.BFNN
2 Note: Google Test filter = CH5_TEST.BFNN
3 [=====] Running 1 test from 1 test suite.
4 [=====] Global test environment set-up.
5 [=====] 1 test from CH5_TEST
6 [ RUN ] CH5_TEST.BFNN
7 Failed to find match for field 'intensity'.
8 Failed to find match for field 'intensity'.
9 I0116 13:40:57.001132 267085 test_nn.cc:36] points: 18869, 18779
10 I0116 13:41:04.886138 267085 sys_utils.h:32] Method Brute-Force Matching (Single-
    threaded) average call time/iterations: 1576.98/5 ms.
11 I0116 13:41:05.291601 267085 sys_utils.h:32] Method Brute-Force Matching (Multi-
    threaded) average call time/iterations: 81.0873/5 ms.

```

As shown, for point clouds with around 18,000 points, the single-threaded brute-force matching takes approximately 1.5 seconds, while the multi-threaded version requires only 81 milliseconds. These benchmarks are highly dependent on machine performance. The tests were conducted on an i9-12900KF machine, and readers may observe different results (possibly significant variations) on their own systems. However, the relative speed differences between algorithms remain consistent for evaluation purposes.

The advantage of brute-force matching is that it computes matches for every pair of points, ensuring correctness. Subsequent methods may not guarantee this. We now use brute-force matching results as a benchmark to evaluate the performance of other approaches.

5.2.2 Grid and Voxel Methods

Brute-force search (or linear search) essentially involves traversing and searching through a data structure. Students familiar with data structures would immediately suggest that for sorted containers, binary search is clearly faster than sequential search. With a bit more recall, one might remember that binary search reduces the complexity to $O(\log_2 N)$ compared to linear search. This line of thinking leads to the binary trees and K-d trees discussed in Section 5.2.3. Tree structures index the data itself. On the other hand, since point clouds are spatial data, we can also index them spatially. Depending on the indexing method, this leads to 2D grid or 3D voxel approaches, and further to quadtrees and octrees. This section focuses on the latter category.

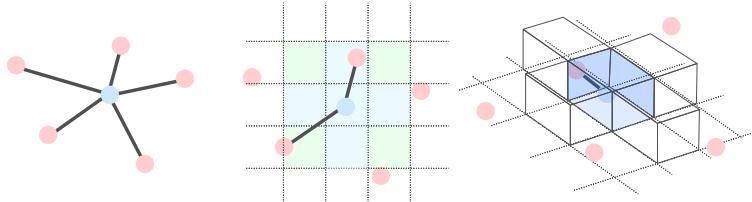


Figure 5-8: Nearest neighbor search: grid and voxel methods. In the grid method, space is divided into 2D projected grids, and the nearest points are searched within the surrounding grids of the query point. In voxel-based methods, space is divided into 3D voxels, and the search is performed in adjacent voxels. For simplicity, only four surrounding voxels are shown here; in practice, the top and bottom voxels should also be included, totaling six.

Grid-Based Nearest Neighbor Search

If we partition the point cloud spatially into grids, we can compute the grid position for each point and index all points spatially, as shown in Figure 5-8. When searching for the nearest

neighbor, we first compute the grid cell containing the query point and then search for its nearest neighbor in the surrounding grid cells. The nearest neighbor in the grid can be easily defined, such as the adjacent cells above, below, left, and right, significantly narrowing the search scope. However, there are a few considerations when generating the grid:

- Based on the density of the point cloud, we need to predefine the **resolution** of the grid, which is a **hyperparameter**—a parameter related to the computational task. If the grid cells are too large, each cell will contain many points, reducing the efficiency of nearest neighbor search. If the cells are too small, the number of cells increases, and due to the sparsity of the point cloud, the nearest neighbor might not lie in the adjacent cells, potentially missing the true nearest neighbor. This method is highly sensitive to **hyperparameters** and requires empirical tuning.
- Since grid boundaries are discrete, when searching for the nearest neighbor, we should not only search within the current grid cell but also in its **surrounding** cells. The definition of **surrounding** can vary. For 2D grids, we might consider the four adjacent cells (up, down, left, right) as "surrounding," or include the four corners for a total of eight cells. Clearly, the more surrounding cells included, the more cells need to be searched, reducing algorithmic efficiency. Thus, a reasonable value must be chosen in practice. This issue also applies to voxel methods, where we can similarly define six adjacent voxels or include the eight corners for a total of 14 neighboring voxels.
- Due to the limited scope of grid searches, it is possible to fail to find the nearest neighbor for a given point. This is quite different from brute-force search, which can match all points and thus guarantees finding the nearest neighbor for every point. Grid methods, however, do not always guarantee this. Therefore, in addition to evaluating the computational efficiency of grid methods, their **correctness** must also be assessed. The same applies to subsequent algorithms.

Voxel-Based Nearest Neighbor Search

Voxel-based nearest neighbor search is very similar to the grid method. While the grid method partitions space into 2D grids, the voxel method divides space into 3D **voxels**—think of them as small cubes forming a partitioning scheme. A voxel is directly adjacent to six surrounding voxels; including diagonals adds another eight, for a total of 14 adjacent voxels.

Implementation

Since 2D grids and voxel methods are very similar, we implement them using a template class. We use the dimensionality as a template parameter and employ 2D or 3D integer vectors as grid indices to unify the implementation of 2D and 3D grids, keeping the code as compact as possible. Additionally, we define neighboring relationships through enumerations, allowing users to specify the desired adjacency type.

First, let's look at the basic definition of grid-based nearest neighbor search:

Listing 5.14: src/ch5/gridnn.hpp

```

1 /**
2 * Grid-based nearest neighbor search
3 * @tparam dim Template parameter, using 2D or 3D voxels
4 */
5 template <int dim>
6 class GridNN {
7     public:
```

```

8  using KeyType = Eigen::Matrix<int, dim, 1>;
9  using PtType = Eigen::Matrix<float, dim, 1>;
10
11 enum class NearbyType {
12     CENTER, // Only consider the center
13     // For 2D
14     NEARBY4, // Up, down, left, right
15     NEARBY8, // Up, down, left, right + four corners
16
17     // For 3D
18     NEARBY6, // Up, down, left, right, front, back
19 };
20
21 private:
22     float resolution_ = 0.1; // Resolution
23     float inv_resolution_ = 10.0; // Inverse of resolution
24
25     NearbyType nearby_type_ = NearbyType::NEARBY4;
26     std::unordered_map<KeyType, std::vector<size_t>, hash_vec<dim>> grids_; // Grid
27         data
28     CloudPtr cloud_;
29
30     std::vector<KeyType> nearby_grids_; // Nearby grids
31 };

```

We define neighboring relationships as the enumeration type ‘NearbyType‘, and the actual grid data is stored in an ‘std::unordered_map‘ (hash table). Since point clouds are sparse, the corresponding grids are also sparse, so empty grids need not be retained where no data exists. Hash tables are highly effective in this application scenario. We define the key of this table as a 2D or 3D integer vector and implement the hash function as follows:

Listing 5.15: src/common/eigen_types.h

```

1 // Vector hash
2 template <int N>
3 struct hash_vec {
4     inline size_t operator()(const Eigen::Matrix<int, N, 1>& v) const;
5 };
6
7 template <>
8 inline size_t hash_vec<2>::operator()(const Eigen::Matrix<int, 2, 1>& v) const {
9     return size_t((v[0] * 73856093) ^ (v[1] * 471943)) % 10000000;
10 }
11
12 template <>
13 inline size_t hash_vec<3>::operator()(const Eigen::Matrix<int, 3, 1>& v) const {
14     return size_t((v[0] * 73856093) ^ (v[1] * 471943) ^ (v[2] * 83492791)) % 10000000;
15 }

```

The ‘hash_vec‘ function is a template function with two specializations for 2D and 3D spatial hash functions, respectively. According to the literature [85], spatial hash functions can be implemented by multiplying each dimension’s data by a large prime number, then taking the XOR, and finally applying modulo with a large integer. For a spatial point $\mathbf{p} = [p_x, p_y, p_z]$, using three large prime numbers n_1, n_2, n_3 and a large integer N , its hash function can be defined as:

$$\text{hash}(\mathbf{p}) = ((p_x n_1) \text{ xor } (p_y n_2) \text{ xor } (p_z n_3)) \bmod N. \quad (5.5)$$

Similarly, we can define a hash function for 2D spatial points. Combined with the template parameter ‘dim‘ of the ‘GridNN‘ class, these can be used as implementations for the hash function in ‘std::unordered_map‘. Then, we define their nearest neighbors in the ‘nearby_grids_‘ member variable:

Listing 5.16: src/ch5/gridnn.hpp

```

1 template <>
2 void GridNN<2>::GenerateNearbyGrids() {
3     if (nearby_type_ == NearbyType::CENTER) {
4         nearby_grids_.emplace_back(KeyType::Zero());
5     } else if (nearby_type_ == NearbyType::NEARBY4) {
6         nearby_grids_ = {Vec2i(0, 0), Vec2i(-1, 0), Vec2i(1, 0), Vec2i(0, 1), Vec2i(0, -1)};
7     } else if (nearby_type_ == NearbyType::NEARBY8) {
8         nearby_grids_ = {
9             Vec2i(0, 0), Vec2i(-1, 0), Vec2i(1, 0), Vec2i(0, 1), Vec2i(0, -1),
10            Vec2i(-1, -1), Vec2i(-1, 1), Vec2i(1, -1), Vec2i(1, 1),
11        };
12    }
13}
14
15 template <>
16 void GridNN<3>::GenerateNearbyGrids() {
17     if (nearby_type_ == NearbyType::CENTER) {
18         nearby_grids_.emplace_back(KeyType::Zero());
19     } else if (nearby_type_ == NearbyType::NEARBY6) {
20         nearby_grids_ = {KeyType(0, 0, 0), KeyType(-1, 0, 0), KeyType(1, 0, 0), KeyType(0, 1, 0),
21                         KeyType(0, -1, 0), KeyType(0, 0, -1), KeyType(0, 0, 1)};
22     }
23}

```

Using specialized template classes, we can define different neighbor generation methods for 2D and 3D voxels. The 2D grid allows selecting 0, 4, or 8 nearest neighbors, while the 3D voxel allows selecting 0 or 6 nearest neighbors (the 14-neighbor case is left as an exercise for the reader). Now, let's implement the nearest neighbor search logic:

1. First, compute the grid cell containing the given point.
2. Based on the nearest neighbor definition, search the nearby grid cells.
3. Collect the results from step 2 and use brute-force matching to compute the nearest neighbor within these grid cells.

Step 3 can reuse the brute-force matching code from earlier. This process is the same for both 2D and 3D grids, so we use the same interface at the code level. The single nearest neighbor search function is as follows:

Listing 5.17: src/ch5/gridnn.hpp

```

1 template <int dim>
2 bool GridNN<dim>::GetClosestPoint(const PointType& pt, PointType& closest_pt, size_t&
3     idx) {
4     // Search for the nearest neighbor in the grid cells around pt
5     std::vector<size_t> idx_to_check;
6     auto key = Pos2Grid(ToEigen<float>, dim>(pt));
7
8     std::for_each(nearby_grids_.begin(), nearby_grids_.end(), [&key, &idx_to_check, this
9         ](const KeyType& delta) {
10         auto dkey = key + delta;
11         auto iter = grids_.find(dkey);
12         if (iter != grids_.end()) {
13             idx_to_check.insert(idx_to_check.end(), iter->second.begin(), iter->second.end()
14         );
15     });
16
17     if (idx_to_check.empty()) {
18         return false;
19     }
20
21     // Brute-force NN in cloud_[idx]
22     idx = bfmn_point(cloud_, idx_to_check, ToVec3f(pt));
23     closest_pt = cloud_->points[idx];
24     return true;
25 }

```

```

23 }
24
25 template <int dim>
26 Eigen::Matrix<int, dim, 1> GridNN<dim>::Pos2Grid(const Eigen::Matrix<float, dim, 1>&
27   pt) {
28   return (pt * inv_resolution_).template cast<int>();
}

```

The point cloud version simply adds a concurrent wrapper around this function:

Listing 5.18: src/ch5/gridnn.hpp

```

1 template <int dim>
2 bool GridNN<dim>::GetClosestPointForCloudMT(CloudPtr ref, CloudPtr query,
3 std::vector<std::pair<size_t, size_t>& matches> {
4   // Essentially the same as the serial version, but matches must be preallocated,
5   // with invalid matches filled on failure
6   std::vector<size_t> index(query->size());
7   std::for_each(index.begin(), index.end(), [idx = 0](size_t& i) mutable { i = idx++; });
8   matches.resize(index.size());
9
10  std::for_each(std::execution::par_unseq, index.begin(), index.end(), [this, &matches
11    , &query][const size_t& idx] {
12    PointType cp;
13    size_t cp_idx;
14    if (GetClosestPoint(query->points[idx], cp, cp_idx)) {
15      matches[idx] = {cp_idx, idx};
16    } else {
17      matches[idx] = {math::kINVALID_ID, math::kINVALID_ID};
18    }
19  });
20
return true;
}

```

Now let's test the performance of various 2D and 3D voxel methods under different nearest neighbor definitions. Note that in addition to testing performance, we should also focus on whether these nearest neighbors are correct. In fact, grid-based nearest neighbor search may encounter two types of errors:

1. The nearest neighbor detected by the grid method is not actually the nearest neighbor in reality. This is called a **false positive** (FP). The number of false positives in an experiment is denoted as FP .
2. A true nearest neighbor in reality is not detected by the grid method. This is called a **false negative** (FN). The number of false negatives in an experiment is denoted as FN .

Using the definitions of FP and FN , we can define the algorithm's **precision** and **recall**. Let m be the total number of nearest neighbors computed by the algorithm and n be the total number of ground truth nearest neighbors. Then, precision and recall are defined as:

$$\text{precision} = \frac{1 - FP}{m}, \quad \text{recall} = \frac{1 - FN}{n} \quad (5.6)$$

Precision describes the correctness of the nearest neighbors detected by the algorithm, while recall describes the proportion of true results that the algorithm successfully detects. A well-performing algorithm should achieve high values in both metrics, but this may come at the cost of slower performance. For example, brute-force matching can achieve 100% precision and recall, but its computational time is often unacceptable.

When testing nearest neighbor algorithms, we can evaluate their precision and recall using the following method:

Listing 5.19: src/ch5/test_nn.cc

```

1 /**
2 * Evaluate the correctness of nearest neighbor matches
3 * @param truth Ground truth matches
4 * @param esti Estimated matches
5 */
6 void EvaluateMatches(const std::vector<std::pair<size_t, size_t>>& truth,
7 const std::vector<std::pair<size_t, size_t>>& esti) {
8     int fp = 0; // false-positive: exists in esti but not in truth
9     int fn = 0; // false-negative: exists in truth but not in esti
10
11    /// Check if a match exists in another container
12    auto exist = [](<const std::pair<size_t, size_t>& data, <const std::vector<std::pair<
13        size_t, size_t>>& vec) -> bool {
14        return std::find(vec.begin(), vec.end(), data) != vec.end();
15    };
16
17    for (const auto& d : esti) {
18        if (!exist(d, truth)) {
19            fp++;
20        }
21    }
22
23    for (const auto& d : truth) {
24        if (!exist(d, esti)) {
25            fn++;
26        }
27    }
28
29    float precision = 1.0 - float(fp) / esti.size();
30    float recall = 1.0 - float(fn) / truth.size();
31    LOG(INFO) << "precision: " << precision << ", recall: " << recall << ", fp: " << fp
32        << ", fn: " << fn;
33}

```

Next, we test the grid-based nearest neighbor method:

Listing 5.20: src/ch5/test_nn.cc

```

1 TEST(CH5_TEST, GRID_NN) {
2     // Code for loading point clouds omitted
3     std::vector<std::pair<size_t, size_t>> truth_matches;
4     sad::bfnn_cloud(first, second, truth_matches);
5
6     // Compare different types of grids
7     sad::GridNN<2> grid0(0.1, sad::GridNN<2>::NearbyType::CENTER), grid4(0.1, sad::
8         GridNN<2>::NearbyType::NEARBY4),
9     grid8(0.1, sad::GridNN<2>::NearbyType::NEARBY8);
10    sad::GridNN<3> grid3(0.1, sad::GridNN<3>::NearbyType::NEARBY6);
11
12    grid0.SetPointCloud(first);
13    grid4.SetPointCloud(first);
14    grid8.SetPointCloud(first);
15    grid3.SetPointCloud(first);
16
17    // Evaluate various versions of Grid NN
18    LOG(INFO) << "=====";
19    std::vector<std::pair<size_t, size_t>> matches;
20    sad::evaluate_and_call(
21        [&first, &second, &grid0, &matches]() { grid0.GetClosestPointForCloud(first, second,
22            matches); },
23        "Grid0 Single-threaded", 10);
24    EvaluateMatches(truth_matches, matches);
25
26    LOG(INFO) << "=====";
27    sad::evaluate_and_call(
28        [&first, &second, &grid0, &matches]() { grid0.GetClosestPointForCloudMT(first,
29            second, matches); },
30        "Grid0 Multi-threaded", 10);
31    EvaluateMatches(truth_matches, matches);
32
33    /// Other test methods are similar and omitted

```

32 }

We primarily compare the runtime and nearest neighbor performance of each method. Readers should observe the same precision and recall metrics, though runtime may vary.

Listing 5.21: Terminal output:

```

1 ./bin/test_nn --gtest_filter=CH5_TEST.GRID_NN
2 I0116 17:04:58.055471 276361 test_nn.cc:125] =====
3 I0116 17:04:58.065711 276361 sys_utils.h:32] Method Grid0 Single-threaded average call
   time/iterations: 1.02376/10 ms.
4 I0116 17:04:58.065724 276361 test_nn.cc:65] truth: 18869, esti: 8518
5 I0116 17:04:58.099488 276361 test_nn.cc:91] precision: 0.486382, recall: 0.219566, fp:
   4375, fn: 14726
6 I0116 17:04:58.099493 276361 test_nn.cc:132] =====
7 I0116 17:04:58.104143 276361 sys_utils.h:32] Method Grid0 Multi-threaded average call
   time/iterations: 0.464818/10 ms.
8 I0116 17:04:58.104161 276361 test_nn.cc:65] truth: 18869, esti: 18779
9 I0116 17:04:58.158778 276361 test_nn.cc:91] precision: 0.486382, recall: 0.219566, fp:
   4375, fn: 14726
10 I0116 17:04:58.158783 276361 test_nn.cc:138] =====
11 I0116 17:04:58.202162 276361 sys_utils.h:32] Method Grid4 Single-threaded average call
   time/iterations: 4.33758/10 ms.
12 I0116 17:04:58.202165 276361 test_nn.cc:65] truth: 18869, esti: 13272
13 I0116 17:04:58.246877 276361 test_nn.cc:91] precision: 0.646775, recall: 0.454926, fp:
   4688, fn: 10285
14 I0116 17:04:58.246881 276361 test_nn.cc:144] =====
15 I0116 17:04:58.254035 276361 sys_utils.h:32] Method Grid4 Multi-threaded average call
   time/iterations: 0.715278/10 ms.
16 I0116 17:04:58.254041 276361 test_nn.cc:65] truth: 18869, esti: 18779
17 I0116 17:04:58.308115 276361 test_nn.cc:91] precision: 0.646775, recall: 0.454926, fp:
   4688, fn: 10285
18 I0116 17:04:58.308118 276361 test_nn.cc:150] =====
19 I0116 17:04:58.379315 276361 sys_utils.h:32] Method Grid8 Single-threaded average call
   time/iterations: 7.11945/10 ms.
20 I0116 17:04:58.379319 276361 test_nn.cc:65] truth: 18869, esti: 14613
21 I0116 17:04:58.425294 276361 test_nn.cc:91] precision: 0.728735, recall: 0.564365, fp:
   3964, fn: 8220
22 I0116 17:04:58.425297 276361 test_nn.cc:156] =====
23 I0116 17:04:58.433573 276361 sys_utils.h:32] Method Grid8 Multi-threaded average call
   time/iterations: 0.827275/10 ms.
24 I0116 17:04:58.433579 276361 test_nn.cc:65] truth: 18869, esti: 18779
25 I0116 17:04:58.485752 276361 test_nn.cc:91] precision: 0.728735, recall: 0.564365, fp:
   3964, fn: 8220
26 I0116 17:04:58.485755 276361 test_nn.cc:162] =====
27 I0116 17:04:58.513800 276361 sys_utils.h:32] Method Grid 3D Single-threaded average
   call time/iterations: 2.80424/10 ms.
28 I0116 17:04:58.513803 276361 test_nn.cc:65] truth: 18869, esti: 8572
29 I0116 17:04:58.540259 276361 test_nn.cc:91] precision: 0.911339, recall: 0.414012, fp:
   760, fn: 11057
30 I0116 17:04:58.540262 276361 test_nn.cc:168] =====
31 I0116 17:04:58.545367 276361 sys_utils.h:32] Method Grid 3D Multi-threaded average
   call time/iterations: 0.510082/10 ms.
32 I0116 17:04:58.545372 276361 test_nn.cc:65] truth: 18869, esti: 18779
33 I0116 17:04:58.589224 276361 test_nn.cc:91] precision: 0.911339, recall: 0.414012, fp:
   760, fn: 11057

```

From the results, we can see that for grid-based methods, increasing the number of neighboring grids increases runtime, while the multi-threaded version significantly outperforms the single-threaded version. The voxel method performs similarly to the grid method in terms of efficiency⁴, and the multi-threaded version is also significantly better than the single-threaded version. For many real-time applications, nearest neighbor query times below 1 millisecond are acceptable.

In terms of precision and recall, 3D voxels significantly outperform 2D grids, and 2D grids show notable improvements as the number of neighbors increases. The grid resolution

⁴This is because we used `std::unordered_map` to index grid keys. If `std::map` were used, a noticeable efficiency difference would emerge—readers are encouraged to try this.

also affects performance. This test used a grid resolution of 0.1, which is typically too small for autonomous driving datasets. Increasing it to around 0.5 would significantly improve precision and recall but at the cost of performance. Readers are encouraged to experiment with different grid sizes to observe performance under various parameters. Note that this test evaluates single nearest neighbor cases; k -nearest neighbor metrics are generally harder to achieve.

Figure 5-9 illustrates the false negative and false positive issues inherent in grid-based nearest neighbor search. Since grids fundamentally impose rigid spatial partitioning, problems arise when a point lies near partition boundaries. In the left portion of the figure, the red point should be the true nearest neighbor of the blue point, but because it falls outside the neighboring grid cells being searched, the algorithm fails to detect it. Conversely, in the right portion, while the left red point has a greater Euclidean distance than the right red point, it gets incorrectly identified as the nearest neighbor simply because it's the only candidate within the searched neighboring grids.

Expanding the neighborhood search range may mitigate these issues probabilistically. However, even with expanded ranges, partition boundaries persist, meaning false negatives and false positives will inevitably remain at these transitional zones.

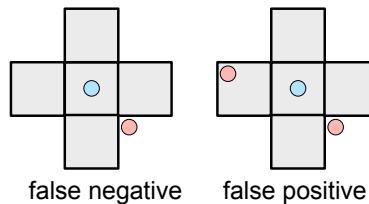


Figure 5-9: Schematic illustration of false positive/negative issues in grid-based nearest neighbor search

This naturally leads to the question: Could we abandon fixed-distance partitioning in favor of more intelligent spatial subdivision methods? This is precisely the rationale behind K-d trees. However, compared to the tree-based data structures we'll discuss later, grid and voxel methods achieve respectable performance using simpler data structures while being inherently parallelizable. Together with tree structures, voxel-based approaches form the foundation of most modern matching algorithms and SLAM systems [86–89].

The key trade-off emerges: While tree structures provide more adaptive spatial partitioning that better handles boundary cases, their increased algorithmic complexity often offsets their theoretical advantages in practical implementations where parallel processing and memory locality significantly impact real-world performance. This explains why many state-of-the-art systems employ hybrid approaches, combining the brute-force efficiency of voxel methods with the adaptive precision of tree structures where needed.

Notably, the precision/recall metrics shown in our earlier benchmarks (0.91 precision for 3D voxels vs 0.73 for 2D 8-neighbor grids) demonstrate that dimensionality plays a crucial role - the additional spatial information in 3D inherently provides better disambiguation between true and false neighbors, even using similar search methodologies. This dimensional advantage becomes particularly important in applications like autonomous driving where the vertical dimension often provides critical discriminative information.

5.2.3 Binary Trees and K-d Trees

Let us now return to the earlier idea: searching in a sorted container can significantly save time. Following this line of thought, we can propose data structures similar to binary search, such as Binary Search Trees (BST) and their high-dimensional counterpart: K-dimensional trees (K-d trees). Since point clouds belong to three-dimensional space, we will focus on introducing K-d trees.

From data structure knowledge, searching in a sorted container using binary search is more efficient than linear search: binary search has a complexity of $O(\log_2 N)$, while linear search is $O(N)$. The binary search process itself is tree-like: for a given element x and container V , we first compare x with the central element of the container. If x is smaller, we continue comparing it with the left half of the container; otherwise, we compare it with the right half. Based on this relationship, we can reorganize the container using a tree data structure to facilitate faster searches.

This process does not require predefined partitioning thresholds. Additionally, binary trees have a space complexity of $O(N)$ and a time complexity of $O(\log_2 N)$, making them an ideal search method. The only drawback is that this approach is only effective for one-dimensional data. For high-dimensional data, points may be well-separated in one dimension but overlap in another, making binary trees and binary search unsuitable for direct application.

The K-d tree [90], originally proposed by Bentley Jon Louis, is a high-dimensional extension of binary trees, as illustrated in Figure 5-10. A K-d tree is also a type of binary tree, where each node consists of left and right branches. In binary trees, we can use a single dimension to distinguish between left and right branches. However, in K-d trees, since we need to partition high-dimensional data, we use **hyperplanes** to separate the left and right branches (though for 3D points, the hyperplane is simply an ordinary 2D plane).

There are methodological differences in **how to partition** the data. Theoretically, finding a hyperplane to separate two high-dimensional point sets can be viewed as a support vector machine (SVM) classification problem [91]. However, for K-d trees in SLAM, where both construction and search processes need to run in real-time, we typically choose simpler partitioning methods. The simplest is the **axis-aligned splitting plane**. Despite its intimidating name, it simply involves splitting the point cloud along any one of the axes, making it very easy to implement.

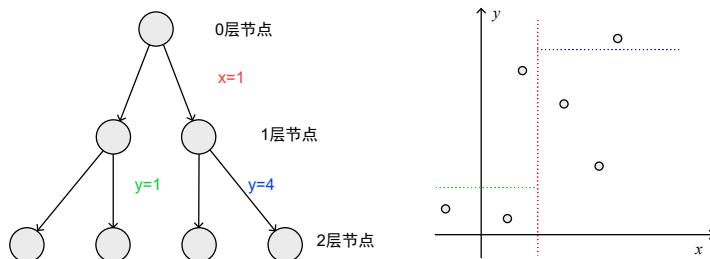


Figure 5-10: Schematic of a K-d tree. The left side shows the tree structure in the program, while the right side illustrates the partitioning relationships of each node.

We can construct a K-d tree for a point cloud of any dimensionality, referred to as the **building process** or **tree construction**. Subsequently, we can perform kNN searches for any point in space, referred to as the **search process**. Depending on the search method,

K-d trees can be categorized into **range search** (Search by range) and **K-nearest neighbor search** (Search by K nearest neighbors), with minor differences in their implementations.

In a K-d tree, we use a tree structure to represent the spatial relationships of the point cloud, with the following rules: 1. Each node has left and right branches. 2. Leaf nodes represent points in the original point cloud. In practice, we can store point indices instead of the points themselves to save space. 3. Non-leaf nodes store a splitting axis and a splitting threshold to define how the left and right branches are partitioned. For example, $x = 1$ can be stored as splitting along the first axis with a threshold of 1. We define the left branch as taking values less than the threshold and the right branch as taking values greater than the threshold.

With these conventions, we can implement the construction and search algorithms for K-d trees. Below, we briefly describe the algorithmic steps and then provide implementations and discussions of the results. Since K-d trees are fundamentally tree structures, most algorithms can be concisely implemented using recursion.

5.2.4 Construction of K-d Trees

In the construction process of a K-d tree, we primarily consider how to partition a given point cloud. Different partitioning strategies exist depending on the method used. Traditional approaches either alternate coordinate axes in a fixed order [92] or calculate the dispersion of the current point cloud along each axis and select the axis with the highest dispersion as the splitting axis. Here, we focus on the latter method. Additionally, there are various variants such as implicit K-d trees [93], min-max K-d trees [94], and relaxed K-d trees [95], which employ different strategies to handle partitioning or leaf node storage. For now, we will focus on the basic K-d tree.

Construction of a K-d Tree:

1. **Input:** Point cloud data $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$, where $\mathbf{x}_i \in \mathbb{R}^k$.
2. Consider inserting subset $\mathbf{X}_n \subset \mathbf{X}$ into node n :
3. If \mathbf{X}_n is empty, exit.
4. If \mathbf{X}_n contains only one point, mark it as a leaf node and exit.
5. Calculate the variance of \mathbf{X}_n along each axis and select the axis j with the highest dispersion. Use the mean $m_j = \mathbf{X}_n[j]$ as the splitting threshold.
6. For each $\mathbf{x} \in \mathbf{X}_n$, if $\mathbf{x}[j] < m_j$, insert it into the left child node; otherwise, insert it into the right child node.
7. Recursively apply the above steps until all points are inserted into the tree.

The above algorithm can be easily implemented recursively.

5.2.5 Searching in K-d Trees

Searching in a K-d tree essentially involves traversing a binary tree. Similar to binary trees, traversal methods such as pre-order, in-order, and post-order can be applied. The unique feature of K-d trees is that we can prune unnecessary branches to improve search efficiency.

Figure 5-11 illustrates the process of searching for a query point in a K-d tree. Note that although the query point (blue) falls on the left side of the K-d tree, its nearest neighbor may

not necessarily be on the left. The position of the splitting plane is determined by the point cloud distribution during tree construction. During the search, the nearest neighbor can be on either side. However, due to the splitting plane, points on the right side have a minimum distance d to the query point, which is the perpendicular distance from the query point to the splitting plane. If a point on the left side is found with a distance smaller than d , the right branch can be pruned. Conversely, if the nearest neighbor on the left is farther than d , the right branch must be searched. This is the fundamental principle of K-d tree traversal.

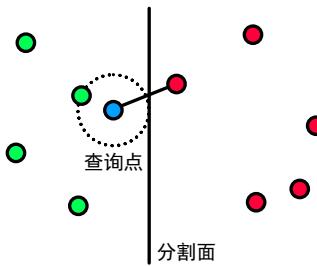


Figure 5-11: Schematic of K-d tree pruning. In this figure, the distance from the splitting plane of the right branch exceeds the current nearest neighbor distance, indicating that points on the right must be farther than the current best solution. Thus, the right branch can be skipped.

Based on this principle, the nearest neighbor search in a K-d tree can be described as follows:

Nearest Neighbor Search in K-d Trees:

1. **Input:** K-d tree T , query point \mathbf{x} .
2. **Output:** Nearest neighbor of \mathbf{x} .
3. Let the current node be n_c , initially the root node. Let d be the current minimum distance.
 - (a) If n_c is a leaf, compute the distance between n_c and \mathbf{x} . If it is smaller than d , update n_c as the nearest neighbor and backtrack to its parent.
 - (b) If n_c is not a leaf, determine which side of n_c \mathbf{x} lies on. If the side containing \mathbf{x} has not been explored, prioritize exploring that side.
 - (c) Determine whether to explore the other side of n_c . Compute the distance d' from \mathbf{x} to the splitting plane of n_c . If $d' < d$, the other side must be explored; otherwise, skip it.
 - (d) If both sides have been explored or need not be explored, backtrack to the parent node until n_c becomes the root node.

For the k -nearest neighbor problem, the nearest neighbor becomes a set of points, and the distance threshold d is updated dynamically:

k -Nearest Neighbor Search in K-d Trees:

1. **Input:** K-d tree T , query point \mathbf{x} , number of neighbors k .

2. **Output:** Set of k -nearest neighbors N .
3. Let the current node be n_c , initially the root node. Define $S(n_c)$ as the k -nearest neighbor search under n_c :
 - (a) If n_c is a leaf, compute its distance to \mathbf{x} . If it is smaller than the largest distance in N , add n_c to N . If $|N| > k$, remove the farthest point in N .
 - (b) Determine which side of n_c \mathbf{x} lies on and recursively call $S(n_c.\text{left})$ or $S(n_c.\text{right})$.
 - (c) Determine whether to explore the other side of n_c . The exploration condition is: if $|N| < k$, explore; if $|N| = k$ and the distance from \mathbf{x} to the splitting plane is less than the largest distance in N , explore.
 - (d) If the other side need not be explored, return; otherwise, continue the search on the other side.

In the worst case, this process may traverse the entire tree. If the query point lies near multiple splitting planes and the other side contains closer points, the K-d tree degenerates to linear search (brute-force), and its performance may even be worse due to the overhead of node traversal. The time complexity for searching a single nearest neighbor is logarithmic $O(\log_2 N)$, while searching for k neighbors depends on the distribution of the point cloud. If the point cloud is well-distributed and the initial k neighbors are close, pruning can significantly reduce the search space. However, for large k or poorly distributed point clouds, more branches may need to be traversed.

In practice, it is often unnecessary to strictly find the exact k -nearest neighbors (e.g., if the k -th neighbor is much farther away). Thus, optimizations can be applied, such as setting a maximum number of points to visit or a timeout to terminate the search early. Another common optimization is to introduce a ratio α for the pruning condition. Let d_{\max} be the maximum distance in the current k -nearest neighbors and d_{split} be the distance to the splitting plane. If $d_{\text{split}} > \alpha d_{\max}$, the branch is pruned. Choosing $\alpha \leq 1$ allows for more aggressive pruning.

Implementation of K-d Tree Construction

Below, we will briefly implement a K-d tree. While there are many open-source implementations of K-d trees, the teaching process often uses simpler and more understandable versions. We will first implement our own K-d tree and then compare it with the K-d tree in PCL, as well as with the algorithms mentioned earlier.

First, let's define the basic structure of the K-d tree. Each node contains pointers to its left and right children, as well as information about the splitting plane:

Listing 5.22: src/ch5/kdtree.cc

```

1 struct KdTreeNode {
2     int id_ = -1;
3     int point_idx_ = 0;           // Point index
4     int axis_index_ = 0;          // Splitting axis
5     float split_thresh_ = 0.0;    // Splitting threshold
6     KdTreeNode* left_ = nullptr;   // Left subtree
7     KdTreeNode* right_ = nullptr;  // Right subtree
8     KdTreeNode* up_ = nullptr;     // Parent node
9
10    bool IsLeaf() const { return left_ == nullptr && right_ == nullptr; } // Whether it
11    is a leaf
12};

```

Each node stores the splitting axis and threshold. For example, if the splitting axis is the first axis (e.g., x-axis) and the threshold is 0.5, points with $x < 0.5$ will be placed in the left subtree, while points with $x \geq 0.5$ will be placed in the right subtree. Additionally, we record the ID of each node to facilitate indexing the tree without writing recursive code. Below are the basic member variables of the K-d tree:

Listing 5.23: src/ch5/kdtree.h

```

1 class KdTree {
2     private:
3         int k_ = 5;                                // Number of nearest neighbors for KNN
4         std::shared_ptr<KdTreeNode> root_ = nullptr; // Root node
5         std::vector<Vec3f> cloud_;                // Input point cloud
6         std::map<int, KdTreeNode*> nodes_;        // For bookkeeping
7         size_t size_ = 0;                          // Number of leaf nodes
8         int tree_node_id_ = 0;                     // Assigns IDs to K-d tree nodes
9     };

```

The K-d tree class holds a pointer to the root node, allowing traversal of the entire tree. We also maintain a map of node IDs to node pointers. Next, let's implement the tree construction process. Building the tree requires an input point cloud. We record the point cloud indices in the leaf nodes:

Listing 5.24: src/ch5/kdtree.cc

```

1 bool KdTree::BuildTree(const CloudPtr &cloud) {
2     // Input validation code omitted
3     IndexVec idx(cloud->size());
4     for (int i = 0; i < cloud->points.size(); ++i) {
5         idx[i] = i;
6     }
7
8     Insert(idx, root_.get());
9
10    return true;
11}
12
13 void KdTree::Insert(const IndexVec &points, KdTreeNode *node) {
14     nodes_.insert({node->id_, node});
15
16     if (points.empty()) {
17         return;
18     }
19
20     if (points.size() == 1) {
21         // Leaf node
22         size_++;
23         node->point_idx_ = points[0];
24         return;
25     }
26
27     IndexVec left, right;
28     FindSplitAxisAndThresh(points, node->axis_index_, node->split_thresh_, left, right);
29
30     const auto create_if_not_empty = [&node, this](KdTreeNode *&new_node, const IndexVec &index) {
31         if (!index.empty()) {
32             new_node = new KdTreeNode;
33             new_node->up_ = node;
34             new_node->id_ = tree_node_id_++;
35
36             Insert(index, new_node);
37         }
38     };
39
40     create_if_not_empty(node->left_, left);
41     create_if_not_empty(node->right_, right);
42 }
43
44 bool KdTree::FindSplitAxisAndThresh(const IndexVec &point_idx, int &axis, float &th,
45     IndexVec &left, IndexVec &right) {

```

```

45 // Calculate variance along each axis using functions from math_utils.h
46 Vec3f var;
47 Vec3f mean;
48 math::ComputeMeanAndCovDiag(point_idx, mean, var, [this](const int &idx) { return
49     cloud_[idx]; });
50 int max_i, max_j;
51 var.maxCoeff(&max_i, &max_j);
52 axis = max_i;
53 th = mean[axis];
54
55 if (var.squaredNorm() < 1e-7) {
56     // Edge case: All points have the same value. Split into left and right halves
57     // arbitrarily.
58     // Rare in real data but possible in synthetic data.
59     for (int i = 0; i < point_idx.size(); ++i) {
60         if (i < point_idx.size() / 2) {
61             left.emplace_back(point_idx[i]);
62         } else {
63             right.emplace_back(point_idx[i]);
64         }
65     }
66     return true;
67 }
68
69 for (const auto &idx : point_idx) {
70     if (cloud_[idx][axis] < th) {
71         left.emplace_back(idx);
72     } else {
73         right.emplace_back(idx);
74     }
75 }
76
77 if (point_idx.size() > 1) {
78     // For non-trivial splits, ensure both subtrees are non-empty
79     assert(left.empty() == false && right.empty() == false);
80 }
81
82 return true;
83 }
```

Here's the translation preserving all LaTeX commands exactly as in the original:

5.2.6 Implementation of K-d Tree Construction

The tree construction process is straightforward, implemented through recursive calls to the `Insert` function. If the input parameter `points` contains only one point, the current `node` becomes a leaf, and the point index is directly assigned. Otherwise, we calculate the variance along each axis, select the axis with the highest variance as the splitting axis, and use the mean value as the threshold. To handle edge cases where all points share identical coordinates (resulting in zero variance), we include a variance check.

Below is a test case for the tree construction process. We define four points on the $z = 0$ plane and observe the resulting tree structure:

Listing 5.25: src/ch5/test_nn.cc

```

1 TEST(CH5_TEST, KDTREE_BASICS) {
2     sad::CloudPtr cloud(new sad::PointCloudType);
3     sad::PointType p1, p2, p3, p4;
4     p1.x = 0; p1.y = 0; p1.z = 0;
5     p2.x = 1; p2.y = 0; p2.z = 0;
6     p3.x = 0; p3.y = 1; p3.z = 0;
7     p4.x = 1; p4.y = 1; p4.z = 0;
8
9     cloud->points.push_back(p1);
10    cloud->points.push_back(p2);
11    cloud->points.push_back(p3);
12    cloud->points.push_back(p4);
13 }
```

```

14     sad::KdTree kdtree;
15     kdtree.BuildTree(cloud);
16     kdtree.PrintAll();
17
18     SUCCEED();
19 }
```

Compile and run the program:

Listing 5.26: Terminal input:

```

1 bin/test_nn --gtest_filter=CH5_TEST.KDTREE_BASICS
2 Note: Google Test filter = CH5_TEST.KDTREE_BASICS
3 =====
4 [ RUN      ] Global test environment set-up.
5 [ RUN      ] 1 test from CH5_TEST
6 [ RUN      ] CH5_TEST.KDTREE_BASICS
7 I0118 10:25:14.149652 295100 kdtree.cc:241] node: 0, axis: 0, th: 0.5
8 I0118 10:25:14.149777 295100 kdtree.cc:241] node: 1, axis: 1, th: 0.5
9 I0118 10:25:14.149780 295100 kdtree.cc:239] leaf node: 2, idx: 0
10 I0118 10:25:14.149780 295100 kdtree.cc:239] leaf node: 3, idx: 2
11 I0118 10:25:14.149781 295100 kdtree.cc:241] node: 4, axis: 1, th: 0.5
12 I0118 10:25:14.149782 295100 kdtree.cc:239] leaf node: 5, idx: 1
13 I0118 10:25:14.149783 295100 kdtree.cc:239] leaf node: 6, idx: 3
14 [ OK ] CH5_TEST.KDTREE_BASICS (0 ms)
```

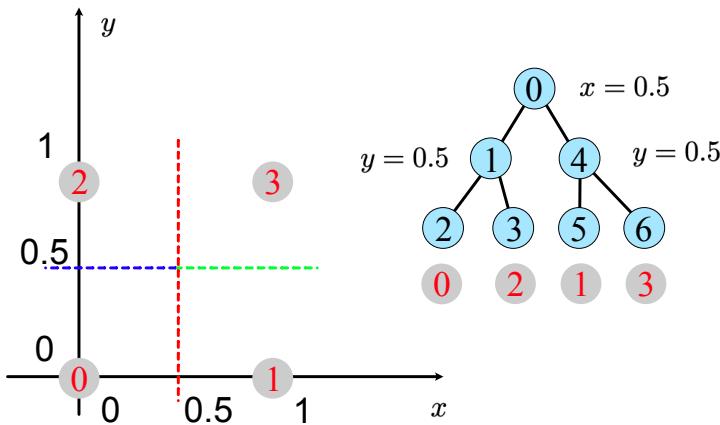


Figure 5-12: Schematic of the K-d tree construction for this example. Gray circles show point cloud IDs, while blue circles indicate K-d tree node IDs.

The experimental results are illustrated in Figure 5-12, displaying the splitting thresholds and leaf node information of the K-d tree. The constructed tree for these four points has two levels, with each splitting plane positioned at the midpoint between two points. Notably, since each split divides the remaining point cloud into two equal subsets, this K-d tree remains balanced, avoiding significant imbalance between left and right subtrees.

Key observations: 1. **Axis Selection:** The first split occurs along the x-axis (axis 0) at $x = 0.5$, separating points p_1, p_3 (left) from p_2, p_4 (right). 2. **Recursive Splitting:** Each subset is further split along the y-axis (axis 1) at $y = 0.5$, resulting in four leaf nodes. 3. **Balance:** The tree depth is minimized due to equal partitioning, ensuring optimal search performance.

This example demonstrates how K-d trees adaptively partition space based on data distribution, a property critical for efficient nearest neighbor searches in higher-dimensional datasets. The balance achieved here is a direct consequence of uniform point spacing, though

real-world data may require additional balancing techniques (e.g., median-based splits) to maintain efficiency.

Implementing K-Nearest Neighbors in K-d Trees

Now let's implement the K-nearest neighbors (KNN) algorithm for K-d trees. When $K = 1$, this naturally becomes the nearest neighbor algorithm, so we don't need to implement a separate nearest neighbor search. First, we define a structure to record nodes and distances, along with its sorting method:

Listing 5.27: src/ch5/kdtree.h

```

1 // For recording KNN results
2 struct NodeAndDistance {
3     NodeAndDistance(KdTreeNode* node, float dis2) : node_(node), distance2_(dis2) {}
4     KdTreeNode* node_ = nullptr;
5     float distance2_ = 0; // Squared distance for comparison
6
7     bool operator<(const NodeAndDistance& other) const { return distance2_ < other.
8         distance2_; }

```

We then use a priority queue to manage the KNN results. This queue maintains order during insertion, making it suitable for storing nearest neighbor results. The K-d tree nearest neighbor search is implemented as follows:

Listing 5.28: src/ch5/kdtree.cc

```

1 // Compute K nearest neighbors
2 bool KdTree::GetClosestPoint(const PointType &pt, std::vector<int> &closest_idx, int k
3 ) {
4     if (k > size_) {
5         LOG(ERROR) << "cannot set k larger than cloud size: " << k << ", " << size_;
6         return false;
7     }
8     k_ = k;
9
10    std::priority_queue<NodeAndDistance> knn_result;
11    Knn(ToVec3f(pt), root_.get(), knn_result);
12
13    // Sort and return results
14    closest_idx.resize(knn_result.size());
15    for (int i = closest_idx.size() - 1; i >= 0; --i) {
16        // Insert in reverse order
17        closest_idx[i] = knn_result.top().node_->point_idx_;
18        knn_result.pop();
19    }
20    return true;
21}
22
23 // Search for nearest neighbors of pt under node and add to result queue
24 void KdTree::Knn(const Vec3f &pt, KdTreeNode *node, std::priority_queue<
25     NodeAndDistance> &knn_result) const {
26     if (node->IsLeaf()) {
27         // If leaf, check if it can be inserted
28         ComputeDisForLeaf(pt, node, knn_result);
29     }
30
31     // Determine which side pt falls on and prioritize searching that subtree
32     // Then check if the other subtree needs to be searched
33     KdTreeNode *this_side, *that_side;
34     if (pt[node->axis_index_] < node->split_thresh_) {
35         this_side = node->left_;
36         that_side = node->right_;
37     } else {
38         this_side = node->right_;
39         that_side = node->left_;

```

```

40
41     Knn(pt, this_side, knn_result);
42     if (NeedExpand(pt, node, knn_result)) { // Note: comparison is with current best
43         Knn(pt, that_side, knn_result);
44     }
45 }
46
47 void KdTree::ComputeDisForLeaf(const Vec3f &pt, KdTreeNode *node,
48 std::priority_queue<NodeAndDistance> &knn_result) const {
49     // Compare with current results and insert if better than worst distance
50     float dis2 = Dis2(pt, cloud_[node->point_idx_]);
51     if (knn_result.size() < k_) {
52         // Not enough results yet
53         knn_result.push({node, dis2});
54     } else {
55         // Already have k results, compare with current worst
56         if (dis2 < knn_result.top().distance2_) {
57             knn_result.push({node, dis2});
58             knn_result.pop();
59         }
60     }
61 }
62
63 bool KdTree::NeedExpand(const Vec3f &pt, KdTreeNode *node, std::priority_queue<
64     NodeAndDistance> &knn_result) const {
65     if (knn_result.size() < k_) {
66         return true;
67     }
68     // Check splitting plane distance to see if better results might exist
69     float d = pt[node->axis_index_] - node->split_thresh_;
70     if ((d * d) < knn_result.top().distance2_) {
71         return true;
72     } else {
73         return false;
74     }
75 }
```

The nearest neighbor function recursively calls the Knn function to perform the search. For leaf nodes, it compares the node's distance with the worst distance in the current KNN results. If better, it inserts into the priority queue. The NeedExpand function determines whether to search the other subtree based on the splitting plane distance.

Let's test our K-d tree implementation. We compare against brute-force results for ground truth and benchmark against PCL's K-d tree:

Listing 5.29: Terminal input:

```

1 bin/test_nn --gtest_filter=CH5_TEST.KDTREE_KNN
2 I0118 17:07:02.307432 318029 sys_utils.h:32] Method Kd Tree build avg time/calls:
3     4.34059/1 ms.
4 I0118 17:07:02.307545 318029 test_nn.cc:234] Kd tree leaves: 18869, points: 18869
5 I0118 17:07:03.029914 318029 sys_utils.h:32] Method Kd Tree 5NN multi-thread avg time/
6     calls: 3.01146/1 ms.
7 I0118 17:07:03.030046 318029 test_nn.cc:65] truth: 93895, esti: 93895
8 I0118 17:07:04.396924 318029 test_nn.cc:93] precision: 1, recall: 1, fp: 0, fn: 0
9 I0118 17:07:04.396939 318029 test_nn.cc:246] building kdtree pcl
10 I0118 17:07:04.398665 318029 sys_utils.h:32] Method Kd Tree build avg time/calls:
11     1.68209/1 ms.
12 I0118 17:07:04.398671 318029 test_nn.cc:252] searching pcl
13 I0118 17:07:05.411710 318029 sys_utils.h:32] Method Kd Tree 5NN in PCL avg time/calls:
14     12.9858/1 ms.
15 I0118 17:07:04.411908 318029 test_nn.cc:65] truth: 93895, esti: 93895
16 I0118 17:07:05.779804 318029 test_nn.cc:93] precision: 1, recall: 1, fp: 0, fn: 0
17 I0118 17:07:05.779814 318029 test_nn.cc:274] done.
```

The K-d tree achieves perfect precision and recall (100%). Compared to PCL's implementation, our K-d tree takes longer to build (likely due to maintaining an additional node list) but demonstrates faster concurrent nearest neighbor searches.

Implementing K-d Tree Construction and Nearest Neighbor Search

Through code implementation, we observe that the most critical part of the K-d tree nearest neighbor algorithm is **pruning**, and the pruning condition is that no closer nearest neighbor exists on the other side of the tree structure. Let the current farthest nearest neighbor distance be d_{max} , and the splitting plane distance be d_{split} . The pruning criterion can then be expressed as:

$$d_{split} > d_{max}. \quad (5.7)$$

However, in the worst-case scenario, if the current nearest neighbor found is poor, we might need to search distant branches for a potentially closer neighbor. This is a situation we wish to avoid. Therefore, we can slightly modify the above criterion by introducing a scaling factor α :

$$d_{split} > \alpha d_{max}. \quad (5.8)$$

When $\alpha < 1$, the pruning condition is relaxed, making the K-nearest neighbor search faster, but we no longer guarantee finding the **exact nearest neighbor** (since their branches may have been pruned). This approach is referred to as **Approximate Nearest Neighbor (ANN)**. The ANN problem is more flexible than the KNN problem, allowing for various algorithmic and data structure approximations. In our code, we implement it as follows:

Listing 5.30: src/ch5/kdtree.cc

```

1 bool KdTree::NeedExpand(const Vec3f &pt, KdTreeNode *node, std::priority_queue<
2     NodeAndDistance> &knn_result) const {
3     if (knn_result.size() < k_) {
4         return true;
5     }
6
7     if (approximate_) {
8         float d = pt[node->axis_index_] - node->split_thresh_;
9         if ((d * d) < knn_result.top().distance2_ * alpha_) {
10             return true;
11         } else {
12             return false;
13         }
14     } else {
15         // Check the splitting plane distance to see if a closer neighbor exists
16         float d = pt[node->axis_index_] - node->split_thresh_;
17         if ((d * d) < knn_result.top().distance2_) {
18             return true;
19         } else {
20             return false;
21         }
22     }
}

```

The performance of the K-d tree with approximate nearest neighbor enabled (with $\alpha = 0.1$) is as follows:

Listing 5.31: Terminal output:

```

1 I0118 17:31:41.829752 320541 sys_utils.h:32] Method Kd Tree build average call time/
2 count: 4.34565/1 ms.
3 I0118 17:31:41.829856 320541 test_nn.cc:227] Kd tree leaves: 18869, points: 18869
4 I0118 17:31:42.553129 320541 sys_utils.h:32] Method Kd Tree 5NN multithreaded average
5 call time/count: 2.10658/1 ms.
6 I0118 17:31:42.553233 320541 test_nn.cc:65] truth: 93895, esti: 93895
7 I0118 17:31:44.371816 320541 test_nn.cc:91] precision: 0.771319, recall: 0.771319, fp:
8 21472, fn: 21472
9 I0118 17:31:44.371834 320541 test_nn.cc:239] building kdtree pcl
10 I0118 17:31:44.373656 320541 sys_utils.h:32] Method Kd Tree build average call time/
11 count: 1.80198/1 ms.
12 I0118 17:31:44.373662 320541 test_nn.cc:244] searching pcl
13 I0118 17:31:44.387229 320541 sys_utils.h:32] Method Kd Tree 5NN in PCL average call
14 time/count: 13.5384/1 ms.

```

```

10 I0118 17:31:44.387405 320541 test_nn.cc:65] truth: 93895, esti: 93895
11 I0118 17:31:45.723769 320541 test_nn.cc:91] precision: 1, recall: 1, fp: 0, fn: 0
12 I0118 17:31:45.723780 320541 test_nn.cc:266] done.

```

It can be seen that the approximate method leads to some decline in precision and recall, with both metrics dropping to around 0.77, but there is an improvement in search speed. Compared to the algorithms in the previous section, the K-d tree is significantly better than brute-force search but slower than 2D and 3D voxel-based methods. If construction time is considered, it is even slower. In terms of performance, the K-d tree allows tuning the α parameter, enabling a balance between speed and accuracy. In contrast, grid-based methods, while very fast, often struggle to achieve satisfactory recall rates.

The K-d tree in this chapter is implemented recursively, making the code concise, but it may encounter stack overflow issues with extremely large point clouds. Readers can modify it to an iterative implementation. Additionally, there are many improved versions of the K-d tree, which we will not implement here. Interested readers can explore open-source K-d tree algorithms such as [96] for comparison. Furthermore, to save space, we have omitted discussions on other topics, such as how to delete a point from a K-d tree, how to balance a K-d tree, etc. Readers interested in these aspects are encouraged to refer to relevant literature.

5.2.7 Quadtree and Octree

The K-d tree methods introduced earlier use a binary tree as the basic data structure. However, can we have more branches beyond just binary trees? The answer is yes. In two-dimensional and three-dimensional spaces, we have two corresponding approaches: the **Quadtree** [97] and the **Octree** [98, 99]. If we do not partition space using grids, other tree-based methods can also be derived. Since these two approaches follow the same underlying principle—only differing in spatial dimensionality—we will discuss them together.

In a quadtree, each node has four children, while in an octree, each node has eight. This naturally corresponds to physical space: a rectangle can be divided into four equal parts by its center, and a 3D cube can be split into eight equal parts. The parent node represents the larger rectangle/cube, and the child nodes represent the subdivided rectangles/cubes, forming the quadtree and octree structures. This structure inherently defines the rules for space partitioning. Therefore, similar to the K-d tree, we can construct a quadtree/octree model for a 2D/3D point cloud and use analogous methods to perform nearest neighbor searches. Due to their more uniform spatial partitioning, quadtrees and octrees can also be used to describe data covering an entire space, such as maps.

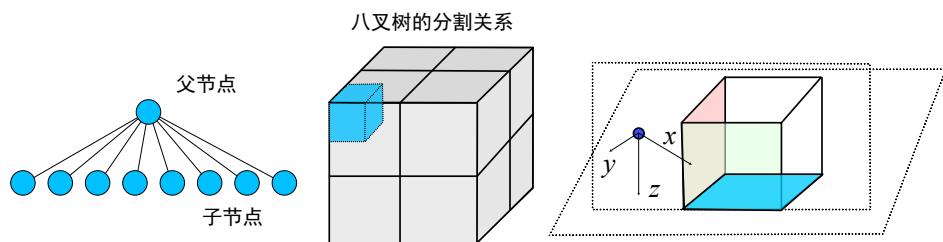


Figure 5-13: Schematic diagram of octree principles and distance computation. Left: The data structure of an octree; Middle: The partitioning method of an octree; Right: The distance calculation from an external point to a cube.

Octree Construction

Our focus is on 3D point cloud problems, so here we implement an octree algorithm, which also serves as a performance comparison experiment for readers. To maintain consistency, we try to mimic the K-d tree interface when implementing the octree, which also demonstrates their similarity. The main differences between octrees and K-d trees are:

1. K-d trees use splitting planes to distinguish point sets, while octrees use cubic forms. For this purpose, we implemented a `Box3D` structure to handle the relationship between points and the octree.
2. When building a K-d tree, the splitting planes are dynamically determined; in octrees, the subdivision of cubes is fixed (dividing into eight parts from the center). This means an octree node can always be expanded, but its child nodes may contain no point cloud. In this implementation, we retain those leaf nodes without corresponding point clouds. Of course, readers can choose to delete these leaf nodes, which would reduce the total number of nodes in the tree.
3. During initial octree construction, we compute the bounding box of the entire point cloud as the root node's boundary box. This bounding box doesn't need to be a perfect cube. We allow it to be longer in certain dimensions, as long as subsequent subdivisions still follow the **eight-way split** rule. Readers can also enforce a perfect cube bounding box, but this may make the tree deeper and reduce search efficiency.
4. The nearest neighbor search process in octrees is similar to K-d trees and also involves pruning. We use the **maximum perpendicular distance from the query point to the outer boundary of the bounding box** as the pruning criterion (see Figure 5-13 for illustration). In this schematic, the query point is outside the grid in the x-direction but inside in the y and z directions. Therefore, the lower bound for the distance between the query point and the point cloud inside the cube should be the distance in the x-direction. If two or three axes are outside the cube, the longest axis should be taken as the distance lower bound. Readers can visualize this themselves. Of course, this lower bound could be estimated more precisely (e.g., by calculating the distance between the query point and the eight vertices of the cube), but we should ensure the distance calculation method is as simple as possible while maintaining effective pruning.

With these differences in mind, we describe the octree construction and search algorithms:

Octree Construction:

1. Input: Point cloud data $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$, where $\mathbf{x}_i \in \mathbb{R}^k$.
2. Consider inserting subset $\mathbf{X}_n \subseteq \mathbf{X}$ into node n :
3. If \mathbf{X}_n is empty, exit;
4. If \mathbf{X}_n contains only one point, mark it as a leaf node and exit;
5. Expand n following the **eight-way split** rule.
6. For each $\mathbf{x} \in \mathbf{X}_n$, record which child node contains \mathbf{x} . Then recursively apply the construction method to the child node and its corresponding point cloud.

7. Repeat the above steps until all points are inserted into the tree.

Octree Search

The k-nearest neighbor search algorithm for octrees can also be obtained by slightly modifying the K-d tree approach:

Octree k-Nearest Neighbor Search:

1. Input: Octree T , query point \mathbf{x} , number of neighbors k ;
2. Output: k-nearest neighbor set N ;
3. Let the current node be n_c (initially the root node). Define function $S(n_c)$ as performing k-nearest neighbor search under n_c :
 - (a) If n_c is a leaf, calculate whether the distance between n_c and \mathbf{x} is smaller than the maximum distance in N ; if so, add n_c to N . If $|N| > k$, remove the farthest matching point from N ;
 - (b) Determine which child node of n_c contains \mathbf{x} . If \mathbf{x} is outside n_c 's bounding box, expand all child nodes; if inside, prioritize expanding the child node containing \mathbf{x} .
 - (c) Determine whether to expand other child nodes of n_c . The expansion condition: if $|N| < k$, expansion is mandatory; if $|N| = k$ and the previously calculated distance between \mathbf{x} and n_c is less than the maximum matching distance in N , also expand;
 - (d) If n_c 's child nodes don't need expansion, return; otherwise, continue calling the neighbor search algorithm for other nodes.

Code Implementation

Now let's implement the octree described earlier. An octree node consists of its bounding box and eight child nodes, defined as follows:

Listing 5.32: src/ch5/octo_tree.h

```

1 struct Box3D {
2     Box3D() = default;
3     Box3D(float min_x, float max_x, float min_y, float max_y, float min_z, float max_z)
4         : min_{min_x, min_y, min_z}, max_{max_x, max_y, max_z} {}
5
6     float min_[3] = {0};
7     float max_[3] = {0};
8 };
9
10 // Octree node
11 struct OctoTreeNode {
12     int id_ = -1;                                // Point index, -1 means invalid
13     int point_idx_ = -1;                          // Whether bounding box is set
14     bool box_set_ = false;                         // Bounding box
15     Box3D box_;                                 // Child nodes
16     OctoTreeNode* children[8] = {nullptr};        // Parent node
17     OctoTreeNode* parent_ = nullptr;               // Parent node
18 };

```

Each bounding box consists of minimum and maximum values along three axes. The actual code also implements its distance calculation functions, which we won't show entirely here. Now let's look at the tree construction code:

Listing 5.33: src/ch5/octo_tree.cc

```

1  bool OctoTree::BuildTree(const CloudPtr &cloud) {
2    // Some validation code omitted
3    // Generate root node's bounding box
4    root_>SetBox(ComputeBoundingBox());
5    Insert(idx_, root_.get());
6    return true;
7  }
8
9  void OctoTree::Insert(const IndexVec &points, OctoTreeNode *node) {
10    nodes_.insert({node->id_, node});
11
12    if (points.empty()) {
13      return;
14    }
15
16    if (points.size() == 1) {
17      size_++;
18      node->point_idx_ = points[0];
19      return;
20    }
21
22    /// Continue expanding this node as long as point count isn't 1
23    std::vector<IndexVec> children_points;
24    ExpandNode(node, points, children_points);
25
26    /// Perform insertion for child nodes
27    for (size_t i = 0; i < 8; ++i) {
28      Insert(children_points[i], node->children[i]);
29    }
30  }
31
32  void OctoTree::ExpandNode(OctoTreeNode *node, const IndexVec &parent_idx, std::vector<
33    IndexVec> &children_idx) {
34    children_idx.resize(8);
35    for (int i = 0; i < 8; ++i) {
36      node->children[i] = new OctoTreeNode();
37      node->children[i]->parent_ = node;
38      node->children[i]->id_ = tree_node_id_++;
39    }
40
41    const Box3D &b = node->box_; // Current node's box
42    // Center point
43    float c_x = 0.5 * (node->box_.min_[0] + node->box_.max_[0]);
44    float c_y = 0.5 * (node->box_.min_[1] + node->box_.max_[1]);
45    float c_z = 0.5 * (node->box_.min_[2] + node->box_.max_[2]);
46
47    // Schematic of 8 sub-boxes
48    // First layer: top-left 1, top-right 2, bottom-left 3, bottom-right 4
49    // Second layer: top-left 5, top-right 6, bottom-left 7, bottom-right 8
50    //   ---> x /-----/-----/ | |
51    //   / | /-----/-----/ | |
52    //   y | z | | | | | | | |
53    //   | | |-----|-----| | | |
54    //   | | | | | | | | | |
55    //   |-----|-----|-----| |
56    node->children[0]->SetBox({b.min_[0], c_x, b.min_[1], c_y, b.min_[2], c_z});
57    node->children[1]->SetBox({c_x, b.max_[0], b.min_[1], c_y, b.min_[2], c_z});
58    node->children[2]->SetBox({b.min_[0], c_x, c_y, b.max_[1], b.min_[2], c_z});
59    node->children[3]->SetBox({c_x, b.max_[0], c_y, b.max_[1], b.min_[2], c_z});
60
61    node->children[4]->SetBox({b.min_[0], c_x, b.min_[1], c_y, c_z, b.max_[2]});
62    node->children[5]->SetBox({c_x, b.max_[0], b.min_[1], c_y, c_z, b.max_[2]});
63    node->children[6]->SetBox({b.min_[0], c_x, c_y, b.max_[1], c_z, b.max_[2]});
64    node->children[7]->SetBox({c_x, b.max_[0], c_y, b.max_[1], c_z, b.max_[2]});
65
66    // Assign points to child nodes
67    for (const auto &idx : parent_idx) {
68      const auto pt = cloud_[idx];
69      for (int i = 0; i < 8; ++i) {
70        if (node->children[i]->box_.Inside(pt)) {
71          children_idx[i].emplace_back(idx);
72          break;
73        }
74      }
75    }
76  }

```

```

73     }
74   }
75 }
76 }
```

Special attention should be paid to the calculation order of width, height and depth in child node bounding boxes. Without the schematic diagram, mistakes can easily occur. Similar to K-d trees, we start from the root node and recursively call the Insert function to insert all point clouds into the octree. Note that once an octree node is expanded, it must have eight child nodes, even if there are fewer than eight points. Such an octree may contain some empty leaf nodes.

K-Nearest Neighbor Search Implementation

Now let's implement the K-nearest neighbor search. The algorithm logic is very similar to the K-d tree, with only a few key differences to note:

Listing 5.34: src/ch5/octo_tree.cc

```

1 bool OctoTree::GetClosestPoint(const PointType &pt, std::vector<int> &closest_idx, int
2     k) const {
3     if (k > size_) {
4         LOG(ERROR) << "cannot set k larger than cloud size: " << k << ", " << size_;
5         return false;
6     }
7
8     std::priority_queue<NodeAndDistanceOcto> knn_result;
9     Knن(ToVec3f(pt), root_.get(), knn_result);
10
11    // Sort and return results
12    closest_idx.resize(knn_result.size());
13    for (int i = closest_idx.size() - 1; i >= 0; --i) {
14        // Insert in reverse order
15        closest_idx[i] = knn_result.top().node->point_idx_;
16        knn_result.pop();
17    }
18    return true;
19 }
20
21 void OctoTree::Knn(const Vec3f &pt, OctoTreeNode *node, std::priority_queue<
22     NodeAndDistanceOcto> &knn_result) const {
23     if (node->IsLeaf()) {
24         if (node->point_idx_ != -1) {
25             // For leaf nodes, check if the point is a nearest neighbor
26             ComputeDisForLeaf(pt, node, knn_result);
27             return;
28         }
29     }
30
31     // Determine which cell contains pt, prioritize searching that subtree
32     // Then check if other subtrees need searching
33     // If pt is outside, prioritize the nearest subtree
34     int idx_child = -1;
35     float min_dis = std::numeric_limits<float>::max();
36     for (int i = 0; i < 8; ++i) {
37         if (node->children[i]->box_.Inside(pt)) {
38             idx_child = i;
39             break;
40         } else {
41             float d = node->box_.Dis(pt);
42             if (d < min_dis) {
43                 idx_child = i;
44                 min_dis = d;
45             }
46         }
47     }
48
49     // First check idx_child
```

```

49 |     Knn(pt, node->children[idx_child], knn_result);
50 |
51 | // Then check others
52 | for (int i = 0; i < 8; ++i) {
53 |     if (i == idx_child) {
54 |         continue;
55 |     }
56 |
57 |     if (NeedExpand(pt, node->children[i], knn_result)) {
58 |         Knn(pt, node->children[i], knn_result);
59 |     }
60 | }
61 |
62 |
63 | bool OctoTree::NeedExpand(const Vec3f &pt, OctoTreeNode *node,
64 | std::priority_queue<NodeAndDistanceOcto> &knn_result) const {
65 |     if (knn_result.size() < k_) {
66 |         return true;
67 |     }
68 |
69 |     if (approximate_) {
70 |         float d = node->box_.Dis(pt);
71 |         if ((d * d) < knn_result.top().distance_ * alpha_) {
72 |             return true;
73 |         } else {
74 |             return false;
75 |         }
76 |     } else {
77 |         // Without FLANN, perform normal search
78 |         float d = node->box_.Dis(pt);
79 |         if ((d * d) < knn_result.top().distance_) {
80 |             return true;
81 |         } else {
82 |             return false;
83 |         }
84 |     }
85 | }

```

Since each node has more branches, the code involves more loop traversals compared to the binary branching of K-d trees. Additionally, note that nearest neighbor points may not necessarily lie inside octree cells - they can be outside. When query points fall outside an octree cell, we prioritize expanding subtrees closer to the query point rather than following numerical order.

Below are the performance metrics and nearest neighbor evaluation results for our octree implementation:

Listing 5.35: Terminal output:

```

1 bin/test_nn --gtest_filter=CH5_TEST.OCTREE_KNN
2 I0119 16:29:34.406015 343713 sys_utils.h:32] Method Octo Tree build average call time/
  count: 18.802/1 ms.
3 I0119 16:29:34.406155 343713 test_nn.cc:320] Octo tree leaves: 18869, points: 18869
4 I0119 16:29:34.406157 343713 test_nn.cc:323] testing knn
5 I0119 16:29:34.414115 343713 sys_utils.h:32] Method Octo Tree 5NN multithreaded
  average call time/count: 7.95114/1 ms.
6 I0119 16:29:34.414139 343713 test_nn.cc:328] comparing with bfnn
7 I0119 16:29:35.099203 343713 test_nn.cc:65] truth: 93895, esti: 93895
8 I0119 16:29:36.522886 343713 test_nn.cc:91] precision: 1, recall: 1, fp: 0, fn: 0
9 I0119 16:29:36.522902 343713 test_nn.cc:334] done.

```

Without using approximate nearest neighbor (ANN), the octree can find exact K-nearest neighbors. We've also implemented ANN parameters that readers can experiment with. Overall, due to the increased number of child nodes and more complex splitting plane calculations compared to K-d trees, the octree's construction time and KNN search time are slower than our custom K-d tree implementation (though still faster than PCL's K-d tree). When enabling ANN, the octree's performance improves, but at the cost of not achieving 100

5.2.8 Other Tree-Based Methods

In practice, indexing spatial data to achieve fast neighborhood queries is an ancient and widespread problem. Beyond SLAM, we can find its applications in many other fields. Both closely related and distant domains—including **pattern recognition**, **classification**, **computer vision**, **coding theory**, **recommendation systems**, **speech recognition**, **chemistry**, **biology**, and others—all involve nearest neighbor problems [100].

In SLAM, we focus on **low-dimensional** data structures that can be **quickly constructed and queried**, thus favoring simpler models. We also expect these structures to adapt rapidly to changes, such as adding new point clouds to the map, meaning K-d trees or octrees must support dynamic updates. In other applications, however, people deal with high-dimensional structured data (e.g., user demographics), tolerate longer construction times, but demand faster query speeds, giving rise to various **spatial data indexing** techniques. Many databases already support spatial data indexing. Broadly speaking, spatial indexing methods can be categorized as follows:

1. **Tree- and forest-based spatial partitioning methods:** - Ball trees [101–103] - R-trees [104] - R*-trees [105] - Randomized K-d trees [95] - AABB trees [106], etc. While tree variants are extremely diverse, most share similar principles, differing mainly in spatial partitioning strategies⁵. For example, ball trees partition data using hyperspheres to ensure disjoint subtrees, while R-trees use minimum bounding boxes (MBBs) to organize data, and AABB trees follow a similar approach but are more commonly used in collision detection.
2. **Space-filling curves:** - Hilbert curves [107, 108] - Z-order curves [109], etc. These methods employ fractal curves to map high-dimensional spaces to one dimension while preserving locality, enabling efficient high-dimensional searches in lower dimensions.
3. **Locality-Sensitive Hashing (LSH)** [85, 110]: LSH operates inversely to space-filling curves, hashing high-dimensional data into a lower-dimensional space while probabilistically preserving neighborhood relationships. In fact, our earlier grid-based methods already used such hashing to store grids—did you notice?

Most spatial indexing methods mentioned here are better suited for **static, known datasets**. For instance, R-trees or R*-trees are ideal for querying elements within a specific bounding box on a map. However, SLAM is inherently **dynamic**, requiring frequent creation and adjustment of point cloud maps, where constructing and destroying complex data structures can be time-consuming. Thus, SLAM tends to favor simpler, easily maintainable methods and often opts for approximate nearest neighbors over strict k -NN. Grids, voxels, and K-d trees remain the most preferred choices in SLAM.

We will not conduct comprehensive comparative experiments on these spatial indexing methods here. Readers may refer to survey literature [111, 112] for performance comparisons and analyses of common spatial indexing algorithms.

5.2.9 Summary

This section introduced several common K-nearest neighbor (KNN) solutions in SLAM: brute-force search, grid-based methods, K-d trees, and octrees. As a summary, we present

⁵Back in the 1990s, proposing one's own improved algorithm was quite straightforward.

the performance comparison of these methods in Figure 5-14 for reference. Typically, multi-threaded implementations significantly outperform single-threaded ones. Readers may verify whether their experimental results align with ours.

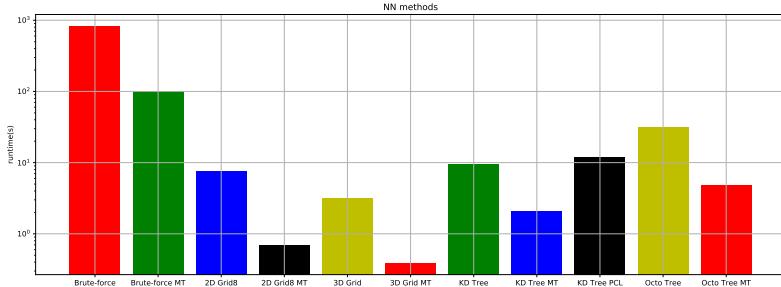


Figure 5-14: Comparison of methods discussed in this section. From left to right: brute-force matching, multithreaded brute-force, 8-neighbor 2D grid, multithreaded 8-neighbor 2D grid, 3D voxel, multithreaded 3D voxel, KD-tree, multithreaded KD-tree, PCL’s KD-tree, octree, and multithreaded octree. Note that due to the significantly longer computation time of brute-force matching, a logarithmic axis is used.

In terms of computation time, for our given dataset, 3D/2D grid-based methods performed the best, followed by K-d trees, while brute-force search was clearly the slowest. However, if the point cloud contains more points, the number of points within each grid cell will also increase, leading to linear growth in computational cost with grid cell density. Thus, grid methods are unsuitable for dense point clouds. In contrast, K-d trees exhibit logarithmic complexity growth. It can be inferred that beyond a certain scale, tree-based methods like K-d trees and octrees will become more advantageous. These tree-based approaches can also balance computation time and accuracy through approximate nearest neighbor (ANN) techniques.

5.3 Fitting Problems

Next, we introduce another important topic in point cloud processing algorithms: the extraction and estimation of basic geometric elements. Sometimes such problems are categorized as **detection** or **clustering** problems, leaning more towards perception methods. For example, many applications in autonomous driving focus heavily on extracting semantic elements like vehicles and pedestrians from point clouds. These elements can serve as data sources for subsequent decision-making and planning [114]. In SLAM, however, we are more concerned with how to use these elements to assist in **registration** between point clouds. Therefore, in traditional SLAM applications, the elements we focus on are typically basic and static, rather than dynamic or semantic. Registering a point to a plane is relatively straightforward, but registering one vehicle to another requires significantly more work⁶.

In registration problems, a common approach is to first use a nearest-neighbor structure to find several nearest neighbors for a point, then fit these neighbors to a fixed geometric shape. Finally, the vehicle’s pose is adjusted so that the scanned LiDAR points align with these shapes. The previous section covered various solutions to the nearest-neighbor problem, while this section focuses on extracting linear features like lines and planes from point

⁶Even though the algorithm for registering two vehicles likely still relies on basic point-to-point methods.

clouds. We will see that these can be elegantly unified under the same framework (linear least squares) and solved using linear algebra.

5.3.1 Plane Fitting

As with many other problems, **linear** problems are often the simplest cases. Linear fitting of point clouds is the most basic component. The linear fitting problem for point clouds can be approached from several perspectives, and comparing these viewpoints provides different insights. The same problem may be referred to by different names across fields, yet their solutions are deeply interconnected. A linear fitting problem might be called **linear regression** (regressing the parameters of a line) or **principal component analysis (PCA)** (analyzing the primary axes of a point cloud's distribution).

Let's first consider plane fitting. Given a point cloud $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ composed of n points, where each point has 3D Euclidean coordinates $\mathbf{x}_k \in \mathbb{R}^3$, we seek plane parameters \mathbf{n}, d such that:

$$\forall k \in [1, n], \mathbf{n}^\top \mathbf{x}_k + d = 0, \quad (5.9)$$

where $\mathbf{n} \in \mathbb{R}^3$ is the normal vector and $d \in \mathbb{R}$ is the intercept.

Clearly, this problem has four unknowns, while each point provides one equation. With multiple points, noise typically makes the system overdetermined and unsolvable. Thus, we often seek a least squares solution (Linear Least Square) to minimize the error:

$$\min_{\mathbf{n}, d} \sum_{k=1}^n \|\mathbf{n}^\top \mathbf{x}_k + d\|_2^2. \quad (5.10)$$

Using homogeneous coordinates can further simplify the problem. The homogeneous coordinates of a 3D point are four-dimensional, though in practice we simply append a 1:

$$\tilde{\mathbf{x}} = [\mathbf{x}^\top, 1]^\top \in \mathbb{R}^4. \quad (5.11)$$

Thus, $\tilde{\mathbf{n}} = [\mathbf{n}^\top, d]^\top \in \mathbb{R}^4$ is also a homogeneous vector, and the problem can be rewritten as:

$$\min_{\tilde{\mathbf{n}}} \sum_{k=1}^n \|\tilde{\mathbf{x}}_k^\top \tilde{\mathbf{n}}\|_2^2. \quad (5.12)$$

The subscript 2 denotes the L2-norm (standard Euclidean norm), and the superscript 2 indicates the squared sum of norms. This is a **summation-form** linear least squares problem, which can also be expressed in matrix form. Stacking all points into a matrix:

$$\tilde{\mathbf{X}} = [\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_n], \quad (5.13)$$

the summation can be omitted:

$$\min_{\tilde{\mathbf{n}}} \|\tilde{\mathbf{X}}^\top \tilde{\mathbf{n}}\|_2^2. \quad (5.14)$$

This is essentially solving a linear algebra problem: given any matrix \mathbf{A} (not necessarily square), we seek a non-zero vector \mathbf{x} that minimizes \mathbf{Ax} . Of course, if $\mathbf{x} = 0$, the product is trivially zero, but we want non-trivial solutions, so we impose $\mathbf{x} \neq 0$. Moreover, scaling \mathbf{x} by a non-zero constant k scales \mathbf{Ax} by k (and its squared norm by k^2). Thus, we disregard the magnitude of \mathbf{x} and focus on its direction by enforcing $\|\mathbf{x}\| = 1$.

For \mathbf{A} , we impose no constraints. In the point cloud plane extraction problem, \mathbf{A} is an $\mathbb{R}^{n \times 4}$ matrix, and \mathbf{x} is a unit vector in \mathbb{R}^4 . We can then ask: for which \mathbf{x} does \mathbf{Ax} attain its maximum or minimum?

Next, we will first discuss how to solve such problems in general linear algebra before returning to the plane fitting problem. Moving from the abstract to the concrete is always easier.

Various Solutions to Linear Least Squares

Eigenvalue Solution Algebraically, linear least squares refers to finding $\mathbf{x}^* \in \mathbb{R}^n$ for a given matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ such that:

$$\mathbf{x}^* = \arg \min_{\mathbf{x}} \|\mathbf{Ax}\|_2^2 = \arg \min_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}^\top \mathbf{A} \mathbf{x}, \quad \text{s.t., } \|\mathbf{x}\| = 1. \quad (5.15)$$

We observe that $\mathbf{A}^\top \mathbf{A}$ is a real symmetric matrix. According to linear algebra theory, real symmetric matrices can always be diagonalized via eigenvalue decomposition:

$$\mathbf{A}^\top \mathbf{A} = \mathbf{V} \Lambda \mathbf{V}^{-1}, \quad (5.16)$$

where Λ is a diagonal matrix of eigenvalues, which we assume are arranged in descending order as $\lambda_1, \dots, \lambda_n$. \mathbf{V} is an orthogonal matrix whose column vectors $\mathbf{v}_1, \dots, \mathbf{v}_n$ are the corresponding eigenvectors, forming an orthonormal basis. Any vector \mathbf{x} can be expressed as a linear combination of this basis:

$$\mathbf{x} = \alpha_1 \mathbf{v}_1 + \dots + \alpha_n \mathbf{v}_n. \quad (5.17)$$

It follows that⁷:

$$\mathbf{V}^{-1} \mathbf{x} = \mathbf{V}^\top \mathbf{x} = [\alpha_1, \dots, \alpha_n]^\top. \quad (5.18)$$

Thus, the objective function becomes:

$$\|\mathbf{Ax}\|_2^2 = \sum_{k=1}^n \lambda_k \alpha_k^2, \quad (5.19)$$

with the constraint $\|\mathbf{x}\| = 1$ implying $\alpha_1^2 + \dots + \alpha_n^2 = 1$. Since the eigenvalues λ_k are arranged in descending order, the minimum is achieved by setting $\alpha_1 = 0, \dots, \alpha_{n-1} = 0, \alpha_n = 1$, i.e., $\mathbf{x}^* = \mathbf{v}_n$.

We conclude that the optimal solution to the linear least squares problem is the **eigenvector corresponding to the smallest eigenvalue**. Since this problem aims to solve $\mathbf{Ax} = \mathbf{0}$, it can also be referred to as the **null space solution**. Note that the eigenvalue decomposition is performed on $\mathbf{A}^\top \mathbf{A}$ rather than directly on \mathbf{A} (as \mathbf{A} may not be diagonalizable, whereas $\mathbf{A}^\top \mathbf{A}$, being real symmetric, is guaranteed to be diagonalizable).

Singular Value Solution The aforementioned problem can also be approached from another perspective using **Singular Value Decomposition (SVD)**. Since any matrix can be decomposed via SVD, we perform SVD on \mathbf{A} to obtain:

$$\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^\top, \quad (5.20)$$

where \mathbf{U} and \mathbf{V} are orthogonal matrices, and Σ is a diagonal matrix called the **singular value matrix**, with its diagonal elements being the singular values of \mathbf{A} , typically arranged in descending order.

Substituting the SVD result into the linear least squares problem, since \mathbf{U} is orthogonal, it cancels out when computing the L2-norm. We observe that this approach is essentially equivalent to the eigenvalue method:

$$\mathbf{x}^\top \mathbf{A}^\top \mathbf{A} \mathbf{x} = \mathbf{x}^\top \mathbf{V} \Sigma^2 \mathbf{V}^\top \mathbf{x}. \quad (5.21)$$

⁷Alternatively, from the eigenvector perspective: $\mathbf{A}^\top \mathbf{A} \mathbf{x} = \sum_{k=1}^n \alpha_k \lambda_k \mathbf{v}_k$, yielding the same result.

Thus, similar to the eigenvalue solution, we take \mathbf{x} as the last column of \mathbf{V} . In fact, the relationship between the singular values of \mathbf{A} and the eigenvalues of $\mathbf{A}^\top \mathbf{A}$ is well-documented in many matrix theory textbooks [115]. Here, we take this opportunity to reintroduce this concept through a practical problem.

Furthermore, fitting an $(N - 1)$ -dimensional hyperplane to points in N -dimensional space can also be viewed as solving the null space problem in least squares. In summary, to fit a plane to a set of points, we simply:

1. Arrange all point coordinates into matrix \mathbf{A} ,
2. Compute the right singular vector corresponding to the smallest singular value of \mathbf{A} , or compute the eigenvector corresponding to the smallest eigenvalue of $\mathbf{A}^\top \mathbf{A}$.

Either approach yields the desired solution.

5.3.2 Plane Fitting Implementation

Let's now implement plane fitting starting from a point cloud. Most linear algebra operations can be handled using Eigen. We implement the plane fitting function in the common math library:

Listing 5.36: src/common/math_utils.h

```

1 template <typename S>
2 bool FitPlane(std::vector<Eigen::Matrix<S, 3, 1>>& data, Eigen::Matrix<S, 4, 1>&
3     plane_coeffs, double eps = 1e-2) {
4     if (data.size() < 3) {
5         return false;
6     }
7
8     Eigen::MatrixXd A(data.size(), 4);
9     for (int i = 0; i < data.size(); ++i) {
10         A.row(i).head<3>() = data[i].transpose();
11         A.row(i)[3] = 1.0;
12     }
13
14     Eigen::JacobiSVD svd(A, Eigen::ComputeThinV);
15     plane_coeffs = svd.matrixV().col(3);
16
17     // check error eps
18     for (int i = 0; i < data.size(); ++i) {
19         double err = plane_coeffs.template head<3>().dot(data[i]) + plane_coeffs[3];
20         if (err * err > eps) {
21             return false;
22         }
23     }
24
25     return true;
26 }
```

We use fast SVD decomposition, computing only the last column of the SVD result for matrix \mathbf{A} . After computation, we also verify the squared error against points doesn't exceed a preset threshold. Now let's write a test program that:

1. Takes randomly generated plane parameters as ground truth
2. Samples points on the plane
3. Adds noise
4. Performs plane fitting:

Listing 5.37: src/ch5/linear_fitting.cc

```

1 void PlaneFittingTest() {
2     Vec4d true_plane_coeffs(0.1, 0.2, 0.3, 0.4);
3     true_plane_coeffs.normalize();
4
5     std::vector<Vec3d> points;
```

```

6 // Generate simulated plane points randomly
7 cv::RNG rng;
8 for (int i = 0; i < FLAGS_num_tested_points_plane; ++i) {
9     // Generate random point, compute 4th dimension, add noise, then normalize
10    Vec3d p(rng.uniform(0.0, 1.0), rng.uniform(0.0, 1.0), rng.uniform(0.0, 1.0));
11    double n4 = -p.dot(true_plane_coeffs.head<3>()) / true_plane_coeffs[3];
12    p = p / (n4 + 1e-18); // Prevent division by zero
13    p += Vec3d(rng.gaussian(FLAGS_noise_sigma), rng.gaussian(FLAGS_noise_sigma), rng.
14               gaussian(FLAGS_noise_sigma));
15    points.emplace_back(p);
16
17    // Verify point-to-plane error
18    LOG(INFO) << "res of p: " << p.dot(true_plane_coeffs.head<3>()) +
19                  true_plane_coeffs[3];
20 }
21
22 Vec4d estimated_plane_coeffs;
23 if (sad::math::FitPlane(points, estimated_plane_coeffs)) {
24     LOG(INFO) << "estimated coeffs: " << estimated_plane_coeffs.transpose()
25     << ", true: " << true_plane_coeffs.transpose();
26 } else {
27     LOG(INFO) << "plane fitting failed";
28 }
29 }
```

Now compile and run the test program to compare estimated plane parameters with ground truth:

Listing 5.38: Terminal output:

```

1 ./bin/linear_fitting
2 I0121 11:04:49.878834 208913 linear_fitting.cc:21] testing plane fitting
3 I0121 11:04:49.879319 208913 linear_fitting.cc:46] res of p: -0.00149684
4 I0121 11:04:49.879382 208913 linear_fitting.cc:46] res of p: -0.00221244
5 ...
6 I0121 11:04:49.879462 208913 linear_fitting.cc:51] estimated coeffs: 0.186755 0.363656
   0.546692 0.730757, true: 0.182574 0.365148 0.547723 0.730297
```

We can observe the difference between ground truth and estimated parameters is around 3 decimal places. Additionally, linear methods don't depend on initial values and work well even when plane point coordinates are far from the origin.

5.3.3 Line Fitting

Let us now consider a problem very similar to plane fitting: line fitting. We still assume the point set \mathbf{X} consists of n 3D points. However, we can describe a straight line in several different ways, such as treating the line as the intersection of two planes, or using a point on the line plus a direction vector to define the line. The latter approach is more intuitive.

Let a point \mathbf{x} on the line satisfy the equation:

$$\mathbf{x} = \mathbf{dt} + \mathbf{p}, \quad (5.22)$$

where $\mathbf{d}, \mathbf{p} \in \mathbb{R}^3$, $t \in \mathbb{R}$. Here, \mathbf{d} is the direction vector of the line, satisfying $\|\mathbf{d}\| = 1$; \mathbf{p} is a point on the line l , and t is the line parameter. We aim to solve for \mathbf{d} and \mathbf{p} , totaling 6 unknowns. Clearly, when the given point set is large, this remains an overdetermined system, requiring the construction of a least squares problem for solution.

For any point \mathbf{x}_k not on l , we can use the Pythagorean theorem to compute the squared perpendicular distance from the point to the line:

$$f_k^2 = \|\mathbf{x}_k - \mathbf{p}\|^2 - \|(\mathbf{x}_k - \mathbf{p})^\top \mathbf{d}\|^2, \quad (5.23)$$

and then formulate the least squares problem to solve for \mathbf{d} and \mathbf{p} :

$$(\mathbf{d}, \mathbf{p})^* = \arg \min_{\mathbf{d}, \mathbf{p}} \sum_{k=1}^n f_k^2, \quad \text{s.t. } \|\mathbf{d}\| = 1. \quad (5.24)$$

Since the error term for each point is already squared, we simply need to sum them.

Next, we separate the \mathbf{d} and \mathbf{p} components. First, consider $\frac{\partial f_k^2}{\partial \mathbf{p}}$:

$$\frac{\partial f_k^2}{\partial \mathbf{p}} = -2(\mathbf{x}_k - \mathbf{p}) + 2 \underbrace{(\mathbf{x}_k - \mathbf{p})^\top \mathbf{d}}_{\text{scalar, } = \mathbf{d}^\top (\mathbf{x}_k - \mathbf{p})} \mathbf{d}, \quad (5.25)$$

$$= (-2)(\mathbf{I} - \mathbf{d}\mathbf{d}^\top)(\mathbf{x}_k - \mathbf{p}). \quad (5.26)$$

Thus, the derivative of the overall objective function with respect to \mathbf{p} is:

$$\frac{\partial \sum_{k=1}^n f_k^2}{\partial \mathbf{p}} = \sum_{k=1}^n (-2)(\mathbf{I} - \mathbf{d}\mathbf{d}^\top)(\mathbf{x}_k - \mathbf{p}), \quad (5.27)$$

$$= (-2)(\mathbf{I} - \mathbf{d}\mathbf{d}^\top) \sum_{k=1}^n (\mathbf{x}_k - \mathbf{p}). \quad (5.28)$$

To find the extremum of the least squares problem, set this equal to zero, yielding:

$$\mathbf{p} = \frac{1}{n} \sum_{k=1}^n \mathbf{x}_k, \quad (5.29)$$

indicating that \mathbf{p} should be the centroid of the point cloud. Thus, we can first determine \mathbf{p} and then consider \mathbf{d} . With \mathbf{p} solved, let $\mathbf{y}_k = \mathbf{x}_k - \mathbf{p}$, treating \mathbf{y}_k as known, and simplify the error term:

$$f_k^2 = \mathbf{y}_k^\top \mathbf{y}_k - \mathbf{d}^\top \mathbf{y}_k \mathbf{y}_k^\top \mathbf{d}. \quad (5.30)$$

Clearly, the first error term does not contain \mathbf{d} and is unaffected by the choice of \mathbf{d} , so it can be omitted. Minimizing the second term is equivalent to maximizing its negation:

$$\mathbf{d}^* = \arg \max_{\mathbf{d}} \sum_{k=1}^n \mathbf{d}^\top \mathbf{y}_k \mathbf{y}_k^\top \mathbf{d} = \sum_{k=1}^n \|\mathbf{y}_k^\top \mathbf{d}\|_2^2. \quad (5.31)$$

If we define:

$$\mathbf{A} = \begin{bmatrix} \mathbf{y}_1^\top \\ \vdots \\ \mathbf{y}_n^\top \end{bmatrix}, \quad (5.32)$$

then the problem becomes:

$$\mathbf{d}^* = \arg \max_{\mathbf{d}} \|\mathbf{A}\mathbf{d}\|_2^2. \quad (5.33)$$

This problem is still very similar to (5.12), except that we seek to **maximize** rather than **minimize**. For plane fitting, we minimized this problem; for line fitting, we maximize it. Following the previous discussion, taking \mathbf{d} as the eigenvector corresponding to the smallest eigenvalue or singular value yields the minimizing solution; conversely, taking the eigenvector corresponding to the largest eigenvalue yields the maximizing solution. Thus, the solution to this problem should take \mathbf{d} as the right singular vector corresponding to the largest singular value of \mathbf{A} , or the eigenvector corresponding to the largest eigenvalue of $\mathbf{A}^\top \mathbf{A}$.

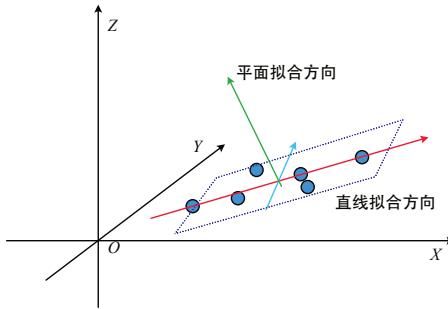


Figure 5-15: Schematic diagram of line fitting and plane fitting for point clouds (TODO: replace)

5.3.4 Implementation of Line Fitting

Now let's implement the line fitting method described earlier. Similarly, we first implement the fitting function in the math library, then compare the fitted results with ground truth in a test program.

Listing 5.39: src/common/math_utils.h

```

1 template <typename S>
2 bool FitLine(std::vector<Eigen::Matrix<S, 3, 1>>& data, Eigen::Matrix<S, 3, 1>& origin
3   , Eigen::Matrix<S, 3, 1>& dir,
4   double eps = 0.2) {
5   if (data.size() < 2) {
6     return false;
7   }
8
9   origin = std::accumulate(data.begin(), data.end(), Eigen::Matrix<S, 3, 1>::Zero().
10   eval() / data.size());
11
12   Eigen::MatrixXd Y(data.size(), 3);
13   for (int i = 0; i < data.size(); ++i) {
14     Y.row(i) = (data[i] - origin).transpose();
15   }
16
17   Eigen::JacobiSVD svd(Y, Eigen::ComputeFullV);
18   dir = svd.matrixV().col(0);
19
20   // check eps
21   for (const auto& d : data) {
22     if (dir.template cross(d - origin).template squaredNorm() > eps) {
23       return false;
24     }
25   }
26
27   return true;
28 }
```

Note that we need to compute the full \mathbf{V} matrix of the SVD since we want the largest singular value. The rest of the computation remains consistent with the mathematical formulation. Below is the test program:

Listing 5.40: src/ch5/linear_fitting.cc

```

1 void LineFittingTest() {
2   // Ground truth line parameters
3   Vec3d true_line_origin(0.1, 0.2, 0.3);
4   Vec3d true_line_dir(0.4, 0.5, 0.6);
5   true_line_dir.normalize();
6
7   // Generate random points along the line using parametric equation
```

```

8   std::vector<Vec3d> points;
9   cv::RNG rng;
10  for (int i = 0; i < fLI::FLAGS_num_tested_points_line; ++i) {
11    double t = rng.uniform(-1.0, 1.0);
12    Vec3d p = true_line_origin + true_line_dir * t;
13    p += Vec3d(rng.gaussian(FLAGS_noise_sigma), rng.gaussian(FLAGS_noise_sigma), rng.
14      gaussian(FLAGS_noise_sigma));
15    points.emplace_back(p);
16  }
17
18  Vec3d esti_origin, esti_dir;
19  if (csad::math::FitLine(points, esti_origin, esti_dir)) {
20    LOG(INFO) << "estimated origin: " << esti_origin.transpose() << ", true: " <<
21    true_line_origin.transpose();
22    LOG(INFO) << "estimated dir: " << esti_dir.transpose() << ", true: " <<
23    true_line_dir.transpose();
24  } else {
25    LOG(INFO) << "line fitting failed";
26  }
27

```

Similarly, we first set the ground truth line parameters, add noise to sampled points along the line, then estimate the line parameters from these samples. The test results are:

Listing 5.41: Terminal output:

```

1 ./bin/linear_fitting
2 I0121 12:14:10.936178 212707 linear_fitting.cc:24] testing line fitting
3 I0121 12:14:10.936190 212707 linear_fitting.cc:77] estimated origin: 0.102906
4   0.204955 0.305633, true: 0.1 0.2 0.3
I0121 12:14:10.936200 212707 linear_fitting.cc:78] estimated dir:  0.45294 0.569855
  0.685646, true: 0.455842 0.569803 0.683763

```

At this point, we have demonstrated how to fit lines and planes to 3D point clouds. Interestingly, these two problems share remarkable similarities and can even be reduced to **finding the maximum and minimum solutions of the same problem**, as illustrated in Figure 5-15. Line fitting seeks the direction of maximum variance, while plane fitting seeks the direction of minimum variance; line fitting corresponds to the eigenvector of the largest eigenvalue, while plane fitting corresponds to the smallest eigenvalue. This aligns perfectly with our intuition.

Extending to higher or lower dimensions, the plane fitting discussed here becomes fitting an $N - 1$ dimensional hyperplane to points in N -dimensional space, while line fitting becomes fitting a 1-dimensional subspace. When points lie in 2D space ($N = 2$), plane fitting reduces to line fitting, making the two problems identical. In higher dimensions, we can pose more complex questions - for example, what form should we fit for 5D point clouds in 4D or 3D space? While these questions may sound abstract, they have practical applications in numerous cases. Of course, people typically refer to them as **data** rather than point clouds. These $N - 2$, $N - 3$ dimensional fittings are also called **data dimensionality reduction**, implemented through SVD decomposition by selecting different columns of the **V** matrix or eliminating components with near-zero singular values while retaining those with larger singular values. Like squeezing water from a sponge, directions with small singular values represent **noise** while those with large singular values contain the **essential information**. These methods find applications in data compression, feature extraction, and more, with different names across domains. In Principal Component Analysis (PCA) [117], they're called **maximum principal components** or **minimum principal components**. In low-rank approximation problems [118], line fitting can be viewed as a rank-1 approximation. Alternatively, in subspace analysis [119], line fitting constructs the range space (or row/column space) while plane fitting constructs the null space. Linear problems often exhibit intricate connections, with conclusions that are universally applicable. All these problems can be

modeled as linear least squares problems, with SVD or eigenvalue decomposition remaining the core solution approaches.

5.4 Summary

This chapter introduced fundamental point cloud representations and basic point cloud-related problems, such as nearest neighbor search and fitting solutions. Starting from first principles, we implemented brute-force nearest neighbor search, K-d trees, octrees, and other nearest neighbor methods, and conducted comparative experiments with PCL versions in terms of efficiency and accuracy. Since we only need to consider LiDAR point clouds without accommodating various other point cloud templates, our implementations of K-d trees and other data structures are more concise than PCL’s versions. These data structures are highly useful and will be employed in subsequent chapters to implement point cloud registration, odometry, and other algorithmic modules.

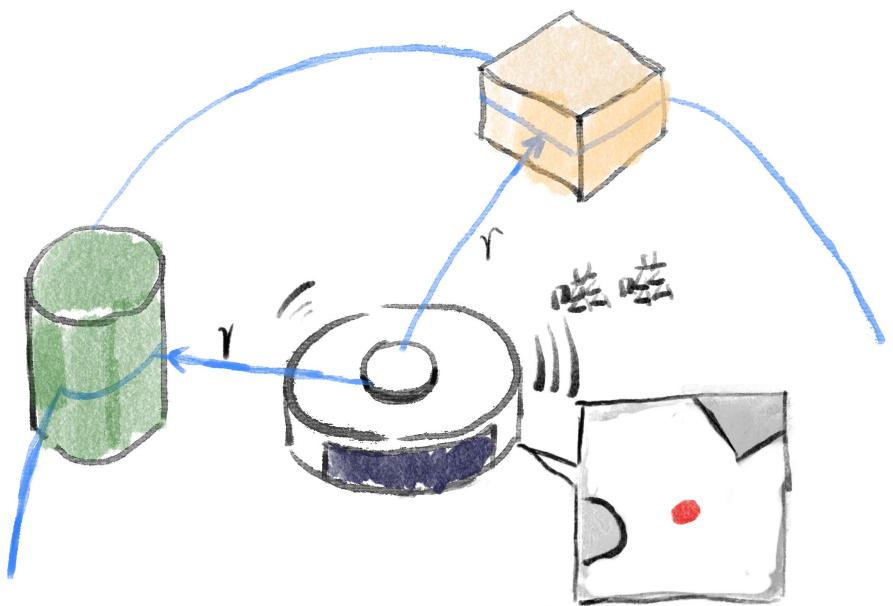
Exercises

1. Define NEARBY14 in 3D voxels and implement 14-cell nearest neighbor search.
2. Extend the K-d tree point cloud type to a template class.
3. Incorporate bounding boxes into the K-d tree node structure to achieve more precise pruning.
4. Attempt to improve the lower bound accuracy of the distance between query points and bounding boxes in octrees, and observe whether nearest neighbor performance improves.
5. Derive that the solution to Equation (5.33) is the eigenvector corresponding to the largest eigenvalue of $\mathbf{A}^\top \mathbf{A}$.
6. Compare the nearest neighbor algorithms in this chapter with common approximate nearest neighbor algorithms such as nanoflann [120], Faiss [121], and nmslib [122], and evaluate their performance in point cloud nearest neighbor search.

Chapter 6

2D Laser Localization and Mapping

In the previous chapter, we introduced the basic principles of laser measurement, as well as the nearest neighbor and fitting methods for point clouds. These serve as the foundation for most laser point cloud registration methods. However, people often treat 2D and 3D laser processing scenarios differently. Overall, 2D laser registration is easier and more suitable for introducing image-like processing methods. This chapter will introduce the relatively simpler 2D laser SLAM, while the next chapter will cover 3D laser SLAM systems. Readers can compare the differences between the two systems from theoretical and methodological perspectives.



2D 场景可以简化很多问题。

6.1 Basic Principles of 2D Laser SLAM



Figure 6-1: Robots using 2D laser SLAM and their corresponding lidars. Cleaning robots typically mount lidars on top, while service robots have built-in lidars after slotting at the base.

All real-world sensors naturally operate in three-dimensional space, inherently without any distinction of dimensionality. However, most wheeled robots move only on a fixed plane, unlike aircraft that freely change their posture. Cleaning robots operate on horizontal ground, while wall-climbing robots work on vertical planes. Some robots, such as hotel food delivery robots, may have a certain height in their main body, but the part primarily responsible for movement—and thus the main focus of SLAM algorithms—is two-dimensional, as shown in Figure 6-1.

Compared to point clouds in three-dimensional space, 2D SLAM can be viewed as a laser SLAM algorithm operating from a top-down perspective. From this viewpoint, laser scan data and map data can be simplified into two-dimensional forms. They closely resemble images, and the map itself can even be stored as an image. Some image feature extraction and matching algorithms can also be applied to 2D SLAM. 2D SLAM is crucial for applications like cleaning robots and AGVs (Automated Guided Vehicles) and was once the dominant focus of SLAM technology [7] (it remains the most widely deployed field today). Historically, it has given rise to many well-known methods, such as FastSLAM [123], GMapping [124], and others.

However, due to the assumption of planar motion, when the robot body or the environment contains significant three-dimensional objects, some fundamentally unsolvable problems arise at the system level. For example, most 2D SLAM solutions assume obstacles are at the same height as the laser sensor. If the environment contains obstacles at other heights or objects whose shapes vary noticeably with height (e.g., a tabletop and its legs are clearly different), 2D maps struggle to represent such objects, and the robot may collide with them. Another example is when the robot moves on an inclined slope, where the scanned object distances differ geometrically from the actual distances. These scenarios violate the 2D motion assumption and are inherent limitations of the system, making them difficult to resolve within the framework of 2D SLAM. The 3D point cloud SLAM introduced in the next chapter can effectively address these shortcomings caused by 2D assumptions.

On the other hand, early 2D SLAM systems often treated the map as a single 2D image, an approach that was simplistic and not well-suited for handling loop closures. This chapter will introduce 2D SLAM from a more modern perspective, adopting a framework similar

to that of 3D laser SLAM. The content arrangement will also emphasize the similarities between the two.

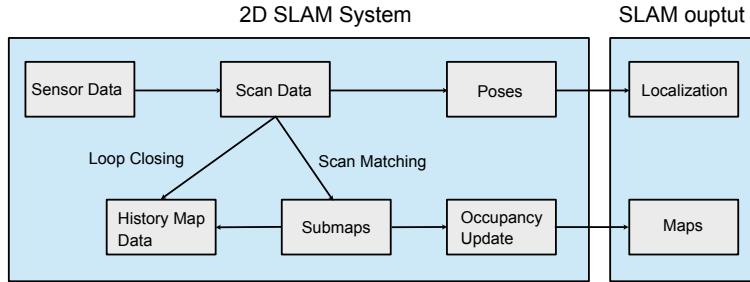


Figure 6-2: The basic pipeline of submap-based 2D SLAM.

Figure 6-2 illustrates a typical 2D SLAM framework. Here is a brief overview of its workflow:

1. First, the 2D laser sensor outputs range measurements at a fixed frequency. Each full cycle of data is called a **scan**¹.
2. To estimate the robot's pose for this scan, we need to **match** (or **register**) it against something. This process is called **scan matching**. We can match the scan either against the previous scan or against the map, so scan matching can be further divided into *scan-to-scan* and *scan-to-map* modes. The principles are largely the same, and they can be used flexibly in practice. In this chapter, we will implement common 2D scan matching algorithms, such as point-to-point and point-to-line methods.
3. After estimating the pose of this scan, we integrate it into the map. Of course, a scan is essentially a point cloud, so the simplest approach is to place all scans into the map in chronological order. However, this may suffer from cumulative errors or moving objects. Modern SLAM solutions often adopt a more flexible **submap** approach, grouping nearby laser scans into a submap and then stitching the submaps together [3]. In the submap model, each submap is internally fixed and does not require repeated computation. At the same time, submaps have their own independent coordinate systems, and the poses between them can be adjusted and optimized. Thus, when handling loop closures, submaps can be treated as basic units. Early SLAM solutions often relied on a single global map [124]. Submaps represent an intermediate management approach between single frames and a full map, making loop closure detection and map updates more convenient. This chapter will also adopt the submap model for map construction.
4. Finally, how should the scanned map be stored and updated? Many robot maps need to distinguish between **obstacles** and **navigable areas**. To represent these concepts, we will use an **occupancy grid map** for map management [125, 126]. Occupancy grid maps can effectively filter out the impact of moving objects, resulting in cleaner maps.

In this chapter, we will work with readers to implement the mainstream 2D SLAM algorithms discussed above. We will implement several key scan matching algorithms, build

¹Hereafter, the term "scan" will refer to the laser scan data within one cycle. Since terms like "scan-to-scan" are already widely used in the industry, we will not deliberately translate "scan."

them into local submaps, and then use loop closure corrections to construct a complete occupancy grid map. Among the algorithms mentioned here, scan matching is the core of many subsequent processes. We can use traditional methods like the Iterative Closest Point (ICP) algorithm for scan matching or leverage the characteristics of 2D to implement image-based methods such as Gaussian likelihood fields.

6.2 Scan Matching Algorithms

6.2.1 Point-to-Point Scan Matching

Let us begin by introducing the scan matching methods in 2D SLAM. A single 2D scan is represented by a set of angle-distance pairs, denoted as $(\rho, r)_i$, where ρ is the angle relative to the robot's own frame, r is the measured distance, and $i = 0, \dots, N$ indicates multiple measurement points. The value of N depends on the angular resolution of the laser sensor. In implementation, these data points are often stored in an array. These measurements are in polar coordinates and can be naturally converted to Cartesian coordinates, expressed as $(x, y)_i$.

Visualizing 2D Lidar Data Using OpenCV

Starting from this chapter, we will use real-world data collected from actual robots to verify whether our algorithms perform satisfactorily in real-world scenarios. This section and subsequent chapters will require some ROS bag files. Due to their large size, readers are advised to download the necessary datasets from the code repository associated with this book. If storage space is limited, you may choose to download only representative datasets for each chapter. The program in this section requires the data package located in the ‘2dmapping/’ directory, while other chapters will use datasets from their respective directories.

To facilitate testing different algorithms across various datasets, we have implemented an abstract interface for ROS bag processing. Readers only need to define callback functions for different message types. For example, in the demo code shown here, we need to read 2D scan messages from the bag file and pass them to a visualization program for rendering. In other chapters, these data may be fed into matching algorithms for registration. We leverage C++ lambda functions to achieve this flexible invocation:

Listing 6.1: src/ch6/test_2dlidar_io.cc

```

1 sad::RosbagIO rosbag_io(FLAGS_bag_path);
2 rosbag_io
3 .AddScan2DHandle("/pavo_scan_bottom",
4   [](Scan2d::Ptr scan) {
5     cv::Mat image;
6     sad::Visualize2DScan(scan, SE2(), image, Vec3b(255, 0, 0));
7     cv::imshow("scan", image);
8     cv::waitKey(20);
9     return true;
10 })
11 .Go();
12
13 void Visualize2DScan(Scan2d::Ptr scan, const SE2& pose, cv::Mat& image, const Vec3b&
14   color, int image_size, float resolution, const SE2& pose_submap) {
15   if (image.data == nullptr) {
16     image = cv::Mat(image_size, image_size, CV_8UC3, cv::Vec3b(255, 255, 255));
17   }
18   for (size_t i = 0; i < scan->ranges.size(); ++i) {
19     if (scan->ranges[i] < scan->range_min || scan->ranges[i] > scan->range_max) {
20       continue;
21     }

```

```

22
23     double real_angle = scan->angle_min + i * scan->angle_increment;
24     double x = scan->ranges[i] * std::cos(real_angle);
25     double y = scan->ranges[i] * std::sin(real_angle);
26
27     if (real_angle < scan->angle_min + 30 * M_PI / 180.0 || real_angle > scan->
28         angle_max - 30 * M_PI / 180.0) {
29         continue;
30     }
31
32     Vec2d psubmap = pose_submap.inverse() * (pose * Vec2d(x, y));
33
34     int image_x = int(psubmap[0] * resolution + image_size / 2);
35     int image_y = int(psubmap[1] * resolution + image_size / 2);
36     if (image_x >= 0 && image_x < image.cols && image_y >= 0 && image_y < image.rows)
37     {
38         image.at<cv::Vec3b>(image_y, image_x) = cv::Vec3b(color[0], color[1], color[2]);
39     }
40
41 // Draw the robot's position
42 Vec2d pose_in_image =
43 pose_submap.inverse() * (pose.translation() * double(resolution) + Vec2d(image_size
44 / 2, image_size / 2));
cv::circle(image, cv::Point2f(pose_in_image[0], pose_in_image[1]), 5, cv::Scalar(
    color[0], color[1], color[2]), 2);
}

```

As shown, this program uses the ‘sad::RosbagIO’ class to read the ‘pavo_scan_bottom’ messages from the bag file and passes them to a visualization function. The visualization function converts the lidar’s range and angle measurements into Cartesian coordinates and renders them onto an image at a specified resolution. If the robot’s pose is provided, the visualization can also display the scan data in motion. Here, we demonstrate the scan data in the robot’s body frame by setting the input pose to the origin.

Now, please compile the ‘test_2dlidar_io’ program and run the following command to view the laser scan data in the given bag file:

Listing 6.2: Terminal command

```
/bin/test_2dlidar_io --bag_path ./dataset/sad/2dmapping/floor1.bag
```

Readers should see laser scan data similar to Figure 6-3. Since the actual robot was moving, you should also observe structures from different locations in the scene. With good spatial imagination, one should be able to infer the robot’s movement direction and surrounding environment.

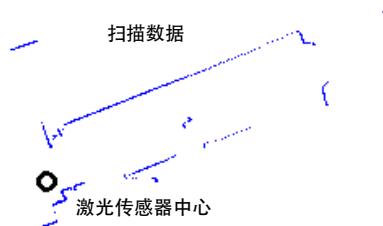


Figure 6-3: Example of single 2D scan data

In scan data like Figure 6-3, we call the actual laser hit points **end points**. End points have two physical meanings: 1. The end point itself represents an actual existing obstacle; 2. Along the line connecting the sensor to the end point, no other obstacles exist.

Note that the second meaning requires calculating the line from the sensor to the end point (the sensor doesn't measure this line - it only measures the end point, so we need to compute this line ourselves). If we want to calculate which grid cells this line passes through, it involves **ray casting algorithm** [127] and **rasterization algorithm** [128]. We'll mention these again later in grid map construction and provide a simple implementation. However, in scan matching algorithms, we focus more on the first meaning and often ignore the second.

Under this premise, a single scan can be viewed as a simple 2D point set.

Now let us derive the mathematical model for scan matching algorithms. From the perspective of state estimation, 2D laser scan data can be denoted as observation data \mathbf{z} . It is obtained when the robot at pose \mathbf{x} observes a certain map \mathbf{m} . Thus, the observation model can be simply expressed as:

$$\mathbf{z} = \mathbf{h}(\mathbf{x}, \mathbf{m}) + \mathbf{w}, \quad (6.1)$$

where \mathbf{w} is the noise term. Our goal is to estimate \mathbf{x} based on the observed \mathbf{z} and \mathbf{m} . According to Bayesian estimation theory, \mathbf{x} can be obtained through **Maximum a Posteriori** (MAP) or **Maximum Likelihood Estimation** (MLE):

$$\mathbf{x}_{\text{MLE}} = \arg \max p(\mathbf{x}|\mathbf{z}, \mathbf{m}) = \arg \max p(\mathbf{z}|\mathbf{x}, \mathbf{m}). \quad (6.2)$$

If we only consider the scan-to-scan problem, \mathbf{m} can simply be written as the previous scan data. The key then becomes how to define the detailed form of the observation equation, i.e., how to compute the residual term for each observation. Here we present several typical solutions: point-to-point scan matching (ICP) [129], point-to-line scan matching (PL-ICP [130] or ICL [131]), and the Gaussian likelihood field method (or CSM [132]). In 3D matching algorithms, we will further introduce other methods such as point-to-plane [133, 134] and NDT [74, 135, 136]. Since 2D scan matching does not involve surface elements, we will only discuss point-to-point and point-to-line algorithms here.

The specific definition of the observation equation involves several issues:

1. How to select the points to be matched. In principle, all scanned points should participate in matching, but for efficiency considerations, **sampling** can be applied. There are many sampling methods, from uniform or random sampling to normal- or feature-based sampling, all of which can be used in practice.
2. How to determine which specific map point corresponds to a scan point $(x, y)_i$. This is also known as the **data association** problem. This problem is typically solved using the nearest neighbor method introduced in the previous section, i.e., assuming that under the current estimated pose, the closest map point to the observed point is the matching point. In field-based methods, grid cells or fields in the map can also be used as matching points.
3. After determining the scan point $(x, y)_i$ and its corresponding map point \mathbf{m}_i , how to compute the residual. This involves the modeling of residuals. The complete laser scan noise model (beam model) is complex with many parameters [137], and as a state estimation model, it is not smooth enough. In practice, we usually simplify it, and in the simplest case, we can directly model it as a 2D Gaussian distribution, i.e., $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \Sigma)$.

As can be seen, a scan matching algorithm involves many choices at different stages, and there are numerous variants of basic methods in both industry and academia. We will introduce the origins of these variants starting from basic methods, but we will not attempt to cover all scan matching algorithms. To maintain consistency, we will use the same mathematical notation to describe the problems and provide code implementations for each algorithm.

First, let us look at the simplest point-to-point matching problem, also known as the Iterative Closest Point (ICP) algorithm [138]. The ICP algorithm divides the scan matching problem into two steps: **data association** and **pose estimation**, and alternates between these two steps until convergence. In fact, regardless of how data association and pose estimation are specifically solved, as long as the algorithm involves alternating between these two steps, we can refer to it as an **ICP-like** algorithm [87, 139, 140]. When the matching relationship is known, ICP can be solved in closed form, but this approach discards the possibility of further filtering outliers and makes point-to-point and point-to-plane methods appear different (note that from an optimization perspective, they are unified). For consistency, we will describe the problem in terms of residuals and optimization.

The pose of a 2D laser is described by translation and rotation angles², and can be simply written as:

$$\mathbf{x} = [x, y, \theta]^\top. \quad (6.3)$$

Here, \mathbf{x} describes a transformation from the robot's coordinate frame B to the world frame W , denoted as $\mathbf{x} = \mathbf{T}_{WB}$ according to the convention of this book. Note that submap frames and their coordinate systems will be introduced later, so it is necessary to clarify the transformation relationship of \mathbf{x} . Suppose a scan point \mathbf{p}_i^B in the robot's frame has distance and angle r_i, ρ_i . Then, based on the current laser pose, it can be transformed to the world frame:

$$\mathbf{p}_i^W = [x + r_i \cos(\rho_i + \theta), y + r_i \sin(\rho_i + \theta)]^\top. \quad (6.4)$$

In 3D space, this can be written as:

$$\mathbf{p}_i^W = \mathbf{T}_{WB}\mathbf{p}_i^B. \quad (6.5)$$

In the program, since the SE3 and SE2 interfaces are consistent, we do not deliberately distinguish between 3D and 2D poses in mathematical notation.

Assuming we find a nearest neighbor \mathbf{q}_i^W near \mathbf{p}_i^W , we can easily construct the residual between \mathbf{p}_i^W and \mathbf{q}_i^W :

$$\mathbf{e}_i = \mathbf{p}_i^W - \mathbf{q}_i^W, \quad (6.6)$$

This residual describes the Euclidean geometric distance between two points. Clearly, this error uses world coordinates and is related to the robot's pose at that time. Thus, the robot pose estimation problem can be transformed into a least-squares problem with variables x, y, θ :

$$(x, y, \theta)^* = \arg \min_{\mathbf{x}} \sum_{i=1}^n \|\mathbf{e}_i\|_2^2. \quad (6.7)$$

This least-squares problem can be solved by many existing solvers.

To solve the least-squares problem, we should provide the derivatives of \mathbf{e} with respect to each state variable. The obvious advantage of 2D poses is that we no longer need to use manifold notation and can directly use the decomposed x, y, θ ³. Based on the above

²In the program, we use the SE2 interface, which is essentially the same as the SE3 interface. We can use the same notation for SE3 and SE2, such as matrix multiplication. In some literature, 2D poses are also referred to as **three-degree-of-freedom** poses.

³Of course, it is possible to unify them using manifold notation, but it is unnecessary.

definitions, we can easily obtain:

$$\frac{\partial \mathbf{e}_i}{\partial x} = [1, 0]^\top, \quad (6.8a)$$

$$\frac{\partial \mathbf{e}_i}{\partial y} = [0, 1]^\top, \quad (6.8b)$$

$$\frac{\partial \mathbf{e}_i}{\partial \theta} = [-r_i \sin(\rho_i + \theta), r_i \cos(\rho_i + \theta)]^\top. \quad (6.8c)$$

We can organize this into matrix form:

$$\frac{\partial \mathbf{e}_i}{\partial \mathbf{x}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -r_i \sin(\rho_i + \theta) & r_i \cos(\rho_i + \theta) \end{bmatrix} \in \mathbb{R}^{3 \times 2}. \quad (6.9)$$

In the subsequent experimental section, we will use this Jacobian matrix to solve the Gauss-Newton method. It is important to note that if the state variables x, y, θ change, \mathbf{q}_i will also change, altering the entire problem. If the state variables are initially set far from the optimal solution, \mathbf{q}_i may be an incorrect point, making ICP-like methods highly dependent on the initial value of optimization. We will continue to discuss this issue later.

6.2.2 Implementation of Point-to-Point ICP (Gauss-Newton)

Below we implement a 2D point-to-point ICP method by manually coding the Gauss-Newton approach. In each Gauss-Newton iteration, we recompute the nearest neighbors between points and then solve for the pose increment. The key points here are: (1) implementing nearest neighbor search, and (2) implementing Gauss-Newton iteration.

Since the nearest neighbor data structure from the previous lecture used 3D points rather than 2D points, here we employ PCL's K-d tree for nearest neighbor search with 2D points. Thus, when setting the target point cloud, we need to build a K-d tree for it. Our 2D ICP class interface is as follows:

Listing 6.3: src/ch6/icp_2d.h

```

1  class Icp2d {
2      public:
3          using Point2d = pcl::PointXYZ;
4          using Cloud2d = pcl::PointCloud<Point2d>;
5          Icp2d() {}
6
7          /// Set the target scan
8          void SetTarget(Scan2d::Ptr target) {
9              target_scan_ = target;
10             BuildTargetKdTree();
11         }
12
13         /// Set the source scan to be aligned
14         void SetSource(Scan2d::Ptr source) { source_scan_ = source; }
15
16         /// Perform alignment using Gauss-Newton method
17         bool AlignGaussNewton(SE2& init_pose);
18
19     private:
20         // Build K-d tree for the target point cloud
21         void BuildTargetKdTree();
22
23         pcl::search::KdTree<Point2d> kdtree_;
24         Cloud2d::Ptr target_cloud_; // Target cloud in PCL format
25
26         Scan2d::Ptr target_scan_ = nullptr;
27         Scan2d::Ptr source_scan_ = nullptr;
28     };

```

The ‘AlignGaussNewton‘ function implements 2D ICP based on Gauss-Newton iteration:

Listing 6.4: src/ch6/icp_2d.cc

```

1  bool Icp2d::AlignGaussNewton(SE2& init_pose) {
2      int iterations = 10;
3      double cost = 0, lastCost = 0;
4      SE2 current_pose = init_pose;
5      const float max_dis2 = 0.01;           // Maximum squared distance for nearest
6          neighbors
7      const int min_effect_pts = 20;        // Minimum number of effective points
8
9      for (int iter = 0; iter < iterations; ++iter) {
10         Mat3d H = Mat3d::Zero();
11         Vec3d b = Vec3d::Zero();
12         cost = 0;
13
14         int effective_num = 0; // Number of effective points
15
16         // Traverse source points
17         for (size_t i = 0; i < source_scan_->ranges.size(); ++i) {
18             float r = source_scan_->ranges[i];
19             if (r < source_scan_->range_min || r > source_scan_->range_max) {
20                 continue;
21             }
22
23             float angle = source_scan_->angle_min + i * source_scan_->angle_increment;
24             float theta = current_pose.so2().log();
25             Vec2d pw = current_pose * Vec2d(r * std::cos(angle), r * std::sin(angle));
26             Point2d pt;
27             pt.x = pw.x();
28             pt.y = pw.y();
29
30             // Nearest neighbor search
31             std::vector<int> nn_idx;
32             std::vector<float> dis;
33             kdtree_.nearestKSearch(pt, 1, nn_idx, dis);
34
35             if (nn_idx.size() > 0 && dis[0] < max_dis2) {
36                 effective_num++;
37                 Mat3d J;
38                 J << 1, 0, 0, 1, -r * std::sin(angle + theta), r * std::cos(angle + theta);
39                 H += J * J.transpose();
40
41                 Vec2d e(pt.x - target_cloud_->points[nn_idx[0]].x, pt.y -
42                         target_cloud_->points[nn_idx[0]].y);
43                 b += -J * e;
44
45                 cost += e.dot(e);
46             }
47         }
48
49         if (effective_num < min_effect_pts) {
50             return false;
51         }
52
53         // Solve for dx
54         Vec3d dx = H.ldlt().solve(b);
55         if (isnan(dx[0])) {
56             break;
57         }
58
59         cost /= effective_num;
60         if (iter > 0 && cost >= lastCost) {
61             break;
62         }
63
64         LOG(INFO) << "iter " << iter << " cost = " << cost << ", effect num: " <<
65             effective_num;
66
67         current_pose.translation() += dx.head<2>();
68         current_pose.so2() = current_pose.so2() * SO2::exp(dx[2]);
69         lastCost = cost;
70
71     }
72 }
```

```

68 }
69
70     init_pose = current_pose;
71     LOG(INFO) << "estimated pose: " << current_pose.translation().transpose()
72     << ", theta: " << current_pose.so2().log();
73
74     return true;
75 }
```

The Jacobian matrix here corresponds to the theoretical part introduced earlier. We limit the maximum squared distance for nearest neighbors (set to 0.01) and count the number of valid nearest neighbors, then compute their average error. Finally, the ‘current_pose’ obtained from Gauss-Newton iteration is filled into the return result.

We also write a test program to evaluate the results of 2D ICP:

Listing 6.5: src/ch6/test_2d_icp_s2s.cc

```

1 rosbag_io.AddScan2DHandle("/pavo_scan_bottom",
2 [&](Scan2d::Ptr scan) {
3     current_scan = scan;
4
5     if (last_scan == nullptr) {
6         last_scan = current_scan;
7         return true;
8     }
9
10    sad::Icp2d icp;
11    icp.SetTarget(last_scan);
12    icp.SetSource(current_scan);
13
14    SE2 pose;
15    if (FLAGS_method == "point2point") {
16        icp.AlignGaussNewton(pose);
17    } else if (FLAGS_method == "point2plane") {
18        icp.AlignGaussNewtonPoint2Plane(pose);
19    }
20
21    cv::Mat image;
22    sad::Visualize2DScan(last_scan, SE2(), image, Vec3b(255, 0, 0)); // target in
23    blue
24    sad::Visualize2DScan(current_scan, pose, image, Vec3b(0, 0, 255)); // source in
25    red
26    cv::imshow("scan", image);
27    cv::waitKey(20);
28
29    last_scan = current_scan;
30    return true;
31 }
32 .Go();
```

This program registers the current scan data to the previous scan and visualizes the results using OpenCV. The previous frame is displayed in blue, while the current frame is shown in red. After registration, the two scans should align well with each other. Running this program allows real-time observation of the registration effect, as shown in Figure 6-4. Readers can also monitor metrics such as the objective function value and the number of valid points for each ICP iteration in the terminal. However, due to the robot’s motion, there will inevitably be some discrepancies between the two scans. Previously unexplored areas will appear in the current frame, and dynamic objects or motion distortion in the scan data itself may also interfere with the registration results. We can adjust the thresholds in ICP or parameters in the optimization model to mitigate the impact of dynamic objects to some extent.

This test program is also compatible with the point-to-plane ICP interface discussed later. Readers can use different gflags to test it.



Figure 6-4: Registration results of point-to-point ICP

6.2.3 Point-to-Line Scan Matching

In addition to point-to-point methods, ICP can also utilize other error formulations. The most common alternatives are point-to-line or point-to-plane approaches. Since 2D lidar data doesn't contain planes, we focus on the point-to-line formulation (which can be considered a lower-dimensional version of point-to-plane). The overall workflow of point-to-line ICP remains similar to point-to-point ICP, except that during nearest neighbor search, we need to find multiple neighbors (e.g., k neighbors), fit a line to these points, and then compute the perpendicular distance from the target point to this line. This method is called Point-to-line ICP (PL-ICP) [130].

Let these k nearest neighbors be $(\mathbf{x}_1, \dots, \mathbf{x}_k)$, $\forall i \in 1, \dots, k$, $\mathbf{x}_i \in \mathbb{R}^2$. In 3D space, the fitted line can be described by a direction vector \mathbf{d} and origin \mathbf{p} , with fitting methods already introduced in Section 5.3.3. While lines in 3D space are more complex, in 2D space they can be simplified to a slope-intercept model. Let the line equation be:

$$ax + by + c = 0, \quad (6.10)$$

where a, b, c are line parameters. The line fitting can then be formulated as a least-squares parameter estimation problem:

$$(a, b, c)^* = \arg \min \sum_{i=1}^N \|ax_i + by_i + c\|_2^2. \quad (6.11)$$

We simply arrange the point coordinates into a matrix:

$$\mathbf{A} = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \vdots & & \\ x_k & y_k & 1 \end{bmatrix}, \quad (6.12)$$

and find the minimum singular vector of \mathbf{A} .

After obtaining the line parameters (a, b, c) from the nearest neighbors, the perpendicular distance from any point (x, y) to this line can be expressed as:

$$d = \frac{ax + by + c}{\sqrt{a^2 + b^2}}. \quad (6.13)$$

Since the denominator is a constant that can be ignored, we can directly use the residual:

$$e = ax + by + c, \quad (6.14)$$

as the objective function. The line equation also provides the corresponding Jacobian matrix:

$$\frac{\partial e}{\partial x} = a, \quad \frac{\partial e}{\partial y} = b. \quad (6.15)$$

Thus, the fitted line results can guide the optimization direction. We will see similar results in 3D point-to-plane ICP later.

Now we incorporate the lidar's pose into the above discussion. Let the lidar's position and orientation be $\mathbf{x} = (x, y, \theta)$. For a lidar point with distance and angle (r_i, ρ_i) , we transform it to world coordinates to get \mathbf{p}_i^w . With line parameters (a_i, b_i, c_i) fitted from its nearest neighbors, the Jacobian matrix of its residual e_i with respect to the pose can be expressed using the chain rule:

$$\frac{\partial e_i}{\partial \mathbf{x}} = \frac{\partial e_i}{\partial \mathbf{p}_i^w} \frac{\partial \mathbf{p}_i^w}{\partial \mathbf{x}}, \quad (6.16)$$

where the latter term is given in Equation (6.9), and the former term is determined by the line parameters. Multiplying them together yields:

$$\frac{\partial e_i}{\partial \mathbf{x}} = [a_i, b_i, -a_i r_i \sin(\rho_i + \theta) + b_i r_i \cos(\rho_i + \theta)]^\top. \quad (6.17)$$

6.2.4 Implementation of Point-to-Line ICP (Gauss-Newton)

Below we implement the algorithm described in the previous section. Its overall workflow is consistent with point-to-point ICP, and we only need to add an interface to the existing ICP class:

Listing 6.6: src/ch6/icp_2d.cc

```

1  bool Icp2d::AlignGaussNewtonPoint2Plane(SE2& init_pose) {
2      int iterations = 10;
3      double cost = 0, lastCost = 0;
4      SE2 current_pose = init_pose;
5      const float max_dis = 0.3; // Maximum distance for nearest neighbors
6      const int min_effect_pts = 20; // Minimum number of effective points
7
8      for (int iter = 0; iter < iterations; ++iter) {
9          Mat3d H = Mat3d::Zero();
10         Vec3d b = Vec3d::Zero();
11         cost = 0;
12
13         int effective_num = 0; // Number of effective points
14
15         // Traverse source points
16         for (size_t i = 0; i < source_scan_->ranges.size(); ++i) {
17             float r = source_scan_->ranges[i];
18             if (r < source_scan_->range_min || r > source_scan_->range_max) {
19                 continue;
20             }
21
22             float angle = source_scan_->angle_min + i * source_scan_->angle_increment;
23             float theta = current_pose.so2().log();
24             Vec2d pw = current_pose * Vec2d(r * std::cos(angle), r * std::sin(angle));
25             Point2d pt;
26             pt.x = pw.x();
27             pt.y = pw.y();
28
29             // Find 5 nearest neighbors
30             std::vector<int> nn_idx;
31             std::vector<float> dis;
```

```

32     kdTree_.nearestKSearch(pt, 5, nn_idx, dis);
33
34     std::vector<Vec2d> effective_pts; // Effective points
35     for (int j = 0; j < nn_idx.size(); ++j) {
36         if (dis[j] < max_dis) {
37             effective_pts.emplace_back(
38                 Vec2d(target_cloud_->points[nn_idx[j]].x,
39                       target_cloud_->points[nn_idx[j]].y));
40         }
41     }
42
43     if (effective_pts.size() < 3) {
44         continue;
45     }
46
47     // Fit line and assemble J, H and error
48     Vec3d line_coeffs;
49     if (math::FitLine2D(effective_pts, line_coeffs)) {
50         effective_num++;
51         Vec3d J;
52         J << line_coeffs[0], line_coeffs[1],
53             -line_coeffs[0] * r * std::sin(angle + theta) + line_coeffs[1] * r * std::cos(angle + theta);
54         H += J * J.transpose();
55
56         double e = line_coeffs[0] * pw[0] + line_coeffs[1] * pw[1] + line_coeffs[2];
57         b += -J * e;
58
59         cost += e * e;
60     }
61
62     if (effective_num < min_effect_pts) {
63         return false;
64     }
65
66     // solve for dx
67     Vec3d dx = H.ldlt().solve(b);
68     if (isnan(dx[0])) {
69         break;
70     }
71
72     cost /= effective_num;
73     if (iter > 0 && cost >= lastCost) {
74         break;
75     }
76
77     LOG(INFO) << "iter " << iter << " cost = " << cost << ", effect num: " <<
78     effective_num;
79
80     current_pose.translation() += dx.head<2>();           current_pose.so2() =
81     current_pose.so2() *
82     SO2::exp(dx[2]);
83     lastCost = cost;
84
85     init_pose = current_pose;
86     LOG(INFO) << "estimated pose: " << current_pose.translation().transpose()
87     << ", theta: " << current_pose.so2().log();
88
89     return true;
}

```

In the implementation, we search for five nearest neighbors around the target point and use them to fit a local line segment. The 2D line fitting algorithm is provided in common/-math_utils.h:

Listing 6.7: src/common/math_utils.h

```

1 template <typename S>
2 bool FitLine2D(const std::vector<Eigen::Matrix<S, 2, 1>>& data, Eigen::Matrix<S, 3,
3   1>& coeffs) {
4   if (data.size() < 2) {

```

```

4     return false;
5 }
6
7 Eigen::MatrixXd A(data.size(), 3);
8 for (int i = 0; i < data.size(); ++i) {
9     A.row(i).head<2>() = data[i].transpose();
10    A.row(i)[2] = 1.0;
11 }
12
13 Eigen::JacobiSVD svd(A, Eigen::ComputeThinV);
14 coeffs = svd.matrixV().col(2);
15 return true;
16 }
```

Note its similarity to the 3D plane fitting algorithm. Finally, the test case from the previous section can be used to examine the registration effect of point-to-line ICP. Since its results are similar to point-to-point ICP, we won't include additional figures here - readers are encouraged to experiment themselves (using the test program from the previous section with `-method=point2plane`). Generally speaking, point-to-line ICP performs better than point-to-point ICP, though at the cost of greater computational requirements due to the need to compute multiple nearest neighbors.

6.2.5 Likelihood Field Method

Point-to-point or point-to-line ICP can be used for both scan-to-scan matching and scan-to-map registration. If we store the map as discrete 2D points, ICP-like methods can be applied to map matching in the same way. However, in 2D SLAM, we typically store the map as an **occupancy grid map** with a certain resolution. This image-like map has an update mechanism that provides some filtering effect against dynamic objects (which we will implement in the next section). Thus, we can design a registration method that aligns scan data with grid maps in an ICP-like manner. The likelihood field method (also known as Gaussian Likelihood Field) introduced in this section is precisely such an approach for registering scan data with grid maps [7].

In point-to-point ICP, we compute Euclidean distance errors between target points and their nearest neighbors in another point cloud. These errors grow with the squared distance between points and ultimately form the objective function through summation. Intuitively, we can imagine a **spring** installed between each point and its nearest neighbor. The collective pull of these springs eventually brings the point cloud to the position of minimum energy. However, in ICP methods, we must reinstall these springs during each iteration, which is computationally expensive.

An alternative approach is to consider that the point cloud generates a **field** in space rather than installing springs between points. This field attracts nearby point clouds, with its attractive force decaying quadratically with distance. This is essentially the idea behind the likelihood field method. We can define a decaying field around each point in the map. Unlike physical fields, however, the fields in computer programs have defined **effective ranges** and **resolutions**. The field can decay quadratically or follow a Gaussian distribution with distance. When a measured point falls near the field, we can use the field's value as the error function for that point.

The likelihood field method can be used to register either two scan datasets or a scan dataset with a map dataset. More commonly, it works in conjunction with grid maps for map matching. To perform registration, we first need to generate this likelihood field. In this section, we generate the likelihood field only for point cloud data. Later, after introducing occupancy grid maps, we can also generate likelihood field maps for grid maps. The likelihood field can be further bound with submaps to achieve simple and fast registration. Here,

we "draw" a distance-decaying circle around each point. These circles are fixed and can be precomputed.



Figure 6-5: An example of a likelihood field

Figure 6-5 shows a 2D scan dataset and its corresponding likelihood field. Visually, we can observe that the likelihood field radiates from each scan point and gradually decays with distance. We can customize its range and decay characteristics. The likelihood field essentially describes a distance function between each pixel and its nearest scan point, also referred to as a distance transform map in some applications [141]. With the likelihood field, we no longer need nearest neighbor structures like K-d trees to find the closest point for a given point; instead, we can directly use the field's readings.

Next, we derive the scan matching algorithm based on the likelihood field. The readings from the likelihood field can directly serve as the objective function for registration. Consider a point \mathbf{p}_i^B transformed by pose \mathbf{x} to obtain \mathbf{p}_i^W in the world coordinate frame. Simultaneously, there exists a likelihood field π in the world coordinate frame⁴. The reading of this point in the likelihood field π is $\pi(\mathbf{p}_i^W)$. Thus, \mathbf{x} can be obtained by solving the optimization problem:

$$\mathbf{x}^* = \arg \min_{\mathbf{x}} \sum_{i=1}^n \|\pi(\mathbf{p}_i^W)\|_2^2. \quad (6.18)$$

The Jacobian matrix of the π function with respect to pose \mathbf{x} can be decomposed via the chain rule:

$$\frac{\partial \pi}{\partial \mathbf{x}} = \frac{\partial \pi}{\partial \mathbf{p}_i^W} \frac{\partial \mathbf{p}_i^W}{\partial \mathbf{x}}. \quad (6.19)$$

The latter term is given in Equation (6.9), so we focus on the former term.

Since the likelihood field is stored as an image, \mathbf{p}_i^W must be sampled at a certain resolution. Let the transformation from \mathbf{p}_i^W to its image coordinates \mathbf{p}_i^f be:

$$\mathbf{p}_i^f = \alpha \mathbf{p}_i^W + \mathbf{c}, \quad (6.20)$$

⁴Note that the likelihood field doesn't necessarily need to be maintained in the world coordinate frame; later discussions will show it is primarily maintained in submap coordinates.

where α is the scaling factor and $\mathbf{c} \in \mathbb{R}^2$ is the offset of the image center. Note that image coordinates typically start from the top-left corner, while object coordinates usually originate from the center, so the offset is typically half the image dimensions. The derivative of function π with respect to \mathbf{p}_i^W is:

$$\frac{\partial \pi}{\partial \mathbf{p}_i^W} = \frac{\partial \pi}{\partial \mathbf{p}_i^f} \frac{\partial \mathbf{p}_i^f}{\partial \mathbf{p}_i^W} = \alpha [\Delta \pi_x, \Delta \pi_y]^\top, \quad (6.21)$$

where $[\Delta \pi_x, \Delta \pi_y]$ represents the gradient of the likelihood field in the image. Since we define the likelihood function for each point as a smooth function, its gradient is equally reliable. Multiplying these two matrices yields the Jacobian matrix of each residual with respect to the pose:

$$\frac{\partial \pi}{\partial \mathbf{x}} = [\alpha \Delta \pi_x, \alpha \Delta \pi_y, -\alpha \Delta \pi_x r_i \sin(\rho_i + \theta) + \alpha \Delta \pi_y r_i \cos(\rho_i + \theta)]^\top. \quad (6.22)$$

Using this Jacobian matrix, we can implement registration based on the Gauss-Newton method.

6.2.6 Implementation of Likelihood Field Method (Gauss-Newton)

When implementing the likelihood field method, we need to generate the corresponding likelihood field when setting the target point cloud. Each point in the likelihood field can use a pre-generated, fixed-size template, which is then "pasted" onto each point of the target point cloud.

Listing 6.8: src/ch6/likelihood_field.cc

```

1  class LikelihoodField {
2      public:
3          /// 2D field template, generated when setting target scan or map
4          struct ModelPoint {
5              ModelPoint(int dx, int dy, float res) : dx_(dx), dy_(dy), residual_(res) {}
6              int dx_ = 0;
7              int dy_ = 0;
8              float residual_ = 0;
9          };
10
11     private:
12         std::vector<ModelPoint> model_; // 2D template
13     };
14
15     void LikelihoodField::BuildModel() {
16         const int range = 20; // Template size in pixels
17         for (int x = -range; x <= range; ++x) {
18             for (int y = -range; y <= range; ++y) {
19                 model_.emplace_back(x, y, std::sqrt((x * x) + (y * y)));
20             }
21         }
22     }
23
24     void LikelihoodField::SetTargetScan(Scan2d::Ptr scan) {
25         target_ = scan;
26
27         // Generate field function on target points
28         field_ = cv::Mat(1000, 1000, CV_32F, 30.0);
29
30         for (size_t i = 0; i < scan->ranges.size(); ++i) {
31             if (scan->ranges[i] < scan->range_min || scan->ranges[i] > scan->range_max) {
32                 continue;
33             }
34
35             double real_angle = scan->angle_min + i * scan->angle_increment;
36             double x = scan->ranges[i] * std::cos(real_angle) * resolution_ + 500;

```

```

37     double y = scan->ranges[i] * std::sin(real_angle) * resolution_ + 500;
38
39     // Fill field function around (x,y)
40     for (auto& model_pt : model_) {
41         int xx = int(x + model_pt.dx_);
42         int yy = int(y + model_pt.dy_);
43         if (xx >= 0 && xx < field_.cols && yy >= 0 && yy < field_.rows &&
44             field_.at<float>(yy, xx) > model_pt.residual_) {
45             field_.at<float>(yy, xx) = model_pt.residual_;
46         }
47     }
48 }
49

```

We use a 1000×1000 pixel image to store the likelihood field data. This class generates a template with a 20-pixel edge length during construction and then applies this template to each point.

Once the likelihood field is generated, we can use the previously described Gauss-Newton iteration method to register two scan datasets.

Listing 6.9: src/ch6/likelihood_field.cc

```

1  bool LikelihoodField::AlignGaussNewton(SE2& init_pose) {
2      int iterations = 10;
3      double cost = 0, lastCost = 0;
4      SE2 current_pose = init_pose;
5      const int min_effect_pts = 20; // Minimum number of effective points
6      const int image_boarder = 20; // Image border margin
7
8      for (int iter = 0; iter < iterations; ++iter) {
9          Mat3d H = Mat3d::Zero();
10         Vec3d b = Vec3d::Zero();
11         cost = 0;
12
13         int effective_num = 0; // Number of effective points
14
15         // Traverse source points
16         for (size_t i = 0; i < source_->ranges.size(); ++i) {
17             float r = source_->ranges[i];
18             if (r < source_->range_min || r > source_->range_max) {
19                 continue;
20             }
21
22             float angle = source_->angle_min + i * source_->angle_increment;
23             float theta = current_pose.so2().log();
24             Vec2d pw = current_pose * Vec2d(r * std::cos(angle), r * std::sin(angle));
25
26             // Image coordinates in the field
27             Vec2i pf = (pw * resolution_ + Vec2d(500, 500)).cast<int>();
28
29             if (pf[0] >= image_boarder && pf[0] < field_.cols - image_boarder && pf[1] >=
30                 image_boarder &&
31                 pf[1] < field_.rows - image_boarder) {
32                 effective_num++;
33
34                 // Image gradient
35                 float dx = 0.5 * (field_.at<float>(pf[1], pf[0] + 1) - field_.at<float>(pf
36                     [1], pf[0] - 1));
37                 float dy = 0.5 * (field_.at<float>(pf[1] + 1, pf[0]) - field_.at<float>(pf
38                     [1] - 1, pf[0]));
39
40                 Vec3d J;
41                 J << resolution_ * dx, resolution_ * dy,
42                     -resolution_ * dx * r * std::sin(angle + theta) + resolution_ * dy * r * std
43                         ::cos(angle + theta);
44                 H += J * J.transpose();
45
46                 float e = field_.at<float>(pf[1], pf[0]);
47                 b += -J * e;
48
49                 cost += e * e;
50             }
51
52         }
53
54     }
55
56     return cost < 1e-6;
57 }
58

```

```

47 }
48
49 if (effective_num < min_effect_pts) {
50     return false;
51 }
52
53 // solve for dx
54 Vec3d dx = H.ldlt().solve(b);
55 if (isnan(dx[0])) {
56     break;
57 }
58
59 cost /= effective_num;
60 if (iter > 0 && cost >= lastCost) {
61     break;
62 }
63
64 LOG(INFO) << "iter " << iter << " cost = " << cost << ", effect num: " <<
65     effective_num;
66
67 current_pose.translation() += dx.head<2>();
68 current_pose.so2() = current_pose.so2() * S02::exp(dx[2]);
69 lastCost = cost;
70
71 init_pose = current_pose;
72 return true;
73 }
```

We only need to replace the residuals and Jacobians from the previous ICP with those from the likelihood field. Readers can run the provided test program to view the matching results of the 2D likelihood field method:

Listing 6.10: Terminal input:

```
bin/test_2d_icp_likelihood
```

In addition to displaying the registration results, this program also shows real-time single-frame likelihood field data. Readers should observe that the likelihood field aligns with the scan data, as shown in Figure 6-6.

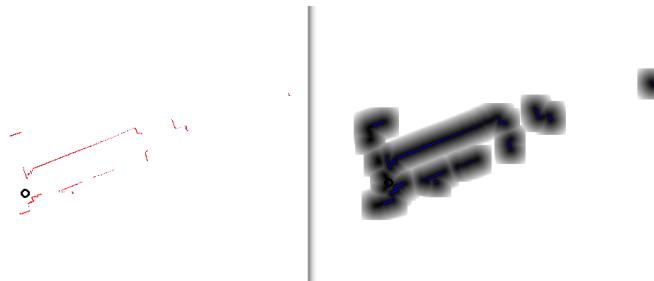


Figure 6-6: Real-time scan data and likelihood field data

6.2.7 Implementation of Likelihood Field Method (g2o)

Below we demonstrate how to use the g2o optimizer [142] to implement a likelihood field-based scan matching algorithm. By using an optimizer, we can more conveniently employ different iteration strategies and set robust kernel functions to achieve more robust matching algorithms. In fact, all the registration methods implemented earlier can be adapted to an optimizer-based approach. Now we define the SE(2) pose vertex and the observation error edges corresponding to each scan point.

Listing 6.11: src/ch6/g2o_types.h

```

1  class VertexSE2 : public g2o::BaseVertex<3, SE2> {
2      public:
3          EIGEN_MAKE_ALIGNED_OPERATOR_NEW
4
5      void setToOriginImpl() override { _estimate = SE2(); }
6      void oplusImpl(const double* update) override {
7          _estimate.translation()[0] += update[0];
8          _estimate.translation()[1] += update[1];
9          _estimate.so2() = _estimate.so2() * SO2::exp(update[2]);
10     }
11 }
12
13 class EdgeSE2LikelihoodFiled : public g2o::BaseUnaryEdge<1, double, VertexSE2> {
14     public:
15         EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
16         EdgeSE2LikelihoodFiled(const cv::Mat& field_image, double range, double angle,
17             float resolution
18             = 10.0) : field_image_(field_image), range_(range), angle_(angle), resolution_(resolution) {}
19
20     void computeError() override {
21         VertexSE2* v = (VertexSE2*)_vertices[0];
22         SE2 pose = v->estimate();
23         Vec2d pw = pose * Vec2d(range_ * std::cos(angle_), range_ * std::sin(angle_));
24         Vec2i pf = (pw * resolution_ + Vec2d(field_image_.rows / 2, field_image_.cols / 2)).cast<int>();
25
26         if (pf[0] >= image_boarder_ && pf[0] < field_image_.cols - image_boarder_ && pf
27             [1] >=
28             image_boarder_ && pf[1] < field_image_.rows - image_boarder_) {
29             _error[0] = field_image_.at<float>(pf[1], pf[0]);
30         } else {
31             _error[0] = 0;
32             setLevel(1);
33         }
34     }
35
36     void linearizeOplus() override {
37         VertexSE2* v = (VertexSE2*)_vertices[0];
38         SE2 pose = v->estimate();
39         float theta = pose.so2().log();
40         Vec2d pw = pose * Vec2d(range_ * std::cos(angle_), range_ * std::sin(angle_));
41         Vec2i pf = (pw * resolution_ + Vec2d(field_image_.rows / 2, field_image_.cols / 2)).cast<int>();
42
43         if (pf[0] >= image_boarder_ && pf[0] < field_image_.cols - image_boarder_ && pf
44             [1] >=
45             image_boarder_ && pf[1] < field_image_.rows - image_boarder_) {
46             // Image gradient
47             float dx = 0.5 * (field_image_.at<float>(pf[1], pf[0] + 1) - field_image_.at<
48                 float>(pf[1], pf[0] - 1));
49             float dy = 0.5 * (field_image_.at<float>(pf[1] + 1, pf[0]) - field_image_.at<
50                 float>(pf[1] - 1, pf[0]));
51
52             _jacobianOplusXi << resolution_ * dx, resolution_ * dy,
53             -resolution_ * dx * range_ * std::sin(angle_ + theta) +
54             resolution_ * dy * range_ * std::cos(angle_ + theta);
55         } else {
56             _jacobianOplusXi.setZero();
57             setLevel(1);
58         }
59     }
60
61     private:
62         const cv::Mat& field_image_;
63         double range_;
64         double angle_;
65         float resolution_ = 10.0;
66         inline static const int image_boarder_ = 10;
67     };

```

The Jacobian matrix here is consistent with our previous derivation, except that the like-

lihood field image has been moved inside the class for quick lookup of corresponding field function values and their gradients. Next, we only need to convert the iteration process from the Gauss-Newton method into an optimization problem.

Listing 6.12: src/ch6/likelihood_field.cc

```

1  bool LikelihoodField::AlignG2O(SE2& init_pose) {
2    using BlockSolverType = g2o::BlockSolver<g2o::BlockSolverTraits<3, 1>>;
3    using LinearSolverType = g2o::LinearSolverCholmod<BlockSolverType::PoseMatrixType
4      >;
5    auto* solver = new g2o::OptimizationAlgorithmLevenberg(
6      g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>()));
7    g2o::SparseOptimizer optimizer;
8    optimizer.setAlgorithm(solver);
9
10   auto* v = new VertexSE2();
11   v->setId(0);
12   v->setEstimate(init_pose);
13   optimizer.addVertex(v);
14
15   // Traverse source points
16   for (size_t i = 0; i < source_->ranges.size(); ++i) {
17     float r = source_->ranges[i];
18     if (r < source_->range_min || r > source_->range_max) {
19       continue;
20     }
21
22     float angle = source_->angle_min + i * source_->angle_increment;
23     auto e = new EdgeSE2LikelihoodFiled(field_, r, angle, resolution_);
24     e->setVertex(0, v);
25     e->setInformation(Eigen::Matrix<double, 1, 1>::Identity());
26     optimizer.addEdge(e);
27   }
28
29   optimizer.setVerbose(true);
30   optimizer.initializeOptimization();
31   optimizer.optimize(10);
32
33   init_pose = v->estimate();
34   return true;
}

```

This implements the g2o-based 2D scan matching algorithm. By adding `-method=g2o` to the test program in Section 6.2.5, you can test the optimizer version of likelihood field matching. Since the results are similar, we won't include result images in this section. Readers can also implement versions based on Ceres or other optimizers following similar principles. Additionally, we can perform linear interpolation on the likelihood field image (Figure 6-6) to obtain more accurate error functions. We leave these two aspects as exercises.

6.3 Occupancy Grid Map

6.3.1 Principle of Occupancy Grid Map

After performing scan matching, we obtain the relative motion between two sets of scan data. This process is equivalent to the `**localization**` problem in SLAM. Now, let us examine the `**mapping**` component.

If we use the scan-to-map approach for scan matching, we can derive its pose relative to the map. Naturally, we can merge this scan data into the map to form a local map. However, there are some considerations in the actual map-building process. For example, should the map be constructed all at once or piece by piece? Should all scan points be combined into a single map, or should strategies like overlapping and refreshing be implemented? Early 2D SLAM solutions often adopted a simpler, monolithic map-building approach, but this

method has many limitations. Therefore, this book introduces a more flexible management model based on **grid maps** and **submaps**.

First, let us discuss the **occupancy grid map**. An occupancy grid is a map format that stores occupancy probabilities in a grid-like (or image-like) structure. A grid is a very simple form of a 2D map. It divides the map into many small planar cells, each storing custom information. The organization of this information is highly flexible. If obstacle information is stored, it is called an obstacle grid map, which can be used for path planning [143]. In some applications, semantic information is also stored in grids, known as semantic grids [144]. Grid maps are often associated with images, where each grid cell corresponds to a pixel. The storage and visualization of grid maps can be implemented using image libraries like OpenCV. Overall, grid maps are a widely used method for representing dense information on a 2D plane.

In many robotic applications, people are only concerned with the **traversability** of each grid cell and not more complex semantic information (though passenger vehicles are an exception, which is why grid maps are rarely used in outdoor vehicles). Therefore, we only need to indicate whether a grid cell is occupied by an obstacle. The presence of an obstacle is probabilistic. Before scanning the map, we have no knowledge of whether obstacles exist, so the occupancy probability should initially be set to 0.5. If a grid cell is observed multiple times to contain an obstacle, its occupancy probability gradually increases to 1. Conversely, if a grid cell is repeatedly observed to be traversable, its probability decreases to 0. The endpoint measured by a 2D laser sensor indicates that the grid is occupied, while the line connecting the sensor to the endpoint indicates that the grid is traversable. Thus, 2D laser scan data can be easily converted into a grid format.

It is worth noting that some applications use **occupancy** grids to represent obstacles, while others use **traversability** probability to indicate whether a grid is passable. These two probabilities are inversely related but functionally equivalent. In an occupancy grid, a probability of 1 means the grid contains an obstacle, whereas in a traversability grid, a probability of 1 means the grid is obstacle-free. In practice, both approaches can be used freely.

On the other hand, grid maps can be dynamically updated, and not every grid cell's probability converges to 0 or 1. If a robot observes a grid cell as an obstacle multiple times, its occupancy probability will continuously rise. If a cell is repeatedly observed as empty, its occupancy probability should decrease. If an obstacle initially exists but later moves away, the grid cell's probability will first increase and then decrease. In summary, an occupancy grid map should satisfy the following descriptions:

1. It stores the probability of each grid cell being occupied by an obstacle in a grid-like format. This probability should be a floating-point number between 0 and 1.
2. The grid has a certain resolution and is typically dense.
3. The occupancy probability changes with observations. Mathematically, the update logic for occupancy probability should comply with probabilistic principles. However, from an engineering perspective, simpler metrics like observation counts can also be used.

Figure 6-7 shows an example of a 2D LiDAR scan being converted into grid cells. When a laser beam is emitted from the sensor, the endpoint corresponds to an occupied grid cell, while the path from the sensor center to the endpoint is considered traversable. Note that this geometric model is only valid for **2D** robots. If the robot has significant height or the laser beams are tilted, this model no longer applies. In such cases, the 2D LiDAR occupancy

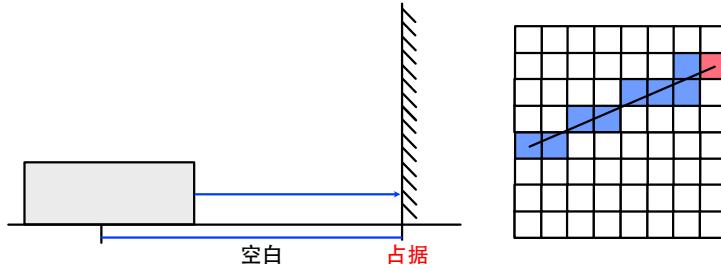


Figure 6-7: Occupancy grid and ray casting process for 2D LiDAR

grid can only indicate whether obstacles exist **at this specific height**. Obstacles at other heights, such as low steps, tables, or hanging objects, may still render the robot unable to pass. Therefore, if the robot's motion cannot be simplified to 2D, the map must account for the influence of obstacles at other heights.

Due to the resolution limitations of grid maps, converting continuous laser beams into probability updates for each grid cell involves a **rasterization** process, as illustrated on the right side of Figure 6-7 . Rasterization is a concept in computer graphics that describes the conversion of geometric shapes into grid-based outputs. In 2D grid maps, we can choose to compute the rasterization result for each laser beam. However, if the LiDAR's angular resolution is high or the measurement range is long, this process can be time-consuming. Alternatively, we can precompute a fixed-size template region. The former requires rasterizing each laser scan line, while the latter involves rasterizing each template point. We will implement both algorithms and compare their performance.

6.3.2 Map Generation Based on Bresenham's Algorithm

Bresenham's algorithm is a rasterization method for straight lines, commonly used for vectorization of geometric lines [145]. It can be implemented entirely with integer operations, making it highly efficient. Since grid map coordinates are inherently integer-based, Bresenham's algorithm is well-suited for updating grid maps.

Let the robot's origin in the map coordinate system be \mathbf{p}_1 , and an endpoint be \mathbf{p}_2 , both with integer coordinates. We aim to fill a straight line from \mathbf{p}_1 to \mathbf{p}_2 in the map, marking these cells as traversable. The Bresenham algorithm proceeds as follows:

1. Compute $[dx, dy] = p_2 - p_1$, representing the direction of coordinate growth.
2. Compare $|dx|$ and $|dy|$, selecting the larger one as the primary growth axis (assume the x -axis for now).
3. Initialize (x, y) at \mathbf{p}_1 . Since the line's slope is dy/dx , each time x increments by 1, the error between the discrete point and the true line increases by dy/dx . When this error exceeds 0.5, increment y by 1 and decrease the error by 1.
4. Repeat until (x, y) reaches \mathbf{p}_2 .

Step 3 involves floating-point operations, but we prefer integer-only computation. Thus, we multiply all terms and conditions in Step 3 by $2dx$ and subtract dx , transforming it into:

1. Compute $[dx, dy] = p_2 - p_1$, representing the direction of coordinate growth.

2. Compare $|dx|$ and $|dy|$, selecting the larger one as the primary growth axis (assume the x -axis for now).
3. Initialize (x, y) at \mathbf{p}_1 and set the initial error $e = -dx$. Each time x increments by 1, e increases by $2dy$. If $e > 0$, increment y by 1 and subtract $2dx$ from e .
4. Repeat until (x, y) reaches \mathbf{p}_2 .

This avoids floating-point and division operations, using only additions and multiplications. Similarly, if the y -axis is the primary growth direction, swap the roles of x and y .

The implementation of Bresenham's algorithm in the grid map is as follows:

Listing 6.13: src/ch6/occupancy_map.cc

```

1 void OccupancyMap::BresenhamFilling(const Vec2i& p1, const Vec2i& p2) {
2     int dx = p2.x() - p1.x();
3     int dy = p2.y() - p1.y();
4     int ux = dx > 0 ? 1 : -1;
5     int uy = dy > 0 ? 1 : -1;
6
7     dx = abs(dx);
8     dy = abs(dy);
9     int x = p1.x();
10    int y = p1.y();
11
12    if (dx > dy) {
13        // Increment along x
14        int e = -dx;
15        for (int i = 0; i < dx; ++i) {
16            x += ux;
17            e += 2 * dy;
18            if (e >= 0) {
19                y += uy;
20                e -= 2 * dx;
21            }
22
23            if (Vec2i(x, y) != p2) {
24                SetPoint(Vec2i(x, y), false);
25            }
26        }
27    } else {
28        int e = -dy;
29        for (int i = 0; i < dy; ++i) {
30            y += uy;
31            e += 2 * dx;
32            if (e >= 0) {
33                x += ux;
34                e -= 2 * dy;
35            }
36            if (Vec2i(x, y) != p2) {
37                SetPoint(Vec2i(x, y), false);
38            }
39        }
40    }
41
42    /// Set whether a grid cell is occupied
43    void OccupancyMap::SetPoint(const Vec2i& pt, bool occupy) {
44        int x = pt[0], y = pt[1];
45        if (x < 0 || y < 0 || x >= occupancy_grid_.cols || y >= occupancy_grid_.rows) {
46            if (occupy) {
47                has_outside_pts_ = true;
48            }
49            return;
50        }
51
52    /// Apply upper and lower bounds
53    uchar value = occupancy_grid_.at<uchar>(y, x);
54    if (occupy) {
55        if (value > 117) {
56            occupancy_grid_.ptr<uchar>(y)[x] -= 1;
57        }
58    }
59}
```

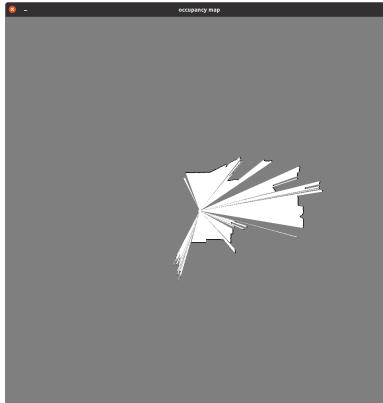


Figure 6-8: Grid map generated from a single scan

```

58     }
59 } else {
60   if (value < 137) {
61     occupancy_grid_.ptr<uchar>(y)[x] += 1;
62   }
63 }
64 }
```

Now, readers are invited to compile and run the ‘test_occupancy_grid’ program:

Listing 6.14: Terminal input:

```
bin/test_occupancy_grid --bag_path ./dataset/sad/2dmapping/floor1.bag
```

You can specify different filling methods using the ‘–method’ option. The program will display the grid map results generated from a single scan, as shown in Figure 6-8 . The final grid map is the superposition of these individual grid maps. As can be observed, the obstacles and traversable areas obtained from 2D LiDAR scans match our intuitive expectations.

The LiDAR used in our experiments does not have 360-degree coverage, leaving a blind spot behind the robot. For most robots with substantial height, it’s impossible to completely suspend sensors in midair, inevitably causing partial occlusion of sensor data.

In this chapter’s test program, we can observe that the template-based algorithm requires approximately 10-20ms to complete template filling, while the line-based method takes less than 1 millisecond - a significant difference. This is because the template method processes significantly more points than the line method (hundreds of thousands versus hundreds). If the LiDAR’s angular resolution increases or the detection range extends, the template method can limit its computation range to save processing time.

In template-based grid map updating, the sizes of submaps and templates should be set according to the actual sensor range. If the template is too small, distant scan data may not be fully utilized; if submap size is insufficient, frequent submap expansion may occur, introducing unnecessary computations. Here we’ve used parameters suitable for a 20m LiDAR range, with both grid template and submap sizes configured accordingly. For readers using longer-range LiDARs, these parameters should be appropriately increased. Note that algorithm efficiency will change noticeably with parameter adjustments - readers are encouraged to experiment with different settings.

6.4 Submaps

6.4.1 Principle of Submaps

Next, we integrate the matching algorithm with grid maps, utilizing the grid map update mechanism to combine matched data.

Furthermore, we can group several matched results together to form **submaps**. Submaps serve as an intermediate data organization level between individual scan data and the global map. They flexibly aggregate multiple scans in chronological order and can be used in both 2D LiDAR SLAM and 3D LiDAR SLAM systems.

Each submap is assigned an adjustable pose denoted by $\mathbf{T}_{WS} \in \text{SE}(2)$, where W represents the world coordinate system and S represents the submap coordinate system. During scan matching, the scan-to-map algorithm essentially computes the pose relationship \mathbf{T}_{SC} between the current LiDAR scan and the submap, where C denotes the current scan coordinate system. Thus, the world coordinate of each scan can be expressed as:

$$\mathbf{T}_{WC} = \mathbf{T}_{WS}\mathbf{T}_{SC}. \quad (6.23)$$

This approach decouples the **submap pose** variable \mathbf{T}_{WS} . During scan matching, we compute \mathbf{T}_{SC} for the current LiDAR scan, which remains fixed once determined. When adjusting the overall map shape, we only need to modify each submap's \mathbf{T}_{WS} without altering internal submap contents (i.e., per-scan \mathbf{T}_{SC}). Consequently, loop closure detection can treat submaps as fundamental units without considering individual keyframe poses in the world coordinate system.

We implement a submap-based 2D LiDAR SLAM system following this logic:

1. Each submap is associated with a likelihood field and a grid map.
2. The current scan is always matched against the current submap to determine its pose within the submap⁵.
3. Keyframes are selected based on travel distance or rotation thresholds.
4. If the robot moves beyond the current submap's bounds or the submap contains excessive keyframes, a new submap is created. The new submap centers on the current frame with $\mathbf{T}_{WS} = \mathbf{T}_{WC}$, initially empty. For matching convenience, recent keyframes from the old submap are copied to the new one.
5. Merging all submaps' grid maps yields the global map.

This workflow is not limited to 2D LiDAR SLAM. The submap paradigm can also be applied to 3D LiDAR or visual SLAM systems, though implementation becomes more complex than single-map approaches.

6.4.2 Implementation of Submaps

At the implementation level, we encapsulate both grid map and likelihood field objects within the ‘Submap’ class, while placing the mapping algorithm workflow in the ‘mapping_2d’ class.

The core structure of the ‘Submap’ class is as follows:

⁵This step is flexible in implementation. With sufficient computational resources, matching against historical submaps is also feasible.

Listing 6.15: src/ch6/submap.h

```

1 class Submap {
2     public:
3     Submap(const SE2& pose) : pose_(pose) {
4         Vec2f center = pose_.translation().cast<float>();
5         occu_map_.SetCenter(center);
6         field_.SetCenter(center);
7     }
8
9     /// Match frame against this submap to compute frame->pose
10    bool MatchScan(std::shared_ptr<Frame> frame);
11
12    /// Add a frame to the occupancy grid
13    void AddScanInOccupancyMap(std::shared_ptr<Frame> frame);
14
15    void AddKeyFrame(std::shared_ptr<Frame> frame) { frames_.emplace_back(frame); }
16
17    private:
18    SE2 pose_; // submap pose, Tw
19    size_t id_ = 0;
20
21    std::vector<std::shared_ptr<Frame>> frames_; // keyframes in this submap
22    LikelihoodField field_; // for scan matching
23    OccupancyMap occu_map_; // for grid map generation
24};

```

The submap's primary functions are scan matching and grid map updates, implemented through internal object calls:

Listing 6.16: src/ch6/submap.cc

```

1 bool Submap::MatchScan(std::shared_ptr<Frame> frame) {
2     field_.SetSourceScan(frame->scan_);
3     field_.AlignG2O(frame->pose_submap_);
4     frame->pose_ = pose_* frame->pose_submap_; // T_w_c = T_w_s * T_s_c
5
6     return true;
7 }
8
9 void Submap::AddScanInOccupancyMap(std::shared_ptr<Frame> frame) {
10    occu_map_.AddLidarFrame(frame, OccupancyMap::GridMethod::MODEL_POINTS); // update
11    grid cells
12    field_.SetFieldImageFromOccuMap(occu_map_.GetOccupancyGrid()); // update
13    likelihood field
14}

```

We employ g2o-based likelihood field methods for registration, incorporating boundary checks and robust kernels to mitigate moving object interference. Readers may refer to the source code for implementation details. The outer mapping workflow proceeds as follows:

Listing 6.17: src/ch6/mapping_2d.cc

```

1 bool Mapping2D::ProcessScan(Scan2d::Ptr scan) {
2     current_frame_ = std::make_shared<Frame>(scan);
3     current_frame_->id_ = frame_id_++;
4
5     LOG(INFO) << "processing frame " << current_frame_->id_;
6     if (last_frame_) {
7         // initialize pose from last frame
8         current_frame_->pose_ = last_frame_->pose_;
9     }
10
11    // perform scan-to-map matching
12    if (!first_scan_) {
13        // skip matching for first scan (directly add to occupancy map)
14        current_submap_->MatchScan(current_frame_);
15    }
16
17    first_scan_ = false;
18    current_submap_->AddScanInOccupancyMap(current_frame_);
19}

```

```

20 |     if (IsKeyFrame()) {
21 |         AddKeyFrame();
22 |
23 |         if (current_submap_>HasOutsidePoints() || (current_submap_->NumFrames() > 50) {
24 |             /// Trigger new submap when leaving current bounds or exceeding keyframe limit
25 |             ExpandSubmap();
26 |         }
27 |
28 |
29 |         last_frame_ = current_frame_;
30 |
31 |         return true;
32 |     }
33 |
34 |     bool Mapping2D::IsKeyFrame() {
35 |         if (last_keyframe_ == nullptr) {
36 |             return true;
37 |         }
38 |
39 |         SE2 delta_pose = last_keyframe_->pose_.inverse() * current_frame_->pose_;
40 |         if (delta_pose.translation().norm() > keyframe_pos_th_ ||
41 |             fabs(delta_pose.so2().log()) > keyframe_ang_th_) {
42 |             return true;
43 |         }
44 |
45 |         return false;
46 |     }
47 |
48 |     void Mapping2D::AddKeyFrame() {
49 |         LOG(INFO) << "add keyframe " << keyframe_id_;
50 |         current_frame_->keyframe_id_ = keyframe_id_++;
51 |
52 |         current_submap_->AddKeyFrame(current_frame_);
53 |         last_keyframe_ = current_frame_;
54 |     }
55 |
56 |     void Mapping2D::ExpandSubmap() {
57 |         // archive current submap and create new one
58 |         all_submaps_.emplace_back(current_submap_);
59 |
60 |         current_submap_ = std::make_shared<Submap>(current_frame_->pose_);
61 |         current_submap_->SetId(submap_id_++);
62 |         current_submap_->AddKeyFrame(current_frame_);
63 |         current_submap_->AddScanInOccupancyMap(current_frame_);
64 |
65 |         LOG(INFO) << "create submap " << current_submap_->GetId();
66 |     }

```

The system continuously matches current scans against the active submap. When the robot moves beyond threshold distances, new keyframes are created and added to the current submap. Excessive keyframes trigger new submap creation, with old submaps archived for global mapping.

The test program for this section only needs to read LiDAR data from ROS bags, requiring no additional inputs. This completes our pure LiDAR-based 2D SLAM system:

Listing 6.18: src/ch6/test_2d_mapping.cc

```

1  sad::RosbagIO rosbag_io(fLS::FLAGS_bag_path);
2  sad::Mapping2D mapping;
3
4  if (mapping.Init(FLAGS_with_loop_closing) == false) {
5      return -1;
6  }
7
8  rosbag_io.AddScan2DHandle("/pavo_scan_bottom", [&](Scan2d::Ptr scan) { return
9      mapping.ProcessScan(scan); }).Go();
10 cv::imwrite("./data/ch6/global_map.png", mapping.ShowGlobalMap(1000));
11 return 0;

```

Readers can run the `test_2d_mapping` program to observe the submap switching

process along with each submap's grid and likelihood field visualization. By default, loop closure detection is disabled in this section. The next section will use the same test program with loop closure enabled to demonstrate its effects.

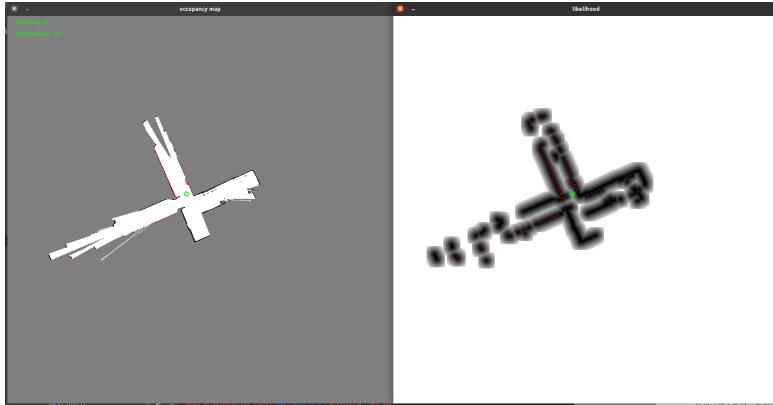


Figure 6-9: A single submap and its corresponding likelihood field in 2D LiDAR SLAM

Figure 6-9 shows an active submap and its likelihood field during operation. During testing, these visualizations update dynamically, allowing readers to observe both scan matching and submap transitions. The current LiDAR pose and scan data are overlaid in contrasting colors for clearer real-time localization and mapping visualization.

In addition to the active submap, the program outputs a global map image for debugging purposes (Figure 6-10). The actual image dimensions are configurable - readers may specify larger sizes for higher-resolution outputs. Since we haven't yet implemented loop closure, accumulated drift creates visible ghosting effects when the robot revisits areas. The next section will address this through loop detection algorithms to eliminate such cumulative errors.

6.5 Loop Closure Detection and Correction

Loop closure detection constitutes a critical component in SLAM systems. Without it, both LiDAR odometry and wheel-inertial odometry methods inevitably accumulate drift over time. The previous section's example clearly demonstrates this issue - when the robot completes a loop and returns to previously mapped areas, new submaps exhibit noticeable misalignment with historical ones. This naturally suggests that aligning current scans/submaps with historical maps and adjusting inter-submap pose relationships could effectively eliminate accumulated errors. While theoretically sound, several practical challenges emerge:

1. **Detection Scope:** Which historical submaps should be examined? This defines the **loop detection** problem. Intuitively, submaps spatially proximate to current scans should undergo alignment. However, spatial relationships rely on estimated trajectories that may contain significant drift, potentially causing missed detections. Effective implementation requires approximate bounds on accumulated error magnitude.
2. **Registration Methods:** Loop closure registration differs fundamentally from odometric alignment. Odometry assumes **continuous motion** with small displacements between optimizations. Loop closure initial guesses, however, depend on **accumulated error magnitude** and may deviate significantly from optimal alignments. Stan-



Figure 6-10: Global map without loop closure. When the robot returns from upper regions to the central area, noticeable ghosting appears between submaps.

dard ICP and likelihood field methods suffer from strong initial value dependence. Practical solutions incorporate:

- Grid searching [146]
- Particle filters [147]
- Branch-and-bound (BAB) [3]
- Image pyramids [148]

Among these, branch-and-bound and pyramid approaches offer particularly robust performance with similar underlying principles.

3. **Map Correction:** Upon loop detection, global pose adjustment can operate on either keyframes or submaps. Submap-based optimization benefits from significantly smaller problem dimensions (fewer submaps than keyframes), making it ideal when the frontend already employs submap management. However, submap-level correction cannot address intra-submap distortions, motivating some systems to retain keyframe-level optimization.

6.5.1 Multi-Resolution Loop Closure Detection

We now introduce the pyramid-based loop closure detection method, also known as **coarse-to-fine** or multi-resolution registration. Rather than being a single algorithm, this represents a general strategy for addressing initialization problems - an approach widely applicable beyond point cloud registration. For instance:

- Branch-and-bound serves both as a coarse registration method and a fundamental technique in integer programming
- Pyramid methods are equally vital for likelihood field registration and optical flow computation [149]

From a search perspective, these methods efficiently explore solution spaces while maintaining optimality. Since both branch-and-bound and coarse-to-fine approaches utilize multi-resolution grid maps, we classify them collectively as multi-resolution loop closure techniques.



Figure 6-11: Multi-resolution registration visualization. Left: Original resolution field; Right: Downsampled fields. Red: Unaligned scans; Green: Registered scans.

This section focuses on multi-resolution likelihood field matching. While fundamentally still a scan matching problem, it specifically addresses scenarios with poor initial pose estimates. Our implementation extends the standard likelihood field (20 pixels/meter resolution) with additional lower-resolution fields. As shown in Figure 6-11, we employ four field layers where each higher level halves the image dimensions. The coarsest level's blurred obstacle representations permit larger initial pose errors while maintaining discriminative power.

The implementation (src/ch6/multi_resolution_likelihood_field.cc) follows standard frame-to-frame matching but with adjusted parameters and multiple field layers:

Listing 6.19: src/ch6/multi_resolution_likelihood_field.cc

```

1 void MRLikelihoodField::BuildModel() {
2     const int range = 20; // Template pixel radius
3
4     /// Pyramid field construction
5     field_ = {
6         cv::Mat(125, 125, CV_32F, 30.0), // Level 0 (coarsest)
7         cv::Mat(250, 250, CV_32F, 30.0),
8         cv::Mat(500, 500, CV_32F, 30.0),
9         cv::Mat(1000, 1000, CV_32F, 30.0), // Level 3 (finest)
10    };
11
12    // Template generation
13    for (int x = -range; x <= range; ++x) {
14        for (int y = -range; y <= range; ++y) {
15            model_.emplace_back(x, y, std::sqrt((x * x) + (y * y)));
16        }
17    }
18
19    bool MRLikelihoodField::AlignG2O(SE2& init_pose) {
20        num_inliers_.clear();
21        inlier_ratio_.clear();
22
23        // Hierarchical registration
24        for (int l = 0; l < levels_; ++l) {
25            if (!AlignInLevel(l, init_pose)) {
26                return false; // Abort if any level fails
27            }
28        }
29    }
30
31    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
32        // ...
33    }
34
35    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
36        // ...
37    }
38
39    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
40        // ...
41    }
42
43    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
44        // ...
45    }
46
47    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
48        // ...
49    }
50
51    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
52        // ...
53    }
54
55    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
56        // ...
57    }
58
59    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
60        // ...
61    }
62
63    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
64        // ...
65    }
66
67    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
68        // ...
69    }
70
71    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
72        // ...
73    }
74
75    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
76        // ...
77    }
78
79    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
80        // ...
81    }
82
83    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
84        // ...
85    }
86
87    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
88        // ...
89    }
90
91    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
92        // ...
93    }
94
95    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
96        // ...
97    }
98
99    void MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
100       // ...
101   }
102 }
```

```

29     }
30     return true;
31 }
32
33 bool MRLikelihoodField::AlignInLevel(int level, SE2& init_pose) {
34 // G2O optimization setup (omitted for brevity)
35 ...
36
37 // Adaptive parameters
38 const double rk_delta[] = {0.2, 0.3, 0.6, 0.8}; // Robust kernel thresholds
39 const float inlier_ratio_th = 0.4; // Minimum inlier percentage
40
41 // Edge construction and optimization
42 ...
43
44 // Validation
45 if (num_inliers > 100 && inlier_ratio > inlier_ratio_th) {
46     init_pose = v->estimate();
47     return true;
48 }
49 return false;
50 }
```

The pyramid matching proceeds sequentially - failure at any level terminates the process. Notably, loop closure scenarios often exhibit partial matches due to:

1. Significant initial pose errors
2. Limited LiDAR FOV (270° in our experiments)
3. Non-identical revisit poses

We therefore set a conservative inlier threshold (40%) to balance sensitivity and robustness. This critical parameter trades off between: - **Strictness**: Higher thresholds require precise pose recurrence - **Sensitivity**: Lower thresholds permit more detections but increase false positives

While we demonstrate coarse-to-fine registration, other methods like branch-and-bound [3] similarly leverage multi-resolution maps. Both approaches share core principles but differ in implementation details - the former performs full optimization at each level, while the latter evaluates discrete candidates. Given their conceptual similarity, we focus on one representative implementation.

6.5.2 Submap-Based Loop Closure Correction

If loop closure detection successfully establishes the registration relationship between the current frame and a historical submap, we initiate a loop closure correction. This problem can again be modeled as a pose graph problem on SE(2). Moreover, since we have already constructed submaps earlier, the pose graph problem can utilize only submap poses as optimization nodes.

Consider the relationship between the current frame and a historical submap. We compute the pose of the current frame within the historical submap through multi-resolution matching. Let the pose of the historical submap S_1 itself be \mathbf{T}_{WS_1} , the pose of the current frame be \mathbf{T}_{WC} , and the pose of the submap S_2 containing the current frame be \mathbf{T}_{WS_2} . The multi-resolution matching actually computes the relative pose \mathbf{T}_{S_1C} . Thus, we can convert this result into the pose transformation between the historical submap and the current submap:

$$\mathbf{T}_{S_1S_2} = \mathbf{T}_{S_1C} \mathbf{T}_{WC}^{-1} \mathbf{T}_{WS_2} \quad (6.24)$$

Thereby obtaining the relative pose relationship between S_1 and S_2 . This process is illustrated in Figure 6-12.

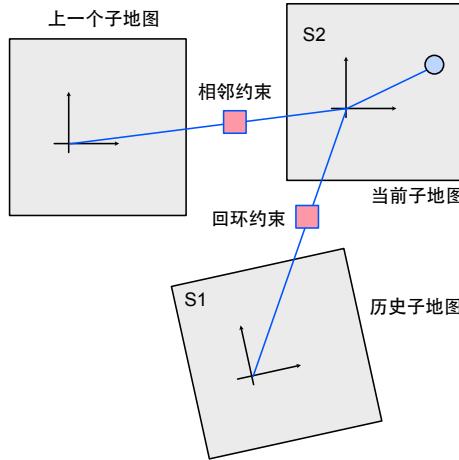


Figure 6-12: Schematic diagram of submap pose graph optimization.

In loop closure optimization, we treat each submap pose as an optimization variable and construct a pose graph for optimization. The observations in this pose graph primarily come from two sources: first, the relative pose observations between adjacent submaps, and second, the relative pose relationships between two submaps computed by loop closure detection. Again considering the relative pose between submap 1 and submap 2, assuming the relative pose observation computed by loop closure detection is $\mathbf{T}_{S_1 S_2}$, its residual term is:

$$\mathbf{e} = \text{Log}(\mathbf{T}_{WS_1}^{-1} \mathbf{T}_{WS_2} \mathbf{T}_{S_1 S_2}^{-1}) \in \mathbb{R}^3. \quad (6.25)$$

Its Jacobian matrix is rather cumbersome, so we leave it to automatic differentiation. To prevent incorrect loop closures, we also add a loop closure verification process: the accumulated error after correction should not be too large, otherwise the loop closure will be rejected as an outlier. The implementation code for loop closure detection is as follows:

Listing 6.20: src/ch6/loop_closing.cc

```

1 class LoopClosing {
2     public:
3     /// A loop closure constraint
4     struct LoopConstraints {
5         LoopConstraints(size_t id1, size_t id2, const SE2& T12) : id_submap1_(id1),
6             id_submap2_(id2), T12_(T12) {}
7         size_t id_submap1_ = 0;
8         size_t id_submap2_ = 0;
9         SE2 T12_; // Relative pose
10        bool valid_ = true;
11    };
12
13    /// Add the most recent submap, which may still be under construction
14    void AddNewSubmap(std::shared_ptr<Submap> submap);
15
16    /// Add a completed submap
17    void AddFinishedSubmap(std::shared_ptr<Submap> submap);
18
19    /// Perform loop closure detection for a new frame and update its pose and submap
20    /// poses
21    void AddNewFrame(std::shared_ptr<Frame> frame);
22
23    private:

```

```

22 // Detect possible loop closures between the current frame and historical maps
23 bool DetectLoopCandidates();
24
25 // Match the current frame with historical submaps
26 void MatchInHistorySubmaps();
27
28 // Perform pose graph optimization between submaps
29 void Optimize();
30
31 std::shared_ptr<Frame> current_frame_ = nullptr;
32 size_t last_submap_id_ = 0; // ID of the most recent submap
33
34 std::map<size_t, std::shared_ptr<Submap>> submaps_; // All submaps
35
36 // Mapping between submaps and their multi-resolution fields
37 std::map<std::shared_ptr<Submap>, std::shared_ptr<MRLikelihoodField>>
38 submap_to_field_;
39
40 std::vector<size_t> current_candidates_; // Potential loop closure candidates
41 std::map<std::pair<size_t, size_t>, LoopConstraints> loop_constraints_; // Loop
42 constraints indexed by constrained submap pairs
43
44 // Parameters
45 inline static constexpr float candidate_distance_th_ = 15.0; // Distance threshold
46 // between candidate frame and submap center
47 inline static constexpr int submap_gap_ = 1; // Minimum submap
48 // index difference for loop closure
49 inline static constexpr double loop_rk_delta_ = 1.0; // Robust kernel
50 // threshold for loop closure
51 };
52
53 void LoopClosing::AddFinishedSubmap(std::shared_ptr<Submap> submap) {
54 auto mr_field = std::make_shared<MLLikelihoodField>();
55 mr_field->SetPose(submap->GetPose());
56 mr_field->SetFieldImageFromOccuMap(submap->GetOccuMap().GetOccupancyGrid());
57 submap_to_field_.emplace(submap, mr_field);
58 }
59
60 void LoopClosing::AddNewSubmap(std::shared_ptr<Submap> submap) {
61 submaps_.emplace(submap->GetId(), submap);
62 last_submap_id_ = submap->GetId();
63 }
64
65 void LoopClosing::AddNewFrame(std::shared_ptr<Frame> frame) {
66 current_frame_ = frame;
67 if (!DetectLoopCandidates()) {
68 return;
69 }
70 }
71
72 bool LoopClosing::DetectLoopCandidates() {
73 // Require sufficient separation between current frame and historical submaps
74 has_new_loops_ = false;
75 if (last_submap_id_ < submap_gap_) {
76 return false;
77 }
78
79 current_candidates_.clear();
80
81 for (auto& sp : submaps_) {
82 if ((last_submap_id_ - sp.first) <= submap_gap_) {
83 // Skip recently created submaps
84 continue;
85 }
86
87 // Skip if valid constraint already exists between these submaps
88 auto hist_iter = loop_constraints_.find(std::pair<size_t, size_t>(sp.first,
89

```

```

    last_submap_id_));
90  if (hist_iter != loop_constraints_.end() && hist_iter->second.valid_) {
91      continue;
92  }
93
94  Vec2d center = sp.second->GetPose().translation();
95  Vec2d frame_pos = current_frame_->pose_.translation();
96  double dis = (center - frame_pos).norm();
97  if (dis < candidate_distance_th_) {
98      current_candidates_.emplace_back(sp.first);
99  }
100 }
101
102 return !current_candidates_.empty();
103 }
104
105 void LoopClosing::MatchInHistorySubmaps() {
106     // First save the scan, pose and submap to offline files for MR matching
107     // current_frame_->Dump("./data/ch6/frame_" + std::to_string(current_frame_->id_) +
108     // ".txt");
109
110     for (const size_t& can : current_candidates_) {
111         auto mr = submap_to_field_.at(submaps_[can]);
112         mr->SetSourceScan(current_frame_->scan_);
113
114         auto submap = submaps_[can];
115         SE2 pose_in_target_submap = submap->GetPose().inverse() * current_frame_->pose_;
116         // T_S1_C
117         SE2 init_guess = pose_in_target_submap;
118
119         if (mr->AlignG2O(pose_in_target_submap)) {
120             // Set constraints from current submap to target submap
121             // T_S1_S2 = T_S1_C * T_C_W * T_W_S2
122             SE2 T_this_cur =
123                 pose_in_target_submap * current_frame_->pose_.inverse() * submaps_[
124                     last_submap_id_->GetPose()];
125             loop_constraints_.emplace(std::pair<size_t, size_t>(can, last_submap_id_), T_this_cur);
126             LoopConstraints(can, last_submap_id_, T_this_cur));
127             has_new_loops_ = true;
128         }
129     }
130
131     current_candidates_.clear();
132 }
```

This code constructs a multi-resolution likelihood field for each submap upon completion, then matches recent scans with completed submaps. If multi-resolution matching succeeds, it calls the Optimize function to perform loop closure correction. The correction results directly affect each submap's pose. The Optimize function implements g2o graph optimization setup, which is similar to previously listed g2o programs and thus omitted here. To simplify the implementation, we do not run loop closure detection as a separate thread (which would require lock management for many data structures), but instead execute it sequentially in the main 2D mapping function. Meanwhile, we display submap coordinate systems and their loop closure relationships on the global map. Running the test_2d_mapping program with `-with_loop_closing=true` shows this process in real-time. The global map after loop closure correction is shown in Figure 6-13, where compared with the earlier Figure 6-10, clear improvements can be seen in the central section.

Thus, we have completed a relatively comprehensive 2D LiDAR-only SLAM program based on a submap management framework. Such grid maps can be directly used for localization and navigation if saved. Many real-world robots or functionally simple autonomous vehicles achieve self-localization and navigation through similar approaches. However, for brevity, this implementation omits many engineering details that readers may further refine.

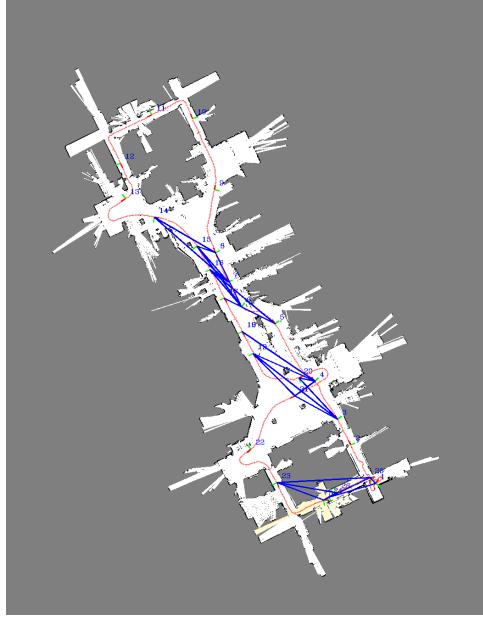


Figure 6-13: Global map with loop closure detection and inter-submap loop constraints.

6.5.3 Discussion

Below we present several potential improvements to the programs in this section. As this book primarily focuses on algorithmic principles and aims to avoid excessive engineering details that could complicate the code, the following content is mainly narrative in nature.

Laser Motion Compensation, Reflectivity Issues, etc.

First, let us examine the laser sensor itself. A laser sensor rotates periodically and is not designed to account for the simultaneous rotation of the robot's chassis. If the chassis is stationary, the laser sensor should physically rotate 360 degrees when completing a full scan. However, if the robot itself is also rotating, the actual rotation angle when the laser completes a scan may be slightly more or less than 360 degrees. This deviation depends on the speed and direction of the chassis's rotation. Similarly, translational motion of the chassis affects the actual measured distance of each laser point. This phenomenon is referred to as **motion distortion**.

Motion distortion can be mitigated using motion compensation algorithms. The basic idea is to acquire the robot's pose at the start and end of each laser scan cycle. Most laser sensors complete a full rotation in 100 milliseconds, so the robot's motion during this period must be accounted for by compensating each laser point. The same issue arises with 3D LiDAR sensors, which will be discussed in the next chapter. However, how do we obtain the poses at the start and end of the scan? After all, when the laser data is first received, the robot's pose has not yet been estimated, while motion compensation relies on this very estimate. In scenarios with an IMU, we can use its data to predict the robot's motion within the 100-millisecond window and thereby compensate for the distortion.

Motion compensation improves the accuracy of submap registration. We will detail the motion compensation algorithm in the chapter on 3D LiDAR SLAM (see Section 7.6.4), as modern 3D LiDARs typically provide timestamps for each point, whereas 2D LiDARs often

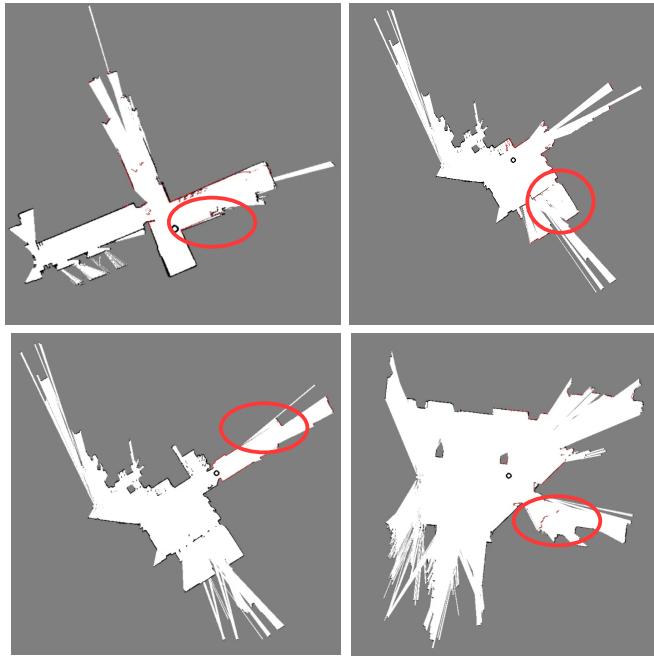


Figure 6-14: Common issues in 2D LiDAR SLAM. Top-left: Without motion compensation, the laser scan covers slightly less than the actual scene; Top-right: When observing glass surfaces, perspective effects may incorrectly mark glass walls as passable areas; Bottom-left: Reflective artifacts at the end of a left wall create a phantom wall segment beyond the actual corridor; Bottom-right: Robot vibrations cause the laser to hit the ground, creating non-existent obstacles.

do not, requiring manual estimation based on the scanning process. This involves additional sensor-specific parameters, complicating the implementation.

On the other hand, LiDAR sensors measure distance based on the time difference between emitted and reflected light. If the target object absorbs, transmits, or specularly reflects the laser beam, the measured distance may be affected. A typical example is glass surfaces, which may partially reflect or transmit the laser, resulting in intermittent obstacles or walls on the map. Mirrors represent another extreme case, where the laser beam is reflected to another area, causing the measured distance in that direction to appear much longer and making the mirror itself undetectable on the map.

If the robot vibrates during motion, the LiDAR sensor may deviate from the horizontal plane, reducing measurement consistency. The laser may also hit the ground or slopes, violating the assumptions of the grid map model. Many robots require manual annotation of sloped areas in 2D SLAM. With an IMU, the robot's 3D orientation can be estimated to determine whether the laser is tilted.

The aforementioned issues can all be observed in the experiments of this section, as shown in Figure 6-14. Therefore, real-world SLAM systems must address significantly more challenges than those covered in theoretical discussions.



Figure 6-15: Degeneration issues in 2D LiDAR under degenerate and open scenarios.

IMU and Robot Odometry Fusion

The 2D SLAM introduced in this section is a pure LiDAR solution. Although LiDAR offers high measurement accuracy, single-line LiDAR has limited observation modes and typically narrow field of view, making pure LiDAR solutions susceptible to various environmental structures. Typical examples include open areas and corridor scenarios. In these environments, 2D scan matching algorithms fundamentally cannot uniquely determine the laser scan's position, potentially exhibiting one or more additional degrees of freedom. We refer to such issues as **degeneracy problems**.

In degenerate scenarios, most pure LiDAR scan matching algorithms struggle to provide correct pose estimation. For instance, in corridor environments, likelihood field methods generate fields concentrated along both corridor walls. If the robot moves along the corridor, new scan data will appear to match perfectly with existing corridor walls. Consequently, the matching algorithm may falsely conclude the robot hasn't moved. Similar phenomena occur with ICP-type methods. Here, we say the robot's pose estimation gains **extra degrees of freedom (gauge freedom)** along the corridor direction, while the perpendicular direction remains constrained, resulting in 1 additional degree of freedom. In open square scenarios, neither translation nor rotation can be uniquely determined, yielding 3 additional degrees of freedom. Beyond corridors and squares, many regular, symmetric environments (perfect circles, single walls, multiple parallel walls, cylindrical surfaces, etc.) also exhibit degeneracy - readers should be able to identify other degenerate scenarios.

Degeneracy problems are particularly pronounced in pure LiDAR SLAM systems. Incorporating additional sensors can partially compensate for these effects. Later chapters will introduce multi-sensor fusion SLAM systems, most commonly combining IMU, wheel encoders, odometry or GPS information with LiDAR to achieve more robust performance. Both loosely-coupled and tightly-coupled approaches can compensate for LiDAR mapping or localization in degenerate scenarios.

Readers can experiment with other datasets provided in this chapter to verify the SLAM system's core functionality (see Figure 6-16). Further improvements can be achieved by implementing motion compensation, degeneracy detection, secondary echo detection, and multi-sensor fusion. However, 2D SLAM ultimately faces practical limitations and is primarily suitable for relatively simple indoor applications. The 3D SLAM introduced in the next chapter demonstrates superior performance in large-scale outdoor environments.

6.6 Summary

This chapter has presented various algorithmic modules for 2D LiDAR SLAM, which historically formed the core of SLAM research. In 2D scenarios, many problems can be simplified.

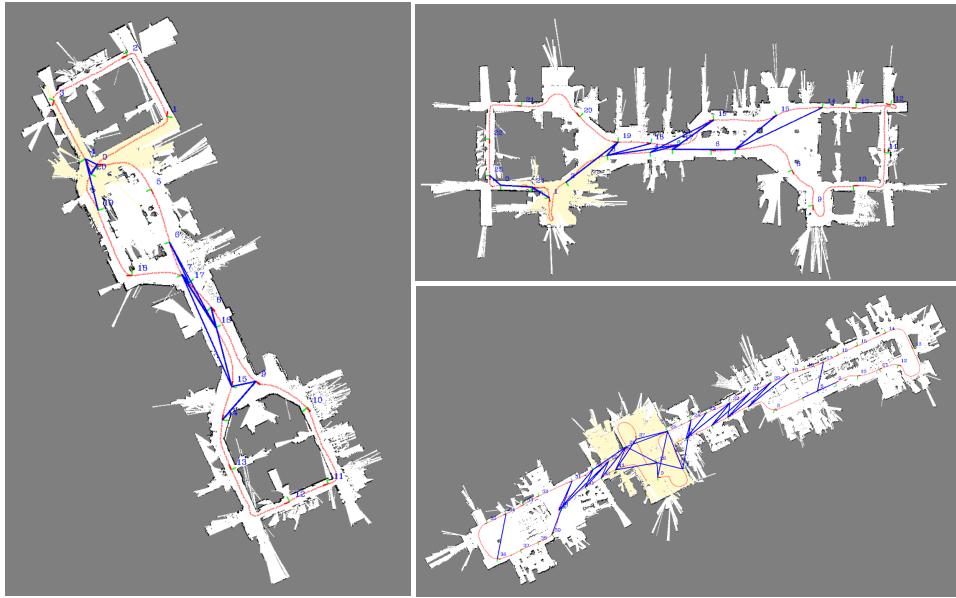


Figure 6-16: Mapping results from additional datasets in this chapter.

We implemented registration algorithms including ICP, ICL and Gaussian fields, managed them using a submap framework, and ultimately constructed global grid maps. We also explained submap management across timestamps and loop closure correction for accumulated errors. The 2D SLAM system described here serves as a prototype for various single-line LiDAR mapping and localization applications.

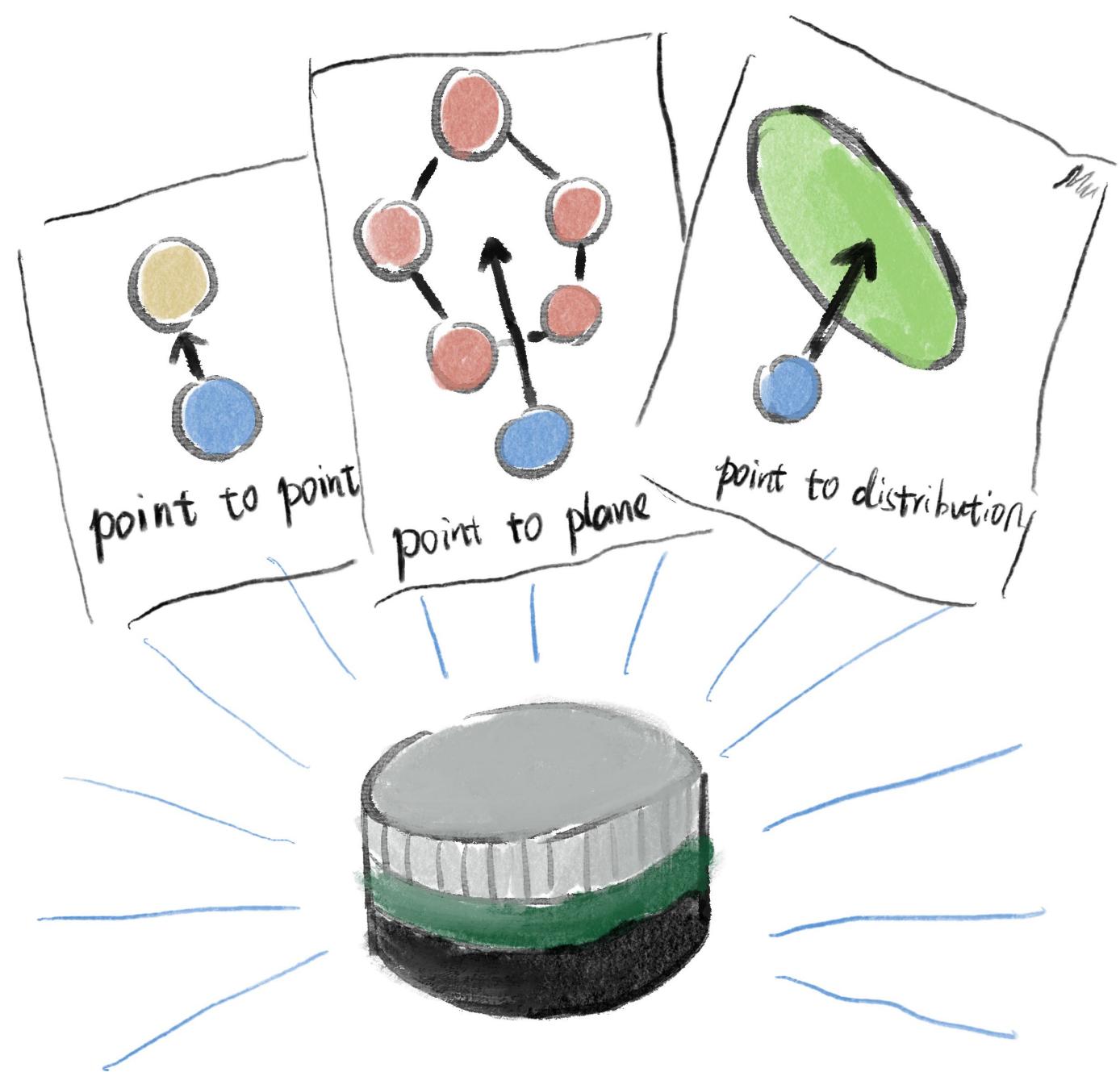
Exercises

1. Implement point-to-point 2D ICP using an optimization framework (based on g2o or ceres).
2. Implement point-to-line 2D ICP using an optimization framework (based on g2o or ceres).
3. Implement likelihood field registration method using an optimization framework (based on g2o or ceres).
4. In the likelihood field method, interpolation of the likelihood field image can be used to obtain more accurate error values and gradient functions. Implement a likelihood field scan matching method using linear interpolation.
5. Implement a degeneracy detection method for 2D LiDAR using line fitting.

Chapter 7

3D LiDAR Localization and Mapping

This chapter introduces the principles of mapping and localization using 3D LiDAR. Compared to 2D LiDAR, 3D LiDAR provides richer information, is less susceptible to occlusion, and can better reconstruct the three-dimensional structure of the environment. We can either directly register the 3D point clouds or extract geometric features before registration. Both approaches are widely used in autonomous driving applications. In this chapter, we will manually implement the core algorithms and assemble them into a LiDAR odometry system.



配准方法可以使用各种不同的
误差类型。

7.1 Working Principles of Multi-beam LiDAR

7.1.1 Mechanical LiDAR

The ranging principle of multi-beam LiDAR is the same as that of single-beam LiDAR. They emit a laser pulse toward the target object, measure the time interval between the transmitted and returned pulses, and then calculate the distance by multiplying the time interval by the speed of light. If the laser penetrates the object during measurement, the receiver may capture multiple echoes. Most LiDAR systems compute the distances from multiple echoes and use the strongest return as the final measured distance. This ranging principle is called Time of Flight (ToF), which is the primary method for most LiDAR sensors and RGB-D cameras. ToF can be further divided into Direct ToF (DToF) and Indirect ToF (IToF), with IToF further categorized into Frequency-Modulated Continuous Wave (FMCW) and Amplitude-Modulated Continuous Wave (AMCW). Due to its lower power consumption and simpler implementation, most LiDARs and RGB-D cameras currently use DToF for ranging.

Compared to single-beam LiDAR, multi-beam LiDAR typically has multiple laser emitters. These are controlled by motors and rotate around an axis at a fixed frequency (e.g., 10 revolutions per second). The laser emitters are arranged at small angular offsets, allowing them to scan objects within a certain field of view when rotating. This rotating configuration is called a **spinning LiDAR** or mechanical LiDAR. Mechanical LiDARs usually have a 360-degree horizontal field of view, while the vertical field of view depends on the emitter arrangement, typically ranging from 30 to 45 degrees. The number of emitters is referred to as the **channel count**. Due to their symmetrical design, the channel count is usually a power of two, with common configurations including 4, 16, 32, 64, and 128 channels. The complexity and cost increase significantly with higher channel counts. Figure 7-1 shows several common 3D LiDARs and their single-frame scan data, demonstrating their significantly richer information compared to 2D LiDARs.



Figure 7-1: Common 3D LiDARs on the market and their single-frame scan data.

The rich information provided by 3D LiDARs greatly simplifies many computational tasks. For example, we can detect vehicles, traffic signs, and other distinct objects in 3D point clouds [150, 151], identify static and dynamic obstacles, and extract lane markings using reflection intensity [152, 153]—tasks that are challenging with 2D LiDAR. From a localization and mapping perspective, we can register multiple scans to create a global point cloud map. Annotators can then label the point cloud to generate a **high-definition (HD) map**, enabling vehicles to achieve **high-precision localization** within the map. This is currently the standard practice for many Level 4 (L4) autonomous vehicles.

In the early years, mechanical LiDARs were primarily supplied by Velodyne. However,



Figure 7-2: Low-speed autonomous vehicles using multi-beam LiDAR as the primary sensor

with recent technological advancements, domestic manufacturers such as RoboSense, He-sai, and DJI have gradually taken over the market, while Velodyne has declined. Today, mechanical LiDARs are available at relatively reasonable prices, with cost-effective 16-channel LiDARs becoming the preferred choice for many low-speed autonomous driving products. Figure 7-2 shows some low-speed autonomous vehicles equipped with multi-beam LiDARs. These vehicles typically use one or more LiDARs for mapping and localization. To ensure unobstructed views, they commonly mount the LiDAR on the roof, allowing 360-degree coverage around the vehicle. However, due to the limited vertical field of view, roof-mounted LiDARs create significant blind spots near the vehicle. Some larger vehicles address this issue by installing one or two supplementary blind-spot LiDARs at the front to prevent collisions. Nevertheless, using multiple LiDARs significantly increases overall costs, posing a major challenge for mass production.

7.1.2 Solid-State LiDAR

Mechanical LiDARs require rotating laser emitters to achieve 360-degree coverage, inevitably incorporating precision mechanical components. These moving parts lead to high costs and vulnerability under harsh conditions like vehicle vibrations. Consequently, LiDARs that measure distances without moving the entire emission/reception assembly—collectively termed **solid-state LiDARs** (see Fig. 7-3)—have emerged.

Solid-state LiDARs employ various measurement principles:

1. **Rotating Mirror (Semi-Solid) Design:** The laser emitter and receiver remain stationary, while a prism deflects the beam to scan different directions. Though the prism undergoes minor motion, this design eliminates rotating emitters. Examples include RoboSense and DJI's mass-produced rotating mirror LiDARs, now deployed in some passenger vehicles.
2. **Pure Solid-State Designs:**
 - *Phased Array:* Scans sequentially via electronic beam steering.
 - *Flash LiDAR:* Illuminates the entire FoV simultaneously for single-shot 3D capture.

These technologies remain less mature but promise fully static operation.



Figure 7-3: Representative solid-state LiDARs and their scan patterns

Both semi- and pure solid-state LiDARs feature a scanning window defining their **Field of View (FoV)**. Typical horizontal FoVs are $<120^\circ$ (vs. 360° for mechanical LiDARs), while vertical FoVs are comparable. Unlike mechanical LiDARs, solid-state variants lack a **channel count** metric¹. For instance, DJI's Livox series employs a "petal" scanning pattern, whereas others mimic mechanical LiDAR's horizontal scanning with equivalent channels. The constrained FoV enables higher scan rates at the cost of fewer points per frame. Ranging accuracy remains comparable as both types use ToF principles.

With lower costs and extended lifespans, solid-state LiDARs are increasingly adopted in production vehicles—primarily for forward obstacle detection rather than L4 HD mapping/localization. While consumer vehicles may use 1–2 front-facing units, L4 systems often combine 4–5 solid-state LiDARs for 360° coverage, introducing calibration/synchronization challenges and eroding cost advantages over mechanical LiDARs.

Most algorithms discussed later are agnostic to LiDAR type (mechanical/solid-state). We provide datasets for both to evaluate SLAM performance. Emerging designs (e.g., spherical/hemispherical FoV solid-state LiDARs) promise broader coverage, reflecting ongoing sensor evolution that will continue driving SLAM innovation.

7.2 Scan Matching for Multi-beam LiDAR

The SLAM framework for multi-beam LiDAR is similar to that of single-beam systems. Following the previous chapter, we first discuss scan matching methods for aligning two point clouds, then address back-end processing. Multi-beam LiDAR typically yields superior scan matching compared to single-beam systems, simplifying back-end design. Most multi-beam SLAM systems forgo submap concepts, directly managing point clouds. Thus, registration primarily employs scan-to-scan or scan-to-map approaches.

7.2.1 Point-to-Point ICP

Theoretical Foundations

Point-to-point Iterative Closest Point (ICP) is a fundamental point cloud registration method. Given two point clouds $S_1 = \{\mathbf{p}_1, \dots, \mathbf{p}_m\}$ and $S_2 = \{\mathbf{q}_1, \dots, \mathbf{q}_n\}$, correct alignment

¹Some manufacturers use "equivalent channels"; e.g., many solid-state LiDARs match 128-channel mechanical performance but with narrower FoVs.

should satisfy for matched pairs $\mathbf{p}_i \in S_1, \mathbf{q}_j \in S_2$:

$$\mathbf{p}_i = \mathbf{R}\mathbf{q}_j + \mathbf{t}. \quad (7.1)$$

Note that we neither assume equal point counts nor initial ordering correspondence between clouds. The ICP workflow proceeds as:

1. Initialize pose estimate $\mathbf{R}_0, \mathbf{t}_0$.
2. Iterate from initial pose. Let $\mathbf{R}_k, \mathbf{t}_k$ denote the k -th iteration estimate.
3. For current pose, establish correspondences via nearest-neighbor search, yielding pairs $(\mathbf{p}_i, \mathbf{q}_i)$.
4. Update pose by minimizing:

$$\mathbf{R}_{k+1}, \mathbf{t}_{k+1} = \arg \min_{\mathbf{R}, \mathbf{t}} \sum_i \|\mathbf{p}_i - (\mathbf{R}\mathbf{q}_i + \mathbf{t})\|_2^2. \quad (7.2)$$

5. Check convergence; if unmet, return to step 3.

ICP essentially **alternates between pose estimation and correspondence matching** [154]. Each iteration uses the current pose to find matches, then solves for an updated pose. Key observations:

- *Alternating Minimization*: This decoupled approach simplifies computation—both matching and pose subproblems are tractable. Modern methods increasingly solve both jointly [155]. While theoretically minimizing error monotonically [140], incorrect matches (from fixed correspondences) can degrade performance.
- *Least-Square Optimization*: Equation (7.2) admits various enhancements: weight matrices, robust kernels, or alternative error metrics [139, 156, 157]. Later discussions on point-line/point-plane ICP [158] exemplify error metric modifications.
- *Feature-Based Matching*: Extracting geometric features (planes, cylinders) from stable autonomous driving scenes can reduce computation and improve robustness by operating on higher-level abstractions.

We implement basic ICP using iterative optimization (analytical solutions exist²) for consistency with later sections and flexibility in least-squares extensions. Defining point-to-point error:

$$\mathbf{e}_i = \mathbf{p}_i - \mathbf{R}\mathbf{q}_i - \mathbf{t}, \quad (7.3)$$

with right-multiplication for \mathbf{R} updates, the Jacobians w.r.t. rotation and translation are:

$$\frac{\partial \mathbf{e}_i}{\partial \mathbf{R}} = \mathbf{R}\mathbf{q}^\wedge, \quad \frac{\partial \mathbf{e}_i}{\partial \mathbf{t}} = -\mathbf{I}. \quad (7.4)$$

²See Lecture 14, Sec. 7.9.

Implementation

The core ICP algorithm is straightforward. Omitting peripheral I/O code, we present a concurrent implementation that significantly outperforms PCL's built-in ICP:

Listing 7.1: ch7/icp_3d.cc

```

1  bool Icp3d::AlignP2P(SE3& init_pose) {
2      LOG(INFO) << "aligning with point to point";
3      assert(target_ != nullptr && source_ != nullptr);
4
5      SE3 pose = init_pose;
6      pose.translation() = target_center_ - source_center_; // Initialize translation
7      LOG(INFO) << "init trans set to " << pose.translation().transpose();
8
9      // Pre-generate point indices
10     std::vector<int> index(source_->points.size());
11     std::iota(index.begin(), index.end(), 0);
12
13     // Concurrent computation buffers
14     std::vector<bool> effect_pts(index.size(), false);
15     std::vector<Eigen::Matrix<double, 3, 6>> jacobians(index.size());
16     std::vector<Vec3d> errors(index.size());
17
18     for (int iter = 0; iter < options_.max_iteration_; ++iter) {
19         // Parallel nearest neighbor search
20         std::for_each(std::execution::par_unseq, index.begin(), index.end(), [&](int idx
21             ) {
22             auto q = ToVec3d(source_->points[idx]);
23             Vec3d qs = pose * q; // Transformed point
24             std::vector<int> nn;
25             kdtree_->GetClosestPointToPointType(qs), nn, 1);
26
27             if (!nn.empty()) {
28                 Vec3d p = ToVec3d(target_->points[nn[0]]);
29                 double dis = (p - qs).norm();
30                 if (dis > options_.max_nn_distance_) return; // Reject outliers
31
32                 effect_pts[idx] = true;
33
34                 // Build residual and Jacobian
35                 Vec3d e = p - qs;
36                 Eigen::Matrix<double, 3, 6> J;
37                 J.block<3, 3>(0, 0) = pose.so3().matrix() * S03::hat(q);
38                 J.block<3, 3>(0, 3) = -Mat3d::Identity();
39
40                 jacobians[idx] = J;
41                 errors[idx] = e;
42             }
43         });
44
45         // Accumulate Hessian and error
46         double total_res = 0;
47         int effective_num = 0;
48         auto H_and_err = std::accumulate(
49             index.begin(), index.end(),
50             std::pair<Mat6d, Vec6d>(Mat6d::Zero(), Vec6d::Zero()),
51             [&](const auto& pre, int idx) {
52                 if (!effect_pts[idx]) return pre;
53                 total_res += errors[idx].squaredNorm();
54                 effective_num++;
55                 return std::pair<Mat6d, Vec6d>(
56                     pre.first + jacobians[idx].transpose() * jacobians[idx],
57                     pre.second - jacobians[idx].transpose() * errors[idx]
58                 );
59             });
60
61         if (effective_num < options_.min_effective_pts_) {
62             LOG(WARNING) << "Insufficient correspondences: " << effective_num;
63             return false;
64         }
65
66         // Solve and update
67         Vec6d dx = H_and_err.first.ldlt().solve(H_and_err.second);

```

```

67     pose.so3() = pose.so3() * S03::exp(dx.head<3>());
68     pose.translation() += dx.tail<3>();
69
70     LOG(INFO) << "iter " << iter << " | res: " << total_res
71     << " | eff: " << effective_num
72     << " | avg res: " << total_res/effective_num
73     << " | dx: " << dx.norm();
74
75     if (dx.norm() < options_.eps_) {
76         LOG(INFO) << "Converged with dx = " << dx.transpose();
77         break;
78     }
79
80     init_pose = pose;
81     return true;
82 }
83

```

The implementation leverages the KD-tree from Chapter 5 for nearest neighbor search. A test program is provided in `test_icp.cc` for evaluating registration performance:

Listing 7.2: `src/ch7/test/test_icp.cc`

```

1  sad::CloudPtr source(new sad::PointCloudType), target(new sad::PointCloudType);
2  pcl::io::loadPCDFile(fLS::FLAGS_source, *source);
3  pcl::io::loadPCDFile(fLS::FLAGS_target, *target);
4
5  bool success;
6
7  sad::evaluate_and_call([&]{
8      sad::Icp3d icp;
9      icp.SetSource(source);
10     icp.SetTarget(target);
11     icp.SetGroundTruth(gt_pose);
12
13     SE3 pose;
14     success = icp.AlignP2P(pose);
15
16     if (success) {
17         LOG(INFO) << "ICP success. Rotation (quat): "
18         << pose.so3().unit_quaternion().coeffs().transpose()
19         << " | Translation: " << pose.translation().transpose();
20
21     sad::CloudPtr aligned(new sad::PointCloudType);
22     pcl::transformPointCloud(*source, *aligned, pose.matrix().cast<float>());
23     pcl::io::savePCDFileBinaryCompressed("./data/ch7/aligned.pcd", *aligned);
24 } else {
25     LOG(ERROR) << "Alignment failed";
26 }
27 }, "ICP P2P", 1);

```

7.3 Experimental Evaluation

To facilitate algorithm benchmarking, we provide simulated datasets derived from the EPFL Statues Dataset³, which contains high-accuracy reconstructed point clouds as shown in Fig. 7-4. We generate test cases by applying random transformations to these models followed by sampling, producing target and source point clouds (available in `data/ch7/EPFL/`). The dataset includes two models (`kneeling_lady` and `aquarius`) with ground truth poses for quantitative evaluation. All registration algorithms in this chapter report per-iteration pose errors against ground truth to assess convergence.

Below demonstrates ICP testing (subsequent algorithms share the same evaluation framework):

³EPFL Statues Dataset: https://lgg.epfl.ch/statues_dataset.php

Listing 7.3: Terminal output

```

1 I0130 16:46:40.795749 84155 icp_3d.cc:13] Aligning with point-to-point ICP
2 I0130 16:46:40.805476 84155 icp_3d.cc:96] Iter 0 | Res: 11.523 | Eff: 44455 | Avg:
3     0.000259 | dx: 0.027
4 I0130 16:46:40.805512 84155 icp_3d.cc:101] Pose error: 0.0689234
5 [...]
6 I0130 16:46:40.828277 84155 icp_3d.cc:105] Converged, dx = [0.0054, -0.0021,
7     -0.0035, -0.0009, 0.0018, -0.0023]
8 I0130 16:46:40.828301 84155 test_icp.cc:54] ICP success. Pose: [0.0265, -0.0107,
9     -0.0235, 0.9993], [-0.0689, -0.1033, 0.0050]
10 I0130 16:46:40.878885 84155 sys_utils.h:32] ICP P2P avg time: 163.421ms (1 runs)
11 [...]
12 I0130 16:46:42.158504 84155 test_icp.cc:140] PCL ICP pose: [0.0292, -0.0092,
13     -0.0195, 0.9993], [-0.0636, -0.0567, 0.0027]
14 I0130 16:46:42.161834 84155 test_icp.cc:146] PCL ICP error: 0.262063
15 I0130 16:46:42.162148 84155 sys_utils.h:32] PCL ICP avg time: 895.009ms (1 runs)

```

Our concurrent implementation demonstrates $5.5\times$ speedup over PCL's ICP with superior accuracy (0.015 vs 0.262 pose error). The decreasing residuals and pose errors confirm stable convergence. Fig. 7-4 visualizes the alignment results⁴. While PCL's result shows visible misalignment (black edges), our implementation achieves sub-centimeter precision. Readers can inspect the results using:

```
1 pcl_viewer ./data/ch7/icp_trans.pcd ./data/ch7/EPFL/kneeling_lady_target.pcd
```

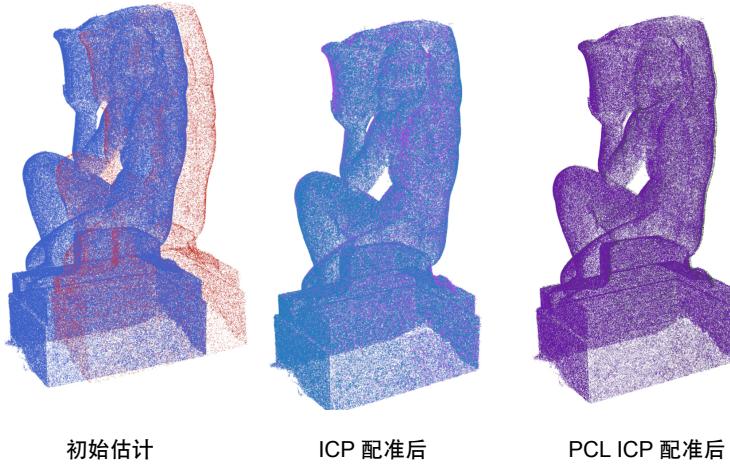


Figure 7-4: Alignment visualization: (Left) Initial pose, (Middle) Our ICP, (Right) PCL ICP

Despite its simplicity, point-to-point ICP faces limitations in autonomous driving scenarios. The sparsity of LiDAR point clouds means consecutive scans rarely sample identical surface points. Matching discrete points directly (as in basic ICP) proves suboptimal. Subsequent sections address this through:

- Multi-point correspondences (point-to-plane ICP)
- Distribution-based matching (NDT)
- Feature-level registration.

⁴Note: Point cloud colors may vary across PCL viewers due to random color assignment.

7.3.1 Point-to-Line and Point-to-Plane ICP

Theoretical Foundations

Point-to-line and point-to-plane ICP are direct extensions of the standard ICP method. Their principles are similar to the 2D point-to-line ICP introduced in the previous chapter, with the optimization variables extended to 3D space and derivatives handled using manifold representations.

Following the definitions from the previous section, let's consider two point clouds S_1 and S_2 with transformation parameters \mathbf{R}, \mathbf{t} . For each transformed point $\mathbf{R}\mathbf{q}_i + \mathbf{t}$, instead of finding a single nearest neighbor \mathbf{p}_i , we find multiple nearest neighbors and fit either a plane or a line to them.

Point-to-Plane ICP For plane fitting, assume the plane parameters are $(\mathbf{n}, d) \in \mathbb{R}^4$, where \mathbf{n} is the unit normal vector and d is the intercept. For any point \mathbf{p} on the plane:

$$\mathbf{n}^\top \mathbf{p} + d = 0. \quad (7.5)$$

The signed distance from a point \mathbf{q}_i to the plane is:

$$e_i = \mathbf{n}^\top (\mathbf{R}\mathbf{q}_i + \mathbf{t}) + d. \quad (7.6)$$

Note that since $|\mathbf{n}| = 1$, no normalization is needed. The derivatives with respect to \mathbf{R} and \mathbf{t} are:

$$\frac{\partial e_i}{\partial \mathbf{R}} = -\mathbf{n}^\top \mathbf{R}\mathbf{q}_i^\wedge, \quad \frac{\partial e_i}{\partial \mathbf{t}} = \mathbf{n}. \quad (7.7)$$

Point-to-Line ICP For line fitting, parameterize the line as:

$$\mathbf{p} = \mathbf{d}\tau + \mathbf{p}_0, \quad (7.8)$$

where \mathbf{d} is the unit direction vector, \mathbf{p}_0 is a point on the line, and τ is a scalar parameter. The perpendicular distance from \mathbf{q}_i to the line is given by the cross product:

$$\mathbf{e}_i = \mathbf{d} \times (\mathbf{R}\mathbf{q}_i + \mathbf{t} - \mathbf{p}_0) = \mathbf{d}^\wedge (\mathbf{R}\mathbf{q}_i + \mathbf{t} - \mathbf{p}_0). \quad (7.9)$$

The derivatives are:

$$\frac{\partial \mathbf{e}_i}{\partial \mathbf{R}} = -\mathbf{d}^\wedge \mathbf{R}\mathbf{q}_i^\wedge, \quad \frac{\partial \mathbf{e}_i}{\partial \mathbf{t}} = \mathbf{d}^\wedge. \quad (7.10)$$

These derivatives are straightforward to derive, and we leave their complete derivation as an exercise for the reader.

7.3.2 Point-to-Line ICP Implementation

The point-to-line ICP follows the same principle as previous methods, with modifications to use line-based residuals. Below shows the key implementation differences:

Listing 7.4: ch7/icp_3d.cc

```

1 std::for_each(std::execution::par_unseq, index.begin(), index.end(), [&](int idx) {
2     auto q = ToVec3d(source_>points[idx]);
3     Vec3d qs = pose * q; // Transformed point
4     std::vector<int> nn;
5     kdtree_->GetClosestPointToPointType(qs), nn, 5); // 5 neighbors for line fitting
6
7     if (nn.size() == 5) {
8         std::vector<Vec3d> nn_eigen;
```

```

9 |     for (int i = 0; i < 5; ++i) {
10|         nn_eigen.emplace_back(ToVec3d(target_->points[nn[i]]));
11|     }
12|
13|     Vec3d direction, point_on_line;
14|     if (!math::Fitline(nn_eigen, point_on_line, direction,
15|                         options_.max_line_distance_)) {
16|         effect_pts[idx] = false;
17|         return;
18|     }
19|
20|     Vec3d err = SO3::hat(direction) * (qs - point_on_line);
21|
22|     if (err.norm() > options_.max_line_distance_) {
23|         effect_pts[idx] = false;
24|         return;
25|     }
26|
27|     effect_pts[idx] = true;
28|
29|     // Build 3x6 Jacobian
30|     Eigen::Matrix<double, 3, 6> J;
31|     J.block<3, 3>(0, 0) = -SO3::hat(direction) * pose.so3().matrix() * SO3::hat(q);
32|     J.block<3, 3>(0, 3) = SO3::hat(direction);
33|
34|     jacobians[idx] = J;
35|     errors[idx] = err;
36| } else {
37|     effect_pts[idx] = false;
38| }
39|};

```

Key implementation notes:

- Uses 5 nearest neighbors for line fitting
- Jacobian becomes 3×6 dimensional
- Residual becomes 3D vector (perpendicular distance components)

Benchmark results show comparable performance to point-to-point ICP:

Listing 7.5: Terminal output

```

1 I0130 16:46:41.100083 84155 icp_3d.cc:232] Aligning with point-to-line
2 I0130 16:46:41.109969 84155 icp_3d.cc:325] Iter 0 | Pose error: 0.0536 | Res: 11.360 |
   Eff: 44503 | Avg: 0.000255 | dx: 0.0431
3 I0130 16:46:41.118880 84155 icp_3d.cc:325] Iter 1 | Pose error: 0.0279 | Res: 3.6516 |
   Eff: 44515 | Avg: 8.20e-05 | dx: 0.0262
4 I0130 16:46:41.127648 84155 icp_3d.cc:325] Iter 2 | Pose error: 0.0143 | Res: 1.013 |
   Eff: 44515 | Avg: 2.28e-05 | dx: 0.0138
5 I0130 16:46:41.135840 84155 icp_3d.cc:325] Iter 3 | Pose error: 0.0073 | Res: 0.292 |
   Eff: 44515 | Avg: 6.58e-06 | dx: 0.00714
6 I0130 16:46:41.135859 84155 icp_3d.cc:333] Converged, dx = [0.00545, -0.00273,
   -0.00242, -0.000307, 0.00164, -0.00229]
7 E0130 16:46:41.135869 84155 test_icp.cc:96] ICP success. Pose: [0.0296, -0.0118,
   -0.0257, 0.999], [-0.0702, -0.1019, 0.00249]
8 I0130 16:46:41.186185 84155 sys_utils.h:32] ICP P2Line avg time: 157.942ms (1 runs)

```

Performance comparison:

- **Accuracy:** Point-to-plane > Point-to-line \approx Point-to-point
- **Efficiency:** All variants significantly outperform PCL's implementation

An important observation arises: Why limit ourselves to a single residual type? Modern approaches often:

1. Classify local geometry (planar/linear) in the target cloud

2. Dynamically select point-to-plane or point-to-line residuals

This feature-adaptive approach, common in autonomous driving, will be explored later. First, we introduce another statistical registration method: NDT (Normal Distributions Transform).

7.3.3 NDT Method

Principles

Whether point-to-line or point-to-plane ICP, their fundamental difference from standard ICP lies in registering points not to individual **points**, but to certain **statistical measures**. Point-to-plane ICP fits local points to planes, while point-to-line ICP fits them to lines. Extending this idea further: Why must we presuppose whether points form planes or lines? Why precisely determine surface/line parameters? We only need the **local statistics** of the point cloud for matching. The most basic statistical measures of a point set are its **mean** and **covariance**. Following this reasoning leads to the traditional NDT (Normal Distribution Transform) method [159, 160].

This section explains NDT principles using simplified notation, which differs slightly from the original NDT papers while maintaining theoretical consistency. The NDT workflow:

1. Partition the target point cloud into voxels at a specified resolution.
2. Compute Gaussian distributions for each voxel. Let μ_k and Σ_k denote the mean and covariance of the k -th voxel.
3. During registration, determine which voxel each transformed point occupies, then establish residuals between the point and the voxel's μ_k, Σ_k .
4. Iteratively refine the pose estimate using Gauss-Newton or Levenberg-Marquardt.

The key step is #3. For a source point \mathbf{q}_i transformed by \mathbf{R}, \mathbf{t} into a voxel with statistics μ_i, Σ_i^5 , we define the voxel residual as:

$$\mathbf{e}_i = \mathbf{R}\mathbf{q}_i + \mathbf{t} - \mu_i, \quad (7.11)$$

The optimal \mathbf{R}, \mathbf{t} solve a weighted least-squares problem:

$$(\mathbf{R}, \mathbf{t})^* = \arg \min_{\mathbf{R}, \mathbf{t}} \sum_i (\mathbf{e}_i^\top \Sigma_i^{-1} \mathbf{e}_i). \quad (7.12)$$

From least-squares theory, this equivalently maximizes the probability of each point belonging to its voxel's distribution - a maximum likelihood estimation (MLE)⁶:

$$(\mathbf{R}, \mathbf{t})^* = \arg \max_{\mathbf{R}, \mathbf{t}} \prod_i P(\mathbf{R}\mathbf{q}_i + \mathbf{t}) \quad (7.13)$$

Here, Σ_i^{-1} provides the weighting: tighter distributions demand closer alignment to μ_i , while dispersed distributions tolerate greater deviation.

⁵In practice, pose errors may place points in adjacent voxels. Most NDT implementations search neighboring voxels to improve convergence.

⁶Readers unfamiliar should consult Chapter 6 of *14 Lectures*.

The Gauss-Newton system becomes:

$$\sum_i (\mathbf{J}_i^\top \Sigma_i^{-1} \mathbf{J}_i) \Delta \mathbf{x} = - \sum_i \mathbf{J}_i^\top \Sigma_i^{-1} \mathbf{e}_i, \quad (7.14)$$

where $\Delta \mathbf{x}$ is the increment and \mathbf{J}_i the Jacobian:

$$\frac{\partial \mathbf{e}_i}{\partial \mathbf{R}} = -\mathbf{R} \mathbf{q}_i^\wedge, \quad \frac{\partial \mathbf{e}_i}{\partial \mathbf{t}} = \mathbf{I}. \quad (7.15)$$

Levenberg-Marquardt can enhance robustness.

Note: This derivation differs significantly from [74], aligning more closely with early 2D NDT [135]. The core principles remain identical - the original formulation used trigonometric expressions before manifold optimization became prevalent in SLAM. We omit the uniform distribution component for clarity, as most implementations incorporate practical improvements [136, 161]. Similarly, other algorithms in this book adapt original formulations to maintain consistent notation.

Implementation

The NDT implementation consists of two main components: voxel construction and registration. The NDT class internally maintains these voxels and their indices. The voxel construction process is similar to the grid method described in Chapter 5, but without requiring nearest neighbor search implementation. The core code is as follows:

Listing 7.6: src/ch7/ndt_3d.cc

```

1 class Ndt3d {
2     public:
3         enum class NearbyType {
4             CENTER, // Consider only center
5             NEARBY6, // Up/down/left/right/front/back
6         };
7
8         using KeyType = Eigen::Matrix<int, 3, 1>; // Voxel index
9         struct VoxelData {
10             VoxelData() {}
11             VoxelData(size_t id) { idx_.emplace_back(id); }
12
13             std::vector<size_t> idx_; // Point indices in cloud
14             Vec3d mu_ = Vec3d::Zero(); // Mean
15             Mat3d sigma_ = Mat3d::Zero(); // Covariance
16             Mat3d info_ = Mat3d::Zero(); // Inverse covariance
17         };
18
19     private:
20         void BuildVoxels();
21
22         /// Generate nearby grids based on neighbor type
23         void GenerateNearbyGrids();
24
25         CloudPtr target_ = nullptr;
26         CloudPtr source_ = nullptr;
27
28         Vec3d target_center_ = Vec3d::Zero();
29         Vec3d source_center_ = Vec3d::Zero();
30         Options options_;
31
32         std::unordered_map<KeyType, VoxelData, hash_vec<3>> grids_; // Voxel data
33         std::vector<KeyType> nearby_grids_; // Nearby voxels
34     };
35
36     void Ndt3d::BuildVoxels() {
37         assert(target_ != nullptr);
38         assert(target_->empty() == false);
39 }
```

```

40  /// Assign points to voxels
41  std::vector<size_t> index(target_->size());
42  std::iota(index.begin(), index.end(), 0);
43
44  std::for_each(index.begin(), index.end(), [this](const size_t& idx) {
45    auto pt = ToVec3d(target_->points[idx]);
46    auto key = (pt * options_.inv_voxel_size_).cast<int>();
47    grids_[key].idx_.emplace_back(idx);
48  });
49
50  /// Compute mean and covariance for each voxel
51  std::for_each(std::execution::par_unseq, grids_.begin(), grids_.end(), [this](auto&
52    v) {
53    if (v.second.idx_.size() > options_.min_pts_in_voxel_) {
54      // Minimum 3 points required
55      math::ComputeMeanAndCov(v.second.idx_, v.second.mu_, v.second.sigma_,
56      [this](const size_t& idx) { return ToVec3d(target_->points[idx]); });
57      // Regularize covariance matrix
58      v.second.info_ = (v.second.sigma_ + Mat3d::Identity() * 1e-3).inverse();
59    }
60  });
61
62  /// Remove voxels with insufficient points
63  for (auto iter = grids_.begin(); iter != grids_.end();) {
64    if (iter->second.idx_.size() > options_.min_pts_in_voxel_) {
65      iter++;
66    } else {
67      iter = grids_.erase(iter);
68    }
69  }
70  LOG(INFO) << "voxels: " << grids_.size();
71 }
72
73 void Ndt3d::GenerateNearbyGrids() {
74   nearby_grids_.clear();
75   if (options_.nearby_type_ == NearbyType::CENTER) {
76     nearby_grids_.emplace_back(KeyType::Zero());
77   } else if (options_.nearby_type_ == NearbyType::NEARBY6) {
78     nearby_grids_ = {KeyType(0, 0, 0), KeyType(-1, 0, 0), KeyType(1, 0, 0),
79                      KeyType(0, 1, 0), KeyType(0, -1, 0), KeyType(0, 0, -1),
80                      KeyType(0, 0, 1)};
81 }
82 }
```

Voxel Neighbor Selection

Users can configure the voxel neighbor search strategy. We can either use only the center voxel or include its six adjacent voxels as neighbors. For any voxel containing at least three points, we compute its mean and covariance.

Note that in simulation scenarios, multiple points may share identical coordinates along certain axes, causing zero values in the covariance matrix diagonal. Similarly, when points within a voxel approximate a plane or line, the covariance matrix may become singular. Therefore, we add a small regularization term (1e-3) to the diagonal before matrix inversion to ensure numerical stability.

The NDT registration implementation:

Listing 7.7: src/ch7/ndt_3d.cc

```

1 bool Ndt3d::AlignNdt(SE3& init_pose) {
2   LOG(INFO) << "Aligning with NDT";
3   assert(!grids_.empty());
4
5   SE3 pose = init_pose;
6   if (options_.remove_centroid_) {
7     pose.translation() = target_center_ - source_center_; // Initialize translation
8     LOG(INFO) << "Initial translation: " << pose.translation().transpose();
```

```

9  }
10
11 // Precompute point indices
12 int num_residual_per_point = options_.nearby_type_ == NearbyType::NEARBY6 ? 7 : 1;
13 std::vector<int> index(source_->size());
14 std::iota(index.begin(), index.end(), 0);
15
16 for (int iter = 0; iter < options_.max_iteration_; ++iter) {
17     // Parallel computation buffers
18     std::vector<bool> effect_pts(index.size() * num_residual_per_point, false);
19     std::vector<Eigen::Matrix<double, 3, 6>> jacobians(effect_pts.size());
20     std::vector<Vec3d> errors(effect_pts.size());
21     std::vector<Mat3d> infos(effect_pts.size());
22
23     // Parallel residual computation
24     std::for_each(std::execution::par_unseq, index.begin(), index.end(), [&](int idx)
25     {
26         Vec3d qs = pose * ToVec3d(source_->points[idx]);
27         Vec3i base_key = (qs * options_.inv_voxel_size_).cast<int>();
28
29         for (int i = 0; i < nearby_grids_.size(); ++i) {
30             Vec3i key = base_key + nearby_grids_[i];
31             int buffer_idx = idx * num_residual_per_point + i;
32
33             if (auto it = grids_.find(key); it != grids_.end()) {
34                 const auto& v = it->second;
35                 Vec3d e = qs - v.mu_;
36                 double res = e.transpose() * v.info_ * e;
37
38                 if (std::isnan(res) || res > options_.res_outlier_th_) {
39                     continue; // Reject outliers
40                 }
41
42                 // Build residual and Jacobian
43                 jacobians[buffer_idx].block<3,3>(0,0) = -pose.so3().matrix() * S03::hat(q);
44                 jacobians[buffer_idx].block<3,3>(0,3) = Mat3d::Identity();
45                 errors[buffer_idx] = e;
46                 infos[buffer_idx] = v.info_;
47                 effect_pts[buffer_idx] = true;
48             }
49         });
50
51     // Accumulate Hessian and error
52     Mat6d H = Mat6d::Zero();
53     Vec6d err = Vec6d::Zero();
54     double total_res = 0;
55     int effective_num = 0;
56
57     for (int i = 0; i < effect_pts.size(); ++i) {
58         if (!effect_pts[i]) continue;
59
60         H += jacobians[i].transpose() * infos[i] * jacobians[i];
61         err -= jacobians[i].transpose() * infos[i] * errors[i];
62         total_res += errors[i].transpose() * infos[i] * errors[i];
63         effective_num++;
64     }
65
66     if (effective_num < options_.min_effective_pts_) {
67         LOG(WARNING) << "Insufficient correspondences: " << effective_num;
68         return false;
69     }
70
71     // Update pose
72     Vec6d dx = H.ldlt().solve(err);
73     pose.so3() = pose.so3() * S03::exp(dx.head<3>());
74     pose.translation() += dx.tail<3>();
75
76     LOG(INFO) << "Iter " << iter << " | Res: " << total_res
77     << " | Eff: " << effective_num
78     << " | Avg: " << total_res/effective_num
79     << " | dx: " << dx.norm();
80
81     if (gt_set_) {

```

```

82     LOG(INFO) << "Pose error: " << (gt_pose_.inverse() * pose).log().norm();
83 }
84
85 if (dx.norm() < options_.eps_) break;
86 }
87
88 init_pose = pose;
89 return true;
90 }
```

Key implementation notes:

- Supports configurable neighbor search (center-only or 6-adjacent)
- Parallelized residual computation using C++17 execution policies
- Regularized covariance matrices (1e-3) for numerical stability
- Outlier rejection based on Mahalanobis distance threshold
- Efficient LDLT decomposition for solving the linear system

The overall workflow remains consistent with our previous discussion, though readers may incorporate additional convergence checks as needed. The inclusion of covariance matrices imposes constraints during Gauss-Newton iterations, accelerating convergence⁷. When executing the test program, readers will observe significantly faster performance from this NDT implementation:

Listing 7.8: Terminal output:

```

1 I0130 17:48:05.746295 88880 ndt_3d.cc:69] Aligning with NDT
2 I0130 17:48:05.746309 88880 ndt_3d.cc:75] Initial translation: -0.0565748 -0.122053
   0.0288451
3 I0130 17:48:05.748436 88880 ndt_3d.cc:168] Iter 0 | Res: 136282 | Eff: 44120 | Avg:
   3.0889 | dx: 0.0323 [0.0212, -0.0005, -0.0209, -0.0078, 0.0069, -0.0071]
4 I0130 17:48:05.748458 88880 ndt_3d.cc:178] Pose error: 0.065853
5 [...]
6 I0130 17:48:05.754640 88880 ndt_3d.cc:182] Converged, dx = [0.0069, -0.0035, -0.0049,
   -0.0005, 0.0021, -0.0031]
7 E0130 17:48:05.754648 88880 test_icp.cc:121] NDT success. Pose: [0.0267, -0.0049,
   -0.0222, 0.9994], [-0.0713, -0.1045, 0.0086]
8 I0130 17:48:05.758247 88880 sys_utils.h:32] NDT avg time: 20.9043ms (1 runs)
9 [...]
10 I0130 17:44:52.928387 88363 test_icp.cc:167] PCL NDT pose error: 0.201841
11 I0130 17:44:52.928720 88363 sys_utils.h:32] PCL NDT avg time: 250.998ms (1 runs)
```

Performance benchmarks show:

- **Center-voxel mode:** 10x faster than PCL implementation
- **6-neighbor mode:** 3-4x faster than PCL
- **Dense point clouds:** Prefer center-voxel for efficiency
- **Sparse point clouds:** Include neighboring voxels for robustness

NDT's elegant formulation and general applicability (unlike ICP's need for geometric assumptions) have established it as a standard benchmark for registration algorithms. However, two fundamental challenges persist:

⁷Similar covariance constraints could be applied in point-to-plane ICP, e.g., prioritizing errors along plane normals rather than uniform optimization.

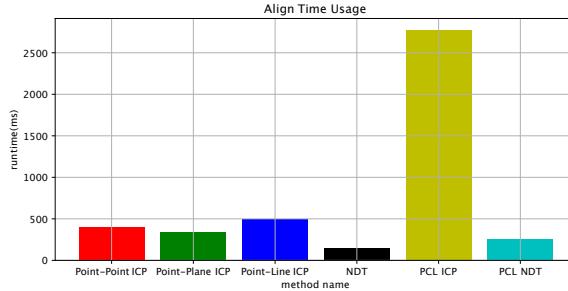


Figure 7-5: Computation time comparison across registration methods

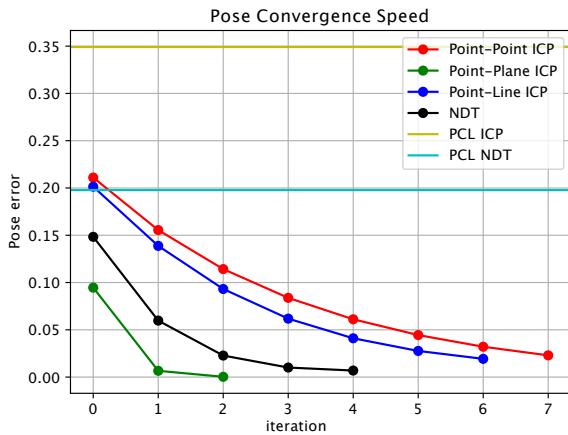


Figure 7-6: Pose convergence curves of different registration methods

- **Initial guess dependency:** Like all registration methods, poor initialization may place points in incorrect voxels, leading to misalignment.
- **Voxel size sensitivity:** The algorithm's discretization introduces a critical hyperparameter:
 - Oversized voxels lose geometric fidelity in dense clouds
 - Undersized voxels contain insufficient points for meaningful statistics

Practical recommendation: Always conduct voxel size analysis during implementation, as this parameter profoundly impacts registration accuracy while lacking universal optimal values. The appropriate scale depends entirely on the scene's spatial characteristics and point density.

7.3.4 Comparative Analysis of Registration Methods vs. PCL Implementations

Using the EPFL dataset provided in this chapter, we evaluate the temporal efficiency and accuracy metrics of various registration methods. Our current implementation facilitates horizontal comparison of:

1. Our point-to-point ICP (Sec. 7.2.1);

2. Our point-to-plane ICP (Sec. 7.2.2);
3. Our point-to-line ICP (Sec. 7.2.3);
4. Our NDT implementation (Sec. 7.3);
5. PCL's ICP implementation;
6. PCL's NDT implementation.

Figures 7-5 and 7-6 present the computational time and error convergence profiles respectively. Since PCL implementations only output final poses without intermediate iterations, their errors are represented as horizontal lines. Key observations from our benchmark:

- **Point-to-plane ICP** achieves the highest pose accuracy with fastest convergence for small models like EPFL datasets
- **NDT** demonstrates competitive speed but slightly inferior final accuracy compared to point-to-plane ICP
- **Basic point-to-point ICP** shows relatively poor performance, serving mainly as a baseline

Further subdivision by nearest neighbor methods (K-dTree vs voxel-based) reveals additional performance variations. We encourage readers to conduct extended comparisons, though our experiments already establish a framework for evaluating registration algorithms.

Critical Note: Small model performance doesn't directly translate to large-scale scenarios. For compact models with smooth surfaces (like EPFL), point-to-plane ICP benefits from rich geometric constraints. NDT's effectiveness becomes voxel-size dependent - overly coarse voxels may prevent optimal convergence. However, autonomous driving scenarios present fundamentally different characteristics:

- Sparse point clouds with large structures
- Higher noise levels
- Reduced surface completeness

Algorithm selection must therefore consider specific application contexts rather than relying on singular dataset performance. The optimal method varies significantly between precision small-object registration and large-scale noisy environment mapping.

7.4 Direct Method LiDAR Odometry

With registration methods like ICP and NDT, we can align multiple point clouds to form local maps and ultimately build a LiDAR odometry module. The simplest approach is to consecutively apply scan-to-scan matching between consecutive frames. In 3D SLAM, point clouds can be easily merged - we can compose a local map from recent scans and register the current frame against this map. This feature-free approach is called direct LiDAR odometry. Depending on the registration method used, we can implement various direct odometry solutions based on different ICP or NDT variants, as shown in the left portion of Fig. 7-7.

For higher precision with slightly more computation, we can enhance the NDT implementation at the voxel level. Instead of concatenating historical point clouds into a local

map, we can incrementally update the Gaussian distributions within NDT voxels using newly registered points. This incremental NDT approach avoids reconstructing the entire NDT structure or KD-trees for each frame, enabling highly efficient implementation. Below we implement both approaches and compare their computational performance.

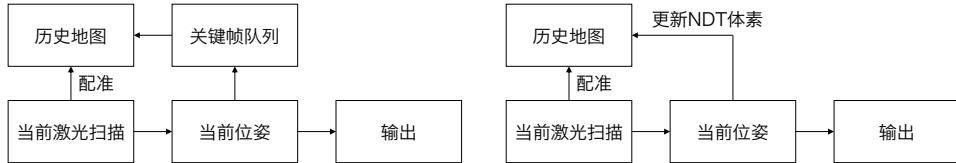


Figure 7-7: Two implementation approaches for LiDAR odometry.

7.4.1 Building LiDAR Odometry with NDT

We first implement the LiDAR odometry following the initial approach. This odometry maintains a local map composed of multiple scans, registers them collectively, and uses the NDT alignment from Section 7.3.3 to estimate the current frame's pose. The core implementation is as follows:

Listing 7.9: src/ch7/direct_ndt_lo.cc

```

1 class DirectNdtLo {
2     public:
3         struct Options {
4             Options() {}
5             double kf_distance_ = 0.5;           // Keyframe distance threshold
6             double kf_angle_deg_ = 30;          // Rotation threshold
7             int num_kfs_in_local_map_ = 30;      // Local map keyframe capacity
8             bool use_pcl_ndt_ = true;           // Use our NDT or PCL's NDT
9             bool display_realtime_cloud_ = true; // Enable real-time visualization
10
11         Ndt3d::Options ndt3d_options_;       // NDT3D configuration
12     };
13
14     DirectNdtLo(Options options = Options()) : options_(options) {
15         if (options_.display_realtime_cloud_) {
16             viewer_ = std::make_shared<PCLMapViewer>(0.5);
17         }
18
19         ndt_ = Ndt3d(options_.ndt3d_options_);
20
21         // Configure PCL NDT
22         ndt_pcl_.setResolution(1.0);
23         ndt_pcl_.setStepSize(0.1);
24         ndt_pcl_.setTransformationEpsilon(0.01);
25     }
26
27     /**
28     * Process a new point cloud scan
29     * @param scan Current frame point cloud
30     * @param pose Estimated pose (output)
31     */
32     void AddCloud(CloudPtr scan, SE3& pose);
33
34     private:
35         /// Align scan with local map
36         SE3 AlignWithLocalMap(CloudPtr scan);
37
38         /// Keyframe decision logic
39         bool IsKeyframe(const SE3& current_pose);
40
41     private:
42         Options options_;
43         CloudPtr local_map_ = nullptr;
44         std::deque<CloudPtr> scans_in_local_map_;

```

```

45     std::vector<SE3> estimated_poses_; // Estimated trajectory
46     SE3 last_kf_pose_; // Last keyframe pose
47
48     pcl::NormalDistributionsTransform<PointType, PointType> ndt_pcl_;
49     Ndt3d ndt_;
50 };
51
52 void DirectNDTLO::AddCloud(CloudPtr scan, SE3& pose) {
53     if (local_map_ == nullptr) {
54         // Initialize with first frame
55         local_map_.reset(new PointCloudType);
56         *local_map_ += *scan;
57         pose = SE3();
58         last_kf_pose_ = pose;
59
60         if (options_.use_pcl_ndt_) {
61             ndt_pcl_.setInputTarget(local_map_);
62         } else {
63             ndt_.SetTarget(local_map_);
64         }
65         return;
66     }
67
68     // Align scan with local map
69     pose = AlignWithLocalMap(scan);
70     CloudPtr scan_world(new PointCloudType);
71     pcl::transformPointCloud(*scan, *scan_world, pose.matrix().cast<float>());
72
73     if (IsKeyframe(pose)) {
74         last_kf_pose_ = pose;
75
76         // Update local map
77         scans_in_local_map_.emplace_back(scan_world);
78         if (scans_in_local_map_.size() > options_.num_kfs_in_local_map_) {
79             scans_in_local_map_.pop_front();
80         }
81
82         local_map_.reset(new PointCloudType);
83         for (auto& scan : scans_in_local_map_) {
84             *local_map_ += *scan;
85         }
86
87         if (options_.use_pcl_ndt_) {
88             ndt_pcl_.setInputTarget(local_map_);
89         } else {
90             ndt_.SetTarget(local_map_);
91         }
92     }
93 }
94
95 SE3 DirectNDTLO::AlignWithLocalMap(CloudPtr scan) {
96     if (options_.use_pcl_ndt_) {
97         ndt_pcl_.setInputSource(scan);
98     } else {
99         ndt_.SetSource(scan);
100    }
101
102    SE3 guess;
103    bool align_success = true;
104
105    if (estimated_poses_.size() < 2) {
106        // Initial alignment without motion model
107        if (options_.use_pcl_ndt_) {
108            pcl::PointCloud<PointType> output;
109            ndt_pcl_.align(output, guess.matrix().cast<float>());
110            guess = Mat4dToSE3(ndt_pcl_.getFinalTransformation().cast<double>());
111        } else {
112            align_success = ndt_.AlignNdt(guess);
113        }
114    } else {
115        // Motion model prediction from last two poses
116        SE3 T1 = estimated_poses_.back();
117        SE3 T2 = estimated_poses_[estimated_poses_.size() - 2];
118        guess = T1 * (T2.inverse() * T1);

```

```

119     if (options_.use_pcl_ndt_) {
120         pcl::PointCloud<PointType> output;
121         ndt_pcl_.align(output, guess.matrix().cast<float>());
122         guess = Mat4dToSE3(ndt_pcl_.getFinalTransformation().cast<double>());
123     } else {
124         align_success = ndt_.AlignNdt(guess);
125     }
126 }
127
128 LOG(INFO) << "Estimated pose - t: " << guess.translation().transpose()
129 << ", q: " << guess.so3().unit_quaternion().coeffs().transpose();
130
131 if (options_.use_pcl_ndt_) {
132     LOG(INFO) << "Transformation probability: " << ndt_pcl_.
133         getTransformationProbability();
134 }
135
136 estimated_poses_.emplace_back(guess);
137 return guess;
138 }
```

This basic odometry continuously aligns incoming scans against a local map using NDT registration. Users can select either PCL's NDT or our custom implementation. Keyframes are selected based on distance or rotation thresholds, with recent keyframes aggregated into a local map serving as the NDT target. For alignment initialization, we employ a motion model derived from the relative movement of the last two frames. The system provides real-time visualization through a PCL-based 3D viewer and saves the merged point cloud as a PCD file upon completion.

The test program for this section is shown below, where gflags can specify whether to use PCL's NDT implementation and the number of nearest neighbors for our NDT:

Listing 7.10: src/ch7/test/test_ndt_lo.cc

```

1 // This program demonstrates NDT-based LiDAR Odometry using ULHK dataset
2 // When using PCL NDT, it rebuilds the NDT tree
3 DEFINE_string(bag_path, "./dataset/sad/ulhk/test2.bag", "path to rosbag");
4 DEFINE_string(dataset_type, "ULHK", "NCLT/ULHK/KITTI/WXB_3D"); // Dataset type
5 DEFINE_bool(use_pcl_ndt, false, "use pcl ndt to align?");
6 DEFINE_bool(use_ndt_nearby_6, false, "use ndt nearby 6?");
7 DEFINE_bool(display_map, true, "display map?");
8
9 int main(int argc, char** argv) {
10     sad::RosbagIO rosbag_io(fLS::FLAGS_bag_path, sad::Str2DatasetType(
11         fFLAGS_dataset_type));
12
13     sad::DirectNDTLO::Options options;
14     options.use_pcl_ndt_ = fLB::FLAGS_use_pcl_ndt;
15     options.ndt3d_options_.nearby_type_ =
16         fFLAGS_use_ndt_nearby_6 ? sad::Ndt3d::NearbyType::NEARBY6 : sad::Ndt3d::NearbyType
17             ::CENTER;
18     options.display_realtime_cloud_ = fFLAGS_display_map;
19     sad::DirectNDTLO ndt_lo(options);
20
21     rosbag_io
22         .AddAssociateHandle([&ndt_lo](sensor_msgs::PointCloud2::Ptr msg) -> bool {
23             sad::common::Timer::Evaluate(
24                 [&]() {
25                     SE3 pose;
26                     ndt_lo.AddCloud(sad::VoxelCloud(sad::PointCloud2ToCloudPtr(msg)), pose);
27                 },
28                 "NDT registration");
29             return true;
30         })
31         .Go();
32
33     if (fFLAGS_display_map) {
34         // Save the generated map
35         ndt_lo.SaveMap("./data/ch7/map.pcd");
36     }
37 }
```

```

35     sad::common::Timer::PrintAll();
36     LOG(INFO) << "done.";
37
38     return 0;
39 }

```

To run this program, readers need to download two test datasets from ULHK, with paths configurable via gflags. Compile and run with:

Listing 7.11: Terminal command:

```
1 bin/test_ndt_lo
```

You should see results similar to Fig. 7-8 (colors may vary - book illustrations typically use white background for printing). Upon completion, the Timer class prints algorithm efficiency:

Listing 7.12: Terminal output:

```

1 I0131 10:51:49.840482 102707 timer.cc:16] >>> ===== Printing run time =====
2 I0131 10:51:49.840484 102707 timer.cc:18] > [ NDT registration ] average time usage:
3           36.349 ms , called times: 3178
4 I0131 10:51:49.840495 102707 timer.cc:23] >>> ===== Printing run time end =====
I0131 10:51:49.840497 102707 test_ndt_lo.cc:56] done.

```

With visualization disabled, our NDT odometry processes each frame in about 15ms (18ms when using 6 nearest neighbors), while PCL's NDT takes about 85ms. Readers can benchmark both implementations on their own machines. The number of nearest neighbors affects registration efficiency, but isn't the dominant factor in the full odometry pipeline - merging local maps and constructing voxels/KD-trees for these maps consume most computational resources.

Key reasons our NDT odometry outperforms PCL's implementation:

1. While NDT is voxel-based, PCL's implementation still requires KD-trees for nearest neighbor searches. The time spent building KD-trees for local maps during scan-to-map registration is non-negligible. Our implementation directly uses voxel neighbors for faster lookups.
2. We parallelize residual and Jacobian calculations during registration, significantly speeding up the process compared to PCL's serial implementation.

However, the current LO pipeline has room for improvement. For example, when adding keyframes, we currently rebuild the entire local map and reset all NDT voxels. While faster than rebuilding KD-trees, this could be optimized further by incrementally updating voxels with new scan data while automatically discarding outdated voxels through a time-based queue - an approach we call **Incremental NDT**. We'll implement this next and compare its performance with the current method.

7.4.2 Incremental NDT Odometry

Principles of Incremental Updates

Implementing incremental NDT odometry raises two key challenges: maintaining dynamically growing voxels and determining how to update Gaussian parameters within each voxel. We first address the problem of updating statistical estimates within individual voxels. Specifically, given an existing Gaussian distribution estimated from historical points in a voxel, how should we update the distribution parameters when new points are added? This derivation requires basic probability theory.

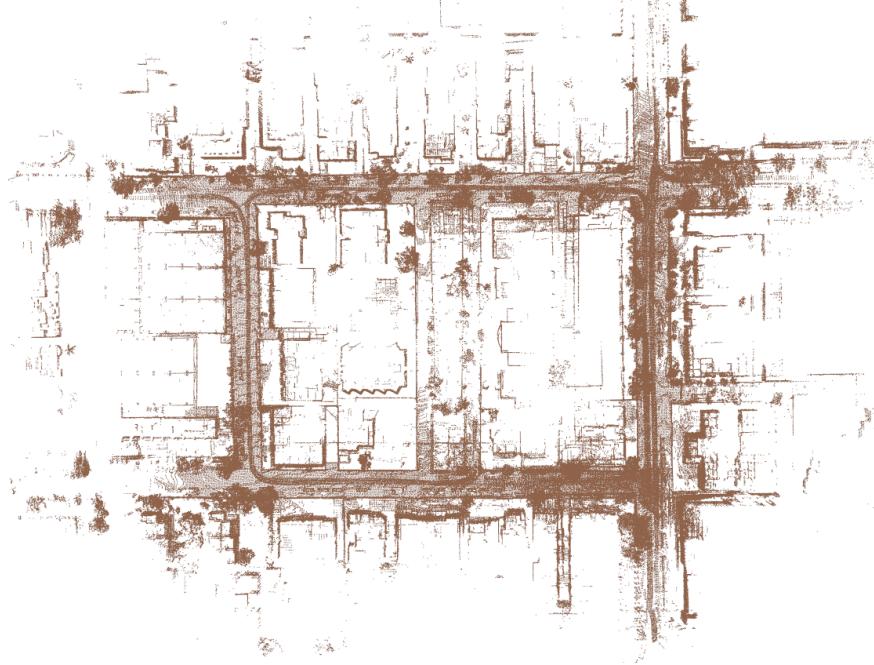


Figure 7-8: Point cloud map generated by our odometry, using ULHK dataset.

Incremental Update of Gaussian Distribution Consider a voxel containing m historical points with Gaussian parameters μ_H, Σ_H . When adding n new points with statistics μ_A, Σ_A , we derive the updated distribution μ, Σ . Let historical points be $\mathbf{x}_1, \dots, \mathbf{x}_m$ and new points $\mathbf{y}_1, \dots, \mathbf{y}_n$.

The mean update is straightforward:

$$\mu = \frac{\sum_{i=1}^m \mathbf{x}_i + \sum_{j=1}^n \mathbf{y}_j}{m+n} = \frac{m\mu_H + n\mu_A}{m+n}. \quad (7.16)$$

For covariance (ignoring Bessel's correction), we start with the sample covariance definition:

$$\Sigma = \frac{1}{m+n} \left(\sum_{i=1}^m (\mathbf{x}_i - \mu)(\mathbf{x}_i - \mu)^\top + \sum_{j=1}^n (\mathbf{y}_j - \mu)(\mathbf{y}_j - \mu)^\top \right). \quad (7.17)$$

Expanding the first term using $\mathbf{x}_i - \mu = (\mathbf{x}_i - \mu_H) + (\mu_H - \mu)$:

$$\begin{aligned} \sum_{i=1}^m (\mathbf{x}_i - \mu)(\mathbf{x}_i - \mu)^\top &= \sum_{i=1}^m [(\mathbf{x}_i - \mu_H) + (\mu_H - \mu)][(\mathbf{x}_i - \mu_H) + (\mu_H - \mu)]^\top \\ &= m\Sigma_H + \sum_{i=1}^m (\mathbf{x}_i - \mu_H)(\mu_H - \mu)^\top \end{aligned} \quad (7.18)$$

$$= m\Sigma_H + \sum_{i=1}^m (\mathbf{x}_i - \mu_H)(\mu_H - \mu)^\top \quad (7.19)$$

$$+ \sum_{i=1}^m (\mu_H - \mu)(\mathbf{x}_i - \mu_H)^\top + m(\mu_H - \mu)(\mu_H - \mu)^\top. \quad (7.20)$$

The cross terms vanish since:

$$\sum_{i=1}^m (\mathbf{x}_i - \boldsymbol{\mu}_H)(\boldsymbol{\mu}_H - \boldsymbol{\mu})^\top = \left(\sum_{i=1}^m \mathbf{x}_i - m\boldsymbol{\mu}_H \right) (\boldsymbol{\mu}_H - \boldsymbol{\mu})^\top = \mathbf{0}. \quad (7.21)$$

Thus we obtain:

$$\sum_{i=1}^m (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\top = m \left(\boldsymbol{\Sigma}_H + (\boldsymbol{\mu}_H - \boldsymbol{\mu})(\boldsymbol{\mu}_H - \boldsymbol{\mu})^\top \right). \quad (7.22)$$

Similarly for the new points:

$$\sum_{j=1}^n (\mathbf{y}_j - \boldsymbol{\mu})(\mathbf{y}_j - \boldsymbol{\mu})^\top = n \left(\boldsymbol{\Sigma}_A + (\boldsymbol{\mu}_A - \boldsymbol{\mu})(\boldsymbol{\mu}_A - \boldsymbol{\mu})^\top \right). \quad (7.23)$$

The final covariance update formula becomes:

$$\Sigma = \frac{m(\Sigma_H + \Delta\mu_H \Delta\mu_H^\top) + n(\Sigma_A + \Delta\mu_A \Delta\mu_A^\top)}{m+n} \quad (7.24)$$

where $\Delta\mu_H = \mu_H - \mu$ and $\Delta\mu_A = \mu_A - \mu$. This formula enables efficient incremental updates of NDT voxel statistics.

7.4.3 Incremental NDT Odometry

Incremental Voxel Maintenance

Beyond updating Gaussian distributions within voxels, the voxel structure itself must dynamically expand as the vehicle moves. However, for long-term odometry operation, we must limit the total number of voxels (e.g., maintaining around 100,000 voxels) by periodically removing older ones. This requires implementing a least recently used (LRU) cache mechanism. We maintain a queue where recently updated voxels move to the front, while voxels exceeding capacity are removed from the back.

The implementation follows:

Listing 7.13: src/ch7/ndt_inc.h

```
1 class IncNdt3d {
2     public:
3         enum class NearbyType {
4             CENTER, // Center voxel only
5             NEARBY6, // 6 adjacent voxels
6         };
7
8         using KeyType = Eigen::Matrix<int, 3, 1>; // Voxel index
9
10        /// Voxel data structure
11        struct VoxelData {
12             VoxelData() {}
13             VoxelData(const Vec3d& pt) {
14                 pts_.emplace_back(pt);
15                 num_pts_ = 1;
16             }
17
18             void AddPoint(const Vec3d& pt) {
19                 pts_.emplace_back(pt);
20                 if (!ndt_estimated_) {
21                     num_pts_++;
22                 }
23             }
24         };
25     }
```

```

24     std::vector<Vec3d> pts_;           // Buffered points
25     Vec3d mu_ = Vec3d::Zero();         // Mean
26     Mat3d sigma_ = Mat3d::Zero();       // Covariance
27     Mat3d info_ = Mat3d::Zero();        // Inverse covariance
28
29     bool ndt_estimated_ = false; // Whether Gaussian parameters are estimated
30     int num_pts_ = 0;             // Total accumulated points
31 };
32
33     /// Add point cloud to voxels
34     void AddCloud(CloudPtr cloud_world);
35
36     /// NDT alignment using Gauss-Newton
37     bool AlignNdt(SE3& init_pose);
38
39     private:
40     /// Update voxel statistics
41     void UpdateVoxel(VoxelData& v);
42
43     CloudPtr source_ = nullptr;
44     Options options_;
45
46     using KeyAndData = std::pair<KeyType, VoxelData>;
47     std::list<KeyAndData> data_; // LRU cache with actual data
48     std::unordered_map<KeyType, std::list<KeyAndData>::iterator, hash_vec<3>> grids_; // Hashmap for fast lookup
49     std::vector<KeyType> nearby_grids_; // Nearby voxel offsets
50 };
51

```

The IncNdt3d class maintains voxel data in a doubly-linked list (for LRU management) while using a hashmap for O(1) access. When new points are added, we update voxel statistics and maintain the cache:

Listing 7.14: src/ch7/ndt_inc.cc

```

1 void IncNdt3d::AddCloud(CloudPtr cloud_world) {
2     std::set<KeyType, less_vec<3>> active_voxels; // Track updated voxels
3     for (const auto& p : cloud_world->points) {
4         auto pt = ToVec3d(p);
5         auto key = (pt * options_.inv_voxel_size_).cast<int>();
6         auto iter = grids_.find(key);
7
8         if (iter == grids_.end()) {
9             // New voxel
10            data_.push_front({key, {pt}});
11            grids_.insert({key, data_.begin()});
12
13            if (data_.size() >= options_.capacity_) {
14                // Remove oldest voxel
15                grids_.erase(data_.back().first);
16                data_.pop_back();
17            }
18        } else {
19            // Existing voxel - add point and update LRU
20            iter->second->second.AddPoint(pt);
21            data_.splice(data_.begin(), data_, iter->second);
22            iter->second = data_.begin();
23        }
24
25        active_voxels.emplace(key);
26    }
27
28    // Parallel voxel updates
29    std::for_each(std::execution::par_unseq, active_voxels.begin(), active_voxels.end()
30    (),
31    [this](const auto& key) { UpdateVoxel(grids_[key]->second); });
32
33    void IncNdt3d::UpdateVoxel(VoxelData& v) {
34        if (v.ndt_estimated_ && v.num_pts_ > options_.max_pts_in_voxel_) {
35            return; // Skip if already well-estimated
36        }
37

```

```

37     if (!v.ndt_estimated_ && v.pts_.size() > options_.min_pts_in_voxel_) {
38         // Initial estimation for new voxel
39         math::ComputeMeanAndCov(v.pts_, v.mu_, v.sigma_, [](<const> Vec3d& p) { return p;
40             });
41         v.info_ = (v.sigma_ + Mat3d::Identity() * 1e-3).inverse();
42         v.ndt_estimated_ = true;
43         v.pts_.clear();
44     }
45     else if (v.ndt_estimated_ && v.pts_.size() > options_.min_pts_in_voxel_) {
46         // Incremental update for existing voxel
47         Vec3d cur_mu, new_mu;
48         Mat3d cur_var, new_var;
49         math::ComputeMeanAndCov(v.pts_, cur_mu, cur_var, [](<const> Vec3d& p) { return p;
50             });
51         math::UpdateMeanAndCov(v.num_pts_, v.pts_.size(), v.mu_, v.sigma_,
52         cur_mu, cur_var, new_mu, new_var);
53
54         v.mu_ = new_mu;
55         v.sigma_ = new_var;
56         v.num_pts_ += v.pts_.size();
57         v.pts_.clear();
58
59         // Regularize covariance
60         Eigen::JacobiSVD svd(v.sigma_, Eigen::ComputeFullU | Eigen::ComputeFullV);
61         Vec3d lambda = svd.singularValues();
62         if (lambda[1] < lambda[0] * 1e-3) lambda[1] = lambda[0] * 1e-3;
63         if (lambda[2] < lambda[0] * 1e-3) lambda[2] = lambda[0] * 1e-3;
64
65         Mat3d inv_lambda = Vec3d(1.0 / lambda[0], 1.0 / lambda[1], 1.0 / lambda[2]).asDiagonal
66         ();
67         v.info_ = svd.matrixV() * inv_lambda * svd.matrixU().transpose();
68     }
69 }
```

Key features:

- Each voxel tracks whether its Gaussian parameters have been estimated
- Points are buffered until reaching `min_pts_in_voxel` threshold
- Well-estimated voxels (with `max_pts_in_voxel`) skip updates
- Voxel updates are fully parallelizable
- Covariance regularization ensures numerical stability

The Gaussian update computation corresponds to the formulas presented earlier, with a straightforward implementation:

Listing 7.15: src/common/math_utils.h

```

1 template <typename S, int D>
2 void UpdateMeanAndCov(int hist_m, int curr_n, const Eigen::Matrix<S, D, 1>&
3     hist_mean,
4     const Eigen::Matrix<S, D, D>& hist_var, const Eigen::Matrix<S, D, 1>& curr_mean,
5     const Eigen::Matrix<S, D, D>& curr_var, Eigen::Matrix<S, D, 1>& new_mean,
6     Eigen::Matrix<S, D, D>& new_var) {
7     new_mean = (hist_m * hist_mean + curr_n * curr_mean) / (hist_m + curr_n);
8     new_var = (hist_m * (hist_var + (hist_mean - new_mean) * (hist_mean - new_mean).
9         transpose())
10    + curr_n * (curr_var + (curr_mean - new_mean) * (curr_mean - new_mean).transpose()
11    ))
12    / (hist_m + curr_n);
13 }
```

The LiDAR odometry algorithm (src/ch7/incremental_ndt_lo.cc) is simpler than previous implementations, as it only needs to process keyframes by continuously adding them to the internal NDT structure and calling the alignment function. The local map is now maintained within NDT, eliminating the need for explicit point cloud concatenation.

Listing 7.16: src/ch7/incremental_ndt_lo.cc

```

1 class IncrementalNDTLO {
2     public:
3         struct Options {
4             Options() {}
5             double kf_distance_ = 0.5;           // Keyframe distance threshold
6             double kf_angle_deg_ = 30;          // Rotation threshold
7             bool display_realtime_cloud_ = true; // Enable visualization
8             IncNdt3d::Options ndt3d_options_;   // NDT configuration
9         };
10
11    IncrementalNDTLO(Options options = Options()) : options_(options) {
12        if (options_.display_realtime_cloud_) {
13            viewer_ = std::make_shared<PCLMapViewer>(0.5);
14        }
15        ndt_ = IncNdt3d(options_.ndt3d_options_);
16    }
17
18    /**
19     * Process a new point cloud scan
20     * @param scan Current frame
21     * @param pose Estimated pose (output)
22     * @param use_guess Whether to use input pose as initial guess
23     */
24    void AddCloud(CloudPtr scan, SE3& pose, bool use_guess = false);
25
26    private:
27        Options options_;
28        bool first_frame_ = true;
29        std::vector<SE3> estimated_poses_; // Trajectory history
30        SE3 last_kf_pose_;                // Last keyframe pose
31        int cnt_frame_ = 0;
32
33        IncNdt3d ndt_;
34        std::shared_ptr<PCLMapViewer> viewer_ = nullptr;
35    };
36
37    void IncrementalNDTLO::AddCloud(CloudPtr scan, SE3& pose, bool use_guess) {
38        if (first_frame_) {
39            pose = SE3();
40            last_kf_pose_ = pose;
41            ndt_.AddCloud(scan);
42            first_frame_ = false;
43            return;
44        }
45
46        // Alignment against NDT-maintained local map
47        SE3 guess;
48        ndt_.SetSource(scan);
49        if (estimated_poses_.size() < 2) {
50            ndt_.AlignNdt(guess);
51        } else {
52            if (!use_guess) {
53                // Motion model prediction
54                SE3 T1 = estimated_poses_.back();
55                SE3 T2 = estimated_poses_[estimated_poses_.size() - 2];
56                guess = T1 * (T2.inverse() * T1);
57            } else {
58                guess = pose;
59            }
60            ndt_.AlignNdt(guess);
61        }
62
63        pose = guess;
64        estimated_poses_.emplace_back(pose);
65
66        if (IsKeyframe(pose)) {
67            last_kf_pose_ = pose;
68            cnt_frame_ = 0;
69            // Add to NDT's internal map
70            CloudPtr scan_world(new PointCloudType);
71            pcl::transformPointCloud(*scan, *scan_world, guess.matrix().cast<float>());
72            ndt_.AddCloud(scan_world);
73        }
}

```

```

74
75     if (viewer_) {
76         viewer_->SetPoseAndCloud(pose, scan_world);
77     }
78     cnt_frame_++;
79 }
```

The odometry mainly handles state flags and counters, with minimal algorithmic complexity. Readers can test using `test_inc_ndt_lo` with parameters identical to Section 7.4.1. The incremental NDT output (Fig. 7-9) appears similar to Fig. 7-8 in top view but shows noticeable drift in side view - an unavoidable **accumulated error** in odometry that we'll address later with loop closure detection and pose graph optimization.



Figure 7-9: Incremental NDT output (alternate view)

With visualization disabled, our incremental NDT processes each frame in 6ms, compared to 100ms for PCL's NDT - a $>10\times$ speedup. This demonstrates how understanding algorithmic principles enables custom optimizations beyond library limitations. Historical implementations often don't meet modern efficiency standards or anticipate future needs.

Both this and the next section present **pure LiDAR** odometry. While adequate for passenger vehicle datasets, they may struggle with rapidly rotating small vehicles. Later we'll enhance them with IMU and RTK measurements for robust performance.

7.5 Laser Odometry with Feature-based Methods

7.5.1 Feature Extraction

We refer to the odometry that **directly registers point clouds** as the direct method odometry. In contrast, autonomous driving often employs odometry that **first extracts features and then performs registration**, which we call feature-based odometry (or indirect method odometry) [16]. Feature-based odometry requires extracting simple features from the point cloud first, followed by registering only the feature points. Additionally, depending on the different properties of the feature points themselves, different registration methods can be applied to achieve higher accuracy [162]. Feature-based odometry implements different registration methods for different point cloud structures, offering better generalization compared to direct method odometry, which uniformly uses ICP or NDT. Among feature-based odometry systems, the LOAM series [2], including the original LOAM [2] and subsequent improved versions (LeGO-LOAM [163], A-LOAM⁸, F-LOAM⁹), are widely used open-source solu-

⁸<https://github.com/HKUST-Aerial-Robotics/A-LOAM>

⁹<https://github.com/wh200720041/floam>

tions in the autonomous driving industry and serve as the foundation for many subsequent LIO systems. However, the actual LOAM open-source code is quite complex, and we do not intend to explain it line by line in this book. Instead, we will explore the design philosophy of the LOAM series and provide a simple implementation based on its principles.

When discussing feature-based methods, the most pressing questions are: For a multi-beam LiDAR, what kind of features should we extract? And how should we use these features for point cloud registration? There are many types of point cloud features, such as PFH [164], FPFH [165], and various deep learning-based features. We must ask: What kind of features are meaningful for real-time registration? Point cloud features can be used not only for registration but also for database retrieval, comparison, and compression. In real-time SLAM, we have the following requirements for features:

1. Features should reflect the characteristics of the point cloud. For example, point clouds in autonomous driving are typically scanned by multi-beam LiDARs and are not as dense as RGB-D point clouds. Instead, they exhibit distinct **beam** properties, where each point belongs to a specific beam of the LiDAR. Instead of extracting features directly from the entire point cloud, we can extract features from the scan data of individual beams.
2. After feature extraction, it should be straightforward to perform geometric registration on these feature points. Registration methods such as ICP or NDT can be used.
3. Feature extraction should not consume excessive CPU or GPU resources, nor should it require specialized hardware.
4. In system design, we prefer the entire LO or SLAM system to use the same computational architecture, rather than having some parts run on the CPU and others on the GPU, which would lead to unnecessary data transfer and resource consumption.

Therefore, in industrial LO systems, people often use simple feature structures rather than complex ones based on statistical information or neural network computations. Most systems utilize **beam** information for feature extraction. Below, we introduce the feature extraction method used in LOAM-like systems.

7.5.2 Feature Extraction Based on LiDAR Beams

Multi-beam LiDARs inherently provide **beam** information. In addition to the positional data (x, y, z) , the point cloud obtained from a LiDAR includes the following extra information:

1. Which scan line a laser point originates from;
2. The sequential order of laser points on the same scan line;
3. Some drivers also output precise information such as polar coordinates and scan timestamps for each point.

Knowing which points belong to the same scan line and their temporal order is highly useful for designing LiDAR odometry algorithms. Most notably, it eliminates the need to search for nearest neighbors on the same scan line. Additionally, based on the sequential order of these points, we can compute their **curvature** and classify them accordingly. The curvature can be defined as the difference between a point and its neighboring points on the same beam. In the LOAM series of work, researchers proposed a straightforward approach: points with high curvature along the same beam are classified as corner points, while those

with low curvature are classified as planar points. To select the most distinct corner and planar points, one can sample the points with the highest and lowest curvature values, respectively. In multi-beam LiDAR scans, corner points typically lie on the surfaces of vertical objects or at the intersection of two planes, making them suitable for point-to-line ICP in the vertical direction. Meanwhile, planar points can be sampled across different beams and registered using point-to-plane ICP. The entire feature extraction and nearest-neighbor process is illustrated in Fig. 7-10.

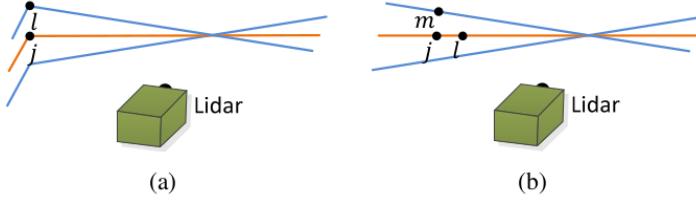


Figure 7-10: Using beam curvature to identify corner and planar points, followed by nearest-neighbor matching. The left side shows points l and j forming a line, while the right side shows points m , j , and l forming a plane. This figure is sourced from [2].

It is worth noting that this method of extracting corner and planar points can also be applied to 2D LiDARs, allowing the same approach to be used in 2D scan matching. Moreover, the extraction method is not limited to this single technique. For example, LeGO-LOAM [163] uses range images to extract ground points, corner points, and planar points, while MULLS employs PCA to extract features such as ground, facades, pillars, and horizontal lines [166]. However, most practical systems involve complex feature extraction processes, the details of which we will not cover here. Readers need only understand that extracting features to some extent helps optimize registration results.

That said, this beam-based method relies heavily on prior knowledge, such as the assumption that the point cloud consists of multiple scan lines and that the LiDAR is horizontally mounted. These implementations are often highly engineering-dependent, tailored to specific LiDAR scan angles and beam configurations. Such assumptions do not hold for solid-state LiDARs or RGB-D camera point clouds, limiting the applicability of this odometry approach.

7.5.3 Implementation of Feature Extraction

We will now implement the feature extraction component. Beam-based feature extraction requires knowledge of the LiDAR's beam information, which some LiDAR drivers provide. However, if the point cloud is converted to a standard PCL format, this information may be lost. While it is possible to compute the elevation angle for each point and infer its beam based on the LiDAR's elevation angle specifications, this complicates the code and introduces additional dependencies, increasing the learning curve for this book. Instead, we provide readers with preprocessed point clouds that include beam information, allowing them to focus solely on computing corner and planar points.

The feature extraction algorithm consists of the following steps:

1. Compute and categorize each point's beam. Since our point clouds already include this information, this step can be skipped.
2. Sequentially compute the curvature for each point on a beam.

- Divide the point cloud into sectors (we use six sectors in this implementation). Select points with the highest curvature as corner points, while the remaining points are classified as planar points.

Note that most feature extraction methods lack rigorous theoretical justification and are instead derived from empirical engineering practices. Readers are encouraged to modify the feature extraction pipeline as needed, adding or removing steps as appropriate. Below is the code implementation:

Listing 7.17: src/ch7/loam_like/feature_extraction.cc

```

1 void FeatureExtraction::Extract(FullCloudPtr pc_in, CloudPtr pc_out_edge, CloudPtr
2   pc_out_surf) {
3   int num_scans = 16;
4   std::vector<CloudPtr> scans_in_each_line; // Point clouds grouped by beam
5   for (int i = 0; i < num_scans; i++) {
6     scans_in_each_line.emplace_back(new PointCloudType);
7   }
8
9   for (auto &pt : pc_in->points) {
10    assert(pt.ring >= 0 && pt.ring < num_scans);
11    PointType p;
12    p.x = pt.x;
13    p.y = pt.y;
14    p.z = pt.z;
15    p.intensity = pt.intensity;
16
17    scans_in_each_line[pt.ring]->emplace_back(p);
18  }
19
20 // Compute curvature
21 for (int i = 0; i < num_scans; i++) {
22   if (scans_in_each_line[i]->points.size() < 131) {
23     continue;
24   }
25
26   std::vector<IdAndValue> cloud_curvature; // Curvature for each beam
27   int total_points = scans_in_each_line[i]->points.size() - 10;
28   for (int j = 5; j < (int)scans_in_each_line[i]->points.size() - 5; j++) {
29     // Leave a margin at both ends, sampling 10 neighboring points for averaging
30     double diffX = scans_in_each_line[i]->points[j - 5].x + scans_in_each_line[i]
31       ->points[j - 4].x +
32     scans_in_each_line[i]->points[j - 3].x + scans_in_each_line[i]->points[j - 2].
33       x +
34     scans_in_each_line[i]->points[j - 1].x - 10 * scans_in_each_line[i]->points[j].
35       x +
36     scans_in_each_line[i]->points[j + 1].x + scans_in_each_line[i]->points[j + 2].
37       x +
38     scans_in_each_line[i]->points[j + 3].x + scans_in_each_line[i]->points[j + 4].
39       x +
40     scans_in_each_line[i]->points[j + 5].x;
41     double diffY = scans_in_each_line[i]->points[j - 5].y + scans_in_each_line[i]
42       ->points[j - 4].y +
43     scans_in_each_line[i]->points[j - 3].y + scans_in_each_line[i]->points[j - 2].
44       y +
45     scans_in_each_line[i]->points[j - 1].y - 10 * scans_in_each_line[i]->points[j].
46       y +
47     scans_in_each_line[i]->points[j + 1].y + scans_in_each_line[i]->points[j + 2].
48       y +
49     scans_in_each_line[i]->points[j + 3].y + scans_in_each_line[i]->points[j + 4].
50       y +
51     scans_in_each_line[i]->points[j + 5].y;
52     double diffZ = scans_in_each_line[i]->points[j - 5].z + scans_in_each_line[i]
53       ->points[j - 4].z +
54     scans_in_each_line[i]->points[j - 3].z + scans_in_each_line[i]->points[j - 2].
55       z +
56     scans_in_each_line[i]->points[j - 1].z - 10 * scans_in_each_line[i]->points[j].
57       z +
58     scans_in_each_line[i]->points[j + 1].z + scans_in_each_line[i]->points[j + 2].
59       z +
60     scans_in_each_line[i]->points[j + 3].z + scans_in_each_line[i]->points[j + 4].
61       z +
62     scans_in_each_line[i]->points[j + 5].z;

```

```

46     scans_in_each_line[i]->points[j + 5].z;
47     IdAndValue distance(j, diffX * diffX + diffY * diffY + diffZ * diffZ);
48     cloud_curvature.push_back(distance);
49 }
50
51 // Select features per sector (360° divided into 6 sectors)
52 for (int j = 0; j < 6; j++) {
53     int sector_length = (int)(total_points / 6);
54     int sector_start = sector_length * j;
55     int sector_end = sector_length * (j + 1) - 1;
56     if (j == 5) {
57         sector_end = total_points - 1;
58     }
59
60     std::vector<IdAndValue> sub_cloud_curvature(cloud_curvature.begin() +
61                                                 sector_start,
62                                                 cloud_curvature.begin() + sector_end);
63
64     ExtractFromSector(scans_in_each_line[i], sub_cloud_curvature, pc_out_edge,
65                         pc_out_surf);
66 }
67
68 void FeatureExtraction::ExtractFromSector(const CloudPtr &pc_in, std::vector<
69     IdAndValue> &cloud_curvature, CloudPtr &pc_out_edge, CloudPtr &pc_out_surf) {
70     // Sort by curvature
71     std::sort(cloud_curvature.begin(), cloud_curvature.end(),
72               [] (const IdAndValue &a, const IdAndValue &b) { return a.value_ < b.value_; });
73
74     int largest_picked_num = 0;
75     int point_info_count = 0;
76
77     /// Select corner points with the highest curvature
78     std::vector<int> picked_points; // Mark selected corner points (neighboring
79     // points are excluded)
80     for (int i = cloud_curvature.size() - 1; i >= 0; i--) {
81         int ind = cloud_curvature[i].id_;
82         if (std::find(picked_points.begin(), picked_points.end(), ind) == picked_points.
83             end()) {
84             if (cloud_curvature[i].value_ <= 0.1) {
85                 break;
86             }
87
88             largest_picked_num++;
89             picked_points.push_back(ind);
90
91             if (largest_picked_num <= 20) {
92                 pc_out_edge->push_back(pc_in->points[ind]);
93                 point_info_count++;
94             } else {
95                 break;
96             }
97
98             for (int k = 1; k <= 5; k++) {
99                 double diffX = pc_in->points[ind + k].x - pc_in->points[ind + k - 1].x;
100                double diffY = pc_in->points[ind + k].y - pc_in->points[ind + k - 1].y;
101                double diffZ = pc_in->points[ind + k].z - pc_in->points[ind + k - 1].z;
102                if ((diffX * diffX + diffY * diffY + diffZ * diffZ) > 0.05) {
103                    break;
104                }
105                picked_points.push_back(ind + k);
106            }
107            for (int k = -1; k >= -5; k--) {
108                double diffX = pc_in->points[ind + k].x - pc_in->points[ind + k + 1].x;
109                double diffY = pc_in->points[ind + k].y - pc_in->points[ind + k + 1].y;
110                double diffZ = pc_in->points[ind + k].z - pc_in->points[ind + k + 1].z;
111                if ((diffX * diffX + diffY * diffY + diffZ * diffZ) > 0.05) {
112                    break;
113                }
114                picked_points.push_back(ind + k);
115            }
116        }
117    }
118 }
```

```

116  /// Select planar points with lower curvature
117  for (int i = 0; i <= (int)cloud_curvature.size() - 1; i++) {
118      int ind = cloud_curvature[i].id_;
119      if (std::find(picked_points.begin(), picked_points.end(), ind) == picked_points.
120          end()) {
121          pc_out_surf->push_back(pc_in->points[ind]);
122      }
123  }

```

The entire process aligns with the description above. We use the ‘test_feature_extraction’ program to evaluate the extraction results on a single point cloud. The program reads raw Velodyne packets and converts them into a point cloud while preserving beam information. The feature extraction algorithm then separates corner and planar points, storing them in two PCD files, as shown in Fig. 7-11. In structured indoor environments, corner and planar points are clearly distinguishable—corners lie on edges, while planar points lie on surfaces. In outdoor open areas, planar points perform well, but corner points are less reliable. Additionally, ground points are often distributed in circular patterns rather than straight lines, making curvature-based classification less effective. Some odometry algorithms opt to filter out ground points or model them separately. Similar approaches are used for ceilings in indoor environments.

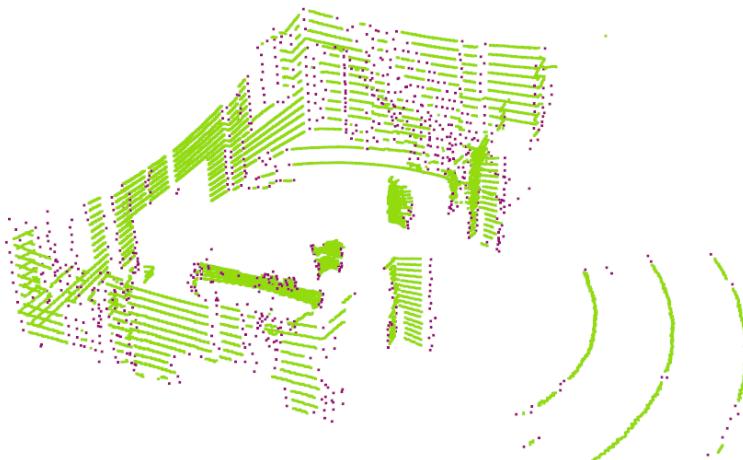


Figure 7-11: Corner and planar points extracted from a single scan. Red points: corners (or edges); green points: planar points.

Feature extraction typically consumes significant computational resources, making feature-based odometry slower than direct registration methods. However, the results are often more stable.

Now let’s implement the feature-based LiDAR odometry. The overall computational flow is similar to the previous NDT-based approach. We construct two local maps for corner and planar points respectively, then incorporate both ICP results into a single optimization problem. The core registration code is as follows:

Listing 7.18: src/ch7/loam-like/loam_like_odom.cc

```

1 class LoamLikeOdom {
2     public:
3         struct Options {
4             Options() {}

```

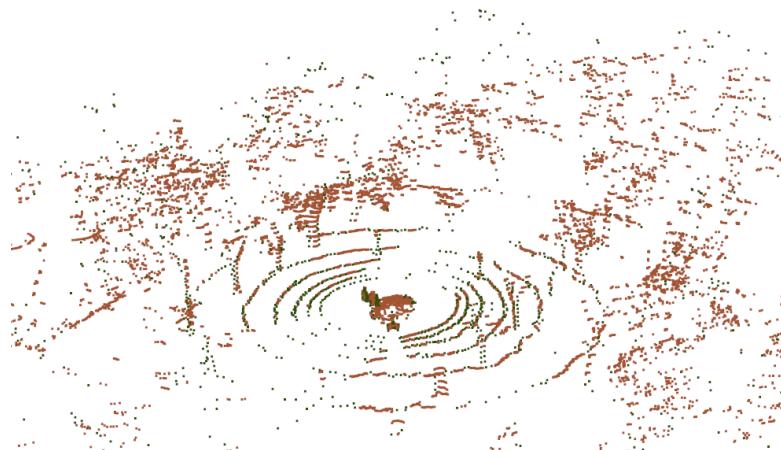


Figure 7-12: Corner and planar points in an outdoor scene. Planar points: reddish-brown; corner points: green.

```

5     int min_edge_pts_ = 20;           // Minimum edge points
6     int min_surf_pts_ = 20;           // Minimum planar points
7     double kf_distance_ = 1.0;        // Keyframe distance threshold
8     double kf_angle_deg_ = 15;        // Rotation angle threshold
9     int num_kfs_in_local_map_ = 30;   // Number of keyframes in local map
10    bool display_realtime_cloud_ = true; // Whether to display real-time cloud
11
12    // ICP parameters
13    int max_iteration_ = 5;           // Maximum iterations
14    double max_plane_distance_ = 0.05; // Plane distance threshold
15    double max_line_distance_ = 0.5;  // Line distance threshold
16    int min_effective_pts_ = 10;      // Minimum effective points
17    double eps_ = 1e-3;              // Convergence threshold
18
19    bool use_edge_points_ = true;     // Whether to use edge points
20    bool use_surf_points_ = true;     // Whether to use planar points
21
22};
23
24 explicit LoamLikeOdom(Options options = Options());
25
26 /**
27 * Process a point cloud, which will be divided into edge and planar points
28 * @param pcd_edge
29 * @param pcd_surf
30 */
31 void ProcessPointCloud(FullCloudPtr full_cloud);
32
33 private:
34     /// Align with local map
35     SE3 AlignWithLocalMap(CloudPtr edge, CloudPtr surf);
36
37     /// Determine if current frame is keyframe
38     bool IsKeyframe(const SE3& current_pose);
39
40     Options options_;
41
42     int cnt_frame_ = 0;
43     int last_kf_id_ = 0;
44
45     CloudPtr local_map_edge_ = nullptr, local_map_surf_ = nullptr; // Local maps
46     std::vector<SE3> estimated_poses_; // Estimated poses for trajectory
47     SE3 last_kf_pose_; // Last keyframe pose
48     std::deque<CloudPtr> edges_, surfs_; // Cached edge and planar points
49
50     CloudPtr global_map_ = nullptr; // Global map for saving
51

```

```

52 std::shared_ptr<FeatureExtraction> feature_extraction_ = nullptr;
53
54 std::shared_ptr<PCLMapViewer> viewer_ = nullptr;
55 KdTree kdTree_edge_, kdTree_surf_;
56 }
57
58 void LoamLikeOdom::ProcessPointCloud(FullCloudPtr cloud) {
59     LOG(INFO) << "processing frame " << cnt_frame_++;
60     // Step 1. Feature extraction
61     CloudPtr current_edge(new PointCloudType), current_surf(new PointCloudType);
62     feature_extraction_->Extract(cloud, current_edge, current_surf);
63
64     if (current_edge->size() < options_.min_edge_pts_ || current_surf->size() <
65         options_.min_surf_pts_) {
66         LOG(ERROR) << "not enough edge/surf pts: " << current_edge->size() << ", " <<
67             current_surf->size();
68         return;
69     }
70
71     LOG(INFO) << "edge: " << current_edge->size() << ", surf: " << current_surf->size()
72     ());
73
74     if (local_map_edge_ == nullptr || local_map_surf_ == nullptr) {
75         // Special handling for first frame
76         local_map_edge_ = current_edge;
77         local_map_surf_ = current_surf;
78
79         kdTree_edge_.BuildTree(local_map_edge_);
80         kdTree_surf_.BuildTree(local_map_surf_);
81
82         edges_.emplace_back(current_edge);
83         surfs_.emplace_back(current_surf);
84         return;
85     }
86
87     /// Align with local map
88     SE3 pose = AlignWithLocalMap(current_edge, current_surf);
89     CloudPtr scan_world(new PointCloudType);
90     pcl::transformPointCloud(*ConvertToCloud<FullPointType>(cloud), *scan_world, pose.
91     matrix());
92
93     CloudPtr edge_world(new PointCloudType), surf_world(new PointCloudType);
94     pcl::transformPointCloud(*current_edge, *edge_world, pose.matrix());
95     pcl::transformPointCloud(*current_surf, *surf_world, pose.matrix());
96
97     if (IsKeyframe(pose)) {
98         LOG(INFO) << "inserting keyframe";
99         last_kf_pose_ = pose;
100        last_kf_id_ = cnt_frame_;
101
102        // Rebuild local map
103        edges_.emplace_back(edge_world);
104        surfs_.emplace_back(surf_world);
105
106        if (edges_.size() > options_.num_kfs_in_local_map_) {
107            edges_.pop_front();
108        }
109        if (surfs_.size() > options_.num_kfs_in_local_map_) {
110            surfs_.pop_front();
111        }
112
113        local_map_surf_.reset(new PointCloudType);
114        local_map_edge_.reset(new PointCloudType);
115
116        for (auto& s : edges_) {
117            *local_map_edge_ += *s;
118        }
119        for (auto& s : surfs_) {
120            *local_map_surf_ += *s;
121        }
122
123        local_map_surf_ = VoxelCloud(local_map_surf_, 1.0);
124        local_map_edge_ = VoxelCloud(local_map_edge_, 1.0);
125    }

```

```

122 LOG(INFO) << "insert keyframe, surf pts: " << local_map_surf_->size()
123 << ", edge pts: " << local_map_edge_->size();
124
125 kdtree_surf_.BuildTree(local_map_surf_);
126 kdtree_edge_.BuildTree(local_map_edge_);
127
128 *global_map_ += *scan_world;
129 }
130
131 LOG(INFO) << "current pose: " << pose.translation().transpose() << ", "
132 << pose.so3().unit_quaternion().coeffs().transpose();
133
134 if (viewer_ != nullptr) {
135   viewer_->SetPoseAndCloud(pose, scan_world);
136 }
137
138 SE3 LoamLikeOdom::AlignWithLocalMap(CloudPtr edge, CloudPtr surf) {
139   // Custom ICP implementation needed here
140   SE3 pose;
141   if (estimated_poses_.size() >= 2) {
142     // Predict pose from last two poses
143     SE3 T1 = estimated_poses_[estimated_poses_.size() - 1];
144     SE3 T2 = estimated_poses_[estimated_poses_.size() - 2];
145     pose = T1 * (T2.inverse() * T1);
146   }
147
148   int edge_size = edge->size();
149   int surf_size = surf->size();
150
151   // Implement concurrent processing
152   for (int iter = 0; iter < options_.max_iteration_; ++iter) {
153     std::vector<bool> effect_surf(surf_size, false);
154     std::vector<Eigen::Matrix<double, 1, 6>> jacob_surf(surf_size); // 1D residual
155       for planes
156     std::vector<double> errors_surf(surf_size);
157
158     std::vector<bool> effect_edge(edge_size, false);
159     std::vector<Eigen::Matrix<double, 3, 6>> jacob_edge(edge_size); // 3D residual
160       for edges
161     std::vector<Vec3d> errors_edge(edge_size);
162
163     std::vector<int> index_surf(surf_size);
164     std::iota(index_surf.begin(), index_surf.end(), 0);
165     std::vector<int> index_edge(edge_size);
166     std::iota(index_edge.begin(), index_edge.end(), 0);
167
168     // Gauss-Newton iteration
169     // Nearest neighbor search for edge points
170     if (options_.use_edge_points_) {
171       std::for_each(std::execution::par_unseq, index_edge.begin(), index_edge.end(),
172           [&](int idx) {
173         Vec3d q = ToVec3d(edge->points[idx]);
174         Vec3d qs = pose * q;
175
176         // Find nearest neighbors
177         std::vector<int> nn_indices;
178
179         kdtree_edge_.GetClosestPoint(ToPointType(qs), nn_indices, 5);
180         effect_edge[idx] = false;
181
182         if (nn_indices.size() >= 3) {
183           std::vector<Vec3d> nn_eigen;
184           for (auto& n : nn_indices) {
185             nn_eigen.emplace_back(ToVec3d(local_map_edge_->points[n]));
186           }
187
188           // Point-to-line residual
189           Vec3d d, p0;
190           if (!math::FitLine(nn_eigen, p0, d, options_.max_line_distance_)) {
191             return;
192           }
193
194           Vec3d err = S03::hat(d) * (qs - p0);
195
196           if (err.squaredNorm() > 0.001) {
197             effect_surf[index_surf[n]] = true;
198             effect_edge[idx] = true;
199           }
200         }
201       });
202     }
203   }
204 }
```

```

193     if (err.norm() > options_.max_line_distance_) {
194         return;
195     }
196
197     effect_edge[idx] = true;
198
199     // Build residual
200     Eigen::Matrix<double>, 3, 6> J;
201     J.block<3, 3>(0, 0) = -S03::hat(d) * pose.so3().matrix() * S03::hat(q);
202     J.block<3, 3>(0, 3) = S03::hat(d);
203
204     jacob_edge[idx] = J;
205     errors_edge[idx] = err;
206 }
207 });
208
209 /// Nearest neighbor search for planar points
210 if (options_.use_surf_points_) {
211     std::for_each(std::execution::par_unseq, index_surf.begin(), index_surf.end(),
212     [&](int idx) {
213         Vec3d q = ToVec3d(surf->points[idx]);
214         Vec3d qs = pose * q;
215
216         // Find nearest neighbors
217         std::vector<int> nn_indices;
218
219         kdtree_surf_.GetClosestPoint(ToPointType(qs), nn_indices, 5);
220         effect_surf[idx] = false;
221
222         if (nn_indices.size() == 5) {
223             std::vector<Vec3d> nn_eigen;
224             for (auto& n : nn_indices) {
225                 nn_eigen.emplace_back(ToVec3d(local_map_surf_->points[n]));
226             }
227
228             // Point-to-plane residual
229             Vec4d n;
230             if (!math::FitPlane(nn_eigen, n)) {
231                 return;
232             }
233
234             double dis = n.head<3>().dot(qs) + n[3];
235             if (fabs(dis) > options_.max_plane_distance_) {
236                 return;
237             }
238
239             effect_surf[idx] = true;
240
241             // Build residual
242             Eigen::Matrix<double>, 1, 6> J;
243             J.block<1, 3>(0, 0) = -n.head<3>().transpose() * pose.so3().matrix() * S03
244             ::hat(q);
245             J.block<1, 3>(0, 3) = n.head<3>().transpose();
246
247             jacob_surf[idx] = J;
248             errors_surf[idx] = dis;
249         }
250     });
251
252     // Accumulate Hessian and error, compute dx
253     double total_res = 0;
254     int effective_num = 0;
255
256     Mat6d H = Mat6d::Zero();
257     Vec6d err = Vec6d::Zero();
258
259     for (const auto& idx : index_surf) {
260         if (effect_surf[idx]) {
261             H += jacob_surf[idx].transpose() * jacob_surf[idx];
262             err += -jacob_surf[idx].transpose() * errors_surf[idx];
263             effective_num++;
264             total_res += errors_surf[idx] * errors_surf[idx];
265         }
}

```

```

266 }
267
268 for (const auto& idx : index_edge) {
269     if (effect_edge[idx]) {
270         H += jacob_edge[idx].transpose() * jacob_edge[idx];
271         err += -jacob_edge[idx].transpose() * errors_edge[idx];
272         effective_num++;
273         total_res += errors_edge[idx].norm();
274     }
275 }
276
277 if (effective_num < options_.min_effective_pts_) {
278     LOG(WARNING) << "effective num too small: " << effective_num;
279     return pose;
280 }
281
282 Vec6d dx = H.inverse() * err;
283 pose.so3() = pose.so3() * SO3::exp(dx.head<3>());
284 pose.translation() += dx.tail<3>();
285
286 // Update
287 LOG(INFO) << "iter " << iter << " total res: " << total_res << ", eff: " <<
288             effective_num
289             << ", mean res: " << total_res / effective_num << ", dnx: " << dx.norm();
290
291 if (dx.norm() < options_.eps_) {
292     LOG(INFO) << "converged, dx = " << dx.transpose();
293     break;
294 }
295
296 estimated_poses_.emplace_back(pose);
297 return pose;
298 }
```

Since we need to handle both point-to-line and point-to-plane registration simultaneously, the code here will be somewhat lengthy. The workflow is identical to what we introduced in the ICP section, with the addition of local map maintenance procedures. Readers should compile and run the ‘test_loam_odom’ program to execute this odometry:

Listing 7.19: Terminal input:

```
bin/test_loam_odom
```

Figure 7-13 shows the mapping results from this section’s odometry program, where readers can clearly observe the structural shapes of various objects.

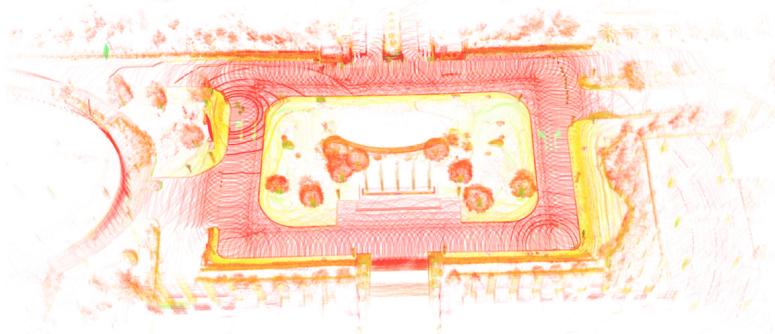


Figure 7-13: Point cloud map generated by the LOAM-like odometry algorithm in this section

In principle, feature-based algorithms classify raw point clouds into simple features and then construct different optimization problems for different feature types. The rationale be-

hind extracting these features and using different ICP methods for corner and planar points remains largely based on empirical debugging experience. In practical testing, corner and planar points cannot always be perfectly classified—they represent approximate results. Many points classified as corners or planes may actually lie on trees or bushes, where point-to-plane ICP might not reflect true physical meaning. Therefore, feature-based point cloud registration methods are primarily designed based on intuitive experience. They may perform better on certain datasets but fail in other scenarios. Many feature extraction processes also lack theoretical foundations, relying instead on empirical design. Systems like LeGO-LOAM and MULLS employ more sophisticated feature extraction methods, while their registration components remain consistent with classical algorithms.

We can further refine the classification by separating ground and ceiling points, categorizing point clouds into **corners**, **less-sharp corners**, **planar points**, **less-flat planar points**, etc., based on beam curvature. Such refinements can improve registration accuracy.

In terms of performance metrics, this odometry processes 16-beam LiDAR data at approximately 20 milliseconds per frame, slightly slower than the incremental NDT odometry discussed earlier. This is partly due to the added feature extraction step and the need to rebuild Kd-trees for nearest-neighbor searches in point-to-plane and point-to-line ICP. Some methods use voxels for ICP nearest-neighbor searches instead—readers are encouraged to implement this themselves.

In addition to the examples demonstrated in the book, readers can also run other provided sample datasets.

7.6 Loosely Coupled LIO Systems

By the previous section, we have systematically introduced LiDAR SLAM systems along with the principles of inertial navigation and integrated navigation. Now it's time to combine them together.

Many LiDAR odometry algorithms utilize IMU measurements to guide the initial guess for LiDAR matching. Such Lidar-inertial Odometry systems are referred to as LIO or LINS systems, which can be viewed as coupled LiDAR-IMU systems. Based on the coupling methodology, they can be classified into "tightly coupled" and "loosely coupled" systems. This section focuses on loosely coupled systems, while the theoretically more complex tightly coupled systems will be introduced in Chapter 8.

In a loosely coupled system, we still maintain a state estimator to compute the vehicle's state. The IMU and wheel odometry provide observations for inertial and velocity states, while **the pose output from LiDAR point cloud matching** serves as pose observations for the system. In this framework, we do not directly incorporate point cloud residuals (such as point-to-line, point-to-plane, or NDT residuals) into the state estimator. Instead, we fuse the **output pose** from point cloud registration with the estimator. This keeps the Kalman filter component and point cloud registration **decoupled** [167]. We can freely choose any odometry pose from previous chapters as the observation source. Conversely, point cloud registration can use the predicted output from the state estimator as the initial pose guess. However, if the point cloud structure degenerates or gets disturbed, the registration algorithm may become unstable, adversely affecting the state estimator.

In contrast, tightly coupled systems directly incorporate point cloud residual equations as observation models into the estimator. This approach essentially embeds an ICP or NDT process within the filter or optimization algorithm. Since ICP and NDT require iterative updates of their nearest neighbors, the corresponding filter should also support iterative updates. Having introduced various registration methods in this chapter and different inertial

navigation approaches in earlier chapters, we can flexibly combine them to create an LIO solution. Here we implement a relatively simple case: using the ESKF from Chapter 3 combined with the incremental NDT odometry from Section 7.4.3 to build a loosely coupled LIO system. This implementation is relatively simple and suitable for teaching purposes, and readers can replace the LiDAR odometry with ICP or feature-based methods.

7.6.1 Coordinate Frame Conventions

With the introduction of IMU, we now have three coordinate frames: world frame (W), IMU frame (I), and LiDAR frame (L). The IMU and LiDAR frames are not coincident, with a transformation between them denoted as \mathbf{T}_{IL} . This parameter varies across different datasets depending on sensor mounting. We record \mathbf{T}_{IL} for each dataset in configuration files and load it during program initialization.

For point cloud registration, several approaches exist. Since IMU motion models are derived in the IMU frame, we prefer to keep them unchanged without introducing additional variables. Therefore, we choose to transform LiDAR point clouds to the IMU frame using the extrinsic calibration. This makes the IMU frame the central reference for localization and mapping. The local maps in our LiDAR odometry are also maintained in the IMU frame. The transformation is straightforward: for a point \mathbf{p}_L measured by the LiDAR, its IMU frame coordinates become:

$$\mathbf{p}_I = \mathbf{T}_{IL}\mathbf{p}_L = \mathbf{R}_{IL}\mathbf{p}_L + \mathbf{t}_{IL}. \quad (7.25)$$

This approach avoids introducing additional extrinsic parameters into the observation model. An alternative method would keep the LiDAR odometry operating in the LiDAR frame while transforming the output pose to the IMU frame via \mathbf{T}_{IL} . However, that would introduce extra \mathbf{T}_{IL} terms in the observation equations, complicating the Jacobian matrices.

7.6.2 Motion and Observation Equations for Loosely Coupled LIO

Since the entire LIO operates in the IMU frame, the state variable motion equations remain consistent with (??), which we won't reiterate here. Meanwhile, the pose output from LiDAR odometry can be directly treated as observations of the state variables \mathbf{R}, \mathbf{p} . This process is essentially identical to the GNSS observations discussed in Chapter 3 equation (3.64). Abstractly written as $\mathbf{z} = \mathbf{h}(\mathbf{x})$, the translation observation is:

$$\mathbf{p}_{LO} = \mathbf{p}. \quad (7.26)$$

For rotation observations, to facilitate derivative computation, we express it as:

$$\delta\boldsymbol{\theta} = \text{Log}(\mathbf{R}^\top \mathbf{R}_{LO}). \quad (7.27)$$

Here, the observation Jacobian of odometry rotation with respect to error state $\delta\boldsymbol{\theta}$ should be \mathbf{I} , with the residual defined as:

$$\mathbf{r}_\mathbf{R} = -\text{Log}(\mathbf{R}^\top \mathbf{R}_{LO}). \quad (7.28)$$

At the implementation level, we'll use the same observation function as GNSS since they fundamentally represent direct observations of rotation and translation.

7.6.3 Data Preparation for Loosely Coupled System

The implementation of loosely coupled LIO consists of three main components: First, we need to synchronize IMU data with LiDAR data. LiDAR typically operates at 10Hz while

IMU runs at 100Hz. We aim to uniformly process the 10 IMU measurements between consecutive LiDAR scans. Second, we need to handle LiDAR motion compensation, which requires pose estimates during the LiDAR measurement period - these can be obtained from ESKF predictions of IMU data. Third, we should feed the predicted poses from ESKF to the odometry algorithm, then incorporate the registered poses back into ESKF. Additionally, to support various datasets in this book, we define an extra data conversion processor.

The CloudConvert class in this section handles conversion of different point cloud formats to PCL format, with the following main interface:

Listing 7.20: src/ch7/loosly_coupled_lio/cloud_convert.h

```

1  class CloudConvert {
2     public:
3         EIGEN_MAKE_ALIGNED_OPERATOR_NEW
4
5     enum class LidarType {
6         AVIA = 1, // DJI solid-state LiDAR
7         VELO32, // Velodyne 32-beam
8         OUST164, // Ouster 64-beam
9     };
10
11    CloudConvert() = default;
12    ~CloudConvert() = default;
13
14    /**
15     * Process Livox Avia point cloud
16     * @param msg
17     * @param pcl_out
18     */
19    void Process(const livox_ros_driver::CustomMsg::ConstPtr &msg, FullCloudPtr &
20                pcl_out);
21
22    /**
23     * Process sensor_msgs::PointCloud2
24     * @param msg
25     * @param pcl_out
26     */
27    void Process(const sensor_msgs::PointCloud2::ConstPtr &msg, FullCloudPtr &pcl_out)
28    ;
29
30    /// Load parameters from YAML
31    void LoadFromYAML(const std::string &yaml);
32
33    private:
34    void AviaHandler(const livox_ros_driver::CustomMsg::ConstPtr &msg);
35    void Oust64Handler(const sensor_msgs::PointCloud2::ConstPtr &msg);
36    void VelodyneHandler(const sensor_msgs::PointCloud2::ConstPtr &msg);
37
38    FullPointCloudType cloud_full_, cloud_out_; // Output point cloud
39    LidarType lidar_type_ = LidarType::AVIA; // LiDAR type
40    int point_filter_num_ = 1; // Point decimation ratio
41    int num_scans_ = 6; // Number of scan beams
42    float time_scale_ = 1e-3; // Time field scaling factor
43
44};

```

The implementation mainly standardizes point cloud interfaces. After processing by this class, subsequent modules only need to handle FullCloudPtr input. The class manages per-point timestamps, decimation ratios, etc., typically producing much smaller point clouds than raw sensor output. We omit the detailed implementation as it's relatively trivial.

Next, we synchronize IMU and point cloud data using the MessageSync class:

Listing 7.21: src/ch7/loosly_coupled_lio/measure_sync.h

```

1  /// IMU and LiDAR synchronization
2  struct MeasureGroup {
3      MeasureGroup() { this->lidar_.reset(new FullPointCloudType()); };
4
5      double lidar_begin_time_ = 0; // LiDAR packet start time

```

```

6   double lidar_end_time_ = 0;           // LiDAR packet end time
7   FullCloudPtr lidar_ = nullptr;        // LiDAR point cloud
8   std::deque<IMUPtr> imu_;            // IMU measurements between frames
9 };
10
11 /**
12 * Synchronize LiDAR and IMU data
13 */
14 class MessageSync {
15 public:
16   using Callback = std::function<void(const MeasureGroup &)>;
17
18   MessageSync(Callback cb) : callback_(cb), conv_(new CloudConvert) {}
19
20   /// Initialization
21   void Init(const std::string &yaml);
22
23   /// Process IMU data
24   void ProcessIMU(IMUPtr imu);
25
26 /**
27 * Process sensor_msgs::PointCloud2
28 * @param msg
29 */
30 void ProcessCloud(const sensor_msgs::PointCloud2::ConstPtr &msg);
31
32 void ProcessCloud(const livox_ros_driver::CustomMsg::ConstPtr &msg);
33
34 private:
35   /// Attempt synchronization, returns true if successful
36   bool Sync();
37
38   Callback callback_;                      // Callback after synchronization
39   std::shared_ptr<CloudConvert> conv_ = nullptr; // Point cloud converter
40   std::deque<FullCloudPtr> lidar_buffer_;       // LiDAR data buffer
41   std::deque<IMUPtr> imu_buffer_;              // IMU data buffer
42   double last_timestamp_imu_ = -1.0;           // Latest IMU timestamp
43   double last_timestamp_lidar_ = 0;             // Latest LiDAR timestamp
44   std::deque<double> time_buffer_;            // Time buffer
45   bool lidar_pushed_ = false;
46   MeasureGroup measures_;
47   double lidar_end_time_ = 0;
48 };
49
50 bool MessageSync::Sync() {
51   if (!lidar_buffer_.empty() || imu_buffer_.empty()) {
52     return false;
53   }
54
55   if (!lidar_pushed_) {
56     measures_.lidar_ = lidar_buffer_.front();
57     measures_.lidar_begin_time_ = time_buffer_.front();
58
59     lidar_end_time_ = measures_.lidar_begin_time_ +
60                       measures_.lidar->points.back().time / double(1000);
61
62     measures_.lidar_end_time_ = lidar_end_time_;
63     lidar_pushed_ = true;
64   }
65
66   if (last_timestamp_imu_ < lidar_end_time_) {
67     return false;
68   }
69
70   double imu_time = imu_buffer_.front()->timestamp_;
71   measures_.imu_.clear();
72   while (!imu_buffer_.empty() && (imu_time < lidar_end_time_)) {
73     imu_time = imu_buffer_.front()->timestamp_;
74     if (imu_time > lidar_end_time_) {
75       break;
76     }
77     measures_.imu_.push_back(imu_buffer_.front());
78     imu_buffer_.pop_front();
79   }
80 }
```

```

81     lidar_buffer_.pop_front();
82     time_buffer_.pop_front();
83     lidar_pushed_ = false;
84
85     if (callback_) {
86         callback_(measures_);
87     }
88
89     return true;
90 }
```

This class receives raw IMU and LiDAR messages from ROS packages, assembles them into a MeasureGroup through the Sync() function, and passes it to the callback function. Our subsequent loosely/tightly coupled algorithms only need to handle MeasureGroup objects without worrying about data preparation and synchronization implementation details.

7.6.4 Main Workflow of Loosely Coupled System

The main implementation of the loosely coupled system is shown below. It maintains an ESKF object and a MessageSync object to process synchronized point clouds and IMU data.

Listing 7.22: src/ch7/loosely_coupled_lios/loosly_lios.h

```

1  class LooselyLIO {
2  public:
3      EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
4      struct Options {
5          Options() {}
6          bool save_motion_undistortion_pcd_ = false; // Whether to save point clouds
7                      // before/after motion undistortion
8          bool with_ui_ = true;                         // Whether to enable UI
9      };
10
11     LooselyLIO(Options options);
12     ~LooselyLIO() = default;
13
14     /// Initialize from config file
15     bool Init(const std::string &config_yaml);
16
17     /// Point cloud callback
18     void PCLCallBack(const sensor_msgs::PointCloud2::ConstPtr &msg);
19     void LivoxPCLCallBack(const livox_ros_driver::CustomMsg::ConstPtr &msg);
20
21     /// IMU callback
22     void IMUCallBack(IMUPtr msg_in);
23
24     /// Terminate program and exit UI
25     void Finish();
26
27     private:
28         /// Process synchronized IMU and LiDAR data
29         void ProcessMeasurements(const MeasureGroup &meas);
30
31         /// Attempt IMU initialization
32         void TryInitIMU();
33
34         /// Predict states using IMU
35         /// Predicted states during this period will be stored in imu_states_
36         void Predict();
37
38         /// Motion undistortion for point cloud in measures_
39         void Undistort();
40
41         /// Perform registration and observation
42         void Align();
43
44     private:
45         /// modules
46         std::shared_ptr<MessageSync> sync_ = nullptr; // Message synchronizer
47         StaticIMUInit imu_init_;                      // IMU static initialization
```

```

47     std::shared_ptr<sad::IncrementalNDTLO> inc_ndt_lo_ = nullptr;
48
49     /// point clouds data
50     FullCloudPtr scan_undistort_{new FullPointCloudType()}; // scan after
51         undistortion
52     SE3 pose_of_lo_;
53
54     Options options_;
55
56     // flags
57     bool imu_need_init_ = true; // Whether IMU bias needs initialization
58     bool flag_first_scan_ = true; // Whether first LiDAR scan
59     int frame_num_ = 0; // Frame counter
60
61     // EKF data
62     MeasureGroup measures_; // Synchronized IMU and point cloud
63     std::vector<NavStated> imu_states_; // States during ESKF prediction
64     ESKFD eskf_; // ESKF
65     SE3 TIL_; // Extrinsic between LiDAR and IMU
66
67     std::shared_ptr<ui::PangolinWindow> ui_ = nullptr;
68 };

```

The core processing logic resides in the `ProcessMeasurements` function, which includes the following steps:

Listing 7.23: src/ch7/loosely_coupled_lio/loosly_lio.cc

```

1 void LooselyLIO::ProcessMeasurements(const MeasureGroup &meas) {
2     LOG(INFO) << "call meas, imu: " << meas.imu_.size() << ", lidar pts: " << meas.
3         lidar_->size();
4     measures_ = meas;
5
6     if (imu_need_init_) {
7         // Initialize IMU system
8         TryInitIMU();
9         return;
10    }
11
12    // State prediction using IMU data
13    Predict();
14
15    // Motion undistortion for point cloud
16    Undistort();
17
18    // Registration
19    Align();
}

```

The workflow is straightforward: When IMU is uninitialized, it uses the static initialization method from Chapter 3 to estimate IMU biases. After initialization, it first performs prediction using IMU data, then applies motion undistortion to the point cloud using predicted states, and finally registers the undistorted point cloud. The static initialization part was introduced earlier. Now let's examine the implementations of these three functions:

The prediction part is simple - it directly feeds IMU data to the filter and records the nominal states:

Listing 7.24: src/ch7/loosely_coupled_lio/loosly_lio.cc

```

1 void LooselyLIO::Predict() {
2     imu_states_.clear();
3     imu_states_.emplace_back(eskf_.GetNominalState());
4
5     /// Predict IMU states
6     for (auto &imu : measures_.imu_) {
7         eskf_.Predict(*imu);
8         imu_states_.emplace_back(eskf_.GetNominalState());
9     }
10 }

```

Regarding motion undistortion, let's first examine its underlying principle.

Motion Compensation Using IMU Predicted Poses With IMU poses available in the LIO system, we can utilize IMU-predicted poses to perform motion compensation on LiDAR point clouds. This aspect wasn't covered in previous pure LiDAR systems, so we'll introduce it here.

Motion compensation refers to correcting scan distortions caused by vehicle motion. If the LiDAR remains stationary during scanning, the measured distances match the true distances. However, when the LiDAR moves with the vehicle, we must account for the vehicle's motion during scanning. Most LiDARs operate at 10-20Hz with scan durations of 0.1-0.05 seconds. For a vehicle moving at 20m/s, the pose difference between scan start and end could reach 1-2 meters. Faster speeds or larger rotations exacerbate motion distortion. Small autonomous vehicles or robots may also rotate rapidly, potentially causing scan angles to under- or overshoot 360 degrees, resulting in ghosting artifacts within single scans. Therefore, motion compensation becomes essential for fast-moving vehicles.

The principle is straightforward: we need the vehicle poses during scanning. Assume a scan duration t_s . Each LiDAR point carries its timestamp (typically provided by the driver, or calculable from beam angles if the LiDAR model is known). Let $\mathbf{p}_t = (x, y, z)^\top, t \in (0, t_s)$ represent a point's coordinates in the **LiDAR frame**. Meanwhile, we can query LiDAR poses at any $t \in [0, t_s]$ (typically via IMU pose interpolation). Denote the IMU pose at t as $\mathbf{T}(t)_{WI}$ and the end pose as $\mathbf{T}(t_s)_{WI}$. To compensate motion, we transform each point to the scan-end pose:

Given IMU-LiDAR extrinsics \mathbf{T}_{IL} , the compensation formula is:

$$\mathbf{p}' = \mathbf{T}_{LI} \mathbf{T}(t_s)_{IW} \mathbf{T}(t)_{WI} \mathbf{T}_{IL} \mathbf{p}_t. \quad (7.29)$$

Reading right-to-left, this naturally yields coordinates in the end-time LiDAR frame. Omitting the leftmost matrix gives IMU-frame coordinates.

In implementation, we obtain scan-start/end poses through various means. For LIO systems, a natural approach uses relative poses from EKF prediction between scans. The `Predict` function already stores IMU-predicted poses between scans, enabling the following undistortion code:

Listing 7.25: src/ch7/loosely_couple_llo/loosly_llo.cc

```

1 void LooselyLIO::Undistort() {
2     auto cloud = measures_.lidar_;
3     auto imu_state = eskf_.GetNominalState(); // Final state
4     SE3 T_end = SE3(imu_state.R_, imu_state.p_);
5
6     /// Transform all points to end-time pose
7     std::for_each(std::execution::par_unseq, cloud->points.begin(), cloud->points.end()
8         , [&](auto &pt) {
9             SE3 Ti = T_end;
10            NavStated match;
11
12            // pt.time is milliseconds relative to scan start
13            math::PoseInterp<NavStated>(
14                measures_.lidar_begin_time_ + pt.time * 1e-3, imu_states_, [](&const NavStated &s)
15                { return s.timestamp_; }, [](&const NavStated &s) { return s.GetSE3(); }, 
16                Ti, match);
17
17            Vec3d pi = ToVec3d(pt);
18            Vec3d p_compensate = TIL_.inverse() * T_end.inverse() * Ti * TIL_ * pi;
19
20            pt.x = p_compensate(0);
21            pt.y = p_compensate(1);
22            pt.z = p_compensate(2);
23        });
24     scan_undistort_ = cloud;

```

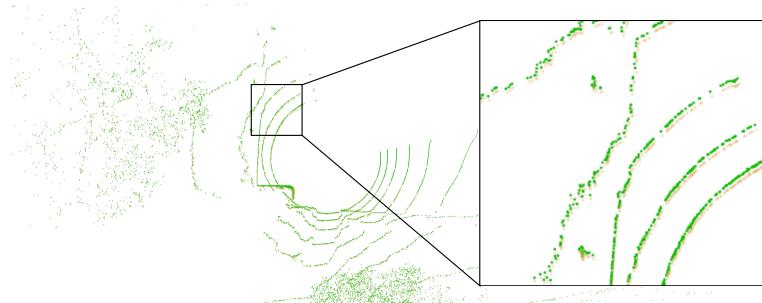


Figure 7-14: Point cloud before/after undistortion (green: corrected)

}

Figure 7-14 shows the undistortion effect. Right-side points (near scan end) require smaller corrections, while left-side points (near scan start) need larger adjustments. Note this method assumes the filter works properly - if the filter fails or poses diverge, the undistortion will likewise diverge.

7.6.5 Registration Module of the Loosely Coupled LIO System

Finally, we present the registration module code of the loosely coupled LIO system. Since the undistorted point cloud has already been obtained in previous steps, we only need to pass it to the NDT odometry, compute the pose, and then feed it back to the Kalman filter:

Listing 7.26: src/ch7/loosely_coupled_lis/loosly_lis.cc



Figure 7-15: Real-time operation results of the loosely coupled LIO program on NCLT dataset

```

34 }
35 frame_num_++;
36 }
```

Now please run the `test_loosely_lio` program to evaluate the performance of this chapter's implementation. The program supports running on three datasets: ULHK, KITTI, and NCLT. You can specify which dataset to use through gflags, for example:

```
./bin/test_loosely_lio --bag_path ./dataset/sad/nclt/20120429.bag --dataset_type
NCLT --config ./config/velodyne_nclt.yaml
```

This will run on a specified NCLT data bag. The NCLT data should appear as shown in Figure 7-15. This example program can also run on solid-state LiDAR datasets, such as the AVIA dataset:

```
bin/test_loosely_lio --bag_path ./dataset/sad/avia/HKU_MB_2020-09-20-13-34-51.bag --
config ./config/avia.yaml --dataset_type=AVIA
```

The AVIA dataset uses DJI's solid-state LiDAR, where you can observe its field of view is significantly narrower than 360-degree mechanical LiDARs (Figure 7-16).

7.7 Summary

This section systematically introduced the fundamental algorithms of a 3D LiDAR SLAM system. We implemented various ICP and NDT methods and extended them into LiDAR odometry. Furthermore, we realized feature-based LiDAR odometry and, combined with the previous Kalman filter, implemented a loosely coupled LIO system based on the Kalman filter. Through the programs in this section, readers can gain deeper insight into how a point cloud system is composed and operates.

In the next section, we will focus on tightly coupled LIO systems. We will introduce tightly coupled LIO based on iterative Kalman filters and optimization. Subsequently, we will incorporate RTK constraints and loop closure detection constraints to achieve complete mapping and localization functionality.

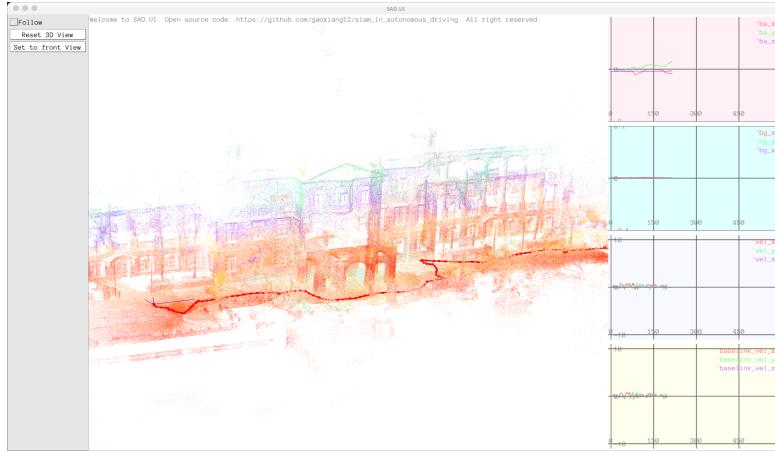


Figure 7-16: Real-time operation results of the loosely coupled LIO program on AVIA dataset

Exercises

1. Derive the right multiplication derivative of \mathbf{R} in Equation (7.4).
2. Use std::reduce to parallelize the Hessian matrix accumulation process.
3. Derive the derivatives of point-to-plane and point-to-line residuals with respect to \mathbf{R} .
4. Explain the relationship between weighted least squares problems and maximum likelihood estimation in NDT.
5. Design and implement a parallel computation pipeline for calculating laser beam curvature.
6. Develop a ground extraction pipeline for the LOAM-like odometry in this book, and apply point-to-plane ICP separately to ground point clouds.
7. Study relevant literature to understand the differences in feature extraction methods between algorithms like LOAM or LeGO-LOAM.
8. *Attempt to implement a loosely coupled LIO system using other methods introduced in this book (point-to-plane ICP, feature-based methods, etc.).

Part III

Applications

Chapter 8

Tightly Coupled LIO System

From this chapter through the next three chapters, we will introduce readers to some practical autonomous driving localization and mapping technologies. This chapter focuses on the tightly coupled Lidar-IMU-Odometry system, known as the tightly coupled LIO system (sometimes referred to as Lins[168] in literature, meaning a system combining LiDAR and inertial navigation). Chapters 9 and 10 will cover offline mapping systems and real-time fusion localization techniques respectively. Overall, autonomous driving applications of SLAM primarily concentrate on two major modules: mapping and localization. Within the localization module, DR or LIO is often used for local position estimation. The tightly coupled LIO system is more complex than the loosely coupled LIO system introduced in the previous chapter. This chapter will first explain its principles before proceeding with implementation.



8.1 Principles and Advantages of Tight Coupling

First, let's address a fundamental question: What is tight coupling, and why do we need a tightly coupled LIO system? The loosely coupled LIO introduced in the previous chapter already demonstrates good performance - what additional benefits can tight coupling provide? In fact, the term “**tightly coupled**” isn't unique to LIO systems; similar concepts exist in traditional integrated navigation and VIO fields[169–171]. Broadly speaking, any state estimation system that considers the intrinsic properties of sensors rather than **modularly** fusing their outputs can be called a tightly coupled system[172]. For example, systems considering IMU observation noise and biases can be called tightly coupled IMU (or INS) systems; those accounting for LiDAR registration residuals are tightly coupled LiDAR systems; while systems incorporating visual feature reprojection errors or RTK sub-states and satellite counts qualify as tightly coupled visual or RTK systems[173, 174]. In loosely coupled systems, we can treat each sensor or algorithm module as a black box, considering only their outputs. The previous chapter demonstrated this approach, which readers should now understand.

So what specific advantages does tight coupling offer? Based on the author's experience, when all algorithm modules function normally, there may be no significant difference between tightly and loosely coupled systems. For instance, when fusing GINS systems with LiDAR odometry, if RTK signals remain valid, the coupling approach shouldn't produce noticeable differences. However, in real-world systems, individual algorithm modules often can't maintain continuous normal operation. Standalone IMU systems quickly diverge without velocity and position observations; independent LiDAR and visual odometry may fail or degrade in structurally poor environments. In loosely coupled systems, when a module fails, we must logically identify the failure and attempt recovery. Tightly coupled systems allow one module's operational status to directly influence others, helping better constrain their working dimensions. Taking loosely coupled LIO as an example: when a vehicle traverses degenerate areas, LiDAR odometry relying solely on point cloud matching may fail, producing erroneous pose estimates with extra degrees of freedom that could mislead the entire system after fusion. In tightly coupled LIO systems, states remain constrained by other sensors, keeping these degrees of freedom within fixed bounds to maintain system validity. While this explanation may seem abstract, this chapter will use practical cases to help readers better understand these concepts.

8.2 LIO System Based on IEKF

8.2.1 State Variables and Motion Equations of IEKF

In a tightly-coupled LIO system, the IMU and LiDAR point cloud registration share the same state model, motion equations, and observation equations. Thus, the most straightforward fusion method is to incorporate both into the EKF model, where the IMU provides constraints during motion and the LiDAR point cloud provides constraints for the observation equation. However, whether using ICP or NDT, LiDAR point cloud registration often requires multiple nearest-neighbor iterations to converge to the correct solution—a point readers should have grasped from the earlier chapters on ICP and NDT. Therefore, we must also adapt the traditional EKF filter into its iterative version: the Iterated Extended Kalman Filter (IEKF). The theory of IEKF is slightly more complex than EKF, but fortunately, we have already derived the error-state Kalman filter in Chapter 3. This section will focus on the iteration process and the integration of LiDAR residuals.

First, let us revisit the ESKF theory introduced earlier. For convenience, we denote its

high-dimensional state variables uniformly as \mathbf{x} , defined in a high-dimensional manifold space \mathcal{M} :

$$\mathbf{x} = [\mathbf{p}, \mathbf{v}, \mathbf{R}, \mathbf{b}_g, \mathbf{b}_a, \mathbf{g}]^\top \in \mathcal{M}. \quad (8.1)$$

We know that its error state can be defined in the usual vector space, i.e., the tangent space of \mathcal{M} near the operating point: $\delta\mathbf{x} \in \mathbb{R}^{18}$. When IMU data arrives, the filter propagates the state variables based on the IMU readings and the current state. The motion process consists of two parts:

1. **State Propagation Using IMU Readings**: Given the IMU readings $\tilde{\omega}, \tilde{\mathbf{a}}$ between time t and $t + \Delta t$, the discrete-time propagation can be written as:

$$\mathbf{p}(t + \Delta t) = \mathbf{p}(t) + \mathbf{v}\Delta t + \frac{1}{2}(\mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a))\Delta t^2 + \frac{1}{2}\mathbf{g}\Delta t^2, \quad (8.2a)$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a)\Delta t + \mathbf{g}\Delta t, \quad (8.2b)$$

$$\mathbf{R}(t + \Delta t) = \mathbf{R}(t)\text{Exp}((\tilde{\omega} - \mathbf{b}_g)\Delta t), \quad (8.2c)$$

$$\mathbf{b}_g(t + \Delta t) = \mathbf{b}_g(t), \quad (8.2d)$$

$$\mathbf{b}_a(t + \Delta t) = \mathbf{b}_a(t), \quad (8.2e)$$

$$\mathbf{g}(t + \Delta t) = \mathbf{g}(t). \quad (8.2f)$$

2. **Observation Process**: While the nominal state is updated according to the above equations, the mean of the error state remains zero, and only its covariance needs to be updated:

$$\mathbf{P}_{\text{pred}} = \mathbf{F}\mathbf{P}\mathbf{F}^\top + \mathbf{Q}, \quad (8.3)$$

where \mathbf{Q} is the noise matrix, \mathbf{P} is the state covariance from the previous time step, and the \mathbf{F} matrix is derived from the linear form of error kinematics:

$$\mathbf{F} = \begin{bmatrix} \mathbf{I} & \mathbf{I}\Delta t & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & -\mathbf{R}(\tilde{\mathbf{a}} - \mathbf{b}_a)^\wedge\Delta t & \mathbf{0} & -\mathbf{R}\Delta t & \mathbf{I}\Delta t \\ \mathbf{0} & \mathbf{0} & \text{Exp}(-(\tilde{\omega} - \mathbf{b}_g)\Delta t) & -\mathbf{I}\Delta t & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix}. \quad (8.4)$$

These details were introduced earlier (see Equation (??)). For the reader's convenience, they are restated here. Since the LiDAR frequency is typically lower than the IMU frequency, multiple IMU readings may exist between two LiDAR data points. We simply propagate the state multiple times using the above equations. For brevity, we omit subscripts and superscripts for multiple propagation steps and denote the propagated state estimate as $\mathbf{x}_{\text{pred}}, \mathbf{P}_{\text{pred}}$. Before applying the observation equation for correction, the estimated mean and covariance are these two quantities.

8.2.2 Iterative Process in the Observation Equation

In a tightly-coupled system, we incorporate the ICP or NDT residuals from the LiDAR as observation equations into the EKF model. However, since both ICP and NDT require iterations to converge, we should also introduce an **iterative process** for the EKF observation step. This iterative process has the following key points:

1. **Linearization of the Error State**: The iterative process first **linearizes** the **error state**. That is, the nominal state starts from \mathbf{x}_{pred} and is continuously updated by the error state $\delta\mathbf{x}$. Since the update $\delta\mathbf{x}$ varies with each iteration, we denote the error state

at the k -th iteration as $\delta\mathbf{x}_k$ and its covariance as \mathbf{P}_k^1 . At $k = 0$, we initialize the process as: $\delta\mathbf{x}_0 = \mathbf{0}, \mathbf{P}_0 = \mathbf{P}_{\text{pred}}$.

2. ****Recomputing Kalman Gain and Update**:** Since the observation equation requires iteration, the Kalman gain and update process must also be recomputed at each step. Thus, quantities related to the update are also subscripted with k , denoted as $\mathbf{K}_k, \mathbf{H}_k$. Before updating, we denote the working point as \mathbf{x}_k , which is essentially obtained by incrementally adding $\delta\mathbf{x}_k$ to \mathbf{x}_0 . From the perspective of nominal and error states, the nominal state at the k -th iteration is \mathbf{x}_k , while the error state is $\mathbf{0}$.

3. ****Practical Tricks for \mathbf{P} Updates**:** During IEKF iterations before convergence, we can consider the filter as still in progress, merely shifting the working point. Thus, \mathbf{P} does not need to be recalculated at every iteration—only the initial \mathbf{P}_{pred} needs to be projected into the current tangent space. The final \mathbf{P} update is performed only after the last iteration, projecting it to the new working point.

4. ****Differences Due to Repeated Linearization**:** In EKF, linearization is performed only once at \mathbf{x}_{pred} . In IEKF, however, the k -th linearization is computed at \mathbf{x}_k . Traditional EKF describes how to update $\mathbf{x}_{\text{pred}}, \mathbf{P}_{\text{pred}}$ to \mathbf{x}, \mathbf{P} , whereas IEKF focuses on updating $\mathbf{x}_{k-1}, \mathbf{P}_{k-1}$ to $\mathbf{x}_k, \mathbf{P}_k$ (though, as noted earlier, \mathbf{P}_k does not need to be explicitly computed). This difference leads to significant variations in the update process between EKF and IEKF, which will be reflected in later implementations.

5. ****Dynamic Observation Model**:** Like ICP and NDT, tightly-coupled LIO should update the nearest-neighbor relationships for each matched point in every iteration. This results in an IEKF observation model **with a variable number of equations**², unlike standard EKF, which has a fixed observation dimension². When the observation dimension is large, certain algebraic transformations are applied to EKF formulas to handle high-dimensional computations.

The overall process is illustrated in Figure 8-1. Starting from $\mathbf{x}_0, \mathbf{P}_0$, we iteratively refine the observation model, compute the incremental update $\delta\mathbf{x}$, and update \mathbf{x} and \mathbf{P} until convergence.

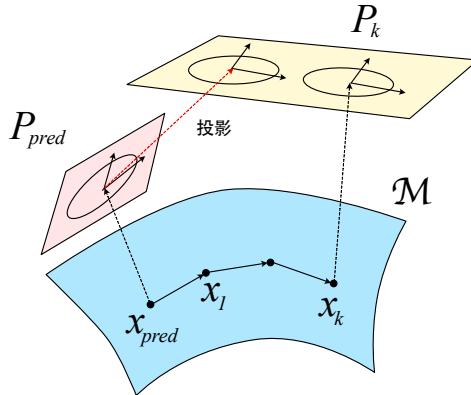


Figure 8-1: Schematic diagram of the Iterated Kalman Filter

Next, we derive this process. Consider the k -th iteration, where the working point is $\mathbf{x}_k, \mathbf{P}_k$, and we aim to compute the incremental update $\delta\mathbf{x}_{k+1}$. First, we clarify the rela-

¹Note that in Chapter 3 of this book, the subscript k was used to denote the k -th time step. Here, however, we focus on the state within a single time step and use k to represent the iteration count rather than the time step. Adding additional superscripts or subscripts for iterations would make the notation overly cumbersome.

²This is specific to LIO and does not imply that IEKF generally alters the number of observation equations.

tionship between \mathbf{x}_k , \mathbf{P}_k and \mathbf{x}_0 , \mathbf{P}_0 . While \mathbf{x}_k is straightforward (obtained by accumulating incremental updates), \mathbf{P}_k requires a tangent space projection, as discussed in Chapter 3 and Equation (3.63). We denote the tangent space Jacobian at the k -th iteration as \mathbf{J}_k :

$$\mathbf{J}_k = \text{diag}(\mathbf{I}_3, \mathbf{I}_3, \mathbf{J}_\theta, \mathbf{I}_3, \mathbf{I}_3, \mathbf{I}_3), \quad \mathbf{J}_\theta = \mathbf{I} - \frac{1}{2}\delta\theta_k^\wedge, \quad \delta\theta_k = \text{Log}(\mathbf{R}_0^\top \mathbf{R}_k). \quad (8.5)$$

This \mathbf{J}_k describes the transformation from \mathbf{P}_{pred} to \mathbf{P}_k . Thus, from the perspective of \mathbf{x}_k , the prior distribution is:

$$\delta\mathbf{x}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{J}_k \mathbf{P}_{\text{pred}} \mathbf{J}_k^\top). \quad (8.6)$$

We denote $\mathbf{P}_k = \mathbf{J}_k \mathbf{P}_{\text{pred}} \mathbf{J}_k^\top$. The observation model for iterative EKF is the same as standard EKF. Referring to the EKF update formula (??), the update process for the iterated Kalman filter is:

$$\mathbf{K}_k = \mathbf{P}_k \mathbf{H}_k^\top (\mathbf{H}_k \mathbf{P}_k \mathbf{H}_k^\top + \mathbf{V})^{-1}, \quad (8.7a)$$

$$\delta\mathbf{x}_k = \mathbf{K}_k (\mathbf{z} - \mathbf{h}(\mathbf{x}_k)). \quad (8.7b)$$

If the filter has not yet converged, \mathbf{P} is not updated—instead, \mathbf{J}_{k+1} is computed from \mathbf{x}_{k+1} , and the initial distribution is projected accordingly. Upon convergence, \mathbf{P}_{k+1} is updated using the Kalman formula:

$$\mathbf{P}_{k+1} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{J}_k \mathbf{P}_{\text{pred}} \mathbf{J}_k^\top. \quad (8.8)$$

In other words, from a covariance perspective, **only the final iteration** is considered effective. The Kalman gain \mathbf{K}_k and linearization matrix \mathbf{H}_k influence only the last update. Intermediate iterations merely adjust the starting point.

If IEKF completes the iteration, \mathbf{P}_{k+1} should also be projected into the tangent space at the final step to maintain consistency. Overall, each IEKF iteration solves a least-squares problem with a prior:

$$\delta\mathbf{x}_k = \arg \min_{\delta\mathbf{x}} \|\mathbf{z} - \mathbf{H}_k(\mathbf{x}_k \boxplus \delta\mathbf{x})\|_{\mathbf{V}}^2 + \|\delta\mathbf{x}_k\|_{\mathbf{P}_k}^2. \quad (8.9)$$

The first term represents the observation residual, while the second term is the prior residual projected into the tangent space. This derivation differs slightly from similar works (e.g., [4, 47]) but is simpler overall. A full implementation would update \mathbf{P}_k at every iteration rather than projecting \mathbf{P}_{pred} , but this would complicate the notation. Alternatively, all increments and covariances could be considered at \mathbf{x}_0 , but this would make the mathematics even more involved.

8.2.3 Efficient Handling of High-Dimensional Observations

In tightly-coupled systems, we directly incorporate NDT or ICP residuals into the observation equation, which significantly increases the dimensionality of the observation space. In traditional integrated navigation systems like RTK, observation equations are typically 3D or 6D. However, when point clouds are included, the observation equation can easily reach thousands or even tens of thousands of dimensions. If we compute the Kalman gain using Equation (8.7)(a), we inevitably encounter the inversion of a very large matrix. Suppose the residual dimension is m , then \mathbf{H}_k becomes an $m \times 18$ matrix, and the term $(\mathbf{H}_k \mathbf{P}_k \mathbf{H}_k^\top + \mathbf{V})$ in the Kalman gain requires the inversion of an $m \times m$ matrix—an operation we must avoid.

In Kalman filters, the **Sherman-Morrison-Woodbury (SMW) identity** [4, 175] is a widely used algebraic transformation that proves highly useful in various Kalman filter derivations. The SMW identity has four equivalent forms, one of which is:

$$\mathbf{AB}(\mathbf{D} + \mathbf{CAB})^{-1} = (\mathbf{A}^{-1} + \mathbf{BD}^{-1}\mathbf{C})^{-1}\mathbf{BD}^{-1}, \quad (8.10)$$

where $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ are matrix blocks satisfying standard matrix multiplication rules. Substituting $\mathbf{P}_k, \mathbf{H}_k, \mathbf{V}$ into this identity yields:

$$\mathbf{K}_k = (\mathbf{P}_k^{-1} + \mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{H}_k)^{-1} \mathbf{H}_k^\top \mathbf{V}^{-1}. \quad (8.11)$$

Note that the inversion inside this formula is now 18×18 , drastically reducing the computational burden. When dealing with high-dimensional observations, we should prefer this form for computing the Kalman gain. Moreover, comparing this with the linear increment equation in NDT (Equation (7.14)), we observe a striking similarity: The term $(\mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{H})^{-1}$ in the Kalman gain corresponds exactly to the coefficient matrix in the normal equation, while $\mathbf{H}^\top \mathbf{V}^{-1}$ aligns with the residual term on the right-hand side of (7.14). This highlights the fundamental nature of the Kalman filter as a balance between prior information and observations.

To further clarify the connection between NDT and the Kalman filter, let us examine their formulations. The Kalman filter's incremental update $\delta \mathbf{x}_k$ is:

$$\delta \mathbf{x}_k = \mathbf{K}_k (\mathbf{z} - \mathbf{h}(\mathbf{x}_k)), \quad (8.12)$$

whereas the least-squares increment in NDT or ICP is:

$$\sum_i (\mathbf{J}_i^\top \Sigma_i^{-1} \mathbf{J}_i) \Delta \mathbf{x} = - \sum_i \mathbf{J}_i^\top \Sigma_i^{-1} \mathbf{e}_i. \quad (8.13)$$

Observant readers will notice the equivalence between these two equations. The matrix inversion in (8.13) matches the Kalman gain in (8.11) when the prior is omitted. The key difference is that the Kalman gain is expressed in matrix form, while ICP or NDT uses summation notation. For later discussions on NDT-LIO, we derive how NDT residuals are incorporated into the Kalman gain. In the experimental section, we will follow this derivation rather than the matrix-based Kalman gain.

Consider the NDT registration of a point cloud with N points. For the j -th point, its residual \mathbf{r}_j is computed based on the voxel it falls into in the target point cloud, with an associated information matrix Σ_j^{-1} (the inverse covariance of the voxel's Gaussian distribution). The squared error \mathbf{e}_j is:

$$\mathbf{e}_j = \mathbf{r}_j^\top \Sigma_j^{-1} \mathbf{r}_j. \quad (8.14)$$

The Jacobian of \mathbf{r}_j with respect to rotation and translation is:

$$\frac{\partial \mathbf{r}_j}{\partial \mathbf{R}} = -\mathbf{R} \mathbf{q}_j^\wedge, \quad \frac{\partial \mathbf{r}_j}{\partial \mathbf{t}} = \mathbf{I}. \quad (8.15)$$

Following the state variable definition in (8.1), we expand the Jacobian to match the state dimensions:

$$\mathbf{J}_j = \left[\frac{\partial \mathbf{r}_j}{\partial \mathbf{t}}, \mathbf{0}_3, \frac{\partial \mathbf{r}_j}{\partial \mathbf{R}}, \mathbf{0}_3, \mathbf{0}_3, \mathbf{0}_3 \right]. \quad (8.16)$$

This Jacobian was introduced in Chapter 7, and we pad it with zero blocks to align with the state vector. In the filter, the j -th block row of \mathbf{H}_k ³ is \mathbf{J}_j , and the noise matrix \mathbf{V} is a block-diagonal matrix composed of Σ_j^{-1} :

$$\mathbf{H}_k = \begin{bmatrix} \cdots, \\ \mathbf{J}_j, \\ \cdots \end{bmatrix}, \quad \mathbf{V} = \text{diag}(\cdots, \Sigma_j, \cdots). \quad (8.17)$$

³Strictly speaking, it should be the $3 \times j$ -th row since \mathbf{J}_j has 3 rows. Interpreted as a block matrix, this is correct.

Since \mathbf{V} is block-diagonal, the term $\mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{H}_k$ in the Kalman gain can be rewritten as a summation:

$$\mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{H}_k = \sum_j \mathbf{J}_j^\top \Sigma_j^{-1} \mathbf{J}_j. \quad (8.18)$$

Similarly, multiplying $\mathbf{H}_k^\top \mathbf{V}^{-1}$ by $(\mathbf{z} - \mathbf{h}(\mathbf{x}_k))$ gives:

$$\mathbf{H}_k^\top \mathbf{V}^{-1} (\mathbf{z} - \mathbf{h}(\mathbf{x}_k)) = [\dots, \mathbf{J}_j^\top, \dots] \text{diag}(\dots, \Sigma_j^{-1}, \dots) \begin{bmatrix} \dots \\ \mathbf{r}_j \\ \dots \end{bmatrix}, \quad (8.19)$$

$$= \sum_j \mathbf{J}_j^\top \Sigma_j^{-1} \mathbf{r}_j. \quad (8.20)$$

For the covariance update in (8.8), substituting \mathbf{K} into $\mathbf{I} - \mathbf{K}_k \mathbf{H}_k$ yields:

$$\mathbf{P}_{k+1} = (\mathbf{I} - (\mathbf{P}_k^{-1} + \mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{H}_k)^{-1} \mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{H}_k) \mathbf{J}_k \mathbf{P}_{\text{pred}} \mathbf{J}_k^\top, \quad (8.21)$$

where $\mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{H}_k$ can again be replaced with the summation form from NDT. Thus, the entire iterative Kalman filter computation is tightly linked with NDT. In this sense, the tightly-coupled system can be viewed as a high-dimensional NDT or ICP with IMU predictions, where these predictions are propagated to the next time step.

8.3 Implementation of IEKF-based LIO

Next we implement the IEKF and its corresponding LIO system. As mentioned earlier, tightly-coupled LIO can compute point cloud residuals through various methods such as point-to-line, point-to-plane, NDT, etc. This section implements an NDT-LIO based on IEKF, while readers may also refer to this book or literature [59, 60, 89] to implement point-to-plane ICP based LIO.

We modify the ESKF from Chapter 3 to design the IEKF interface. Since we want NDT to directly provide Jacobian and information matrices, at the IEKF implementation level, we expect NDT to compute $\mathbf{H}^\top \mathbf{V}^{-1} \mathbf{H}$ and $\mathbf{H}^\top \mathbf{V}^{-1} \mathbf{r}$ (these quantities are calculated internally by NDT), then IESKF uses its return values to iterate the entire filter:

Listing 8.1: src/ch8/lio-iekf/iekf.hpp

```

1 /**
2 * NDT observation function, inputs an SE3 Pose, returns terms from (8.10)
3 * HT V^[-1] H
4 * H\top V^[-1] r
5 * Both can be done in summation form
6 */
7 using CustomObsFunc = std::function<void(const SE3& input_pose, Eigen::Matrix<S, 18,
8   18>& HT_Vinv_H,
9 Eigen::Matrix<S, 18, 1>& HT_Vinv_r)>;
10
11 /// Update filter using custom observation function
12 bool UpdateUsingCustomObserve(CustomObsFunc obs) {
13   SO3 start_R = R_;
14   Eigen::Matrix<S, 18, 1> HTVr;
15   Eigen::Matrix<S, 18, 18> HTVH;
16   Eigen::Matrix<S, 18, Eigen::Dynamic> K;
17   Mat18T Pk, Qk;
18
19   for (int iter = 0; iter < options_.num_iterations_; ++iter) {
20     // call obs function
21     obs(GetNominalSE3(), HTVH, HTVr);
22     // project P

```

```

23   Mat18T J = Mat18T::Identity();
24   J.template block<3, 3>(6, 6) = Mat3T::Identity() - 0.5 * S03::hat(R_.inverse() *
25     start_R).log());
26   Pk = J * cov_ * J.transpose();
27
28   // Kalman update
29   Qk = (Pk.inverse() + HTVH).inverse(); // this intermediate variable can be used
30   // in final update
31   dx_ = Qk * HTVr;
32   LOG(INFO) << "iter " << iter << " dx = " << dx_.transpose() << ", dxn: " << dx_.
33   norm();
34
35   // merge dx into nominal variables
36   UpdateAndReset();
37
38   if (dx_.norm() < options_.quit_eps_) {
39     break;
40   }
41
42   // update P
43   cov_ = (Mat18T::Identity() - Qk * HTVH) * Pk;
44
45   // project P
46   Mat18T J = Mat18T::Identity();
47   J.template block<3, 3>(6, 6) = Mat3T::Identity() - 0.5 * S03::hat(dx_.template block
48     <3, 1>(6, 0));
49   cov_ = J * cov_ * J.inverse();
50
51   dx_.setZero();
52   return true;
53 }
```

This function provides a current estimate \mathbf{x}_k for external algorithms to compute the corresponding $\mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{H}_k$ and $\mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{r}_k$, which are then substituted into the IEKF calculation formula to obtain the current time increment $\delta\mathbf{x}_k$, finally applied to the nominal state variables. Note we denote $(\mathbf{P}_k^{-1} + \mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{H})^{-1}$ as intermediate variable \mathbf{Q}_k in the code. This variable allows us to more simply compute the Kalman filter error state and covariance:

$$\delta\mathbf{x}_k = \mathbf{Q}_k \mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{r}_k, \quad (8.22)$$

$$\mathbf{P}_{k+1} = (\mathbf{I} - \mathbf{Q}_k \mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{H}_k) \mathbf{P}_k. \quad (8.23)$$

Following the previous derivation, we add an interface to Chapter 7's NDT to compute the corresponding matrices and return the results. The entire computation flow is completely consistent with the previous NDT, except NDT no longer needs to compute the updates itself.

Listing 8.2: src/ch7/ndt_inc.cc

```

1 void IncNdt3d::ComputeResidualAndJacobians(const SE3& input_pose, Mat18d& HTVH, Vec18d
2   & HTVr) {
3   assert(grids_.empty() == false);
4   SE3 pose = input_pose;
5
6   // Most steps same as Align(), except z, H, R are handled externally
7   int num_residual_per_point = 1;
8   if (options_.nearby_type_ == NearbyType::NEARBY6) {
9     num_residual_per_point = 7;
10  }
11
12  std::vector<int> index(source_>points.size());
13  for (int i = 0; i < index.size(); ++i) {
14    index[i] = i;
15  }
16
17  int total_size = index.size() * num_residual_per_point;
18
19  std::vector<bool> effect_pts(total_size, false);
20  std::vector<Eigen::Matrix<double, 3, 18>> jacobians(total_size);
```

```

20 std::vector<Vec3d> errors(total_size);
21 std::vector<Mat3d> infos(total_size);
22
23 // gauss-newton iteration
24 // nearest neighbor, can be parallel
25 std::for_each(std::execution::par_unseq, index.begin(), index.end(), [&](int idx) {
26     auto q = ToVec3d(source->points[idx]);
27     Vec3d qs = pose * q; // transformed q
28
29     // compute voxel containing qs and its neighbors
30     Vec3i key = (qs * options_.inv_voxel_size_).cast<int>();
31
32     for (int i = 0; i < nearby_grids_.size(); ++i) {
33         Vec3i real_key = key + nearby_grids_[i];
34         auto it = grids_.find(real_key);
35         int real_idx = idx * num_residual_per_point + i;
36         /// need to check if Gaussian distribution has been estimated
37         if (it != grids_.end() && it->second.ndt_estimated_) {
38             auto& v = it->second; // voxel
39             Vec3d e = qs - v.mu_;
40
41             // check chi2 th
42             double res = e.transpose() * v.info_ * e;
43             if (std::isnan(res) || res > options_.res_outlier_th_) {
44                 effect_pts[real_idx] = false;
45                 continue;
46             }
47
48             // build residual
49             Eigen::Matrix<double, 3, 18> J;
50             J.setZero();
51             J.block<3, 3>(0, 0) = Mat3d::Identity(); // Jacobian w.r.t.
52             p
53             J.block<3, 3>(0, 6) = -pose.so3().matrix() * S03::hat(q); // Jacobian w.r.t.
54             R
55
56             jacobians[real_idx] = J;
57             errors[real_idx] = e;
58             infos[real_idx] = v.info_;
59             effect_pts[real_idx] = true;
60         } else {
61             effect_pts[real_idx] = false;
62         }
63     }
64
65     // accumulate Hessian and error, compute dx
66     double total_res = 0;
67     int effective_num = 0;
68
69     HTVH.setZero();
70     HTVr.setZero();
71
72     std::vector<double> err;
73     const double info_ratio = 0.01; // info feedback factor per point
74
75     for (int idx = 0; idx < effect_pts.size(); ++idx) {
76         if (!effect_pts[idx]) {
77             continue;
78         }
79
80         total_res += errors[idx].transpose() * infos[idx] * errors[idx];
81         effective_num++;
82
83         err.push_back(errors[idx].transpose() * infos[idx] * errors[idx]);
84
85         HTVH += jacobians[idx].transpose() * infos[idx] * jacobians[idx] * info_ratio;
86         HTVr += -jacobians[idx].transpose() * infos[idx] * errors[idx] * info_ratio;
87     }
88 }
```

Here we compute the distributions within NDT voxels in parallel, collect the Jacobians and residuals for each point, and accumulate them into the two output matrices: $\mathbf{H}_k^\top \mathbf{V}^{-1} \mathbf{H}_k$

should be an 18×18 matrix after accumulation, while $\mathbf{H}_k \mathbf{V}^{-1} \mathbf{r}_k$ should be an 18×1 vector after accumulation, which readers can verify through the code. Since NDT typically has significantly more points than the prediction equation, this may bias the estimation toward NDT. We add a scaling factor (0.01) to the information matrix Σ^{-1} here to make the updates more balanced. Readers may adjust this parameter.

At the LIO level, we retain the loosely-coupled framework and only need to modify the registration process to the tightly-coupled form:

Listing 8.3: src/ch8/lio-iekf/lio_iekf.cc

```

1 // subsequent scans, update using NDT with pose
2 ndt_.SetSource(current_scan_filter);
3 iekf_.UpdateUsingCustomObserve([this](const SE3 &input_pose, Mat18d &HTVH, Vec18d &
4 HTVr) {
5     ndt_.ComputeResidualAndJacobians(input_pose, HTVH, HTVr);
6 });
7 // if moved beyond threshold, add point cloud to map
8 SE3 current_pose = iekf_.GetNominalSE3();
9 SE3 delta_pose = last_pose_.inverse() * current_pose;
10 if (delta_pose.translation().norm() > 1.0 || delta_pose.so3().log().norm() > math::
11     deg2rad(10)) {
12     // merge scan into NDT map
13     CloudPtr current_scan_world(new PointCloudType);
14     pcl::transformPointCloud(*current_scan_filter, *current_scan_world, current_pose.
15         matrix());
16     ndt_.AddCloud(current_scan_world);
17     last_pose_ = current_pose;
18 }
```

Internally, IEKF provides NDT with the current state estimate, NDT computes the two matrices mentioned earlier and returns them to IEKF, which then updates its internal state estimate. This gives us the pose corresponding to the current LiDAR scan. On the other hand, if the vehicle moves more than 1 meter or rotates more than 10 degrees, we merge the current LiDAR scan data into the NDT map, preventing the vehicle from continuously spending time updating the map when stationary. We can test the tightly-coupled NDT effect through this chapter's test_lio_iekf program. The test program displays the current point cloud and internal state variables in the UI interface.

Listing 8.4: Terminal input:

```

1 bin/test_lio_iekf --bag_path ./dataset/sad/nclt/20120115.bag --dataset_type NCLT --
config ./config/velodyne_nclt.yaml
```

Readers can test this chapter's tightly-coupled LIO program on any provided dataset, with results shown in Figure 8-2. Since the tightly-coupled Kalman filter iteration process is integrated with NDT, their computational efficiency is extremely high. On my machine, a single 32-beam LiDAR registration takes only 2.7 milliseconds, meaning the entire LiDAR odometry frequency can exceed 300 Hz. Traditional LiDAR odometry methods (Loam[2], LeGo-LOAM[163], LIO-SAM[176]) typically require tens or hundreds of milliseconds. The IEKF and NDT based odometry demonstrates clear advantages in computational efficiency.

8.4 LIO Based on Preintegration

8.4.1 Principles of Preintegrated LIO

Next, we introduce an LIO system based on preintegration and point cloud registration. Similar to Chapter 3, since we have implemented LIO using Kalman filters, we will reimplement it using preintegrated graph optimization to help readers understand their connections

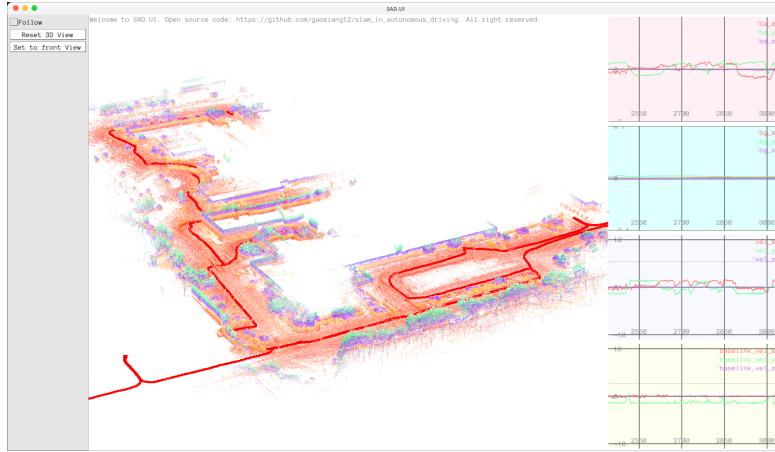


Figure 8-2: Reconstructed point cloud results from tightly-coupled LIO

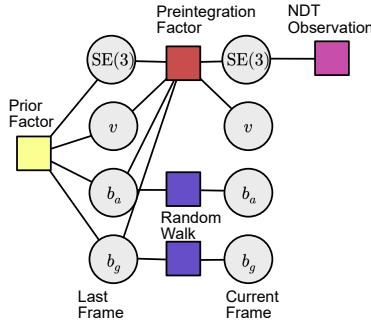


Figure 8-3: Graph optimization model with preintegration

and differences. Like integrated navigation systems, LIO systems with LiDAR point clouds can also be implemented using preintegrated IMU factors combined with LiDAR residuals. Some modern LiDAR SLAM systems adopt this approach. Compared to filtering methods, preintegrated factors can be more conveniently integrated into existing optimization frameworks, making development and implementation easier. However, in practice, the use of preintegration is quite flexible and requires more parameter tuning than EKF systems. For example, LIO-SAM combines preintegrated factors with LiDAR odometry factors to construct the entire optimization problem [176, 177]. In VSLAM systems, preintegrated factors can also be combined with reprojection errors to solve Bundle Adjustment [178]. Here we share some practical experience with preintegration applications:

1. Preintegrated factors typically connect high-dimensional states (typically 15-dimensional) between two keyframes. When converting to a graph optimization problem, we can choose to separate the vertices, e.g., having $\text{SE}(3)$ as one vertex and \mathbf{v} as another, with a preintegration edge connecting 8 vertices. Alternatively, we can combine the high-dimensional state into a single vertex, with preintegration edges connecting two vertices but containing many zero blocks in the Jacobian matrix. In practice, we adopt the former approach, which results in more vertex types but lower-dimensional edges.
2. Since preintegrated factors involve many variables and most measurements are dif-

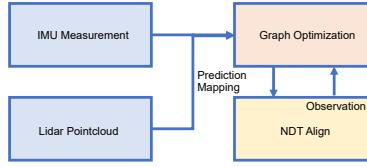


Figure 8-4: Computational framework of preintegrated LIO

ferences between state variables, sufficient observations and constraints on the state variables are necessary to prevent free movement in the null space. For example, the preintegrated velocity observation $\Delta\tilde{\mathbf{v}}_{ij}$ describes the difference between velocities at two keyframes. If we add a constant increment to both keyframe velocities, the velocity error remains unchanged, while adjustments to the displacement terms can keep displacement observations consistent. Therefore, in practice, we impose prior constraints on the previous keyframe and observation constraints on the subsequent keyframe to confine the optimization problem within reasonable bounds.

3. The graph optimization model with preintegration is shown in Figure 8-3. When optimizing two keyframes, we add a prior factor to the previous keyframe, then include preintegration factors and bias random walk factors between the frames, and finally add NDT pose constraints to the next keyframe. After optimization, we use marginalization to compute the covariance of the next keyframe’s pose, which serves as the prior factor for the next optimization round.
4. This graph optimization model closely resembles the GINS system in Chapter 4. However, it’s important to note that LiDAR odometry pose observations depend on prediction data, which is fundamentally different from RTK pose observations. With good RTK signals, we can assume fixed observation accuracy, ensuring convergence in both translation and rotation for filters and graph optimizers. In contrast, if LiDAR odometry uses inaccurate predicted poses, it may produce abnormal observations, causing the entire LIO to diverge. This makes debugging graph optimization-based LIO systems significantly more challenging than GINS systems.
5. To reuse the code from Section 8.3, we maintain the same LIO framework but replace the EKF-based prediction and observation parts with preintegrator components⁴. The computational framework of this LIO system is shown in Figure 8-4. We perform optimization using preintegration between two point clouds. As mentioned earlier, preintegration usage is highly flexible—readers need not strictly follow our implementation and may employ longer preintegration periods or incorporate NDT residuals into graph optimization. However, since preintegrated factors connect many vertices, debugging can be challenging and may lead to error divergence. Starting with an existing system and adding backend optimization is often a good strategy.

8.4.2 Code Implementation

The code in this section is primarily built upon Section 8.3 with the addition of graph optimization methods. Most of the core logic remains unchanged. The LIO system now includes a preintegrator that handles IMU data integration and provides predicted states, while also

⁴In practical systems, filters can serve as the frontend while graph optimization acts as the keyframe backend.

maintaining an incremental NDT odometer for point cloud registration and local map management.

Listing 8.5: src/ch8/lio-preinteg/lio_preinteg.h

```

1 class LioPreinteg {
2     private:
3         /// modules
4         std::shared_ptr<MessageSync> sync_ = nullptr;
5         StaticIMUInit imu_init_;
6
7         /// point clouds data
8         FullCloudPtr scan_undistort_{new FullPointCloudType()}; // scan after
9             undistortion
10        CloudPtr current_scan_ = nullptr;
11
12        // optimization-related
13        NavState last_nav_state_, current_nav_state_; // previous and current states
14        Mat15d prior_info_ = Mat15d::Identity(); // prior constraint
15        std::shared_ptr<IMUPreintegration> preinteg_ = nullptr;
16
17        IMUPtr last_imu_ = nullptr;
18
19        /// NDT data
20        IncNdt3d ndt_;
21        SE3 ndt_pose_;
22        SE3 last_ndt_pose_;
23
24        Options options_;
25        std::shared_ptr<ui::PangolinWindow> ui_ = nullptr;
26    };

```

During the prediction phase, we use poses from preintegration to undistort point clouds:

Listing 8.6: src/ch8/lio-preinteg/lio_preinteg.cc

```

1 void LioPreinteg::Predict() {
2     imu_states_.clear();
3     imu_states_.emplace_back(last_nav_state_);
4
5     /// Predict IMU states
6     for (auto &imu : measures_.imu_) {
7         if (last_imu_ != nullptr) {
8             preinteg_->Integrate(*imu, imu->timestamp_ - last_imu_->timestamp_);
9         }
10
11         last_imu_ = imu;
12         imu_states_.emplace_back(preinteg_->Predict(last_nav_state_, imu_init_.
13             GetGravity()));
14     }

```

During registration, the predicted pose from preintegration is used as input to the NDT odometer, while the NDT output serves as observations for optimization:

Listing 8.7: src/ch8/lio-preinteg/lio_preinteg.cc

```

1 void LioPreinteg::Align() {
2     LOG(INFO) << "==== frame " << frame_num_;
3     ndt_.SetSource(current_scan_filter_);
4
5     current_nav_state_ = preinteg_->Predict(last_nav_state_, imu_init_.GetGravity());
6     ndt_pose_ = current_nav_state_.GetSE3();
7
8     ndt_.AlignNdt(ndt_pose_);
9
10    Optimize();
11}
12
13 void LioPreinteg::Optimize() {
14     using BlockSolverType = g2o::BlockSolverX;
15     using LinearSolverType = g2o::LinearSolverEigen<BlockSolverType::PoseMatrixType>;

```

```

16
17     auto *solver = new g2o::OptimizationAlgorithmLevenberg(
18         g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>()));
19     g2o::SparseOptimizer optimizer;
20     optimizer.setAlgorithm(solver);
21
22     // Previous frame vertices: pose, v, bg, ba
23     auto v0_pose = new VertexPose();
24     v0_pose->setId(0);
25     v0_pose->setEstimate(last_nav_state_.GetSE3());
26     optimizer.addVertex(v0_pose);
27
28     auto v0_vel = new VertexVelocity();
29     v0_vel->setId(1);
30     v0_vel->setEstimate(last_nav_state_.v_);
31     optimizer.addVertex(v0_vel);
32
33     auto v0_bg = new VertexGyroBias();
34     v0_bg->setId(2);
35     v0_bg->setEstimate(last_nav_state_.bg_);
36     optimizer.addVertex(v0_bg);
37
38     auto v0_ba = new VertexAccBias();
39     v0_ba->setId(3);
40     v0_ba->setEstimate(last_nav_state_.ba_);
41     optimizer.addVertex(v0_ba);
42
43     // Current frame vertices: pose, v, bg, ba
44     auto v1_pose = new VertexPose();
45     v1_pose->setId(4);
46     v1_pose->setEstimate(ndt_pose_); // NDT pose as initial value
47     optimizer.addVertex(v1_pose);
48
49     auto v1_vel = new VertexVelocity();
50     v1_vel->setId(5);
51     v1_vel->setEstimate(current_nav_state_.v_);
52     optimizer.addVertex(v1_vel);
53
54     auto v1_bg = new VertexGyroBias();
55     v1_bg->setId(6);
56     v1_bg->setEstimate(current_nav_state_.bg_);
57     optimizer.addVertex(v1_bg);
58
59     auto v1_ba = new VertexAccBias();
60     v1_ba->setId(7);
61     v1_ba->setEstimate(current_nav_state_.ba_);
62     optimizer.addVertex(v1_ba);
63
64     // IMU factor
65     auto edge_inertial = new EdgeInertial(preinteg_, imu_init_.GetGravity());
66     edge_inertial->setVertex(0, v0_pose);
67     edge_inertial->setVertex(1, v0_vel);
68     edge_inertial->setVertex(2, v0_bg);
69     edge_inertial->setVertex(3, v0_ba);
70     edge_inertial->setVertex(4, v1_pose);
71     edge_inertial->setVertex(5, v1_vel);
72     auto *rk = new g2o::RobustKernelHuber();
73     rk->setDelta(200.0);
74     edge_inertial->setRobustKernel(rk);
75     optimizer.addEdge(edge_inertial);
76
77     // Bias random walk
78     auto *edge_gyro_rw = new EdgeGyroRW();
79     edge_gyro_rw->setVertex(0, v0_bg);
80     edge_gyro_rw->setVertex(1, v1_bg);
81     edge_gyro_rw->setInformation(options_.bg_rw_info_);
82     optimizer.addEdge(edge_gyro_rw);
83
84     auto *edge_acc_rw = new EdgeAccRW();
85     edge_acc_rw->setVertex(0, v0_ba);
86     edge_acc_rw->setVertex(1, v1_ba);
87     edge_acc_rw->setInformation(options_.ba_rw_info_);
88     optimizer.addEdge(edge_acc_rw);
89

```

```

90 // Prior for previous frame pose, vel, bg, ba
91 auto *edge_prior = new EdgePriorPoseNavState(last_nav_state_, prior_info_);
92 edge_prior->setVertex(0, v0_pose);
93 edge_prior->setVertex(1, v0_vel);
94 edge_prior->setVertex(2, v0_bg);
95 edge_prior->setVertex(3, v0_ba);
96 optimizer.addEdge(edge_prior);
97
98 /// NDT pose observation
99 auto *edge_ndt = new EdgeGNSS(v1_pose, ndt_pose_);
100 edge_ndt->setInformation(options_.ndt_info_);
101 optimizer.addEdge(edge_ndt);
102
103 // Optimization
104 optimizer.setVerbose(options_.verbose_);
105 optimizer.initializeOptimization();
106 optimizer.optimize(20);
107
108 // Get results
109 last_nav_state_.R_ = v0_pose->estimate().so3();
110 last_nav_state_.p_ = v0_pose->estimate().translation();
111 last_nav_state_.v_ = v0_vel->estimate();
112 last_nav_state_.bg_ = v0_bg->estimate();
113 last_nav_state_.ba_ = v0_ba->estimate();
114
115 current_nav_state_.R_ = v1_pose->estimate().so3();
116 current_nav_state_.p_ = v1_pose->estimate().translation();
117 current_nav_state_.v_ = v1_vel->estimate();
118 current_nav_state_.bg_ = v1_bg->estimate();
119 current_nav_state_.ba_ = v1_ba->estimate();
120
121 // Reset preintegration
122
123 options_.preinteg_options_.init_bg_ = current_nav_state_.bg_;
124 options_.preinteg_options_.init_ba_ = current_nav_state_.ba_;
125 preinteg_ = std::make_shared<IMUPreintegration>(options_.preinteg_options_);
126
127 // Compute prior for current frame
128 // Build Hessian
129 // 15x2, order: v0_pose, v0_vel, v0_bg, v0_ba, v1_pose, v1_vel, v1_bg, v1_ba
130 //          0      6      9     12     15     21     24     27
131 Eigen::Matrix<double> H(30, 30);
132 H.setZero();
133
134 H.block<24, 24>(0, 0) += edge_inertial->GetHessian();
135
136 Eigen::Matrix<double> Hgr = edge_gyro_rw->GetHessian();
137 H.block<3, 3>(9, 9) += Hgr.block<3, 3>(0, 0);
138 H.block<3, 3>(9, 24) += Hgr.block<3, 3>(0, 3);
139 H.block<3, 3>(24, 9) += Hgr.block<3, 3>(3, 0);
140 H.block<3, 3>(24, 24) += Hgr.block<3, 3>(3, 3);
141
142 Eigen::Matrix<double> Har = edge_acc_rw->GetHessian();
143 H.block<3, 3>(12, 12) += Har.block<3, 3>(0, 0);
144 H.block<3, 3>(12, 27) += Har.block<3, 3>(0, 3);
145 H.block<3, 3>(27, 12) += Har.block<3, 3>(3, 0);
146 H.block<3, 3>(27, 27) += Har.block<3, 3>(3, 3);
147
148 H.block<15, 15>(0, 0) += edge_prior->GetHessian();
149 H.block<6, 6>(15, 15) += edge_ndt->GetHessian();
150
151 H = math::Marginalize(H, 0, 14);
152 prior_info_ = H.block<15, 15>(15, 15);
153
154 if (options_.verbose_) {
155     LOG(INFO) << "info trace: " << prior_info_.trace();
156     LOG(INFO) << "optimization done.";
157 }
158
159 NormalizeVelocity();
160 last_nav_state_ = current_nav_state_;
161 }
```

Most of the code here is dedicated to constructing the graph optimization structure, which

corresponds to Figure 8-3. In our implementation, the graph optimization problem for two keyframes consists of 8 vertices, 1 preintegration edge, 2 random walk edges, 1 NDT posterior pose observation, and 1 prior constraint. We didn't directly formulate the NDT observation in graph optimization form because NDT requires updating nearest neighbors during iteration, which isn't directly supported by g2o. After each optimization, we marginalize the previous frame's state. Since g2o doesn't directly support marginalization, we manually assemble the complete Hessian matrix using each edge's Jacobian matrices, which are computed during linearization.

Listing 8.8: src/common/g2o_types.h

```

1 class EdgePriorPoseNavState : public g2o::BaseMultiEdge<15, Vec15d> {
2     public:
3         void computeError();
4         virtual void linearizeOplus();
5
6         Eigen::Matrix<double, 15, 15> GetHessian() {
7             linearizeOplus();
8             Eigen::Matrix<double, 15, 15> J;
9             J.block<15, 6>(0, 0) = -_jacobianOplus[0];
10            J.block<15, 3>(0, 6) = -_jacobianOplus[1];
11            J.block<15, 3>(0, 9) = -_jacobianOplus[2];
12            J.block<15, 3>(0, 12) = -_jacobianOplus[3];
13            return J.transpose() * information() * J;
14        }
15
16        NavState state_;
17    };

```

Finally, we process the \mathbf{H} matrix with the `math::Marginalize` function to marginalize rows 0-14, obtaining the information matrix (15×15 dimension) for the next frame's state. This matrix is then used as prior information in the next optimization problem. Running the `test_li_o_preinteg` program displays the LIO's real-time point cloud, as shown in Figure 8-5.



Figure 8-5: Preintegration-based LIO system

The preintegration optimization demonstrated in this section primarily operates between consecutive LiDAR scans. Readers may modify it to accumulate over longer periods to better highlight differences between preintegration and filtering methods, which we leave as an exercise.

8.5 Summary

This section introduced the tightly-coupled LIO system to the readers. We presented two approaches: the iterative Kalman filter-based method and the pre-integration nonlinear optimization-based method. Overall, both methods work effectively. There is no significant difference in the point cloud quality during normal operation, but the actual debugging difficulty varies. The implementation of the pre-integration system is noticeably more complex. If desired, each NDT registration can be configured as a graph optimization factor to achieve a truly tightly-coupled system, but that would also entail more extensive debugging. We hope readers can gain an understanding of the frontend and backend workings of a tightly-coupled system through the content of this chapter.

Exercises

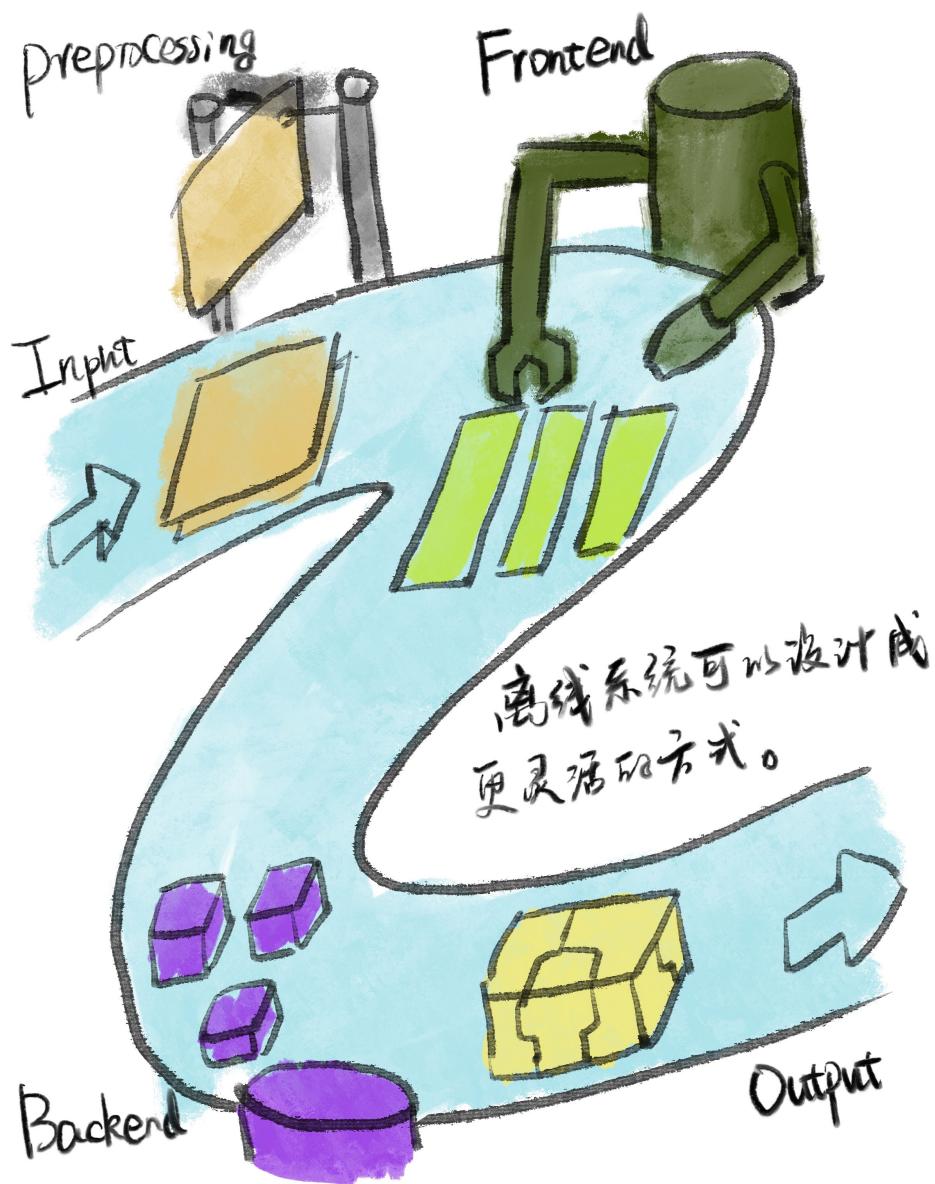
1. Referring to the derivation of NDT-based LIO, derive the Kalman update formula for LIO based on point-to-plane residuals, and explain the relationship between point-to-plane ICP and LIO.
2. Investigate whether the matrices computed by NDT in NDT LIO contain a significant number of zero blocks. If the goal is to reduce the matrix size, can these zero blocks be removed, retaining only the non-zero blocks?
3. In the pre-integration LIO system, increase the pre-integration time—for example, marginalize only after more than 3 seconds instead of after processing each frame.
4. *Incorporate NDT residuals into the pre-integration system to implement an LIO system.

Chapter 9

Mapping for Autonomous Vehicles

This chapter introduces readers to the complete technology stack for point cloud mapping and high-definition (HD) maps in autonomous driving. Full point cloud mapping can be viewed as a comprehensive optimization problem involving RTK, IMU, wheel odometry, and LiDAR. Most L4-level autonomous driving tasks require a complete, RTK-aligned point cloud map for tasks such as map annotation and high-precision localization. We will progressively cover the entire process of point cloud mapping: frontend, backend, loop closure detection, map segmentation, and export. These steps are largely similar across most applications.

In this chapter, we will work with readers to implement a complete point cloud mapping system, where some components can be straightforwardly assembled using algorithmic modules from previous chapters. This system can start from offline data packets, construct, and export point cloud maps that are practically usable for high-precision localization. We will also briefly introduce some vector map-related content. However, since the annotation and generation of vector HD maps are not closely related to SLAM algorithms, we will not require readers to perform map annotation tasks themselves. The next chapter of this book will also use the point cloud maps constructed in this chapter for actual fusion-based localization algorithm testing.



9.1 Point Cloud Mapping Pipeline

Unlike online SLAM systems, mapping systems can operate entirely in offline mode. A major advantage of offline systems is their strong determinism. Each algorithmic step—how it should be executed and what output it should produce—can be planned and determined in advance. There are no scheduling issues related to threads or resources between modules, whereas online systems often need to consider inter-thread dependencies, such as whether to insert new submaps before loop closure detection completes or whether to allow exporting maps with unclosed loops. Therefore, designing a mapping framework as an offline system is easier than developing a real-time SLAM system and enables better automation capabilities.

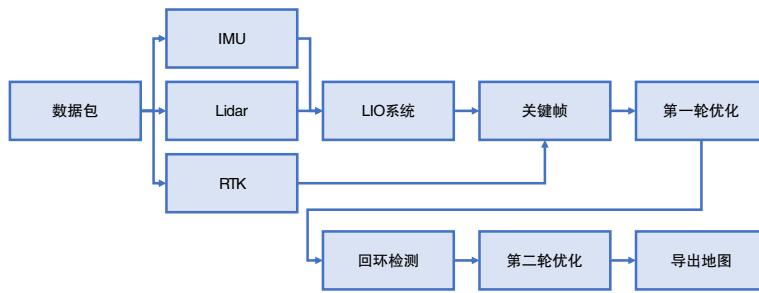


Figure 9-1: The workflow framework of the mapping system

We design the entire mapping pipeline following the approach in [179], as shown in Fig. 9-1.

1. First, we extract IMU, RTK, and LiDAR data from the given ROS bag. Due to dataset variability, not all datasets include wheel odometry (and wheel odometry formats often vary by vehicle). Thus, we primarily use IMU and LiDAR data to construct the LIO system. This part directly utilizes the IEKF LIO code from the previous chapter.
2. Most autonomous vehicles are also equipped with RTK devices or integrated navigation systems. These devices provide the vehicle's position in the physical world but may suffer from signal interference. In the LIO system, we collect point cloud keyframes at certain intervals and assign each keyframe an RTK pose (as an observation) based on RTK or integrated navigation data. This book primarily uses the NCLT dataset (Fig. 9-2) as an example, where the RTK signal follows a single-antenna solution, providing only positional data without orientation.
3. Next, we use LIO for inter-frame motion estimation and RTK poses as absolute coordinate observations to optimize the entire trajectory and determine the validity of RTK for each keyframe. This is called the **first-round optimization**. If RTK functions normally, we obtain a trajectory roughly aligned with RTK. However, in real-world environments, RTK often has invalid observations, which must be identified in the algorithm.
4. Based on the previous step, we perform loop closure detection on the map. The detection algorithm can simply use Euclidean distance-based checks, with NDT or common registration algorithms computing their relative poses. This example employs multi-resolution NDT matching for loop closure detection.

5. Finally, we integrate all this information into a unified pose graph for optimization, eliminating ghosting caused by accumulated errors and removing regions where RTK fails. This is called the **second-round optimization**. Once the poses are finalized, we export the point cloud map accordingly. For easier map loading and visualization, we also segment the point cloud map into tiles. The processed point cloud can then be used for high-precision localization or map annotation.



Figure 9-2: The vehicle used in the NCLT dataset, image from [11]

9.2 Frontend Implementation

First, we implement the frontend. Since previous chapters have already covered a complete LIO system, we only need to add some peripheral logic for data packet parsing. The main frontend code is located in:

Listing 9.1: src/ch9/frontend.cc

```

1 void Frontend::Run() {
2     sad::RosbagIO rosbag_io(bag_path_, DatasetType::NCLT);
3
4     // First extract RTK poses (note NCLT only has translation components)
5     rosbag_io
6         .AddAutoRTKHandle([this](GNSSPtr gnss) {
7             gnss_.emplace(gnss->unix_time_, gnss);
8             return true;
9         })
10        .Go();
11     rosbag_io.CleanProcessFunc(); // No longer need to process RTK
12     RemoveMapOrigin();
13
14     // Then run LIO
15     rosbag_io
16         .AddAutoPointCloudHandle([&](sensor_msgs::PointCloud2::Ptr cloud) -> bool {
17             lio_->PCLCallBack(cloud);
18             ExtractKeyFrame(lio_->GetCurrentState());
19             return true;
20         })
21         .AddImuHandle([&](IMUPtr imu) {
22             lio_->IMUCallBack(imu);
23             return true;
24         })
25         .Go();
26     lio_->Finish();
27
28     // Save running results
29     SaveKeyframes();
30
31     LOG(INFO) << "done.";
32 }
```

As can be seen, the frontend mainly performs the following tasks:

1. Extracts RTK data from the ROS bag and stores it in an RTK message queue, sorted by acquisition time.
2. Uses the first valid RTK data as the map origin, then subtracts this origin from other RTK readings to use them as RTK position observations.
3. Runs LIO using IMU and LiDAR data, extracts keyframes based on distance and angle thresholds, and finally saves the keyframe results.

9.3 Keyframe Extraction

When extracting keyframes, we also retrieve their poses and point clouds from the LIO system:

Listing 9.2: src/ch9/frontend.cc

```

1 void Frontend::ExtractKeyFrame(const sad::NavStated& state) {
2     if (last_kf_ == nullptr) {
3         // First frame
4         auto kf = std::make_shared<Keyframe>(state.timestamp_, kf_id_++, state.GetSE3(),
5                                             lio_->GetCurrentScan());
6         FindGPSPose(kf);
7         kf->SaveAndUnloadScan("./data/ch9/");
8         keyframes_.emplace(kf->id_, kf);
9         last_kf_ = kf;
10    } else {
11        // Calculate motion thresholds between current state and last keyframe
12        SE3 delta = last_kf_->lidar_pose_.inverse() * state.GetSE3();
13        if (delta.translation().norm() > kf_dis_th_ || delta.so3().log().norm() >
14            kf_ang_th_deg_ * math::kDEG2RAD) {
15            auto kf = std::make_shared<Keyframe>(state.timestamp_, kf_id_++, state.GetSE3(),
16                                              lio_->GetCurrentScan());
17            FindGPSPose(kf);
18            keyframes_.emplace(kf->id_, kf);
19            kf->SaveAndUnloadScan("./data/ch9/");
20            LOG(INFO) << "Generated keyframe" << kf->id_;
21            last_kf_ = kf;
22        }
23    }
}

```

These point clouds are sequentially stored in the data/ch9/ directory and then cleared from memory. Meanwhile, we query the RTK pose queue by timestamp, calling a generic pose interpolation function from the math library:

Listing 9.3: src/ch9/frontend.cc

```

1 void Frontend::FindGPSPose(std::shared_ptr<Keyframe> kf) {
2     SE3 pose;
3     GNSSPtr match;
4     if (math::PoseInterp<GNSSPtr>(
5         kf->timestamp_, gnss_, [](const GNSSPtr& gnss) -> SE3 { return gnss->utm_pose_; },
6         pose, match)) {
7         kf->rtk_pose_ = pose;
8         kf->rtk_valid_ = true;
9     } else {
10        kf->rtk_valid_ = false;
11    }
12 /**
13 * Pose interpolation algorithm
14 * @param T Data type
15 * @param query_time Query time
16 */

```

```

17 * @param data Data container
18 * @param take_pose_func Pose extraction predicate
19 * @param result Interpolation result
20 * @param best_match_iter Closest matched element
21 *
22 * NOTE: query_time must be within data's time range (no extrapolation)
23 * Data map must be time-sorted
24 * @return Whether interpolation succeeded
25 */
26 template <typename T>
27 bool PoseInterp(double query_time, const std::map<double, T>& data, const std::function<SE3(const T&)>& take_pose_func,
28 SE3& result, T& best_match) {
29     if (data.empty()) {
30         LOG(INFO) << "data is empty";
31         return false;
32     }
33
34     if (query_time > data.rbegin()>first) {
35         LOG(INFO) << "query time is later than last, " << std::setprecision(18) << ",
36         query: " << query_time
37         << ", end time: " << data.rbegin()>first;
38
39         return false;
40     }
41
42     auto match_iter = data.begin();
43     for (auto iter = data.begin(); iter != data.end(); ++iter) {
44         auto next_iter = iter;
45         next_iter++;
46
47         if (iter->first < query_time && next_iter->first >= query_time) {
48             match_iter = iter;
49             break;
50         }
51
52     auto match_iter_n = match_iter;
53     match_iter_n++;
54     assert(match_iter_n != data.end());
55
56     double dt = match_iter_n->first - match_iter->first;
57     double s = (query_time - match_iter->first) / dt; // s=0: first frame, s=1: next
58     // frame
59
60     SE3 pose_first = take_pose_func(match_iter->second);
61     SE3 pose_next = take_pose_func(match_iter_n->second);
62     result = {pose_first.unit_quaternion().slerp(s, pose_next.unit_quaternion()),
63               pose_first.translation() * (1 - s) + pose_next.translation() * s};
64     best_match = s < 0.5 ? match_iter->second : match_iter_n->second;
65     return true;
66 }
```

This function can retrieve poses from any map structure (requiring user-provided pose extraction method) and perform interpolation. For RTK readings, we simply return its `utm_pose`. Thus, we obtain the corresponding LIO pose, RTK pose, and scanned point cloud for each keyframe. All this data is stored in the `data/ch9/` directory for subsequent algorithm modules.

Now please compile and run the frontend test program:

Listing 9.4: Terminal command:

```
bin/run_frontend --config_yaml ./config/mapping.yaml
```

This program reads basic information like data directory from the YAML configuration file and runs the entire frontend. The computation may take some time. Upon completion, you should see a series of PCD files and a `keyframes.txt` pose data file in `./data/ch9/`, with content similar to:

Listing 9.5: keyframes.txt

```

1 0 1326652772.63223195 0 1 1 0.00323679756335721767 -0.0144500186628650669
  0.00648581949310997503 0.00102604343126653629 -0.000610794694837971455
  -0.00292974916442335183 0.999994995354752558 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
  0 0 1
2 1 1326652810.69380093 0 1 1 -0.00878834742957286877 0.00258137652733943018
  -0.0370536530158143834 0.00556325961406906912 -0.000227692342581170451
  0.0937748334061895006 0.995577861806049347 -0.00207286058563104208
  0.00403151070086457675 0.00941915643412104611 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
3 2 1326652811.20330048 0 1 1 0.00422691408508832616 -0.0517418751056824486
  -0.0160935510000861509 0.0231508342749745313 -0.00193957036575541104
  0.191042271176856737 0.981306846792967979 -0.00323174750160585156
  0.0129979040447824497 0.0161721026594033174 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1
4 3 1326652813.51224709 0 1 1 0.401907347299137463 -0.95978654845477433
  -0.0772027821235120038 0.108749752190681712 0.0143241272304329356
  0.187688702845765748 0.976084659034054059 -0.0201519057154655457
  0.305439419113099575 0.0159999999999627107 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1
5 4 1326652814.32679105 0 1 1 0.764588972100686437 -1.91970590325136792
  -0.0977756250833726193 0.0454533237407165613 -0.0122335876446319734
  0.187788495604154393 0.981080942436958869 -0.268586141930427402
  0.693511614575982205 0.0269999999999868123 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1

```

This file records various poses (RTK, LiDAR odometry, backend-optimized) for each keyframe. Currently only RTK and LiDAR odometry poses are valid since we've only run the frontend. Later we'll use this information for loop closure detection and pose optimization. Now you can merge these keyframe point clouds into a complete map using the frontend-estimated poses:

Listing 9.6: Terminal command:

```
bin/dump_map --pose_source=lidar
```

Readers can examine `dump_map.cc` to see how map merging works. This command generates a `map.pcd` file in `./data/ch9/`, which can be viewed using `pcl_viewer` as shown in Fig 9-3. The frontend structure appears locally accurate but inevitably exhibits accumulated errors, with noticeable ghosting in repeatedly visited areas. Don't worry - subsequent fusion optimization and loop closure detection will naturally eliminate these artifacts.

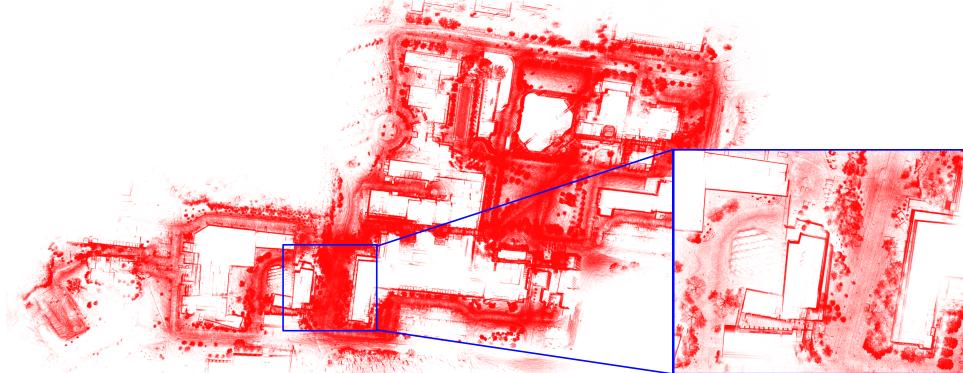


Figure 9-3: NCLT dataset visualization using frontend-estimated poses, showing noticeable ghosting in boxed areas.

The subsequent mapping pipeline can start from the frontend results without reprocessing ROS bags. Compared to raw data, keyframe data is significantly smaller due to distance/angle subsampling (55GB raw NCLT data vs 2GB keyframe data). Next we'll perform joint optimization using RTK and LIO poses, followed by loop closure detection to eliminate accumulated errors.

9.4 Backend Pose Graph Optimization and Outlier Detection

First, let's examine the backend pose graph optimization algorithm. We want the final optimized trajectory to incorporate inputs from various sensors, so the backend graph optimization needs to include these factors:

- RTK factors. RTK factors provide absolute pose observations. If the dataset includes attitude information, we incorporate it. If only translation is available, we only build translation-related constraints. As discussed in Chapter 3, when RTK has extrinsic parameters, its translation observations need transformation before correctly affecting vehicle displacement. This is implemented in our code.
- LiDAR odometry information. We use the previously introduced LO or LIO as local continuity constraints.
- Loop closure factors. If loop closure detection is performed, we load the relevant information and incorporate it into the graph optimization.

These three factors constitute our fundamental pose graph optimization problem. We implement their definitions within the g2o framework:

Listing 9.7: src/common/g2o_types.h

```

1 /**
2 * 3-DOF GNSS constraint
3 */
4 class EdgeGNSSTransOnly : public g2o::BaseUnaryEdge<3, Vec3d, VertexPose> {
5     public:
6         EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
7         EdgeGNSSTransOnly() = default;
8         EdgeGNSSTransOnly(VertexPose* v, const Vec3d& obs, const SE3& TBG = SE3()) {
9             setVertex(0, v);
10            setMeasurement(obs);
11            TBG_ = TBG;
12        }
13
14        void computeError() override {
15            VertexPose* v = (VertexPose*)_vertices[0];
16            _error = (v->estimate() * TBG_).translation() - _measurement;
17        };
18
19        void linearizeOplus() override {
20            // jacobian 3x6
21            VertexPose* v = (VertexPose*)_vertices[0];
22            SE3 TWB = v->estimate();
23            _jacobianOplusXi.setZero();
24            _jacobianOplusXi.block<3, 3>(0, 0) = -TWB.so3().matrix() * SO3::hat(TBG_.
25                translation());
26            _jacobianOplusXi.block<3, 3>(0, 3) = Mat3d::Identity();
27        }
28
29    private:
30        SE3 TBG_;
31    };
32
33 /**
34 * 6-DOF relative motion constraint
35 * Error order: translation first, then rotation
36 * Observation: T12
37 */
38 class EdgeRelativeMotion : public g2o::BaseBinaryEdge<6, SE3, VertexPose, VertexPose> {
39     public:
40         EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
41         EdgeRelativeMotion() = default;

```

```

41     EdgeRelativeMotion(VertexPose* v1, VertexPose* v2, const SE3& obs) {
42         setVertex(0, v1);
43         setVertex(1, v2);
44         setMeasurement(obs);
45     }
46
47     void computeError() override {
48         VertexPose* v1 = (_VertexPose*)_vertices[0];
49         VertexPose* v2 = (_VertexPose*)_vertices[1];
50         SE3 T12 = v1->estimate().inverse() * v2->estimate();
51         _error = (_measurement.inverse() * v1->estimate().inverse() * v2->estimate()).log
52             ();
53     };
54 }
```

Here, `VertexPose` represents the pose vertex, `EdgeGNSSTransOnly` handles single-antenna GNSS observations (including GNSS extrinsic parameters), while LiDAR and loop closure observations are described by the relative motion constraint `EdgeRelativeMotion`. Before optimization, we construct these vertices and edges, set appropriate weights and kernel functions, and then place them in the optimizer.

The relevant code for building the optimization problem is:

Listing 9.8: `src/ch9/optimization.cc`

```

1 void Optimization::BuildProblem() {
2     using BlockSolverType = g2o::BlockSolverX;
3     using LinearSolverType = g2o::LinearSolverEigen<BlockSolverType::PoseMatrixType>;
4     auto* solver = new g2o::OptimizationAlgorithmLevenberg(
5         g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>()));
6
7     optimizer_.setAlgorithm(solver);
8
9     AddVertices();
10    AddRTKEdges();
11    AddLidarEdges();
12    AddLoopEdges();
13 }
```

We won't detail every function for adding vertices and edges in the book - readers can refer to the corresponding code in the repository. After building the complete problem, we first optimize with robust kernels, then remove the kernels from outliers and optimize again. This helps eliminate the impact of outliers. Additionally, if RTK contains no attitude information throughout, the LiDAR odometry trajectory might differ from the RTK trajectory by a rotation. We perform an ICP on the trajectory to estimate this rotation.

Readers can execute an optimization round by running:

Listing 9.9: Terminal command:

```
bin/run_optimization --stage=1
```

The optimization results are written to `keyframes.txt` and can be visualized using:

Listing 9.10: Terminal command:

```
python3 scripts/all_path.py ./data/ch9/keyframes.txt
```

As shown in Fig 9-4, after the first optimization round, the RTK trajectory largely coincides with the optimized trajectory, and some failure regions are well identified. However, point clouds may still show ghosting since we haven't performed loop closure detection yet.

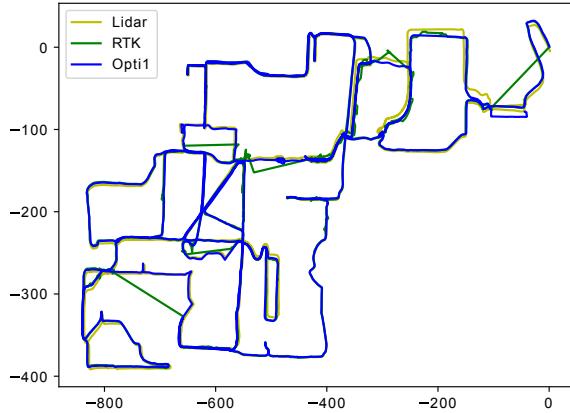


Figure 9-4: Trajectories after the first optimization round.

9.5 Loop Closure Detection

The quality of the point cloud map largely depends on thorough loop closure detection. Without loop closure detection, the shape of the point cloud would primarily rely on the state of the LiDAR odometry. However, LiDAR odometry algorithms only process point clouds from consecutive timestamps, which means that while our map point cloud remains correct under continuous motion, observations from different times and locations are not well-registered. In the loop closure detection step, we aim to design an algorithmic pipeline that enables the program to thoroughly register unaligned regions. Of course, the specific implementation of loop closure detection is flexible—the approach introduced in the book is straightforward and easy to teach. Moreover, since the system operates offline, we can check all relevant regions in one go without waiting for frontend construction results as in real-time operation. During matching, we can also leverage parallel mechanisms to accelerate the loop closure detection process.

We design the loop closure detection steps as follows:

1. First, we traverse the keyframes in the first-round optimized trajectory. For each pair of keyframes that are spatially close but temporally distant, we perform a loop closure detection. We refer to such a keyframe pair as a **checkpoint**. To prevent an excessive number of checkpoints, we set an ID interval, i.e., skipping a fixed number of keyframes between checks. In experiments, this interval is set to 5.
2. For each keyframe pair, we use a **scan-to-map** registration method: extracting a subset of the point cloud near the keyframe as a submap, then registering the scan to this submap. This approach mitigates issues arising from insufficient point density or texture in single scans, though it comes with higher computational costs.
3. Since the computation for each checkpoint is independent, we employ concurrent programming to execute step 2 in parallel, speeding up the process.
4. The actual registration is performed using NDT (Normal Distributions Transform), and we use the NDT score to determine the validity of the loop closure. Unlike odometry methods, loop closure registration often deals with poor initial pose estimates, so we prefer algorithms less reliant on initial guesses. Thus, we enhance the NDT

method with a coarse-to-fine registration process, similar to the multi-resolution 2D registration introduced in Chapter 6.

5. Finally, we record all successful loop closures and read these results during the next optimization call.

The loop closure detection result is described by a loop candidate pair:

Listing 9.11: src/ch9/loopclosure.h

```

1 /**
2 * Loop closure candidate frame
3 */
4 struct LoopCandidate {
5     LoopCandidate() {}
6     LoopCandidate(IdType id1, IdType id2, SE3 Tij) : idx1_(id1), idx2_(id2), Tij_(Tij)
7         {}
8     IdType idx1_ = 0;
9     IdType idx2_ = 0;
10    SE3 Tij_;
11    double ndt_score_ = 0.0;
12};

```

The detection algorithm follows a simple "detect-then-compute" approach:

Listing 9.12: src/ch9/loopclosure.cc

```

1 void LoopClosure::Run() {
2     DetectLoopCandidates();
3     ComputeLoopCandidates();
4
5     SaveResults();
6 }
7
8 void LoopClosure::DetectLoopCandidates() {
9     KFPtr check_first = nullptr;
10    KFPtr check_second = nullptr;
11
12    LOG(INFO) << "detecting loop candidates from pose in stage 1";
13
14    // Essentially a nested loop
15    for (auto iter_first = keyframes_.begin(); iter_first != keyframes_.end(); ++iter_first) {
16        auto kf_first = iter_first->second;
17
18        if (check_first != nullptr && abs(int(kf_first->id_) - int(check_first->id_)) <=
19            skip_id_) {
20            // Keyframe IDs are too close
21            continue;
22        }
23
24        for (auto iter_second = iter_first; iter_second != keyframes_.end(); ++iter_second)
25        {
26            auto kf_second = iter_second->second;
27
28            if (check_second != nullptr && abs(int(kf_second->id_) - int(check_second->id_))
29                <= skip_id_) {
30                // Keyframe IDs are too close
31                continue;
32            }
33
34            if (abs(int(kf_first->id_) - int(kf_second->id_)) < min_id_interval_) {
35                /// If too close on the same trajectory, skip loop closure
36                continue;
37            }
38
39            Vec3d dt = kf_first->opti_pose_1_.translation() - kf_second->opti_pose_1_.
40                translation();
41            double t2d = dt.head<2>().norm(); // x-y distance
42            double range_th = min_distance_;
43        }
44    }
45
46    Vec3d dt = kf_first->opti_pose_1_.translation() - kf_second->opti_pose_1_.
47        translation();
48    double t2d = dt.head<2>().norm(); // x-y distance
49    double range_th = min_distance_;
50}

```

```

40     if (t2d < range_th) {
41         LoopCandidate c(kf_first->id_, kf_second->id_,
42                           kf_first->opti_pose_1_.inverse() * kf_second->opti_pose_1_);
43         loop_candidates_.emplace_back(c);
44         check_first = kf_first;
45         check_second = kf_second;
46     }
47 }
48 }
49 LOG(INFO) << "detected candidates: " << loop_candidates_.size();
50 }
51
52 void LoopClosure::ComputeLoopCandidates() {
53     // Execute computation
54     std::for_each(std::execution::par_unseq, loop_candidates_.begin(), loop_candidates_.end(),
55                  [this](LoopCandidate& c) { ComputeForCandidate(c); });
56     // Save successful candidates
57     std::vector<LoopCandidate> succ_candidates;
58     for (const auto& lc : loop_candidates_) {
59         if (lc.ndt_score_ > ndt_score_th_) {
60             succ_candidates.emplace_back(lc);
61         }
62     }
63     LOG(INFO) << "success: " << succ_candidates.size() << "/" << loop_candidates_.size();
64     loop_candidates_.swap(succ_candidates);
65 }
66

```

The actual computation for loop closure is implemented in ‘ComputeForCandidate’:

Listing 9.13: src/ch9/loopclosure.cc

```

1 void LoopClosure::ComputeForCandidate(sad::LoopCandidate& c) {
2     LOG(INFO) << "aligning " << c.idx1_ << " with " << c.idx2_;
3     const int submap_idx_range = 40;
4     KFPtr kf1 = keyframes_.at(c.idx1_), kf2 = keyframes_.at(c.idx2_);
5
6     auto build_submap = [this](int given_id, bool build_in_world) -> CloudPtr {
7         CloudPtr submap(new PointCloudType);
8         for (int idx = -submap_idx_range; idx < submap_idx_range; idx += 4) {
9             int id = idx + given_id;
10            if (id < 0) {
11                continue;
12            }
13            auto iter = keyframes_.find(id);
14            if (iter == keyframes_.end()) {
15                continue;
16            }
17
18            auto kf = iter->second;
19            CloudPtr cloud(new PointCloudType);
20            pcl::io::loadPCDFile("./data/ch9/" + std::to_string(id) + ".pcd", *cloud);
21            RemoveGround(cloud, 0.1);
22
23            if (cloud->empty()) {
24                continue;
25            }
26
27            // Transform to world frame
28            SE3 Twb = kf->opti_pose_1_;
29
30            if (!build_in_world) {
31                Twb = keyframes_.at(given_id)->opti_pose_1_.inverse() * Twb;
32            }
33
34            CloudPtr cloud_trans(new PointCloudType);
35            pcl::transformPointCloud(*cloud, *cloud_trans, Twb.matrix());
36
37            *submap += *cloud_trans;
38        }
39        return submap;
40    };

```

```

42 auto submap_kf1 = build_submap(kf1->id_, true);
43 kf2->cloud_.reset(new PointCloudType);
44 pcl::io::loadPCDFile("./data/ch9/" + std::to_string(kf2->id_) + ".pcd", *kf2->cloud_);
45 auto submap_kf2 = kf2->cloud_;
46
47 if (submap_kf1->empty() || submap_kf2->empty()) {
48     c.ndt_score_ = 0;
49     return;
50 }
51
52 pcl::NormalDistributionsTransform<PointType, PointType> ndt;
53
54 ndt.setResolution(10.0);
55 ndt.setTransformationEpsilon(0.05);
56 ndt.setStepSize(0.7);
57 ndt.setMaximumIterations(40);
58
59 Mat4f Tw2 = kf2->opti_pose_1_.matrix().cast<float>();
60
61 /// Multi-resolution matching
62 CloudPtr output(new PointCloudType);
63 std::vector<double> res{10.0, 5.0, 4.0, 3.0};
64 for (auto& r : res) {
65     ndt.setResolution(r);
66     auto rough_map1 = VoxelCloud(submap_kf1, r * 0.1);
67     auto rough_map2 = VoxelCloud(submap_kf2, r * 0.1);
68     ndt.setInputTarget(rough_map1);
69     ndt.setInputSource(rough_map2);
70
71     ndt.align(*output, Tw2);
72     Tw2 = ndt.getFinalTransformation();
73 }
74
75 Mat4d T = Tw2.cast<double>();
76 Quatd q(T.block<3, 3>(0, 0));
77 q.normalize();
78 Vec3d t = T.block<3, 1>(0, 3);
79 c.Tij_ = kf1->opti_pose_1_.inverse() * SE3(q, t);
80 c.ndt_score_ = ndt.getTransformationProbability();
81
82 }

```

As shown, we perform registration at grid resolutions of 10, 5, 4, and 3 meters respectively, feeding each registration result into the next iteration. The final score is determined using the 3-meter resolution result.

Now please compile and run the loop closure detection program:

Listing 9.14: Terminal command:

```
bin/run_loopeclosure
```

The loop closure detection will run for some time, with duration depending on the number of matches. Please wait patiently. After completion, we execute the second round of pose optimization, which will incorporate the loop closure results:

Listing 9.15: Terminal command:

```
bin/run_optimization --stage=2
```

The optimized poses will be saved to ‘./data/ch9/after.g2o’. We can visualize the pose graph with loop closures using ‘g2o_viewer’, as shown in Figure 9-5. Observe that correct loop closures are detected in most revisited areas.

Now export the point cloud after the second optimization round, which should show no significant ghosting artifacts:

Listing 9.16: Terminal command:

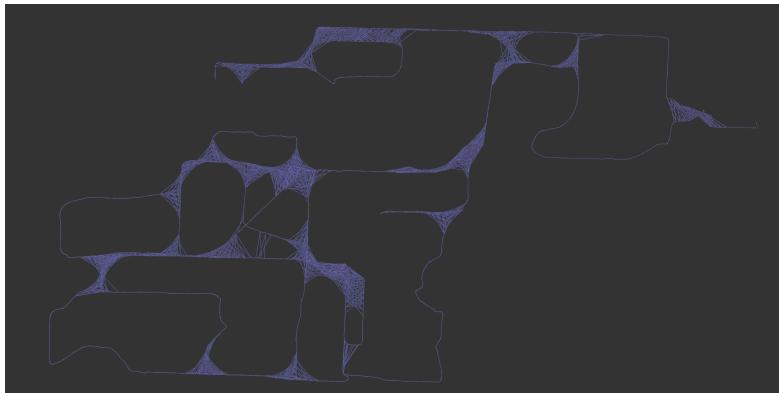


Figure 9-5: Pose graph with loop closures

```
/bin/dump_map --pose_source opti2
```



Figure 9-6: Optimized point cloud map after stage 2

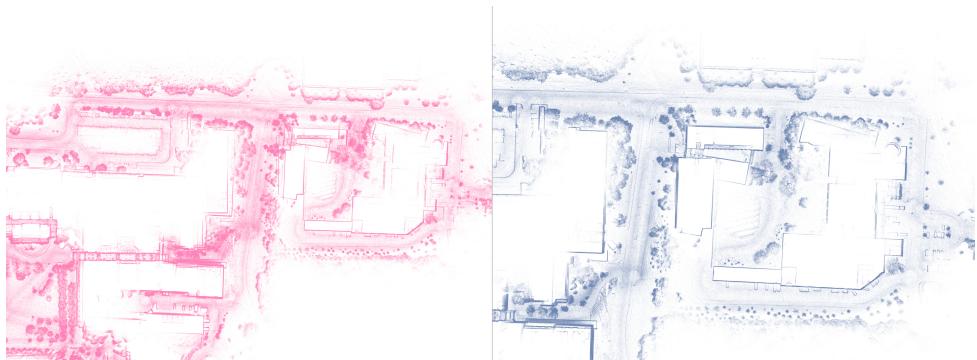


Figure 9-7: Local comparison between LIO point cloud and optimized point cloud. Left: LIO point cloud, Right: Optimized point cloud

The comparison between pre- and post-optimization point clouds is shown in Figure 9-7. Significant improvement in local details demonstrates that loop closure detection and pose optimization are highly effective components of the pipeline. In practice, loop closure range and registration parameters should be adjusted according to sensor accuracy, but for this dataset, the resulting point cloud is already sufficient for our purposes.

Here is the English translation with LaTeX commands preserved:

9.6 Map Export

Finally, we should export the complete point cloud map. While merging all point clouds into a single map is convenient for visualization, this approach is not ideal for real-time localization systems. In most applications, we want to control the loading scale of real-time point clouds—for example, only loading point clouds within a 200-meter radius around the vehicle and unloading distant regions as needed. This helps manage computational load in real-time systems. Below, we reorganize the point clouds generated in this chapter by splitting them into 100-meter grid blocks and design interfaces for block loading and unloading.

The splitting process essentially calculates the grid ID for each point based on its coordinates and assigns it to the corresponding grid. For more robust implementation, we could incorporate parallelization and batch read/write operations. However, since the dataset used in this chapter is relatively small, the benefits of concurrency are limited, so we demonstrate a single-threaded version here.

Listing 9.17: src/ch9/split_map.cc

```

1 int main(int argc, char** argv) {
2     std::map<IdType, KFPtr> keyframes;
3     if (!LoadKeyFrames("./data/ch9/keyframes.txt", keyframes)) {
4         LOG(ERROR) << "failed to load keyframes";
5         return 0;
6     }
7
8     std::map<Vec2i, CloudPtr, less_vec<2>> map_data; // Map data indexed by grid ID
9     pcl::VoxelGrid<PointType> voxel_grid_filter;
10    float resolution = FLAGS_voxel_size;
11    voxel_grid_filter.setLeafSize(resolution, resolution, resolution);
12
13    // Logic similar to dump_map: Find/create grid ID for each point
14    for (auto& kfp : keyframes) {
15        auto kf = kfp.second;
16        kf->LoadScan("./data/ch9/");
17
18        CloudPtr cloud_trans(new PointCloudType);
19        pcl::transformPointCloud(*kf->cloud_, *cloud_trans, kf->opti_pose_2_.matrix());
20
21        // Apply voxel filtering
22        CloudPtr kf_cloud_voxeled(new PointCloudType);
23        voxel_grid_filter.setInputCloud(cloud_trans);
24        voxel_grid_filter.filter(*kf_cloud_voxeled);
25
26        LOG(INFO) << "building kf " << kf->id_ << " in " << keyframes.size();
27
28        // Assign points to grids
29        for (const auto& pt : kf_cloud_voxeled->points) {
30            int gx = int((pt.x - 50.0) / 100);
31            int gy = int((pt.y - 50.0) / 100);
32            Vec2i key(gx, gy);
33            auto iter = map_data.find(key);
34            if (iter == map_data.end()) {
35                // Create new point cloud
36                CloudPtr cloud(new PointCloudType);
37                cloud->points.emplace_back(pt);
38                cloud->is_dense = false;
39                cloud->height = 1;
40                map_data.emplace(key, cloud);
41            } else {
42                iter->second->points.emplace_back(pt);
43            }
44        }
45    }
46}
```

```

41     } else {
42         iter->second->points.emplace_back(pt);
43     }
44 }
45 }
46
47 // Save point clouds and index file
48 LOG(INFO) << "saving maps, grids: " << map_data.size();
49 std::system("mkdir -p ./data/ch9/map_data/");
50 std::system("rm -rf ./data/ch9/map_data/*"); // Clear directory
51 std::ofstream fout("./data/ch9/map_data/map_index.txt");
52 for (auto& dp : map_data) {
53     fout << dp.first[0] << " " << dp.first[1] << std::endl;
54     dp.second->width = dp.second->size();
55     VoxelGrid(dp.second, 0.1);
56
57     pcl::io::savePCDFileBinaryCompressed(
58         "./data/ch9/map_data/" + std::to_string(dp.first[0]) + "_" + std::to_string(dp.
59             first[1]) + ".pcd",
60             *dp.second);
61 }
62 fout.close();
63 LOG(INFO) << "done.";
64 return 0;
65 }
```

After splitting, each map block is exported as a separate PCD file. An index file (map_index.txt) stores grid locations in text format for quick access. The split point clouds can still be visualized using pcl_viewer with color-coded blocks, as shown in Figure 9-8. This demonstrates a 100-meter grid splitting scheme, which readers may adapt to larger or smaller block sizes.



Figure 9-8: Split point cloud map

9.7 Summary

This chapter demonstrated a complete pipeline for point cloud mapping, optimization, and segmentation. The entire mapping process is highly streamlined and automated. By sequentially executing the programs described in this chapter, users can complete a full mapping procedure.

Listing 9.18: src/ch9/run_mapping.cc

```

1 int main(int argc, char** argv) {
2     google::InitGoogleLogging(argv[0]);
3     FLAGS_stderrthreshold = google::INFO;
```

```

4  FLAGS_colorlogtostderr = true;
5  google::ParseCommandLineFlags(&argc, &argv, true);
6
7  LOG(INFO) << "testing frontend";
8  sad::Frontend frontend(FLAGS_config_yaml);
9  if (!frontend.Init()) {
10    LOG(ERROR) << "failed to init frontend.";
11    return -1;
12  }
13
14  frontend.Run();
15
16  sad::Optimization opti(FLAGS_config_yaml);
17  if (!opti.Init(1)) {
18    LOG(ERROR) << "failed to init opti1.";
19    return -1;
20  }
21  opti.Run();
22
23  sad::LoopClosure lc(FLAGS_config_yaml);
24  if (lc.Init() == false) {
25    LOG(ERROR) << "failed to init loop closure.";
26    return -1;
27  }
28  lc.Run();
29
30  sad::Optimization opti2(FLAGS_config_yaml);
31  if (!opti2.Init(2)) {
32    LOG(ERROR) << "failed to init opti1.";
33    return -1;
34  }
35  opti2.Run();
36
37  LOG(INFO) << "done.";
38  return 0;
39
}

```

By simply specifying an NCLT dataset in the configuration file, the program automatically constructs a complete point cloud map with loop closure corrections. Readers are encouraged to test this pipeline on other NCLT datasets and compare their performance. These maps facilitate **high-precision LiDAR localization**, enabling vehicles and robots to achieve accurate pose estimation without RTK. In the next chapter, we will utilize the point cloud maps built here for real-time high-precision localization.

Exercises

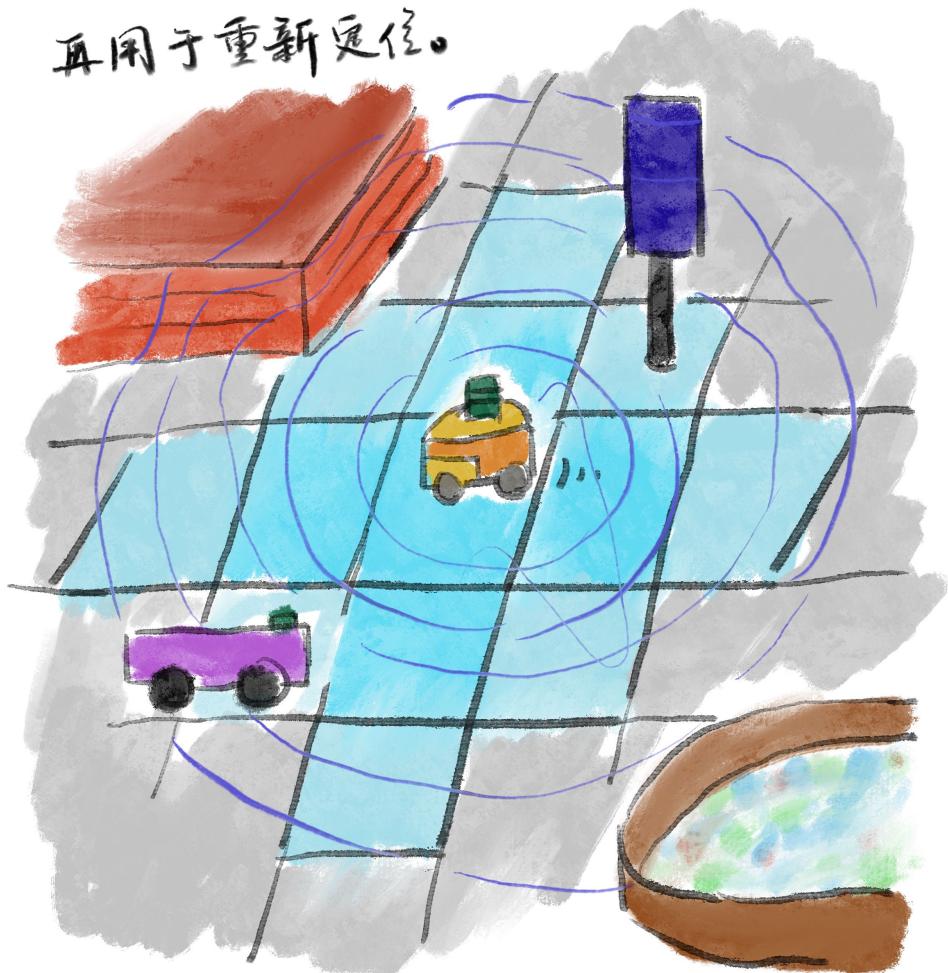
1. Modify the frontend of this chapter's mapping pipeline to use LOAM or its variants, and compare their performance with the LIO frontend employed in this book.
2. Replace the loop closure detection backend with Scan Context or other registration methods to improve its stability.
3. Adapt the NDT registration used for loop closure to the NDT implementation from Chapter 7. Design your own metric for match scoring.

Chapter 10

Real-Time Localization System

This chapter focuses on the real-time LiDAR localization system. With the foundational point cloud map established, we can match current laser scan data with the map to determine the vehicle's position, which is then fused with IMU and other sensor data using filters [18]. However, point cloud localization doesn't directly provide physical world coordinates like RTK does - it first requires an approximate initial position to guide the point cloud registration algorithm toward convergence. Consequently, point cloud localization presents some unique logical challenges in practical applications. Using the point cloud map constructed in the previous chapter, this chapter demonstrates the implementation of point cloud localization and presents a real-time localization solution based on a Kalman filter.

将制作的地图切片，
再用于重新定位。



10.1 Design Scheme for Point Cloud Fusion Localization

Before designing the overall algorithm flow, let us first examine the characteristics of various sensor inputs.

Compared to traditional integrated navigation systems, high-precision localization for autonomous vehicles primarily incorporates an additional input source: LiDAR-based localization. Traditional RTK-IMU integrated navigation systems are less suitable for scenarios like campuses due to the inherent signal quality limitations of RTK. As readers may observe from the NCLT dataset, RTK signals often exhibit instability—such as jitter or dropout—in many areas. On the other hand, the point cloud map constructed in the previous chapter provides an excellent representation of the 3D structure of static environments. Most large-scale scenes, such as buildings, do not change frequently, making point cloud localization a reliable positioning source. Previous chapters of this book have extensively discussed methods for registering scan point clouds with map point clouds. This chapter will employ the NDT method for registration and ultimately integrate the results into a Kalman filter.

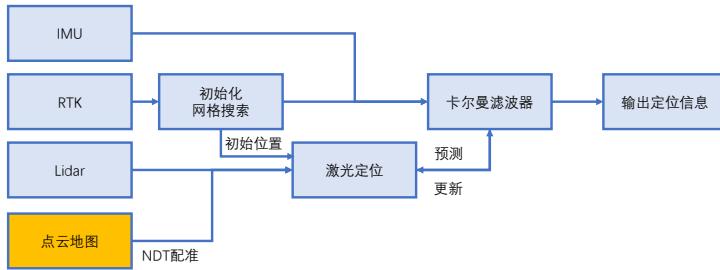


Figure 10-1: Algorithm framework for point cloud fusion localization

From a fusion perspective, point cloud fusion localization can adopt either an error-state Kalman filter (similar to traditional integrated navigation) or pose graph optimization (like modern SLAM systems). Generally, the Kalman filter approach is simpler to design and implement, often yielding smoother results. However, it may converge to incorrect solutions, leading to localization drift that is difficult to correct. In contrast, graph optimization methods facilitate the detection of discrepancies between various factors and the estimated state, enabling logical handling of anomalies—though ensuring smoothness in localization is more challenging (unless manual marginalization, as in Chapter 3, is performed, or optimization libraries like GTSAM with built-in marginalization are used).

First, let us examine the overall algorithm framework (see Fig. 10-1). This chapter demonstrates a Kalman filter-based localization solution. Since the principles of the Kalman filter have already been detailed in Chapter 3, the focus here will be on the fusion of point cloud localization with the Kalman filter.

Point cloud localization requires an initial vehicle position for search, so we design an initialization procedure. When the filter has not yet computed its position, the first valid RTK signal is used to constrain the search range for point cloud localization. Additionally, since the RTK data in the NCLT dataset lacks attitude information, a grid search is employed to determine the vehicle's orientation. Once the Kalman filter converges, its predicted value

serves as the initial guess for point cloud registration¹.

At the point cloud localization level, we use the partitioned point clouds from the previous chapter. Since the point clouds were divided into 100m×100m grid cells earlier, this chapter loads a 3×3 grid (nine cells) around the vehicle. To prevent frequent loading/unloading when the vehicle moves near grid boundaries, an unloading threshold (set to three cells in our implementation) is applied—only cells beyond this range are unloaded.

10.2 Algorithm Implementation

Now we proceed to implement the algorithm described earlier. The code implementation in this chapter reuses portions of the filter from Chapter 3 and the IMU processing code from Chapter 8.

10.2.1 RTK Initialization Search

Following the algorithm flow outlined previously, the system cannot perform point cloud localization until it receives the first valid RTK signal, as the vehicle’s position in the map is unknown. Upon receiving the first valid RTK signal, we conduct a **grid search** around it. Our designed search process primarily determines the vehicle’s initial heading angle by employing a multi-resolution matching method similar to that used in loop detection (Chapter 8).

Listing 10.1: /ch10/fusion.cc

```

1 bool Fusion::SearchRTK() {
2     // Since RTK lacks attitude, we must search within an angular range
3     std::vector<GridSearchResult> search_poses;
4     LoadMap(last_gnss->utm_pose_);
5
6     /// RTK provides no orientation, so we scan angles at fixed intervals
7     double grid_ang_range = 360.0, grid_ang_step = 10; // Angular search range and step
8     for (double ang = 0; ang < grid_ang_range; ang += grid_ang_step) {
9         SE3 pose(SO3::rotZ(ang * math::kDEG2RAD), Vec3d(0, 0, 0) + last_gnss->utm_pose_.
10             translation());
11         GridSearchResult gr;
12         gr.pose_ = pose;
13         search_poses.emplace_back(gr);
14     }
15
16     LOG(INFO) << "grid search_poses: " << search_poses.size();
17     std::for_each(std::execution::par_unseq, search_poses.begin(), search_poses.end(),
18     [this](GridSearchResult& gr) { AlignForGrid(gr); });
19
20     // Select the best-matching result
21     auto max_ele = std::max_element(search_poses.begin(), search_poses.end(),
22     [] (const auto& g1, const auto& g2) { return g1.score_ < g2.score_; });
23     LOG(INFO) << "max score: " << max_ele->score_ << ", pose: \n" << max_ele->
24         result_pose_.matrix();
25     if (max_ele->score_ > rtk_search_min_score_) {
26         LOG(INFO) << "Initialization succeeded, score: " << max_ele->score_ << ">" <<
27             rtk_search_min_score_;
28         status_ = Status::WORKING;
29
30     /// Reset filter state
31     auto state = eskf_.GetNominalState();
32     state.R_ = max_ele->result_pose_.so3();
33     state.p_ = max_ele->result_pose_.translation();
34     state.v_.setZero();
35     eskf_.SetX(state, eskf_.GetGravity());
36 }
```

¹ Alternatively, the point cloud localization could predict the next LiDAR pose based on historical poses, improving system decoupling and simplifying debugging.

```

34     ESKFD::Mat18T cov;
35     cov = ESKFD::Mat18T::Identity() * 1e-4;
36     cov.block<12, 12>(6, 6) = Eigen::Matrix<double, 12, 12>::Identity() * 1e-6;
37     eskf_.SetCov(cov);
38
39     return true;
40 }
41
42 init_has_failed_ = true;
43 last_searched_pos_ = last_gnss_->utm_pose_;
44 return false;
45 }

46 void Fusion::AlignForGrid(sad::Fusion::GridSearchResult& gr) {
47     /// Multi-resolution NDT
48     pcl::NormalDistributionsTransform<PointType, PointType> ndt;
49     ndt.setTransformationEpsilon(0.05);
50     ndt.setStepSize(0.7);
51     ndt.setMaximumIterations(40);

52     ndt.setInputSource(current_scan_);
53     auto map = ref_cloud_;

54     CloudPtr output(new PointCloudType);
55     std::vector<double> res{10.0, 5.0, 4.0, 3.0};
56     Mat4f T = gr.pose_.matrix().cast<float>();
57     for (auto& r : res) {
58         auto rough_map = VoxelCloud(map, r * 0.1);
59         ndt.setInputTarget(rough_map);
60         ndt.setResolution(r);
61         ndt.align(*output, T);
62         T = ndt.getFinalTransformation();
63     }

64     gr.score_ = ndt.getTransformationProbability();
65     gr.result_pose_ = Mat4ToSE3(ndt.getFinalTransformation());
66 }

67

68 }

69 }

70 }
```

10.3 Algorithm Implementation

We concurrently invoke multi-resolution NDT searches to determine the vehicle's initial orientation. If the maximum matching score from these registration results exceeds a preset threshold, we consider the initialization successful. Upon successful RTK initialization, the fusion system transitions to normal operation mode. The code framework here resembles the LIO system described earlier. We retain the point cloud deskewing and IMU-LiDAR message synchronization components while replacing the original LIO registration with map matching:

Listing 10.2: /ch10/fusion.cc

```

1 void Fusion::ProcessMeasurements(const MeasureGroup& meas) {
2     measures_ = meas;
3
4     if (imu_need_init_) {
5         TryInitIMU();
6         return;
7     }
8
9     /// The following three steps remain consistent with LIO,
10    /// except the map matching is now handled by the align function
11    if (status_ == Status::WORKING) {
12        Predict();
13        Undistort();
14    } else {
15        scan_undistort_ = measures_.lidar_;
16    }
17 }
```

```

18     Align();
19 }
20
21 void Fusion::Align() {
22     FullCloudPtr scan_undistort_trans(new FullPointCloudType);
23     pcl::transformPointCloud(*scan_undistort_, *scan_undistort_trans, TIL_.matrix());
24     scan_undistort_ = scan_undistort_trans;
25     current_scan_ = ConvertToCloud<FullPointType>(scan_undistort_);
26     current_scan_ = VoxelCloud(current_scan_, 0.5);
27
28     if (status_ == Status::WAITING_FOR_RTK) {
29         // If recent RTK signal exists, attempt initialization
30         if (last_gnss_ != nullptr) {
31             if (SearchRTK()) {
32                 status_ == Status::WORKING;
33                 ui_>UpdateScan(current_scan_, eskf_.GetNominalSE3());
34                 ui_>UpdateNavState(eskf_.GetNominalState());
35             }
36         } else {
37             LidarLocalization();
38             ui_>UpdateScan(current_scan_, eskf_.GetNominalSE3());
39             ui_>UpdateNavState(eskf_.GetNominalState());
40         }
41     }
42 }
```

At the LiDAR registration level, we load point clouds near the predicted pose and perform NDT registration:

Listing 10.3: /ch10/fusion.cc

```

1  bool Fusion::LidarLocalization() {
2      SE3 pred = eskf_.GetNominalSE3();
3      LoadMap(pred);
4
5      ndt_.setInputCloud(current_scan_);
6      CloudPtr output(new PointCloudType);
7      ndt_.align(*output, pred.matrix().cast<float>());
8
9      SE3 pose = Mat4ToSE3(ndt_.getFinalTransformation());
10     eskf_.ObserveSE3(pose, 1e-1, 1e-2);
11
12     LOG(INFO) << "lidar loc score: " << ndt_.getTransformationProbability();
13
14     return true;
15 }
```

The LoadMap function loads/unloads necessary map blocks based on the given pose:

Listing 10.4: /ch10/fusion.cc

```

1  void Fusion::LoadMap(const SE3& pose) {
2      int gx = int((pose.translation().x() - 50.0) / 100);
3      int gy = int((pose.translation().y() - 50.0) / 100);
4      Vec2i key(gx, gy);
5
6      // Load 9 surrounding blocks around current position
7      std::set<Vec2i, less_vec2> surrounding_index{
8          key + Vec2i(0, 0), key + Vec2i(-1, 0), key + Vec2i(-1, -1), key + Vec2i(-1, 1),
9          key + Vec2i(0, -1),
10         key + Vec2i(0, 1), key + Vec2i(1, 0), key + Vec2i(1, -1), key + Vec2i(1, 1),
11     };
12
13     // Load necessary regions
14     bool map_data_changed = false;
15     int cnt_new_loaded = 0, cnt_unload = 0;
16     for (auto& k : surrounding_index) {
17         if (map_data_index_.find(k) == map_data_index_.end()) {
18             // Map data doesn't exist
19             continue;
20         }
21     }
22 }
```

```

21     if (map_data_.find(k) == map_data_.end()) {
22         // Load this block
23         CloudPtr cloud(new PointCloudType);
24         pcl::io::loadPCDFile(data_path_ + std::to_string(k[0]) + "_" + std::to_string(
25             k[1]) + ".pcd", *cloud);
26         map_data_.emplace(k, cloud);
27         map_data_changed = true;
28         cnt_new_loaded++;
29     }
30
31     // Unload unnecessary regions (slightly larger than loading area to avoid frequent
32     // unloading)
33     for (auto iter = map_data_.begin(); iter != map_data_.end();) {
34         if ((iter->first - key).norm() > 3.0) {
35             // Unload this block
36             iter = map_data_.erase(iter);
37             cnt_unload++;
38         } else {
39             iter++;
40         }
41     }
42
43     LOG(INFO) << "new loaded: " << cnt_new_loaded << ", unload: " << cnt_unload;
44     if (map_data_changed) {
45         // Rebuild NDT target map
46         ref_cloud_.reset(new PointCloudType);
47         for (auto& mp : map_data_) {
48             *ref_cloud_ += *mp.second;
49         }
50
51         LOG(INFO) << "rebuild global cloud, grids: " << map_data_.size();
52         ndt_.setResolution(1.0);
53         ndt_.setInputTarget(ref_cloud_);
54     }
55
56     ui_->UpdatePointCloudGlobal(map_data_);
57 }
```

The map block calculation method remains consistent with the previous chapter. When map blocks change, the KD-tree inside PCL's NDT needs reconstruction. Note this operation can be computationally expensive, potentially causing system lag during map transitions. In practical systems, we could employ a dedicated loading thread to handle this task.

10.3.1 Peripheral Test Code

Finally, we send the unpacked ROS messages to the fusion localization algorithm. The test code is shown below:

Listing 10.5: src/ch10/run_fusion_offline.cc

```

1 int main(int argc, char** argv) {
2     sad::Fusion fusion(FLAGS_config_yaml);
3     if (!fusion.Init()) {
4         return -1;
5     }
6
7     auto yaml = YAML::LoadFile(FLAGS_config_yaml);
8     auto bag_path = yaml["bag_path"].as<std::string>();
9     sad::RosbagIO rosbag_io(bag_path, sad::DatasetType::NCLT);
10
11    /// Deliver various messages to fusion
12    rosbag_io
13        .AddAutoRTKHandle([&fusion](GNSSPtr gnss) {
14            fusion.ProcessRTK(gnss);
15            return true;
16        })
17        .AddAutoPointCloudHandle([&](sensor_msgs::PointCloud2::Ptr cloud) -> bool {
18            fusion.ProcessPointCloud(cloud);
```



Figure 10-2: Test results of fusion localization

```

19     return true;
20 }
21 .AddIMuHandle([&](IMUPtr imu) {
22     fusion.ProcessIMU(imu);
23     return true;
24 })
25 .Go();
26
27 LOG(INFO) << "done.";
28 }
```

To test this section’s code, readers should first specify the test data package in the configuration file’s bag_path field. For fairness, we recommend using different packages from the mapping data. The NCLT dataset was collected across different months. For example, readers could use April’s data for mapping and May/June’s data for localization testing, which includes some dynamic object scenarios.

After executing run_fusion_offline, readers should see the fusion localization interface as shown in Figure 10-2. The gray point cloud displays the map, while the colored point cloud shows the current scan (with cumulative effects retained for some time). The white coordinate frame indicates the vehicle’s current localization. Readers should observe effective localization during most periods, as shown in Figure 10-3.

This section demonstrates the effectiveness of point cloud fusion localization. It shows that point cloud localization typically offers better robustness than RTK. As long as the point cloud map is sufficiently accurate, it can function effectively in most mapped areas without relying on outdoor RTK signal quality.

10.4 Summary

Building upon the point cloud mapping results from Chapter 9, this chapter demonstrates a case study of fusion localization using existing LiDAR point cloud maps. The advantages and disadvantages of point cloud localization can be summarized as follows:

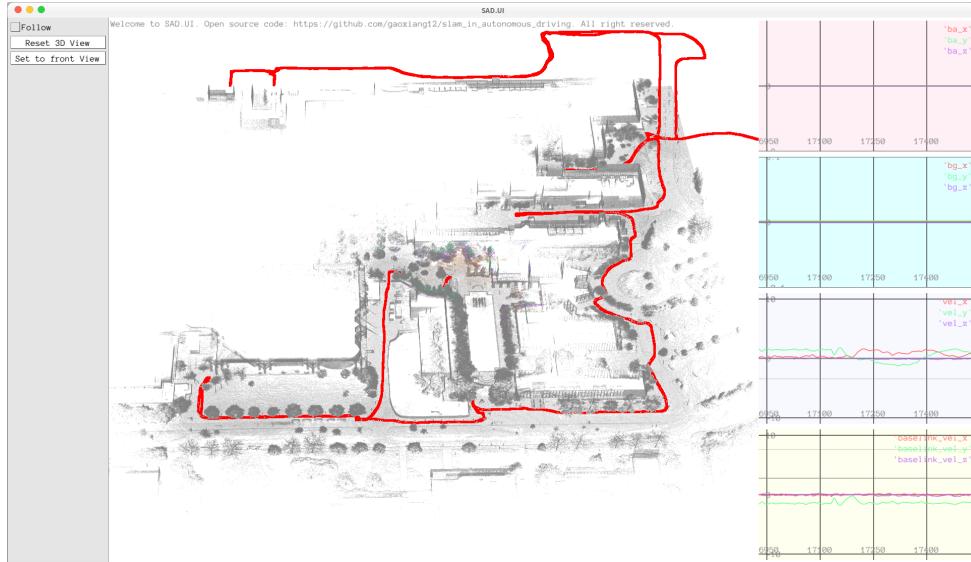


Figure 10-3: Global results of fusion localization

1. Point cloud localization depends on scene structure rather than weather or occlusion conditions, typically offering better environmental adaptability than RTK. It works both indoors and outdoors with few limitations.
2. Point cloud localization results can be integrated into ESKF filters like RTK or form an optimization system as in Chapter 4.
3. For large-scale scenarios, the map should be partitioned, loading only nearby regions for localization.
4. Unlike RTK, point cloud localization requires an initial approximate position. Insufficient initial accuracy may necessitate grid searches of nearby positions and angles.
5. It shows some adaptability to dynamic scenes. As long as major structures (typically buildings) remain unchanged, small dynamic objects (crowds, vehicles, etc.) usually don't significantly interfere.
6. For open areas, point cloud maps can be compressed into 2.5D maps [180–182], greatly reducing storage requirements. Due to space constraints, these compression algorithms aren't detailed here. Readers may refer to the literature for implementation details.

Exercises

1. Test mapping and localization on other datasets provided with this book.
2. Implement your own NDT to replace PCL's NDT, addressing lag during map transitions.
3. Use NDT scores to evaluate matching quality, reinitializing the system when scores are too low.

4. Adapt this program into a ROS node running via subscription callbacks (requires basic ROS knowledge).
5. Experiment with compressing point cloud maps for 2D/2.5D map-based localization [180].

Bibliography

- [1] X. Gao and T. Zhang, *Introduction to visual SLAM: from theory to practice*. Springer Nature, 2021.
- [2] J. Zhang and S. Singh, “Loam: Lidar odometry and mapping in real-time.,” in *Robotics: Science and Systems*, vol. 2, pp. 1–9, Berkeley, CA, 2014.
- [3] W. Hess, D. Kohler, H. Rapp, and D. Andor, “Real-time loop closure in 2d lidar slam,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1271–1278, IEEE, 2016.
- [4] T. D. Barfoot, *State estimation for robotics*. Cambridge University Press, 2017.
- [5] Y. Ma, S. Soatto, J. Kosecka, and S. S. Sastry, *An invitation to 3-d vision: from images to geometric models*, vol. 26. Springer Science & Business Media, 2012.
- [6] J. Solà, “Quaternion kinematics for the error-state kalman filter,” *CoRR*, vol. abs/1711.02508, 2017.
- [7] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT Press, 2005.
- [8] Y. Qin, *Inertial Navigation*. Science Press, 2014.
- [9] G. Yan and J. Weng, *Kalman Filter Algorithm and Combination Navigation Principle for Strap-down Inertial Navigation*. Northwestern Polytechnical University Press, 2019.
- [10] S. L. Gongmin Yan and Y. Qin, *Inertial Instrument Testing and Data Analysis*. National Defense Industry Press, 2012.
- [11] N. Carlevaris-Bianco, A. K. Ushani, and R. M. Eustice, “University of Michigan North Campus long-term vision and lidar dataset,” *International Journal of Robotics Research*, vol. 35, no. 9, pp. 1023–1035, 2015.
- [12] Z. Yan, L. Sun, T. Krajnik, and Y. Ruichek, “EU long-term dataset with multiple sensors for autonomous driving,” in *Proceedings of the 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.
- [13] W. Wen, Y. Zhou, G. Zhang, S. Fahandezh-Saadi, X. Bai, W. Zhan, M. Tomizuka, and L.-T. Hsu, “Urbanloco: A full sensor suite dataset for mapping and localization in urban scenes,” in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2310–2316, IEEE, 2020.
- [14] R. Qian, D. Garg, Y. Wang, Y. You, S. Belongie, B. Hariharan, M. Campbell, K. Q. Weinberger, and W.-L. Chao, “End-to-end pseudo-lidar for image-based 3d object detection,” 2020.
- [15] Y. Zhang, J. Wang, X. Wang, and J. M. Dolan, “Road-segmentation-based curb detection method for self-driving via a 3d-lidar sensor,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 12, pp. 3981–3991, 2018.

- [16] P. Wei, X. Wang, and Y. Guo, “3d-lidar feature based localization for autonomous vehicles,” in *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, pp. 288–293, 2020.
- [17] C. Hungar, S. Jürgens, D. Wilbers, and F. Köster, “Map-based localization with factor graphs for automated driving using non-semantic lidar features,” in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pp. 1–6, 2020.
- [18] L. Wang, Y. Zhang, and J. Wang, “Map-based localization method for autonomous vehicles using 3d-lidar,” *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 276 – 281, 2017. 20th IFAC World Congress.
- [19] L. Sun, C. Peng, W. Zhan, and M. Tomizuka, “A fast integrated planning and control framework for autonomous driving via imitation learning,” in *Dynamic Systems and Control Conference*, vol. 51913, p. V003T37A012, American Society of Mechanical Engineers, 2018.
- [20] I. B. Viana, H. Kanchwala, K. Ahiska, and N. Aouf, “A comparison of trajectory planning and control frameworks for cooperative autonomous driving,” *Journal of Dynamic Systems, Measurement, and Control*, vol. 143, no. 7, 2021.
- [21] L. Zhang, F. Tafazzoli, G. Krehl, R. Xu, T. Rehfeld, M. Schier, and A. Seal, “Hierarchical road topology learning for urban map-less driving,” 2021.
- [22] T. Ort, K. Murthy, R. Banerjee, S. K. Gottipati, D. Bhatt, I. Gilitschenski, L. Paull, and D. Rus, “Maplite: Autonomous intersection navigation without a detailed prior map,” *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 556–563, 2020.
- [23] Y. B. Can, A. Liniger, O. Unal, D. Paudel, and L. V. Gool, “Understanding bird’s-eye view semantic hd-maps using an onboard monocular camera,” 2020.
- [24] A. Sun, “A perception centered self-driving system without hd maps,” *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 10, 2020.
- [25] M. H. Ng, K. Radia, J. Chen, D. Wang, I. Gog, and J. E. Gonzalez, “Bev-seg: Bird’s eye view semantic segmentation using geometry and semantic point cloud,” 2020.
- [26] N. Hendy, C. Sloan, F. Tian, P. Duan, N. Charchut, Y. Xie, C. Wang, and J. Philbin, “Fishing net: Future inference of semantic heatmaps in grids,” 2020.
- [27] J. Levinson and S. Thrun, “Robust vehicle localization in urban environments using probabilistic maps,” in *2010 IEEE international conference on robotics and automation*, pp. 4372–4378, IEEE, 2010.
- [28] R. W. Wolcott and R. M. Eustice, “Visual localization within lidar maps for automated urban driving,” in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 176–183, IEEE, 2014.
- [29] R. Matthaei, G. Bagschik, and M. Maurer, “Map-relative localization in lane-level maps foradas and autonomous driving,” in *2014 IEEE Intelligent Vehicles Symposium Proceedings*, pp. 49–55, IEEE, 2014.
- [30] F. Ghallabi, F. Nashashibi, G. El-Haj-Shhade, and M.-A. Mittet, “Lidar-based lane marking detection for vehicle positioning in an hd map,” in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 2209–2214, IEEE, 2018.
- [31] B. Yang, M. Liang, and R. Urtasun, “Hdnet: Exploiting hd maps for 3d object detection,” in *Conference on Robot Learning*, pp. 146–155, PMLR, 2018.

- [32] R. Spangenberg, D. Goehring, and R. Rojas, “Pole-based localization for autonomous vehicles in urban scenarios,” in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2161–2166, 2016.
- [33] J. Levinson, M. Montemerlo, and S. Thrun, “Map-based precision vehicle localization in urban environments,” in *Robotics: science and systems*, vol. 4, p. 1, Atlanta, GA, USA, 2007.
- [34] H. G. Seif and X. Hu, “Autonomous driving in the city—hd maps as a key challenge of the automotive industry,” *Engineering*, vol. 2, no. 2, pp. 159–162, 2016.
- [35] Y. Zhou, Y. Takeda, M. Tomizuka, and W. Zhan, “Automatic construction of lane-level hd maps for urban scenes,” *arXiv preprint arXiv:2107.10972*, 2021.
- [36] M. Dupuis, M. Strobl, and H. Grezlikowski, “Opendrive 2010 and beyond—status and future of the de facto standard for the description of road networks,” in *Proc. of the Driving Simulation Conference Europe*, pp. 231–242, 2010.
- [37] F. Poggenhans, J.-H. Pauls, J. Janosovits, S. Orf, M. Naumann, F. Kuhnt, and M. Mayr, “Lanelet2: A high-definition map framework for the future of automated driving,” in *2018 21st international conference on intelligent transportation systems (ITSC)*, pp. 1672–1679, IEEE, 2018.
- [38] F. Ghallabi, G. El-Haj-Shhade, M.-A. Mittet, and F. Nashashibi, “Lidar-based road signs detection for vehicle localization in an hd map,” in *2019 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1484–1490, IEEE, 2019.
- [39] W.-C. Ma, I. Tartavull, I. A. Bârsan, S. Wang, M. Bai, G. Mattyus, N. Homayounfar, S. K. Lakshminanth, A. Pokrovsky, and R. Urtasun, “Exploiting sparse semantic hd maps for self-driving vehicle localization,” in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5304–5311, IEEE, 2019.
- [40] M. Elhousni, Y. Lyu, Z. Zhang, and X. Huang, “Automatic building and labeling of hd maps with deep learning,” 2020.
- [41] B. Liao, S. Chen, X. Wang, T. Cheng, Q. Zhang, W. Liu, and C. Huang, “Maptr: Structured modeling and learning for online vectorized hd map construction,” *arXiv preprint arXiv:2208.14437*, 2022.
- [42] T. Qin, P. Li, and S. Shen, “Vins-mono: A robust and versatile monocular visual-inertial state estimator,” *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 1004–1020, 2018.
- [43] T. Barfoot, J. R. Forbes, and P. T. Furgale, “Pose estimation using linearized rotations and quaternion algebra,” *Acta Astronautica*, vol. 68, no. 1-2, pp. 101–112, 2011.
- [44] R. Gilmore, “Baker-campbell-hausdorff formulas,” *Journal of Mathematical Physics*, vol. 15, no. 12, pp. 2090–2092, 1974.
- [45] H. E. Rauch, F. Tung, and C. T. Striebel, “Maximum likelihood estimates of linear dynamic systems,” *AIAA journal*, vol. 3, no. 8, pp. 1445–1450, 1965.
- [46] P. Zarchan, *Progress in astronautics and aeronautics: fundamentals of Kalman filtering: a practical approach*, vol. 208. Aiaa, 2005.
- [47] D. He, W. Xu, and F. Zhang, “Kalman filters on differentiable manifolds,” *arXiv preprint arXiv:2102.03804*, 2021.
- [48] Z. Huai and G. Huang, “Robocentric visual-inertial odometry,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6319–6326, IEEE, 2018.

- [49] J. L. Crassidis, “Sigma-point kalman filtering for integrated gps and inertial navigation,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 42, no. 2, pp. 750–756, 2006.
- [50] O. J. Woodman, “An introduction to inertial navigation,” tech. rep., University of Cambridge, Computer Laboratory, 2007.
- [51] W. Liu, S. Wu, Y. Wen, and X. Wu, “Integrated autonomous relative navigation method based on vision and imu data fusion,” *IEEE Access*, vol. 8, pp. 51114–51128, 2020.
- [52] M. Kleinert and S. Schleith, “Inertial aided monocular slam for gps-denied navigation,” in *2010 IEEE Conference on Multisensor Fusion and Integration*, pp. 20–25, IEEE, 2010.
- [53] M. Li and A. I. Mourikis, “High-precision, consistent ekf-based visual-inertial odometry,” *International Journal of Robotics Research*, vol. 32, pp. 690–711, MAY 2013.
- [54] A. J. Davison, “Real-time simultaneous localisation and mapping with a single camera,” in *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pp. 1403–1410, IEEE, 2003.
- [55] J. Kelly and G. S. Sukhatme, “Visual-inertial sensor fusion: Localization, mapping and sensor-to-sensor self-calibration,” *The International Journal of Robotics Research*, vol. 30, no. 1, pp. 56–79, 2011.
- [56] F. M. Mirzaei and S. I. Roumeliotis, “A kalman filter-based algorithm for imu-camera calibration: Observability analysis and performance evaluation,” *IEEE transactions on robotics*, vol. 24, no. 5, pp. 1143–1156, 2008.
- [57] J. L. Crassidis, “Sigma-point kalman filtering for integrated gps and inertial navigation,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 42, no. 2, pp. 750–756, 2006.
- [58] J.-F. Bonnans, J. C. Gilbert, C. Lemaréchal, and C. A. Sagastizábal, *Numerical optimization: theoretical and practical aspects*. Springer Science & Business Media, 2006.
- [59] W. Xu and F. Zhang, “Fast-lio: A fast, robust lidar-inertial odometry package by tightly-coupled iterated kalman filter,” *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 3317–3324, 2021.
- [60] W. Xu, Y. Cai, D. He, J. Lin, and F. Zhang, “Fast-lio2: Fast direct lidar-inertial odometry,” *arXiv preprint arXiv:2107.06829*, 2021.
- [61] M. Bloesch, M. Burri, S. Omari, M. Hutter, and R. Siegwart, “Iterated extended kalman filter based visual-inertial odometry using direct photometric feedback,” *The International Journal of Robotics Research*, vol. 36, no. 10, pp. 1053–1072, 2017.
- [62] V. Madayastha, V. Ravindra, S. Mallikarjunan, and A. Goyal, “Extended kalman filter vs. error state kalman filter for aircraft attitude estimation,” in *AIAA Guidance, Navigation, and Control Conference*, p. 6615, 2011.
- [63] T. Lupton and S. Sukkarieh, “Efficient integration of inertial observations into visual slam without initialization,” in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1547–1552, IEEE, 2009.
- [64] W. Wen, T. Pfeifer, X. Bai, and L.-T. Hsu, “Factor graph optimization for gnss/ins integration: A comparison with the extended kalman filter,” *NAVIGATION: Journal of the Institute of Navigation*, vol. 68, no. 2, pp. 315–331, 2021.
- [65] C. Hertzberg, R. Wagner, U. Frese, and L. Schröder, “Integrating generic sensor fusion algorithms with sound state representations through encapsulation of manifolds,” *Information Fusion*, vol. 14, no. 1, pp. 57–77, 2013.

- [66] T. Vidal-Calleja, M. Bryson, S. Sukkarieh, A. Sanfeliu, and J. Andrade-Cetto, “On the observability of bearing-only slam,” in *Robotics and Automation, 2007 IEEE International Conference on*, pp. 4114–4119, IEEE, 2007.
- [67] C. Forster, L. Carbone, F. Dellaert, and D. Scaramuzza, “Imu preintegration on manifold for efficient visual-inertial maximum-a-posteriori estimation,” in *Robotics: Science and Systems XI*, no. EPFL-CONF-214687, 2015.
- [68] L. Chang, X. Niu, and T. Liu, “Gnss/imu/odo/lidar-slam integrated navigation system using imu/odo pre-integration,” *Sensors*, vol. 20, no. 17, p. 4702, 2020.
- [69] Z. Yuan, D. Zhu, C. Chi, J. Tang, C. Liao, and X. Yang, “Visual-inertial state estimation with pre-integration correction for robust mobile augmented reality,” in *Proceedings of the 27th ACM International Conference on Multimedia*, pp. 1410–1418, 2019.
- [70] K. Eckenhoff, P. Geneva, and G. Huang, “Closed-form preintegration methods for graph-based visual–inertial navigation,” *The International Journal of Robotics Research*, vol. 38, no. 5, pp. 563–586, 2019.
- [71] R. M. Murray, Z. Li, and S. S. Sastry, *A mathematical introduction to robotic manipulation*. CRC press, 2017.
- [72] T. Lupton and S. Sukkarieh, “Visual-inertial-aided navigation for high-dynamic motion in built environments without initial conditions,” *IEEE Transactions on Robotics*, vol. 28, no. 1, pp. 61–76, 2011.
- [73] S. Leutenegger, S. Lynen, M. Bosse, R. Siegwart, and P. Furgale, “Keyframe-based visual–inertial odometry using nonlinear optimization,” *The International Journal of Robotics Research*, vol. 34, no. 3, pp. 314–334, 2015.
- [74] M. Magnusson, *The three-dimensional normal-distributions transform: an efficient representation for registration, surface analysis, and loop detection*. PhD thesis, Örebro universitet, 2009.
- [75] R. B. Rusu and S. Cousins, “3d is here: Point cloud library (pcl),” in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 1–4, IEEE, 2011.
- [76] D. Delling, P. Sanders, D. Schultes, and D. Wagner, “Engineering route planning algorithms,” in *Algorithmics of large and complex networks*, pp. 117–139, Springer, 2009.
- [77] K. Sabe, M. Fukuchi, J.-S. Gutmann, T. Ohashi, K. Kawamoto, and T. Yoshigahara, “Obstacle avoidance and path planning for humanoid robots using stereo vision,” in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA’04. 2004*, vol. 1, pp. 592–597, IEEE, 2004.
- [78] C. Park, S. Kim, P. Moghadam, C. Fookes, and S. Sridharan, “Probabilistic surfel fusion for dense lidar mapping,” in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pp. 2418–2426, 2017.
- [79] C. Park, P. Moghadam, S. Kim, A. Elfes, C. Fookes, and S. Sridharan, “Elastic lidar fusion: Dense map-centric continuous-time slam,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1206–1213, IEEE, 2018.
- [80] L. Roldão, R. de Charette, and A. Verroust-Blondet, “3d surface reconstruction from voxel-based lidar data,” in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pp. 2681–2686, IEEE, 2019.
- [81] R. Weber, H.-J. Schek, and S. Blott, “A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces,” in *VLDB*, vol. 98, pp. 194–205, 1998.

- [82] Q. Kuang and L. Zhao, “A practical gpu based knn algorithm,” in *Proceedings. The 2009 International Symposium on Computer Science and Computational Technology (ISCSCI 2009)*, p. 151, Citeseer, 2009.
- [83] V. Garcia, E. Debreuve, and M. Barlaud, “Fast k nearest neighbor search using gpu,” in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1–6, IEEE, 2008.
- [84] S. Li and N. Amenta, “Brute-force k-nearest neighbors search on the gpu,” in *International Conference on Similarity Search and Applications*, pp. 259–270, Springer, 2015.
- [85] M. Teschner, B. Heidelberger, M. Müller, D. Pomerantes, and M. H. Gross, “Optimized spatial hashing for collision detection of deformable objects.,” in *Vmv*, vol. 3, pp. 47–54, 2003.
- [86] K. Koide, J. Miura, and E. Menegatti, “A portable three-dimensional lidar-based system for long-term and wide-area people behavior measurement,” *International Journal of Advanced Robotic Systems*, vol. 16, no. 2, p. 1729881419841532, 2019.
- [87] K. Koide, M. Yokozuka, S. Oishi, and A. Banno, “Voxelized gicp for fast and accurate 3d point cloud registration,” *EasyChair Preprint*, no. 2703, 2020.
- [88] F. Huang, W. Wen, J. Zhang, and L.-T. Hsu, “Point wise or feature wise? benchmark comparison of public available lidar odometry algorithms in urban canyons,” *arXiv preprint arXiv:2104.05203*, 2021.
- [89] C. Bai, T. Xiao, Y. Chen, H. Wang, F. Zhang, and X. Gao, “Faster-lio: Lightweight tightly coupled lidar-inertial odometry using parallel sparse incremental voxels,” *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 4861–4868, 2022.
- [90] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- [91] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [92] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf, “Computational geometry,” in *Computational geometry*, pp. 1–17, Springer, 1997.
- [93] M. Groß, C. Lojewski, M. Bertram, and H. Hagen, “Fast implicit kd-trees: Accelerated isosurface ray tracing and maximum intensity projection for large scalar fields,” in *Proc. Computer Graphics and Imaging*, pp. 67–74, Citeseer, 2007.
- [94] B. Duvenhage, “Using an implicit min/max kd-tree for doing efficient terrain line of sight calculations,” in *Proceedings of the 6th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, pp. 81–90, 2009.
- [95] A. Duch, V. Estivill-Castro, and C. Martinez, “Randomized k-dimensional binary search trees,” in *International Symposium on Algorithms and Computation*, pp. 198–209, Springer, 1998.
- [96] Y. Cai, W. Xu, and F. Zhang, “ikd-tree: An incremental k-d tree for robotic applications,” *ArXiv*, vol. abs/2102.10808, 2021.
- [97] H. Samet, “The quadtree and related hierarchical data structures,” *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.
- [98] D. Meagher, “Geometric modeling using octree encoding,” *Computer graphics and image processing*, vol. 19, no. 2, pp. 129–147, 1982.
- [99] C. A. Shaffer and H. Samet, “Optimal quadtree construction algorithms,” *Computer Vision, Graphics, and Image Processing*, vol. 37, no. 3, pp. 402–419, 1987.

- [100] R. P. Haining and R. Haining, *Spatial data analysis: theory and practice*. Cambridge university press, 2003.
- [101] S. M. Omohundro, *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.
- [102] M. Dolatshah, A. Hadian, and B. Minaei-Bidgoli, “Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces,” *arXiv preprint arXiv:1511.00628*, 2015.
- [103] T. Liu, A. W. Moore, A. Gray, and C. Cardie, “New algorithms for efficient high-dimensional nonparametric classification.,” *Journal of Machine Learning Research*, vol. 7, no. 6, 2006.
- [104] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pp. 47–57, 1984.
- [105] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, “The r*-tree: An efficient and robust access method for points and rectangles,” in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pp. 322–331, 1990.
- [106] G. v. d. Bergen, “Efficient collision detection of complex deformable models using aabb trees,” *Journal of graphics tools*, vol. 2, no. 4, pp. 1–13, 1997.
- [107] J. K. Lawder and P. J. H. King, “Querying multi-dimensional data indexed using the hilbert space-filling curve,” *ACM Sigmod Record*, vol. 30, no. 1, pp. 19–24, 2001.
- [108] A. Khoshgozaran and C. Shahabi, “Blind evaluation of nearest neighbor queries using space transformation to preserve location privacy,” in *International symposium on spatial and temporal databases*, pp. 239–257, Springer, 2007.
- [109] J. A. Orenstein and F. A. Manola, “Probe spatial data modeling and query processing in an image database application,” *IEEE transactions on Software Engineering*, vol. 14, no. 5, pp. 611–629, 1988.
- [110] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, “Locality-sensitive hashing scheme based on p-stable distributions,” in *Proceedings of the twentieth annual symposium on Computational geometry*, pp. 253–262, 2004.
- [111] R. P. Mahapatra and P. S. Chakraborty, “Comparative analysis of nearest neighbor query processing techniques,” *Procedia Computer Science*, vol. 57, pp. 1289–1298, 2015.
- [112] N. Bhatia *et al.*, “Survey of nearest neighbor techniques,” *arXiv preprint arXiv:1007.0085*, 2010.
- [113] L. E. Navarro-Serment, C. Mertz, and M. Hebert, “Pedestrian detection and tracking using three-dimensional ladar data,” *The International Journal of Robotics Research*, vol. 29, no. 12, pp. 1516–1528, 2010.
- [114] J. R. Magnus, “Handbook of matrices: H. lütkepohl, john wiley and sons, 1996,” *Econometric Theory*, vol. 14, no. 3, pp. 379–380, 1998.
- [115] S. Wold, K. Esbensen, and P. Geladi, “Principal component analysis,” *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.
- [116] C. Eckart and G. Young, “The approximation of one matrix by another of lower rank,” *Psychometrika*, vol. 1, no. 3, pp. 211–218, 1936.
- [117] D. C. Lay, “Linear algebra and its applications, 3rd updated edition,” 2005.
- [118] J. L. Blanco and P. K. Rai, “nanoflann: a C++ header-only fork of FLANN, a library for nearest neighbor (NN) with kd-trees.” <https://github.com/jlblancoc/nanoflann>, 2014.

- [119] J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with GPUs,” *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.
- [120] L. Boytsov, D. Novak, Y. A. Malkov, and E. Nyberg, “Off the beaten path: Let’s replace term-based retrieval with k-nn search,” in *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016* (S. Mukhopadhyay, C. Zhai, E. Bertino, F. Crestani, J. Mostafa, J. Tang, L. Si, X. Zhou, Y. Chang, Y. Li, and P. Sondhi, eds.), pp. 1099–1108, ACM, 2016.
- [121] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, “Fastslam: A factored solution to the simultaneous localization and mapping problem,” in *Eighteenth National Conference On Artificial Intelligence (AAAI-02)*, pp. 593–598, MIT PRESS, 2002.
- [122] G. Grisetti, C. Stachniss, and W. Burgard, “Improved techniques for grid mapping with rao-blackwellized particle filters,” *IEEE transactions on Robotics*, vol. 23, no. 1, pp. 34–46, 2007.
- [123] S. Thrun, “Learning occupancy grid maps with forward sensor models,” *Autonomous robots*, vol. 15, pp. 111–127, 2003.
- [124] D. Meyer-Delius, M. Beinhofer, and W. Burgard, “Occupancy grid models for robot mapping in changing environments,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 26, pp. 2024–2030, 2012.
- [125] H. Ray, H. Pfister, D. Silver, and T. A. Cook, “Ray casting architectures for volume visualization,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 3, pp. 210–223, 1999.
- [126] J. Pineda, “A parallel algorithm for polygon rasterization,” in *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pp. 17–20, 1988.
- [127] K. S. Arun, T. S. Huang, and S. D. Blostein, “Least-squares fitting of two 3-d point sets,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 5, pp. 698–700, 1987.
- [128] A. Censi, “An icp variant using a point-to-line metric,” in *2008 IEEE International Conference on Robotics and Automation*, pp. 19–25, Ieee, 2008.
- [129] M. Alshawa, “Icl: Iterative closest line a novel point cloud registration algorithm based on linear features,” *Ekscentar*, no. 10, pp. 53–59, 2007.
- [130] E. B. Olson, “Real-time correlative scan matching,” in *2009 IEEE International Conference on Robotics and Automation*, pp. 4387–4393, IEEE, 2009.
- [131] S.-Y. Park and M. Subbarao, “An accurate and fast point-to-plane registration technique,” *Pattern Recognition Letters*, vol. 24, no. 16, pp. 2967–2976, 2003.
- [132] K.-L. Low, “Linear least-squares optimization for point-to-plane icp surface registration,” *Chapel Hill, University of North Carolina*, vol. 4, no. 10, pp. 1–3, 2004.
- [133] P. Biber and W. Straßer, “The normal distributions transform: A new approach to laser scan matching,” in *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*, vol. 3, pp. 2743–2748, IEEE, 2003.
- [134] M. Rapp, M. Barjenbruch, M. Hahn, J. Dickmann, and K. Dietmayer, “Clustering improved grid map registration using the normal distribution transform,” in *2015 IEEE Intelligent Vehicles Symposium (IV)*, pp. 249–254, IEEE, 2015.
- [135] M. Cabaleiro, B. Riveiro, P. Arias, and J. Caamaño, “Algorithm for beam deformation modeling from lidar data,” *Measurement*, vol. 76, pp. 20–31, 2015.

- [136] P. J. Besl and N. D. McKay, "Method for registration of 3-d shapes," in *Sensor fusion IV: control paradigms and data structures*, vol. 1611, pp. 586–606, Spie, 1992.
- [137] A. Segal, D. Haehnel, and S. Thrun, "Generalized-icp.," in *Robotics: science and systems*, vol. 2, p. 435, Seattle, WA, 2009.
- [138] J. Zhang, Y. Yao, and B. Deng, "Fast and robust iterative closest point," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 7, pp. 3450–3466, 2021.
- [139] P. F. Felzenszwalb and D. P. Huttenlocher, "Distance transforms of sampled functions," *Theory of computing*, vol. 8, no. 1, pp. 415–428, 2012.
- [140] R. Kummerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "G2o: a general framework for graph optimization," in *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3607–3613, IEEE, 2011.
- [141] E. G. Tsardoulias, A. Iliakopoulou, A. Kargakos, and L. Petrou, "A review of global path planning methods for occupancy grid maps regardless of obstacle density," *Journal of Intelligent & Robotic Systems*, vol. 84, pp. 829–858, 2016.
- [142] X. Qi, W. Wang, M. Yuan, Y. Wang, M. Li, L. Xue, and Y. Sun, "Building semantic grid maps for domestic robot navigation," *International Journal of Advanced Robotic Systems*, vol. 17, no. 1, p. 1729881419900066, 2020.
- [143] D. Cohen-Or and A. Kaufman, "3d line voxelization and connectivity control," *IEEE Computer Graphics and Applications*, vol. 17, no. 6, pp. 80–87, 1997.
- [144] S. A. Scherer, A. Kloss, and A. Zell, "Loop closure detection using depth images," in *2013 European Conference on Mobile Robots*, pp. 100–106, IEEE, 2013.
- [145] C. Stachniss, G. Grisetti, and W. Burgard, "Recovering particle diversity in a rao-blackwellized particle filter for slam after actively closing loops," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pp. 655–660, IEEE, 2005.
- [146] Z. Qianhao, A. Mai, J. Menke, and A. Yang, "Loop closure detection with rgbd feature pyramid siamese networks," *arXiv preprint arXiv:1811.09938*, 2018.
- [147] D. Fortun, P. Bouthemy, and C. Kervrann, "Optical flow modeling and computation: A survey," *Computer Vision and Image Understanding*, vol. 134, pp. 1–21, 2015.
- [148] Y. Zhou and O. Tuzel, "Voxelnets: End-to-end learning for point cloud based 3d object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4490–4499, 2018.
- [149] S. Shi, X. Wang, and H. Li, "Pointrcnn: 3d object proposal generation and detection from point cloud," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 770–779, 2019.
- [150] G. Sithole and G. Vosselman, "Automatic structure detection in a point-cloud of an urban landscape," in *2003 2nd GRSS/ISPRS Joint Workshop on Remote Sensing and Data Fusion over Urban Areas*, pp. 67–71, IEEE, 2003.
- [151] D. Fernandes, A. Silva, R. Névoa, C. Simões, D. Gonzalez, M. Guevara, P. Novais, J. Monteiro, and P. Melo-Pinto, "Point-cloud based 3d object detection and classification methods for self-driving applications: A survey and taxonomy," *Information Fusion*, vol. 68, pp. 161–191, 2021.
- [152] A. L. Pavlov, G. W. Ovchinnikov, D. Y. Derbyshev, D. Tsetserukou, and I. V. Oseledets, "Aa-icp: Iterative closest point with anderson acceleration," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3407–3412, IEEE, 2018.

- [153] H. Yang, J. Shi, and L. Carlone, “Teaser: Fast and certifiable point cloud registration,” *IEEE Transactions on Robotics*, vol. 37, no. 2, pp. 314–333, 2020.
- [154] H. Pottmann, S. Leopoldseder, and M. Hofer, “Registration without icp,” *Computer Vision and Image Understanding*, vol. 95, no. 1, pp. 54–71, 2004.
- [155] T. Zinßer, J. Schmidt, and H. Niemann, “A refined icp algorithm for robust 3-d correspondence estimation,” in *Proceedings 2003 international conference on image processing (Cat. No. 03CH37429)*, vol. 2, pp. II–695, IEEE, 2003.
- [156] Y. Chen and G. Medioni, “Object modelling by registration of multiple range images,” *Image and vision computing*, vol. 10, no. 3, pp. 145–155, 1992.
- [157] C. Ulas and H. Temeltas, “3d multi-layered normal distribution transform for fast and long range scan matching,” *Journal of Intelligent & Robotic Systems*, vol. 71, no. 1, pp. 85–108, 2013. Times Cited: 0 Ulas, Cihan Temeltas, Hakan 0.
- [158] J. P. Saarinen, H. Andreasson, T. Stoyanov, and A. J. Lilenthal, “3d normal distributions transform occupancy maps: An efficient representation for mapping in dynamic environments,” *The International Journal of Robotics Research*, vol. 32, no. 14, pp. 1627–1644, 2013.
- [159] P.-C. Kung, C.-C. Wang, and W.-C. Lin, “A normal distribution transform-based radar odometry designed for scanning and automotive radars,” in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 14417–14423, IEEE, 2021.
- [160] A. Y. Hata and D. F. Wolf, “Feature detection for vehicle localization in urban environments using a multilayer lidar,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 2, pp. 420–429, 2015.
- [161] T. Shan and B. Englot, “Lego-loam: Lightweight and ground-optimized lidar odometry and mapping on variable terrain,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4758–4765, IEEE, 2018.
- [162] R. B. Rusu, N. Blodow, Z. C. Marton, and M. Beetz, “Aligning point cloud views using persistent feature histograms,” in *2008 IEEE/RSJ international conference on intelligent robots and systems*, pp. 3384–3391, IEEE, 2008.
- [163] R. B. Rusu, N. Blodow, and M. Beetz, “Fast point feature histograms (fpfh) for 3d registration,” in *2009 IEEE international conference on robotics and automation*, pp. 3212–3217, IEEE, 2009.
- [164] Y. Pan, P. Xiao, Y. He, Z. Shao, and Z. Li, “Mulls: Versatile lidar slam via multi-metric linear least square,” in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11633–11640, IEEE, 2021.
- [165] J. Tang, Y. Chen, X. Niu, L. Wang, L. Chen, J. Liu, C. Shi, and J. Hyppä, “Lidar scan matching aided inertial navigation system in gnss-denied environments,” *Sensors*, vol. 15, no. 7, pp. 16710–16728, 2015.
- [166] C. Qin, H. Ye, C. E. Pranata, J. Han, S. Zhang, and M. Liu, “Lins: A lidar-inertial state estimator for robust and efficient navigation,” in *2020 IEEE international conference on robotics and automation (ICRA)*, pp. 8899–8906, IEEE, 2020.
- [167] Y. Yang, P. Geneva, X. Zuo, K. Eckenhoff, Y. Liu, and G. Huang, “Tightly-coupled aided inertial navigation with point and plane features,” in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 6094–6100, IEEE, 2019.
- [168] Y. Liu, F. Liu, Y. Gao, and L. Zhao, “Implementation and analysis of tightly coupled global navigation satellite system precise point positioning/inertial navigation system (gnss ppp/ins) with insufficient satellites for land vehicle navigation,” *Sensors*, vol. 18, no. 12, p. 4305, 2018.

- [169] X. Kong, W. Wu, L. Zhang, and Y. Wang, “Tightly-coupled stereo visual-inertial navigation using point and line features,” *Sensors*, vol. 15, no. 6, pp. 12816–12833, 2015.
- [170] A. Soloviev, “Tight coupling of gps, laser scanner, and inertial measurements for navigation in urban environments,” in *2008 IEEE/ION Position, Location and Navigation Symposium*, pp. 511–525, IEEE, 2008.
- [171] Y. Shi, S. Ji, Z. Shi, Y. Duan, and R. Shibasaki, “Gps-supported visual slam with a rigorous sensor model for a panoramic camera in outdoor environments,” *Sensors*, vol. 13, no. 1, pp. 119–136, 2012.
- [172] D. Schleicher, L. M. Bergasa, M. Ocaña, R. Barea, and E. López, “Real-time hierarchical gps aided visual slam on urban environments,” in *2009 IEEE International Conference on Robotics and Automation*, pp. 4381–4386, IEEE, 2009.
- [173] J. Sherman and W. J. Morrison, “Adjustment of an inverse matrix corresponding to a change in one element of a given matrix,” *The Annals of Mathematical Statistics*, vol. 21, no. 1, pp. 124–127, 1950.
- [174] T. Shan, B. Englot, D. Meyers, W. Wang, C. Ratti, and D. Rus, “Lio-sam: Tightly-coupled lidar inertial odometry via smoothing and mapping,” in *2020 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pp. 5135–5142, IEEE, 2020.
- [175] T. Shan, B. Englot, C. Ratti, and D. Rus, “Lvi-sam: Tightly-coupled lidar-visual-inertial odometry via smoothing and mapping,” in *2021 IEEE international conference on robotics and automation (ICRA)*, pp. 5692–5698, IEEE, 2021.
- [176] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. Montiel, and J. D. Tardós, “Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam,” *IEEE Transactions on Robotics*, vol. 37, no. 6, pp. 1874–1890, 2021.
- [177] X. Gao, Q. Wang, H. Gu, F. Zhang, G. Peng, Y. Si, and X. Li, “Fully automatic large-scale point cloud mapping for low-speed self-driving vehicles in unstructured environments,” in *2021 IEEE Intelligent Vehicles Symposium (IV)*, pp. 881–888, IEEE, 2021.
- [178] R. W. Wolcott and R. M. Eustice, “Robust lidar localization using multiresolution gaussian mixture maps for autonomous driving,” *The International Journal of Robotics Research*, vol. 36, no. 3, pp. 292–319, 2017.
- [179] R. W. Wolcott and R. M. Eustice, “Fast lidar localization using multiresolution gaussian mixture maps,” in *2015 IEEE international conference on robotics and automation (ICRA)*, pp. 2814–2821, IEEE, 2015.
- [180] G. Wan, X. Yang, R. Cai, H. Li, Y. Zhou, H. Wang, and S. Song, “Robust and precise vehicle localization based on multi-sensor fusion in diverse city scenes,” in *2018 IEEE international conference on robotics and automation (ICRA)*, pp. 4670–4677, IEEE, 2018.