Math Toolbox for Robotics v0.1.3

Xinran Wei

October 14, 2025

GitHub: `https://github.com/weixr18/MT4R`

Contents

1 Introduction

Robotics is one of the most exciting technologies of our time. Although the discipline itself is not young, and mature products such as industrial robots have been serving the world for over 30 years, emerging AI technologies—primarily deep learning (DL), computer vision (CV), large language models (LLMs), and reinforcement learning (RL)—are enabling robots to perceive, understand, and make decisions in unprecedented ways. Today, we are closer than ever in history to achieving truly autonomous, general-purpose, and intelligent robots.

So, what kind of technical talents will future robotics require? In an era where LLMs can already generate and modify code and derive formulas, what capabilities and qualities should a forward-looking robotics engineer possess? This book attempts to give an answer—what the future needs may be not a specialist confined to a single domain, but a **generalist robotics engineer** who knows of all major areas of robotics technology.

Therefore, this book differs significantly from most existing robotics textbooks in its structure and content. Our coverage spans all major fields of modern robotics technology rather than focusing on a single technical domain. This includes, but is not limited to: robot modeling and control (e.g. kinematics and dynamics, control theory), robot perception (e.g. vSLAM, filtering), and robot learning (e.g. deep learning, reinforcement learning), among others. We believe that in today's era, having a comprehensive understanding of the foundational aspects across all major areas of robotics—even without being an expert in each—is a crucial cornerstone for developing the next generation of robots.

Compared with existing textbooks, the content scope of this book is exceptionally broad. From a curriculum perspective, this book may cover material equivalent to 10–20 undergraduate and graduate-level courses. Clearly, we do not aim to replace the excellent specialized textbooks already available in these fields. Nor do we strive for comprehensiveness or cutting-edge depth in any specific area. Instead, our goal is to establish a cross-disciplinary technical perspective. For example, we compare traditional control methods (such as PID and LQR) with data-driven approaches (such as the PPO algorithm) under the same problem setting (e.g., the inverted pendulum problem). We discuss the application of optimization methods across different challenges, and so on. We hope that, in facing the demands of next-generation robotics, readers can develop a deeper understanding of foundational robotics technologies through these interdisciplinary perspectives and thereby better integrate and apply them to create truly next-generation robots.

The writing style of this book also differs from that of traditional robotics textbooks. Our positioning is that of a **reference manual**, not a textbook. This means that **this book is intended for engineers, but not for beginners**. It will not pursue absolute mathematical rigor like a math textbook, nor will it include exercises or experiments. However, it does contain the **key concepts, problems, formulas, algorithms, and essential code** from fundamental areas of robotics. Our aim is to help readers grasp the essence of these topics using concise, clear language and notations.

Among all these, what this book values most is the focus on **problems**. From the author's perspective of view, **the problem awareness** is at the core of engineering thinking. An excellent engineer should—and only should—think this way: How to define a problem, how to transform a problem, how to solve a problem, how to discover a new problem... We emphasize that **no algorithm exists in isolation from a problem; every algorithm is born to solve a particular class of problems**. Therefore, readers will see that, from an interdisciplinary perspective, we have systematically identified and defined various robotics-related problems throughout the main text. This is something that many specialized textbooks in related fields often cannot afford to address.

This book is organized by the division of 7 major parts: **Mathematical and Physical Foundations, Fundamentals of Robotics, Fundamentals of Control Theory, Fundamentals of Robotic Control, Deep Learning Methods, Reinforcement Learning Methods, and Visual Navigation Methods**. Each part is labeled with Roman numerals, and subdivided into chapters, and chapters into sections and subsections, with Arabic numerals used for their numbering.

Most of the algorithms in this book are accompanied by corresponding Python code. The core portions of the code are listed after the algorithms. The executable full version of the code has been open-sourced on GitHub: `https://github.com/weixr18/MT4R`. Readers are welcome to refer to it for comparison.

1.1 Acknowledgments

The writing of this book has drawn on a large number of existing textbooks and reference materials. First, I would like to thank the authors of those works: for the robotics fundamentals and control parts, we mainly referred to *Introduction to Robotics* and *Robotics: Modelling, Planning and Control*; for control theory, we referred to Shousong Hu's *Principles of Automatic Control*, Tianwei Wang's *The Beauty of Control (Volumes 1 and 2)*, among others; for deep learning, we referenced *Deep Learning* ("the Flower Book") and Xipeng Qiu's *Neural Networks and Deep Learning*; for reinforcement learning, we referred to Sutton's *Reinforcement Learning* and *EasyRL* ("the Mushroom Book"); for SLAM, we consulted *Fourteen Lectures on Visual SLAM*, etc.

In addition, the writing of this book also benefited from many lecture notes from undergraduate and doctoral courses taken at Tsinghua University and Beihang University. These courses opened the door to the field of robotics for me. I would like to thank the professors for their dedicated teaching: Prof. Mingguo Zhao, Prof. Zongying Shi, and Prof. Tao Zhang from the Department of Automation at Tsinghua (for the course *Intelligent Robotics*); Prof. Huangang Wang and Prof. Li Li (for *Operations Research*); Prof. Lei Huang from the School of Computer Science at Beihang (for *Reinforcement Learning*); Prof. Long Zhao from the School of Automation at Beihang (for *Modern Navigation Technologies*), and many more. Thank you all!

Moreover, many of the formula derivations and algorithmic insights in this book have been inspired by high-quality online resources, including tech blogs, Q&A sites, videos, and more. Readers can find citations to these materials in the footnotes and references of each chapter. I sincerely thank these online creators and contributors for their work.

Finally, I am deeply grateful to my family and friends for their tremendous support during the writing of this book. Thank you all—without you, this book would not have come into existence!

1.2 Copyright Notice

1.3 Contact the Author

We welcome your feedback on this book. Should you have any suggestions or come across any issues—be it in printing, formatting, text, formula derivations, figures, or other elements—please do not hesitate to reach out to us via email at weixr0605@sina.com. Kindly include "[MT4R]" in the subject line of your message. Your input is highly valued and appreciated.

2 Index

This book contains extensive content. To facilitate quick reference and browsing for readers, important symbol definitions, problems, algorithms, etc., are centrally listed here.

2.1 Symbols

2.1.1 Derivative Layout

For multivariate functions, there have traditionally been two conventions for writing derivatives, known as **numerator layout** and **denominator layout**. For a function

$$f(\cdot) : \mathbb{R}^n \to \mathbb{R}$$

the shape of its first derivative differs between the two layouts: numerator layout yields $\mathbb{R}^{1\times n}$, while denominator layout yields $\mathbb{R}^{n\times 1}$. In this book, **we unify all derivative notation to use numerator layout**, i.e.,

$$\frac{\partial f}{\partial x} \in \mathbb{R}^{1\times n}$$

We define the gradient as a variable with the same shape as the input, i.e., it is the transpose of the first partial derivative:

$$\nabla_{\mathbf{x}} f = \left(\frac{\partial f}{\partial x}\right)^T \in \mathbb{R}^n$$

For linear transformations, we have

$$\frac{\partial Ax}{\partial x} = A$$

For differential calculations (i.e., the multiplication of the derivative and the differential of the input), we have

$$dy = \frac{\partial y}{\partial x} dx$$

For gradient descent, we still have

$$x \leftarrow x + \alpha \nabla_{\mathbf{x}} f$$

2.1.2 Array Indexing

Many algorithms in this book are accompanied by Python code. We strive to make variable names in the algorithms and code correspond one-to-one for ease of reader comprehension. However, conventions in programming languages and mathematical formulas differ. In mathematical formulas, when representing a set of variables, subscripts typically start from 1, e.g., $b_1, ..., b_n$. But in Python and most other programming languages, list indices start from 0.

To resolve this issue, we uniformly adopt the following convention: in most cases, by adding redundant space in the code, we ensure that **indices in the code align with those in the formulas**. For example, for a set of variables $b_1, ..., b_n$ in an algorithm, the code will create a variable group b_ns with a total length of $n+1$, where b_ns[n] corresponds to $b[n]$. The extra unused space b_ns[0] is neither accessed nor utilized. If variable indices in the algorithm involve increments or decrements, such as b_{n-1}, the corresponding array in the code is also named b_ns, accessed via the corresponding index b_ns[$n-1$].

However, there are inevitable exceptions to this convention. For instance, for fixed-length arrays or vectors (such as point coordinates in 3D space), adding an extra dimension solely to maintain index consistency is unnecessary. In such cases, we will not include an extra dimension.

2.1.3 Robotics Notation

Parts II and IV of this book cover traditional robotics content. In these parts, for joint space, we use q to represent the robot's joint vector (generally multi-dimensional), and $\dot{q}$ to represent the joint space velocity. If considering only a single (rotational) joint, θ denotes the joint angle. For operational space, we use x_e to represent the robot's end-effector state (pose), and $\dot{x}_e$ or v_e to represent the operational space velocity (generalized velocity, including angular velocity). Operational space velocity can be expressed in two ways: if the angular part is represented by angular velocity, it is denoted as v_{ew}; if represented by the derivative of Euler angles or quaternions, it is denoted as v_{ea}.

We use $J(q)$ to denote the robot's Jacobian matrix, where the geometric Jacobian is $J_w(q)$ and the analytical Jacobian is $J_a(q)$. We use τ to represent the driving torque in joint space, and F_e to represent the external force vector on the end-effector (generalized force, including force and torque). We denote the vector composed of Euler angles as ϑ. In Parts II and IV, if quaternions are needed, to distinguish them from joint angles, we denote quaternions as $\mathbf{q}$.

2.1.4 Optimal Control

In the optimal control related chapters of Part III, we use a colon : to represent array/sequence slicing. This slicing indexing follows the MATLAB style, e.g., $a_:$ represents all elements of the sequence a; the slice $a : b$ represents elements from index a to index b, inclusive. Arrays may start indexing from 0 or 1 as needed.

2.1.5 Reinforcement Learning

Part VI of this book covers reinforcement learning, where we need to handle various random variables and distributions. For key elements in reinforcement learning, we adopt conventional symbols: s represents state, a represents action, and r represents reward. The state space is denoted $\mathcal{S}$, and the action space is denoted $\mathcal{A}$. The lowercase symbols above represent specific values and are used in general contexts.

If we wish to emphasize that something is a random variable sampled from a distribution, we use uppercase symbols: S, A, R. Note that random variables are denoted in uppercase regardless of whether they are continuous or discrete. However, for convenience of expression, we may not strictly distinguish between a random variable and its sampled value.

The distribution of a random variable relates to its nature: continuous variables correspond to probability density functions, discrete variables correspond to probability mass functions. We use uppercase symbols like $P(S_{t+1}|S_t)$ to express the probability function for discrete distributions, and lowercase symbols like $p(S_{t+1}|S_t, A_t)$ to express the probability density function for continuous distributions. It is important to note that often our derivations are set in the context of discrete variables, but the formulas also apply to continuous variables. In such cases, we tend to use lowercase symbols.

Generally, a probability distribution and the value of its corresponding function are distinct concepts. However, for convenience, we sometimes mix the notation. For example, we might use $P(S_{t+1}|S_t)$ to directly express the conditional distribution of state transition.

Sometimes we need to express the value of a probability density function at a specific value; in such cases, we write it as $p(S_{t+1} = s'|S_t = s, A_t = a)$. This notation can be cumbersome, so where it does not cause ambiguity, we simplify it to $p(s'|s, a)$. Furthermore, if we wish to emphasize the probability of a specific random event, we denote it as $Pr()$.

2.2 Abbreviations

In this book, we use a number of widely adopted abbreviations in the field to simplify expressions. Table 1 lists all abbreviations used throughout the book for reference. The index is sorted in the order the abbreviations appear in the main text.

Table 1: Abbreviation Index

Abbreviation	**Full Term**
DL	Deep Learning

Abbreviation Index – Continued from previous page

Abbreviation	Full Term
CV	Computer Vision
LLM	Large Language Model
RL	Reinforcement Learning
LSGD	Line Search Gradient Descent
F-R	Fletcher-Reeves conjugate gradients
G-N	Gauss-Newton method
L-M	Levenberg-Marquardt method
D-H	Denavit-Hartenberg parameters
FK	Forward Kinematics
IK	Inverse Kinematics
LTI	Linear Time-Invariant system
PID	Proportional Integral Derivative control
PD	Proportional Derivative control
KF	Kalman Filter
EKF	Extended Kalman Filter
UKF	Unscented Kalman Filter
HJB	Hamilton-Jacobi-Bellman equation
LQR	Linear Quadratic Regulator
iLQR	iterative Linear Quadratic Regulator
DDP	Derivative Dynamic Programming
MPC	Model Predictive Control
JS	Joint Space
OS	Operating Space
DNN	Deep Neural Network
MLP	Multi-Layer Perceptron
GD	Gradient Descent
SGD	Stochastic Gradient Descent
BSGD	Batch Stochastic Gradient Descent
DRL	Deep Reinforcement Learning
MRP	Markov Reward Process
MDP	Markov Decision Process
MC	Monte Carlo
DP	Dynamic Programming
POMDP	Partially Observable Markov Decision Process
SARSA	State-Action-Reward-State-Action
KL	Kullback-Leibler divergence
GAE	General Advantage Estimation
PPO	Proximal Policy Optimization algorithms
SLAM	Simultaneous Localization and Mapping
vSLAM	Visual Simultaneous Localization and Mapping
VO	Visual Odometer
IMU	Inertial Measurement Unit
VIO	Visual Inertial Odometer
NMS	Non-Maximum Suppression
LoG	Laplacian of Gaussian
DoG	Difference of Gaussian
HoG	Histogram of Gradient
SIFT	Scale Invariant Feature Transform
SURF	Speeded Up Robust Features
FAST	Features from Accelerated Segment Test
BRIEF	Binary Robust Independent Elementary Features

Abbreviation Index – Continued from previous page

Abbreviation	Full Term
ORB	Oriented FAST and Rotated BRIEF
DLT	Direct Linear Transform
EPnP	Effective Perspective n-Point
BA	Bundle Adjustment
MSCKF	Multi-State Constraint Kalman Filter
VINS	Visual-Inertial Navigation System

2.3 Problems

As mentioned in the introduction, **problem awareness** is crucial for robotics engineers. To emphasize this, we explicitly define the problem each algorithm solves and summarize them in the following table for reference.

For short periods, some abbreviations used only in the table below are as follows

- [Eq.Cons.] – Equation Constrained
- [Ineq.Cons.] – Inequation Constrained
- [UnCons.] – Unconstrained
- [Disc.] – [Discrete]
- [Cont.] – [Continuous]
- [M.F.] – [Model-free]
- [M.B.] – [Model-based]

Table 2: Problem Index Table

Domain	Problem Name	Section	Description
Optimization	General Nonlinear Optimization	3.1	Find the optimal variables that minimize the objective function under equality and inequality constraints
Optimization	[Eq.Cons.] Nonlinear Optimization	3.1	Find the optimal variables that minimize the objective function under equality constraints
Optimization	[Ineq.Cons.] Nonlinear Optimization	3.1	Find the optimal variables that minimize the objective function under inequality constraints
Optimization	[UnCons.] Nonlinear Optimization	3.1	Find the optimal variables that minimize the objective function without constraints
Optimization	[UnCons.] Least Squares	3.1	Find the optimal variables that minimize a nonlinear quadratic objective function
Optimization	[UnCons.] QP Problem	3.1	Find the optimal variables that minimize a quadratic objective function without constraints

Problem Index Table – Continued

Domain	Problem Name	Section	Description
Optimization	[Eq.Cons.] QP Problem	3.1	Find the optimal variables that minimize a quadratic objective function under equality constraints
Optimization	General QP Problem	3.1	Find the optimal variables that minimize a quadratic objective function under equality and inequality constraints
Optimization	Line Search Problem	3.2.3	Given an objective function, initial point, and update direction, find the step size that minimizes the function
Interpolation	C^1-continuous Interpolation	4.1	Given sampled points of a scalar function, find an interpolation function with continuous first derivative
Interpolation	C^2-continuous Interpolation	4.1	Given sampled points of a scalar function, find an interpolation function with continuous second derivative
Robotics	Forward Kinematics	6.2	Given fixed parameters and joint vector, compute the end-effector state
Robotics	Inverse Kinematics	6.3	Given fixed parameters and end-effector state, compute the joint vector
Control Theory	Regulation Control	10.1.3	Given an open-loop system, design a controller to stabilize the output at a reference and ensure system stability
Control Theory	Tracking Control	10.1.3	Given an open-loop system, design a controller to track a reference and ensure system stability
Control Theory	State Observation	11.1.1	Given system state and output equations, design an observer to converge observation error
Control Theory	Linear State Observation	11.1.1	Given linear system state and output equations, design an observer to converge observation error
Control Theory	Denoising State Estimation (Filtering) [Disc.]	11.1.2	Given discrete system state and output equations, design a filter to reduce state estimation error
Control Theory	Optimal Filtering [Disc.]	11.1.2	Given discrete system state and output equations, design a filter to minimize state estimation error
Control Theory	Denoising State Estimation (Filtering) [Cont.]	11.1.2	Given continuous system state and output equations, design a filter to reduce state estimation error

Problem Index Table – Continued

Domain	Problem Name	Section	Description
Control Theory	Optimal Filtering [Cont.]	11.1.2	Given continuous system state and output equations, design a filter to minimize state estimation error
Control Theory	Linear Filtering [Disc.]	11.3.1	Given discrete linear system state and output equations, design a filter to reduce estimation error
Control Theory	Linear Optimal Filtering [Disc.]	11.3.1	Given discrete linear system, design a filter to minimize estimation variance
Control Theory	Linear Filtering [Cont.]	11.3.1	Given continuous linear system, design a filter to reduce estimation error
Control Theory	Linear Optimal Filtering [Cont.]	11.3.1	Given continuous linear system, design a filter to minimize estimation variance
Optimal Control	[UnCons.] General Optimal Regulation [Disc.]	12.1	Given a discrete system and cost function, find the optimal regulation control input
Optimal Control	[UnCons.] General Optimal Tracking [Disc.]	12.1	Given a discrete system, cost function, and reference trajectory, find the optimal tracking input
Optimal Control	[UnCons.] Quadratic Optimal Regulation [Disc.]	12.1	Given a discrete system, find the regulation input that minimizes a quadratic cost
Optimal Control	[UnCons.] Quadratic Optimal Tracking [Disc.]	12.1	Given a discrete system, find the tracking input that minimizes a quadratic cost
Optimal Control	[UnCons.] General Optimal Regulation [Cont.]	12.3	Given a continuous system and cost function, find the optimal regulation control input
Optimal Control	[UnCons.] General Optimal Tracking [Cont.]	12.3	Given a continuous system, cost function, and desired trajectory, find the optimal tracking input
Optimal Control	LQR Regulation Control [Disc.]	12.4.1	Given a linear discrete system, find the regulation input that minimizes a quadratic cost
Optimal Control	Infinite-Horizon LQR Regulation [Disc.]	12.4.1	Given a linear discrete system, find the control input that minimizes infinite-horizon quadratic cost
Optimal Control	LQR Regulation Control [Cont.]	12.5	Given a linear continuous system, find the regulation input that minimizes a quadratic cost
Optimal Control	LQR Tracking Control [Disc.]	12.7.1	Given a linear discrete system, find the tracking input that minimizes a quadratic cost
Optimal Control	LQR Smooth Tracking Control [Disc.]	12.7.2	Given a linear discrete system, find the smooth tracking input that minimizes a quadratic cost

Problem Index Table – Continued

Domain	Problem Name	Section	Description
Optimal Control	[Eq.Cons.]Optimal Regulation [Cont.]	12.8	Given a continuous system and cost function, find the optimal regulation input under equality constraints
Optimal Control	Differential-Constrained Optimal Regulation [Cont.]	12.8	Given a continuous system and cost function, find the optimal regulation input under differential constraints
Optimal Control	Integral-Constrained Optimal Regulation [Cont.]	12.8	Given a continuous system and cost function, find the optimal regulation input under integral constraints
Optimal Control	Constrained LQR Regulation [Disc.]	13.3	Given a linear discrete system and constraints, find the regulation input minimizing quadratic cost
Robot Control	Joint-Space Regulation Control	14.1	Given robot parameters, design a controller to stabilize joint angles at reference values
Robot Control	Joint-Space Tracking Control	14.1	Given robot parameters, design a controller to track reference joint angles
Robot Control	Operational-Space Regulation Control	14.1	Given robot parameters, design a controller to stabilize end-effector at reference pose
Robot Control	Operational-Space Tracking Control	14.1	Given robot parameters, design a controller to track reference end-effector pose
Robot Control	Joint-Space Impedance Control	14.2	Given robot parameters and state, design a controller to realize joint-space impedance behavior
Robot Control	Operational-Space Impedance Control	14.2	Given robot parameters and state, design a controller to realize end-effector impedance behavior
Robot Control	Operational-Space Zero-Force Control	14.2	Given robot parameters and state, design a controller to achieve zero-force behavior at end-effector
Robot Control	Joint-Space Admittance Control	14.2	Given robot parameters and external torques, design trajectory to realize joint-space admittance behavior
Robot Control	Operational-Space Admittance Control	14.2	Given robot parameters and external forces, design trajectory to realize end-effector admittance behavior
Robot Control	Constraint Force Control	14.3	Given robot parameters, design a controller to satisfy constraints and track desired force and velocity

Problem Index Table – Continued

Domain	Problem Name	Section	Description
Robot Control	Joint Anti-Disturbance Regulation	15.1.4	Given motor parameters, design a controller to stabilize joint angles and suppress disturbances
Robot Control	Joint Anti-Disturbance Tracking	15.1.4	Given motor parameters, design a controller to track joint angles and suppress disturbances
Deep Learning	Regression Problem	18.1.1	Given a dataset and labels, find the optimal mapping to fit input-output relation
Deep Learning	Classification Problem	18.1.1	Given a dataset and labels, find the optimal mapping to fit input-class relation
Deep Learning	Dimensionality Reduction	18.1.1	Given a dataset, find encoding-decoding mapping to minimize reconstruction error
Deep Learning	Clustering Problem	18.1.1	Given a dataset, find clustering with low intra-class and high inter-class variance
Deep Learning	MLP Gradient Computation	18.3.2	Given MLP loss, input, and parameters, compute gradients of parameters
Deep Learning	MLP Parameter Learning	18.3.2	Given dataset and MLP loss, find optimal parameters
Reinforcement Learning	MRP Value Estimation [M.B.]	20.3	Given MRP model, estimate state values
Reinforcement Learning	MRP Value Estimation [M.F.]	20.3	Without MRP model, estimate state values
Reinforcement Learning	MDP Prediction [M.B.]	20.5	Given MDP model and policy, estimate state values
Reinforcement Learning	MDP Prediction [M.F.]	20.5	Without MDP model, estimate state values under a policy
Reinforcement Learning	MDP Control [M.B.]	20.6	Given MDP model, find optimal policy
Reinforcement Learning	MDP Control [M.F.]	20.7	Without MDP model, find optimal policy
Reinforcement Learning	POMDP Control [M.F.]	20.7	For partially observable and unknown model, find optimal policy
Reinforcement Learning	MDP Control [Parameterized, Model-Free]	22.1	Without MDP model, find optimal parameterized policy
Visual SLAM	Monocular Visual Odometry	24.1.1	Given image sequence, estimate robot pose sequence
Visual SLAM	Stereo Visual Odometry	24.1.2	Given stereo baseline and image sequence, estimate robot pose sequence
Visual SLAM	RGB-D Visual Odometry	24.1.2	Given RGB and depth images, estimate robot pose sequence
Visual SLAM	Monocular Visual-Inertial Odometry	24.1.2	Given image and IMU sequences, estimate robot pose sequence
Visual SLAM	Feature Point Detection	25.2	Given an image, detect representative pixel coordinates

Problem Index Table – Continued

Domain	Problem Name	Section	Description
Visual SLAM	Feature Point Matching	25.2	Given two images with features, find matching pairs
Visual SLAM	Optical Flow Estimation	25.6	Given two images, estimate discrete optical flow
Visual SLAM	2D-2D Pose Estimation	26.1	Given matching 2D pixel coordinates, estimate relative pose matrix
Visual SLAM	3D-3D Pose Estimation	26.1	Given matching 3D coordinates, estimate relative pose matrix
Visual SLAM	3D-2D Pose Estimation	26.1	Given 3D points and 2D projections, estimate relative pose matrix
Visual SLAM	Triangulation	26.2.2	Given relative pose and matching 2D pixels, compute 3D coordinates
Visual SLAM	2D-2D Homography Estimation	26.2.5	Given coplanar 2D matching points, estimate homography matrix
Visual SLAM	2D-2D Motion Estimation	26.2.6	Given 2D matching points, estimate 3D linear and angular velocities

2.4 Algorithms

This book introduces a large number of algorithms related to robotics. In addition to providing the algorithmic procedures, we also extract the key elements of each algorithm (such as the problem it addresses, inputs, outputs, etc.) through an algorithm element table. For some algorithms, Python implementations are also provided. To facilitate reference, all algorithms introduced in this book are listed here. The symbol † in the table indicates that the algorithm has a tested and verified Python implementation.

Most of the algorithms in this book are implemented with Python code (core parts) in the main text. The ◦ symbol in the table indicates that Python code is provided for the algorithm, while the † symbol indicates that the corresponding Python code has passed runtime tests. The executable full version of the code has been open-sourced on GitHub: `https://github.com/weixr18/MT4R`. Readers are welcome to refer to it for comparison.

Table 3: Algorithm Index Table

Algorithm	ID	Task	Solution Type
Gradient Descent†	1	[UnCons.] Nonlinear Optimization	Iterative Solution
Newton's Method†	2	[UnCons.] Nonlinear Optimization	Iterative Solution
Golden Section Method†	3	Line Search Problem	Iterative Solution
Line Search Gradient Descent†	4	[UnCons.] Nonlinear Optimization	Iterative Solution
Damped Newton Method†	5	[UnCons.] Nonlinear Optimization	Iterative Solution
F-R Conjugate Gradient Method†	6	[UnCons.] Nonlinear Optimization	Iterative Solution

Algorithm Index Table – Continued

Algorithm	**ID**	**Task**	**Solution Type**
Lagrange Multiplier Method†	7	[Eq.Cons.] Nonlinear Optimization	Iterative Solution
Barrier Function LSGD†	8	[Ineq.Cons.] Nonlinear Optimization	Iterative Solution
Unconstrained QP Analytical Solution†	9	[UnCons.] QP Problem	Analytical Solution
[Eq.Cons.] QP Analytical Solution†	10	[Eq.Cons.] QP Problem	Analytical Solution
QP Barrier Method†	11	General QP Problem	Iterative Solution
Gradient Descent for LS†	12	[UnCons.] Least Squares	Iterative Solution
Gauss-Newton Method†	13	[UnCons.] Least Squares	Iterative Solution
Levenberg-Marquardt Method†	14	[UnCons.] Least Squares	Iterative Solution
Quadratic Spline Interpolation†	15	C^1-Continuous Interpolation	Analytical Solution
Cubic Spline Interpolation†	16	C^2-Continuous Interpolation	Analytical Solution
Forward Kinematics Analytical Solution◦	17	Forward Kinematics Solving	Analytical Solution
Analytical Jacobian Analytical Solution◦	18	Analytical Jacobian Computation	Analytical Solution
Centroid Jacobian Analytical Solution◦	19	Centroid Jacobian Computation	Analytical Solution
Gradient Descent IK◦	20	Inverse Kinematics Solving	Iterative Solution
Gauss-Newton IK◦	21	Inverse Kinematics Solving	Iterative Solution
Damped Least Squares IK◦	22	Inverse Kinematics Solving	Iterative Solution
Dynamics Analytical Solution	23	Dynamics Term Computation	Analytical Solution
Discrete Kalman Filter†	24	Linear Optimal Filtering [Disc.]	Filter, Recursive
Continuous Kalman Filter	25	Linear Optimal Filtering [Cont.]	Filter, Recursive
Extended Kalman Filter†	26	Optimal Filtering [Disc.]	Filter, Recursive
Error-State Kalman Filter†	27	Optimal Filtering [Disc.]	Filter, Recursive
Unscented Kalman Filter◦	28	Optimal Filtering [Disc.]	Filter, Recursive
Optimal Control - Dynamic Programming	29	(UnCons.) General Optimal Regulation [Disc.]	Model-Based, DP, Analytical Derivation
Discrete LQR Iterative Solution◦	30	LQR Regulation Control [Disc.]	Model-Based, DP, Analytical Solution
Continuous LQR Analytical Solution	31	LQR Regulation Control [Cont.]	Model-Based, DP, Analytical Solution
Nonlinear Optimal Control - DDP	32	(UnCons.) General Optimal Regulation [Disc.]	Model-Based, DP, Iterative Solution
Nonlinear Optimal Control - iLQR	33	(UnCons.) General Optimal Regulation [Disc.]	Model-Based, DP, Iterative Solution
LQR Tracking Control◦	34	LQR Tracking Control [Disc.]	Model-Based, DP, Iterative Solution
LQR Input Increment Control	35	LQR Smooth Tracking Control [Disc.]	Model-Based, DP, Analytical Solution
Unconstrained Linear MPC	36	LQR Regulation Control [Disc.]	Model-Based, MPC, Analytical Solution
[Ineq.Cons.] Linear MPC	37	Constrained LQR Regulation Control [Disc.]	Model-Based, MPC, Iterative Solution
Independent Joint PI Control	38	Joint Disturbance Rejection Regulation Control	Model-Based, Analytical Solution
Independent Joint PID Control	39	Joint Disturbance Rejection Tracking Control	Model-Based, Analytical Solution
JS Gravity Compensation PD Control	40	Joint Space Regulation Control	Model-Based, Analytical Solution
JS Inverse Dynamics PD Control	41	Joint Space Tracking Control	Model-Based, Analytical Solution

Algorithm Index Table – Continued

Algorithm	ID	Task	Solution Type
OS Gravity Compensation PD Control	42	Operational Space Regulation Control	Model-Based, Analytical Solution
OS Inverse Dynamics PD Control	43	Operational Space Tracking Control	Model-Based, Analytical Solution
OS Impedance Control	44	Operational Space Impedance Control	Model-Based, Analytical Solution
OS Zero-Force Control	45	Operational Space Zero-Force Control	Model-Based, Analytical Solution
MLP Forward Propagation†	46	MLP Inference	Analytical Solution
MLP Backpropagation†	47	MLP Gradient Computation	Analytical Solution
GD Optimization†	48	MLP Parameter Learning	Iterative Solution
SGD Optimization†	49	MLP Parameter Learning	Iterative Solution
BSGD Optimization†	50	MLP Parameter Learning	Iterative Solution
BSGD with Learning Rate Decay†	51	MLP Parameter Learning	Iterative Solution
RMSProp Optimization†	52	MLP Parameter Learning	Iterative Solution
Momentum BSGD Optimization†	53	MLP Parameter Learning	Iterative Solution
Adam Optimization†	54	MLP Parameter Learning	Iterative Solution
MLP-BN Forward Propagation†	55	MLP Inference	Analytical Solution
MLP-BN Backpropagation†	56	MLP Gradient Computation	Analytical Solution
MRP Analytical Value Solution	57	MRP Value Estimation [M.B.]	Model-Based, Tabular
MRP-DP Value Estimation	58	MRP Value Estimation [M.B.]	Model-Based, Tabular, Bootstrapping
MRP-MC Value Estimation	59	MRP Value Estimation [M.F.]	Model-Free, Tabular, Monte Carlo
MDP-DP Policy Evaluation◦	60	MDP Prediction Problem [M.B.]	Model-Based, Tabular, Bootstrapping
MDP-MC Policy Evaluation◦	61	MDP Prediction Problem [M.F.]	Model-Free, Tabular, Monte Carlo
MDP-TD(0) Policy Evaluation◦	62	MDP Prediction Problem [M.F.]	Model-Free, Tabular, TD
MDP-TD(n) Policy Evaluation◦	63	MDP Prediction Problem [M.F.]	Model-Free, Tabular, TD
MDP-DP Policy Iteration◦	64	MDP Control Problem [M.B.]	Model-Based, Tabular, Bootstrapping
MDP Deterministic Value Iteration◦	65	MDP Control Problem [M.B.]	Model-Based, Tabular, Bootstrapping
MDP-MC Policy Iteration◦	66	MDP Control Problem [M.F.]	Value-Based, On-Policy?, Tabular
MDP-SARSA Algorithm◦	67	MDP Control Problem [M.F.]	Value-Based, On-Policy, Tabular
MDP-SARSA(n) Algorithm◦	68	MDP Control Problem [M.F.]	Value-Based, On-Policy, Tabular
MDP-Q Learning Algorithm◦	69	MDP Control Problem [M.F.]	Value-Based, Off-Policy, Tabular
Original REINFORCE◦	70	MDP Control Problem [M.F.]	Policy-Based, On-Policy, Policy Gradient
REINFORCE Algorithm◦	71	MDP Control Problem [M.F.]	Policy-Based, On-Policy, Policy Gradient
Actor-Critic Algorithm	72	MDP Control Problem [M.F.]	Policy-Based, On-Policy, Policy Gradient
Actor-Critic Algorithm with GAE	73	MDP Control Problem [M.F.]	Policy-Based, On-Policy, Policy Gradient
Original PPO Algorithm	74	MDP Control Problem [M.F.]	Policy-Based, On-Policy, Policy Gradient
PPO-Penalty Algorithm	75	MDP Control Problem [M.F.]	Policy-Based, On-Policy, Policy Gradient
PPO-Clip Algorithm	76	MDP Control Problem [M.F.]	Policy-Based, On-Policy, Policy Gradient
Harris Corner Detection	78	Feature Point Detection Problem	Traditional CV
SIFT Corner Detection	80	Feature Point Detection Problem	Traditional CV
Optical Flow Feature Matching	81	Feature Point Matching Problem	Traditional CV
L-K Optical Flow	82	Optical Flow Estimation Problem	Traditional CV
Iterative L-K Optical Flow	83	Optical Flow Estimation Problem	Traditional CV
Pyramidal L-K Optical Flow	84	Optical Flow Estimation Problem	Traditional CV

Algorithm Index Table – Continued

Algorithm	ID	Task	Solution Type
Triangulation†	85	Triangulation Problem	Approximate Solution
Pose Recovery from Essential Matrix†	86	2D-2D Point Pose Estimation	Approximate Solution
8-Point Pose Estimation†	87	2D-2D Point Pose Estimation	Approximate Solution
4-Point Homography Estimation○	88	2D-2D Point Homography Estimation	Approximate Solution
6-Point 3D Velocity Estimation†	89	2D-2D Point 3D Velocity Estimation	Approximate Solution
ICP Algorithm†	90	3D-3D Point Pose Estimation	Approximate Solution
DLT Algorithm†	91	3D-2D Point Pose Estimation	Approximate Solution
EPnP Algorithm†	92	3D-2D Point Pose Estimation	Approximate Solution
GN-BA Pose Estimation†	93	3D-2D Point Pose Estimation	Iterative Solution

Part I
Mathematical and Physical Foundations

1 3D Coordinate Systems

1.1 Three-Dimensional Space

1.1.1 Vectors and Coordinates

We live in a three-dimensional world. To clearly express spatial relationships in this 3D world, we need to establish a three-dimensional Cartesian coordinate system.

Take three mutually perpendicular unit vectors $\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}$ as the basis vectors of the Cartesian coordinate system. This set of basis vectors spans the entire three-dimensional space. The directions of these basis vectors are conventionally referred to as the x-axis, y-axis, and z-axis.

Three-dimensional space has a handedness property. By convention, **all 3D coordinate systems in this book are right-handed**. Specifically: extend your right palm with fingers pointing along the x-axis, the direction perpendicular to the palm and outward is the y-axis; make a fist with your right hand while keeping the thumb extended, then the thumb direction represents the z-axis.

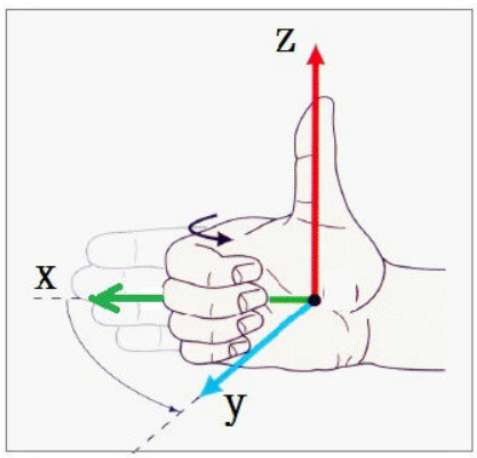

Figure I.1.1: Right-handed coordinate system

A **vector** is a quantity with both magnitude and direction. In three-dimensional space, any directed line segment between two spatial points can also be considered a vector. In physics, many vectors have their own physical meanings. Some vectors can be freely translated without changing their physical significance, such as velocity, acceleration, and momentum; some vectors can slide along their axes without altering their physical meaning, like forces and torques on rigid bodies; some vectors cannot be moved, such as position vectors. In this chapter, we mainly discuss abstract vectors in coordinate systems, also called three-dimensional vectors. Throughout this chapter, all vectors will be denoted with arrow symbols to emphasize their distinction from scalars.

Any vector **a** in three-dimensional space can be orthogonally decomposed onto a set of basis vectors in coordinate system o:

$$\vec{\mathbf{a}} = a_1^o \vec{\mathbf{o_1}} + a_2^o \vec{\mathbf{o_2}} + a_3^o \vec{\mathbf{o_3}} \tag{I.1.1}$$

That is, any vector can be expressed as a linear combination of basis vectors. Given a set of basis vectors, the coefficients of this linear combination can be used to represent the vector in this coordinate system. We call this the **coordinates of the vector**, denoted as

$$\mathbf{a}^o = \begin{pmatrix} a_1^o \\ a_2^o \\ a_3^o \end{pmatrix} \tag{I.1.2}$$

Thus, we have the relationship between vectors, coordinates, and basis vectors:

$$\vec{\mathbf{a}} = [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}] \begin{pmatrix} a_1^o \\ a_2^o \\ a_3^o \end{pmatrix} = [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\mathbf{a}^o \tag{I.1.3}$$

Note here: vectors exist independently of coordinates, and a set of coordinates for a vector always corresponds to a specific set of basis vectors. In subsequent representations, for convenience and when no ambiguity arises, we may use the coordinates of a vector to represent the vector itself. In such cases, we sometimes refer to these default coordinate representations as **vectors**.

As shown earlier, we denote the coordinate system as a superscript on the vector coordinates. Take a set of basis vectors $\mathbf{o_1}, \mathbf{o_2}, \mathbf{o_3}$ as the **default basis vectors**, and the coordinate system is called the **o-system**. We hereby agree: if a vector is represented in the default coordinate system, the superscript can be omitted.

1.1.2 Vectors and Cross Products

Three-dimensional space has vector cross products. The result of a 3D vector cross product is another 3D vector, independent of the chosen coordinate system. The magnitude of the resulting vector is the area of the parallelogram spanned by the two input vectors, and its direction is perpendicular to both input vectors, determined by the right-hand rule according to the order of the cross product, as shown in Figure I.1.2.

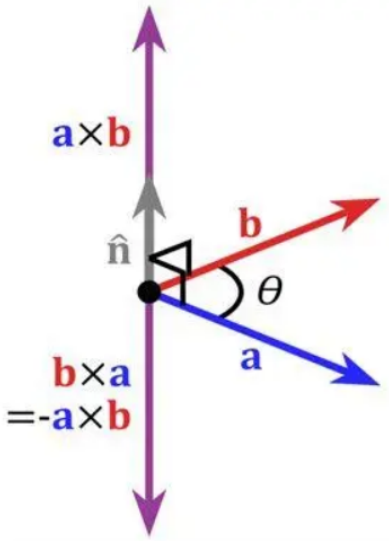

Figure I.1.2: Vector cross product

In the same coordinate system, the cross product of vectors has the following coordinate relationship:

$$\mathbf{a} \times \mathbf{b} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix} \tag{I.1.4}$$

We define the 3×3 matrix formed by vector $\mathbf{a}$ as the **skew-symmetric matrix** of $\mathbf{a}$, namely:

$$\mathbf{a}_\times := \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{I.1.5}$$

Therefore, the cross product of 3D vectors can be written in matrix multiplication form:

$$\mathbf{a} \times \mathbf{b} = (\mathbf{a}_\times)\mathbf{b} \tag{I.1.6}$$

1.2 Rotation Matrices

1.2.1 Two Descriptions of Rotation

Rotation is relative motion. Imagine a ballet dancer spinning on stage. Assume the stage is stationary, with the coordinate system fixed to the stage being the n-system; the coordinate system fixed to the dancer's body is the b-system. Choose an appropriate point in space as the origin for both coordinate systems (to avoid translation complications). The definition of "up" is consistent in both coordinate systems (opposite to the direction of gravity).

If we, as spectators, watch the dancer rotating counterclockwise from the audience seats above the stage, we can say: the b-system is rotating relative to the n-system at a certain angular velocity pointing upward. From the dancer's perspective, she sees the entire world rotating relative to her body. That is, in the b-system, the n-system is rotating relative to the b-system at a certain angular velocity pointing downward. **For the same rotation, there are at least two different perspectives to observe it**. This aligns with our intuitive understanding of relative motion.

Now, let's observe from the spectator's perspective. Suppose we capture a segment of the dancer's rotation—how can we describe this rotation? We have two methods.

- **Vector rotation**: Each vector fixed to the dancer's body rotates relative to the n-system by a certain amount.
- **Coordinate system rotation**: The b-system rotates relative to the n-system by a certain amount.

Note that the difference between these two descriptions lies only in the language used. They both describe the same rotational motion in the three-dimensional world. In fact, **for the same rotational motion, we can always use both descriptions simultaneously**.

So, what is the connection between these two descriptions?

Under the **vector rotation** description, take a vector $\mathbf{a}$ fixed to the dancer's body, with coordinates $\mathbf{a}^n$ in the n-system. Assume after rotation, $\mathbf{a}$ moves to position $\mathbf{a}'$, with coordinates $\mathbf{a}'^n$ in the n-system. For this rotation, for any body-fixed vector $\mathbf{a}$, there exists a matrix R_v satisfying:

$$\mathbf{a}'^n = R_v \mathbf{a}^n \tag{I.1.7}$$

Under the **coordinate system rotation** description, take a vector $\mathbf{c}$ fixed to the stage, with coordinates $\mathbf{c}^n$ in the n-system. Assume at the start of rotation, the b-system and n-system coincide. After this rotation, the b-system and n-system no longer coincide. Now, the vector $\mathbf{c}$ has coordinates $\mathbf{c}^b$ in the b-system. For this rotation, for any stage-fixed vector $\mathbf{c}$, there exists a matrix R_n^b satisfying:

$$\mathbf{c}^b = R_n^b \mathbf{c}^n \tag{I.1.8}$$

In practical use, the R matrices in both descriptions are called **rotation matrices**. Depending on the need, we flexibly switch between the two descriptions. However, **most of the time, we use the coordinate system rotation description**. In fact, these two matrices R_v and R_n^b have a certain relationship:

$$R_v^T = R_n^b$$

Intuitively, this makes sense. The dancer's rotation relative to the stage and the stage's rotation relative to the dancer should be "equal in magnitude but opposite in direction." When represented by orthogonal rotation matrices, they are transposes/inverses of each other.

Next, we formally define rotation matrices using the coordinate system rotation description.

1.2.2 Definition of Rotation Matrices

In three-dimensional space, we can define different coordinate systems. If two coordinate systems share the same origin, then the relationship between them is a three-dimensional rotation—one coordinate system can be transformed into another through certain rotational steps.

To describe the rotational relationship between different coordinate systems, we can write a matrix composed of pairwise inner products of two sets of basis vectors. Suppose in addition to the basis vectors of the O-system, we have another set of basis vectors $\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \vec{\mathbf{e_3}}$ forming another coordinate system, the E-system. We define the **rotation matrix from the O-system to the E-system** ${}^{e}_{o}R$ as follows:

$$
{}^{e}_{o}R := \begin{bmatrix} \vec{\mathbf{e_1}} \cdot \vec{\mathbf{o_1}} & \vec{\mathbf{e_1}} \cdot \vec{\mathbf{o_2}} & \vec{\mathbf{e_1}} \cdot \vec{\mathbf{o_3}} \\ \vec{\mathbf{e_2}} \cdot \vec{\mathbf{o_1}} & \vec{\mathbf{e_2}} \cdot \vec{\mathbf{o_2}} & \vec{\mathbf{e_2}} \cdot \vec{\mathbf{o_3}} \\ \vec{\mathbf{e_3}} \cdot \vec{\mathbf{o_1}} & \vec{\mathbf{e_3}} \cdot \vec{\mathbf{o_2}} & \vec{\mathbf{e_3}} \cdot \vec{\mathbf{o_3}} \end{bmatrix} \tag{I.1.9}
$$

This definition has several key points. First, all rotation matrices represent transformations from one coordinate system to another, requiring two coordinate systems with a specific order. Second, to clearly indicate these two coordinate systems, we write them on the left side of the matrix, with the original coordinate system below and the new coordinate system above.

Thus, this matrix can be used to handle the **coordinate transformation of the same vector from the O-frame to the E-frame**:

$$
{}^{E}_{O}R\mathbf{a}^o = \begin{bmatrix} \vec{\mathbf{e_1}} \cdot \vec{\mathbf{o_1}} & \vec{\mathbf{e_1}} \cdot \vec{\mathbf{o_2}} & \vec{\mathbf{e_1}} \cdot \vec{\mathbf{o_3}} \\ \vec{\mathbf{e_2}} \cdot \vec{\mathbf{o_1}} & \vec{\mathbf{e_2}} \cdot \vec{\mathbf{o_2}} & \vec{\mathbf{e_2}} \cdot \vec{\mathbf{o_3}} \\ \vec{\mathbf{e_3}} \cdot \vec{\mathbf{o_1}} & \vec{\mathbf{e_3}} \cdot \vec{\mathbf{o_2}} & \vec{\mathbf{e_3}} \cdot \vec{\mathbf{o_3}} \end{bmatrix} \begin{pmatrix} a_1^o \\ a_2^o \\ a_3^o \end{pmatrix} = \begin{pmatrix} \vec{\mathbf{e_1}} \cdot \vec{\mathbf{a}} \\ \vec{\mathbf{e_2}} \cdot \vec{\mathbf{a}} \\ \vec{\mathbf{e_3}} \cdot \vec{\mathbf{a}} \end{pmatrix} = \begin{pmatrix} a_1^e \\ a_2^e \\ a_3^e \end{pmatrix} = \mathbf{a}^e
$$

That is,

$$
\mathbf{a}^e = {}^{e}_{o}R\mathbf{a}^o \tag{I.1.10}
$$

On one hand, for the same vector, the rotation matrix can transform its coordinates from one coordinate system to another. On the other hand, the rotation matrix can also transform one set of coordinate basis vectors into another set of coordinate basis vectors:

$$
[\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}] = [\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \vec{\mathbf{e_3}}]\, {}^{e}_{o}R \tag{I.1.11}
$$

In fact, according to the above definition, the columns of the rotation matrix ${}^{e}_{o}R$ are the coordinates of the basis vectors of the O-frame in the E-frame:

$$
{}^{E}_{O}R = \begin{bmatrix} \vec{\mathbf{o}}_1^e & \vec{\mathbf{o}}_2^e & \vec{\mathbf{o}}_3^e \end{bmatrix} \tag{I.1.12}
$$

If there are multiple coordinate systems e_1, e_2, e_3, the rotation matrices can be composed by direct multiplication:

$$
{}^{e_3}_{o}R = {}^{e_3}_{e_2}R\, {}^{e_2}_{e_1}R\, {}^{e_1}_{o}R \tag{I.1.13}
$$

From the definition of the rotation matrix, it is easy to see that any rotation matrix R has the following properties:

$$
R^T = R^{-1} \tag{I.1.14}
$$

By default, the two coordinate systems corresponding to the rotation matrix are right-handed systems, so we also have:

$$
det(R) = 1 \tag{I.1.15}
$$

In subsequent chapters, where no ambiguity arises, we will also denote ${}^{e}_{o}R$ as R_o^e.

1.3 Homogeneous Matrix

In three-dimensional coordinate systems, in addition to rotational relationships, there may also be translational relationships between two coordinate systems. To express this relationship, we define the translation vector:

$$\vec{t}_{oe} = \overrightarrow{OE} \tag{I.1.16}$$

Translation vectors are generally represented in a coordinate system. For example, t^e_{eo} represents the coordinate representation of the position vector from the origin of the E-frame to the origin of the O-frame in the E-frame.

Using the translation vector $\vec{t}^{\,e}_{eo}$ and the rotation matrix R^e_o, we can express the positional relationship between any two three-dimensional coordinate systems in three-dimensional space; we can also perform coordinate transformations for any vector between these two coordinate systems. For example, for any vector $\mathbf{a}^o$, we have:

$$\mathbf{a}^e = R^e_o \mathbf{a}^o + t^e_{eo} \tag{I.1.17}$$

When performing translational transformations between two coordinate systems, two different translation vectors are used: t^e_{eo} and t^o_{oe}. Their relationship also involves the rotation matrix between the two coordinate systems:

$$t^e_{eo} = -R^e_o t^o_{oe} \tag{I.1.18}$$

To make coordinate transformation representations more concise, we can define the **homogeneous matrix**:

$$T^e_o = \begin{bmatrix} R^e_o & t^e_{eo} \\ 0_{1\times 3} & 1 \end{bmatrix} \tag{I.1.19}$$

At the same time, we also augment the three-dimensional coordinates. The transformation between **augmented coordinates** is homogeneous, making the representation more concise:

$$\begin{bmatrix} \mathbf{a}^e \\ 1 \end{bmatrix} = T^e_o \begin{bmatrix} \mathbf{a}^o \\ 1 \end{bmatrix} \tag{I.1.20}$$

When performing continuous mixed coordinate transformations involving rotation and translation between multiple coordinate systems, using augmented coordinates and homogeneous matrix multiplication is very straightforward:

$$T^e_o = T^e_n ... T^3_2 T^2_1 T^1_o \tag{I.1.21}$$

1.4 Derivative of Position Vectors in Moving Coordinate Systems

As mentioned in the previous section, the positional relationship between two coordinate systems in space includes relative translation and relative rotation. Similarly, the relative motion between two coordinate systems can be divided into translational motion and rotational motion. Homogeneous matrices can solve the problem of vector coordinate transformations between relatively stationary coordinate systems but cannot address the problem of deriving position vectors in moving coordinate systems.

Specifically, as mentioned earlier, vectors can be classified into non-movable vectors, sliding vectors, and movable vectors. For non-movable vectors (primarily position vectors), their derivatives in moving coordinate systems are related to the motion between coordinate systems and are more complex than those of movable vectors.

In the 19th century, the scientist Gaspard Gustave de Coriolis studied this problem and summarized an important formula. In the following discussion, we take the static coordinate system as the default o-frame and the moving coordinate system as the e-frame, with their origins denoted as O and E, respectively.

Assume there exists a point P in space that is independent of the coordinate system. Its position vectors (relative to the origins) in the o-frame and e-frame are denoted as:

$$\overrightarrow{OP} = [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\mathbf{p}^o$$
$$\overrightarrow{EP} = [\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \vec{\mathbf{e_3}}]\mathbf{p}^e$$

Note: In this notation, $\mathbf{p}^o$ and $\mathbf{p}^e$ are **projections of different spatial vectors in different coordinate systems**, not projections of the same spatial vector in different coordinate systems. Therefore, they do not satisfy the rotational coordinate transformation (Equation I.1.10) or homogeneous coordinate transformation.

Additionally, we denote the translation vector between the two coordinate systems as:

$$\overrightarrow{OE} = [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\mathbf{t}^o_{oe}$$

For a point in three-dimensional space, we can write the triangular vector equation:

$$\overrightarrow{OP} = \overrightarrow{OE} + \overrightarrow{EP}$$

Expressed using basis vectors and coordinates, it becomes:

$$[\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\mathbf{p}^o = [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\mathbf{t}^o_{oe} + [\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \vec{\mathbf{e_3}}]\mathbf{p}^e$$

Now, suppose that frame e has relative translation and rotation with respect to frame o. The instantaneous velocity of the relative translation is $\mathbf{v}$, and the instantaneous angular velocity of the relative rotation is ω. Thus, we have two fundamental relations.

$$\begin{aligned} \frac{d}{dt}\mathbf{t}^o_{oe} &= \mathbf{v}^o_{oe} \\ \frac{d}{dt}\vec{\mathbf{e_i}} &= \overrightarrow{\omega} \times \vec{\mathbf{e_i}} \end{aligned} \tag{I.1.22}$$

Differentiating the expanded form of the above triangular vector equation with respect to time yields:

$$[\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\frac{d}{dt}\mathbf{p}^o = [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\frac{d}{dt}\mathbf{t}^o_{oe} + \frac{d}{dt}([\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \vec{\mathbf{e_3}}]\mathbf{p}^e)$$

Applying the rules of differentiation, we can expand the latter term as:

$$[\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\frac{d}{dt}\mathbf{p}^o = [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\frac{d}{dt}\mathbf{t}^o_{oe} + [\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \vec{\mathbf{e_3}}]\frac{d}{dt}\mathbf{p}^e + \frac{d}{dt}([\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \vec{\mathbf{e_3}}])\mathbf{p}^e$$

Substituting the fundamental relations, we obtain:

$$[\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\frac{d}{dt}\mathbf{p}^o = [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\frac{d}{dt}\mathbf{t}^o_{oe} + [\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \vec{\mathbf{e_3}}]\frac{d}{dt}\mathbf{p}^e + \overrightarrow{\omega} \times ([\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \vec{\mathbf{e_3}}])\mathbf{p}^e$$

From the coordinate basis transformation relation:

$$[\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \vec{\mathbf{e_3}}] = [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\,{}^o_eR$$

We have:

$$[\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\frac{d}{dt}\mathbf{p}^o = [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\frac{d}{dt}\mathbf{t}^o_{oe} + [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\,{}^o_eR\frac{d}{dt}\mathbf{p}^e + \overrightarrow{\omega} \times [\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}]\,{}^o_eR\mathbf{p}^e$$

Eliminating the coordinate basis, we obtain:

$$\frac{d}{dt}\mathbf{p}^o = \frac{d}{dt}\mathbf{t}^o_{oe} + {}^o_eR\frac{d}{dt}\mathbf{p}^e + \omega^o_{oe} \times {}^o_eR\mathbf{p}^e \tag{I.1.23}$$

This is the **first-order rate relation of the position vector**.

Building on this, under the condition that the relative angular velocity ω is constant, we can further derive the **second-order rate relation of the position vector**:

$$\frac{d^2}{dt^2}\mathbf{p}^o = \frac{d^2}{dt^2}\mathbf{t}^o_{oe} + {}^o_eR\frac{d^2}{dt^2}\mathbf{p}^e + 2\omega^o_{oe} \times {}^o_eR\frac{d}{dt}\mathbf{p}^e + \omega^o_{oe} \times (\omega^o_{oe} \times {}^o_eR\mathbf{p}^e) \tag{I.1.24}$$

In the second-order rate relation, the first term on the right-hand side is called the **transport acceleration**, which arises from the relative accelerated motion of the coordinate frame; the second term is called the **relative acceleration**, representing the projection of the absolute rate; and the last two terms are called the **Coriolis acceleration**, resulting from the relative rotation.

(Reference: *Modern Navigation Technology*)

2 Description of Rotation

In the previous chapter, we briefly introduced the mathematical representation of rotation and translation based on coordinate systems. For robotics, these representations are crucial because robot motion occurs in three-dimensional space. To accurately and efficiently describe robot motion, we typically attach three-dimensional coordinate systems to various moving parts of the robot. This allows us to use the mathematical formulations from the previous chapter to precisely describe the robot's motion state.

For translational motion in three-dimensional space, which has 3 degrees of freedom, we describe it using a 3-dimensional translation vector. For rotational motion in three-dimensional space, which also has 3 degrees of freedom, the rotation matrix introduced in the previous chapter uses 9 elements for description. These 9 elements must satisfy the additional constraints of an orthogonal matrix, making this method less concise in expression and less intuitive in computation. Therefore, we need to introduce other methods for representing coordinate system rotations.

In this chapter, we will introduce three more concise rotation representation methods: **angle-axis** (**rotation vector**), **Euler angles**, and **quaternions**, along with their mutual conversion relationships with rotation matrices.

2.1 Angle-Axis Representation

Any three-dimensional rotation can be expressed as a rotation around a certain axis by a specific angle. This method of describing rotation is called the **angle-axis representation** (also referred to as the **rotation vector**). In the angle-axis representation, we use a 3-dimensional **unit vector** $\mathbf{n}$ to describe the rotation axis and a scalar θ to represent the rotation angle.

Next, we will discuss the relationship between the angle-axis representation and the rotation matrix representation.

2.1.1 Angle-Axis to Rotation Matrix

The angle-axis representation can be converted to a rotation matrix. Here, we consider: under the **vector rotation description**, using the rotation matrix R to represent the angle-axis rotation $\mathbf{n}, \theta$.

Consider any non-zero three-dimensional vector a, which can always be uniquely decomposed into a component parallel to $\mathbf{n}$, $a_\parallel$, and a component perpendicular to $\mathbf{n}$, $a_\perp$.

$$a = a_\parallel + a_\perp$$

where

$$a_\parallel = \mathbf{n}\mathbf{n}^T a$$
$$a_\perp = a - \mathbf{n}\mathbf{n}^T a$$

Consider the effect of the rotation represented by $\mathbf{n}, \theta$ on these components: the rotated $a_\parallel$ remains $a_\parallel$. The rotated $a_\perp$ satisfies two properties: it is perpendicular to $a_\parallel$ and forms an angle θ with $a_\perp$ (with $\mathbf{n}$ as the positive direction of the angle). We denote this vector as $a'_\perp$. Thus,

$$Ra = a'_\perp + a_\parallel$$

How can we mathematically express the properties satisfied by $a'_\perp$? Assuming $a_\perp$ is not zero, let us establish a new coordinate system where the x, y, and z axes are aligned with the directions of $a_\perp$, $\mathbf{n}$, and $a \times \mathbf{n}$, respectively. The unit vectors for each axis are expressed as:

$$\mathbf{x} = \frac{a_\perp}{||a_\perp||}$$
$$\mathbf{y} = \mathbf{n}$$
$$\mathbf{z} = \frac{a \times \mathbf{n}}{||a \times \mathbf{n}||}$$

Clearly, $a'_\perp$ lies in the x-z plane. According to the right-hand rule, the direction of $a'_\perp$ satisfies:

$$\frac{a'_\perp}{||a'_\perp||} = \cos\theta\mathbf{x} - \sin\theta\mathbf{z}$$

Substituting the above expressions, we get:

$$a'_\perp = ||a'_\perp|| \left(\cos\theta \frac{a_\perp}{||a_\perp||} - \sin\theta \frac{a \times \mathbf{n}}{||a \times \mathbf{n}||} \right)$$

Since

$$\begin{aligned} ||a'_\perp|| &= ||a_\perp|| \\ ||a \times \mathbf{n}|| &= ||a|| \sin < a, \mathbf{n} >= ||a_\perp|| \end{aligned}$$

we have:

$$\begin{aligned} a'_\perp &= ||a'_\perp|| \left(\cos\theta \frac{a_\perp}{||a_\perp||} - \sin\theta \frac{a \times \mathbf{n}}{||a \times \mathbf{n}||} \right) \\ &= \cos\theta a_\perp - \sin\theta a \times \mathbf{n} \\ &= \cos\theta(a - \mathbf{n}\mathbf{n}^T a) - \sin\theta(a \times \mathbf{n}) \end{aligned}$$

Therefore,

$$\begin{aligned} Ra &= a'_\perp + a_\parallel \\ &= \cos\theta(a - \mathbf{n}\mathbf{n}^T a) - \sin\theta(a \times \mathbf{n}) + \mathbf{n}\mathbf{n}^T a \\ &= \cos\theta a + (1 - \cos\theta)\mathbf{n}\mathbf{n}^T a + \sin\theta(\mathbf{n} \times a) \end{aligned}$$

That is,

$$R(\mathbf{n}, \theta) = I\cos\theta + (1 - \cos\theta)\mathbf{n}\mathbf{n}^T + \mathbf{n}_\times \sin\theta \tag{I.2.1}$$

This formula is called the **Rodrigues' rotation formula**.

Under the **coordinate system rotation description**, if the rotation from the n frame to the b frame requires rotating around $\mathbf{n}$ by an angle θ, then:

$$R_b^n = I\cos\theta + (1 - \cos\theta)\mathbf{n}\mathbf{n}^T + \mathbf{n}_\times \sin\theta \tag{I.2.2}$$

2.1.2 Rotation Matrix to Angle-Axis

Note that if we take the trace of both sides of the Rodrigues' formula, we get:

$$\mathrm{tr}(R(\mathbf{n}, \theta)) = 3\cos\theta + (1 - \cos\theta)\mathbf{n}^T\mathbf{n} = 1 + 2\cos\theta$$

Thus,

$$\theta = \arccos \frac{\mathrm{tr}(R) - 1}{2} \tag{I.2.3}$$

Additionally, $\mathbf{n}$ is the eigenvector corresponding to the eigenvalue 1 of the matrix $R(\mathbf{n}, \theta)$:

$$R\mathbf{n} = \mathbf{n} \tag{I.2.4}$$

It is important to note that since the angle-axis $(\mathbf{n}m\theta + 2\pi n)$ corresponds to the same rotation matrix, the mapping from angle-axis to rotation matrix is surjective. A single rotation matrix can correspond to multiple angle-axis pairs. The above inverse transformation only provides one "principal value" that satisfies the condition.

2.2 Euler Angles

Euler angles are another method for describing three-dimensional rotations.

Consider the coordinate system rotation problem: suppose the n frame is rotated around the x-axis by an angle p to obtain the bx frame, the n frame is rotated around the y-axis by an angle r to obtain the by frame, and the n frame is rotated around the z-axis by an angle y to obtain the bz frame. Based on Equation I.2.2, we write the rotation matrices R^n_{bx}, R^n_{by}, and R^n_{bz}.

$$
\begin{aligned}
R^n_{bx} =& R_x(p) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(p) & -\sin(p) \\ 0 & \sin(p) & \cos(p) \end{bmatrix} \\
R^n_{by} =& R_y(r) = \begin{bmatrix} \cos(r) & 0 & \sin(r) \\ 0 & 1 & 0 \\ -\sin(r) & 0 & \cos(r) \end{bmatrix} \\
R^n_{bz} =& R_z(y) = \begin{bmatrix} \cos(y) & -\sin(y) & 0 \\ \sin(y) & \cos(y) & 0 \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

Here, we define: the rotation angle around the x-axis as the pitch angle, denoted by p; the rotation angle around the y-axis as the roll angle, denoted by r; and the rotation angle around the z-axis as the yaw angle, denoted by y. The positive direction for each angle is defined when the rotation vector aligns with the positive direction of the respective axis.

For almost any three-dimensional rotation, we can equivalently decompose it into three consecutive axis rotations. Specifically, we first define an axis sequence, such as "Z-X-Y". Then, almost any three-dimensional rotation from the n frame to the b frame can be uniquely equivalent to "first rotating around the z-axis by angle y, then around the x-axis by angle p, and finally around the y-axis by angle r". That is,

$$
R^n_b = R(r, p, y) = R_z(y)R_x(p)R_y(r)
$$

Here, r, p, y are referred to as the **Euler angles**.

A set of Euler angles must specify the order of rotation axes. Since matrix multiplication is not commutative, the same set of Euler angles in a different order will result in distinct rotations. **In this book, unless otherwise specified, we adopt the ZXY rotation order as the default convention for Euler angles.**

For Euler angles in the ZXY order, the relationship between the rotation matrix and the Euler angles is given by:

$$
R^n_b = R(r, p, y) = \begin{bmatrix} \cos(y)\cos(r) - \sin(y)\sin(p)\sin(r) & -\sin(y)\cos(p) & \cos(y)\sin(r) + \sin(y)\sin(p)\cos(r) \\ \sin(y)\cos(r) + \cos(y)\sin(p)\sin(r) & \cos(y)\cos(p) & \sin(y)\sin(r) - \cos(y)\sin(p)\cos(r) \\ -\cos(p)\sin(r) & \sin(p) & \cos(p)\cos(r) \end{bmatrix}
$$

In the above definition, we used the term "almost." This is because any Euler angle sequence is subject to the gimbal lock problem. Gimbal lock refers to a situation where, for certain special rotations, the Euler angle representation becomes redundant or singular. For the ZXY Euler angle sequence, gimbal lock occurs when the pitch angle is $90°$ or $-90°$.

To avoid the gimbal lock problem as much as possible, one can use quaternions, introduced below, to represent rotations.

2.3 Quaternions

A quaternion consists of a vector part and a scalar part. There are two widely used definitions: JPL quaternions (vector-first) and Hamilton quaternions (vector-last). Hamilton quaternions are more commonly used in robotics. Therefore, in this book, we exclusively adopt the Hamilton convention for quaternions.

2.3.1 Definition of Quaternions

A quaternion is a number composed of one real part and three orthogonal imaginary parts:

$$q := q_0 + iq_x + jq_y + kq_z \tag{I.2.5}$$

where i, j, k are three orthogonal imaginary units satisfying:

$$i^2 = j^2 = k^2 = ijk = -1 \tag{I.2.6}$$

Hamilton quaternions follow the right-hand rule:

$$\begin{aligned} ij &= -ji = k \\ jk &= -kj = i \\ ki &= -ik = j \end{aligned} \tag{I.2.7}$$

This definition is referred to as the **imaginary definition of quaternions**.
Alternatively, a quaternion can be expressed as:

$$q = \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \tag{I.2.8}$$

This notation is called the **vector notation of quaternions**. In this form, we can define the norm of a quaternion as:

$$||q|| = q_w^2 + q_x^2 + q_y^2 + q_z^2 \tag{I.2.9}$$

A quaternion with a norm of 1 is called a **unit quaternion**.
Furthermore, a quaternion can be expressed as a scalar part and a vector part:

$$q = q_0 + \vec{q} = \begin{bmatrix} q_0 \\ \vec{q} \end{bmatrix} \tag{I.2.10}$$

This notation is called the **scalar-vector notation of quaternions**.

2.3.2 Basic Operations of Quaternions

In scalar-vector notation, the addition of two quaternions is defined as:

$$p + q := (p_0 + q_0) + (\vec{p} + \vec{q}) \tag{I.2.11}$$

In scalar-vector notation, the multiplication of two quaternions is given by:

$$p \otimes q := (p_0 q_0 - \vec{p} \cdot \vec{q}) + (q_0 \vec{p} + p_0 \vec{q} + \vec{p} \times \vec{q}) \tag{I.2.12}$$

or equivalently:

$$p \otimes q = \begin{bmatrix} p_0 & -\vec{p}^T \\ \vec{p} & p_0 I + \vec{p}_\times \end{bmatrix} \begin{bmatrix} q_0 \\ \vec{q} \end{bmatrix} \tag{I.2.13}$$

In vector notation, quaternion multiplication is expressed as:

$$p \otimes q = \begin{bmatrix} p_0 & -p_1 & -p_2 & -p_3 \\ p_1 & p_0 & -p_3 & p_2 \\ p_2 & p_3 & p_0 & -p_1 \\ p_3 & -p_2 & p_1 & p_0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \tag{I.2.14}$$

Note that since the cross product is not commutative, **quaternion multiplication is non-commutative**. However, quaternion multiplication is associative:

$$p \otimes q \otimes r = p \otimes (q \otimes r) \tag{I.2.15}$$

2.3.3 Quaternion Representation of Rotations

The quaternion representation of rotations is closely related to the angle-axis representation. Given an angle-axis representation $\mathbf{n}, \theta$, under the Hamilton convention, it can be expressed as a (unit) quaternion in scalar-vector notation:

$$q(\theta, \mathbf{n}) = \begin{bmatrix} \cos\frac{\theta}{2} \\ \sin\frac{\theta}{2}\mathbf{n} \end{bmatrix} \tag{I.2.16}$$

Note that a unit quaternion satisfies the condition:

$$||q(\theta, \mathbf{n})|| = 1 \tag{I.2.17}$$

Consider the coordinate frame rotation description, where a rotation from frame n to frame b is a rotation of θ around $\mathbf{n}$. A vector $\mathbf{a}$ is represented as $\mathbf{a}^n$ and $\mathbf{a}^b$ in frames n and b, respectively. We then compute the following expression:

$$\begin{aligned} q(\theta, \mathbf{n}) \otimes \begin{bmatrix} 0 \\ \mathbf{a}^b \end{bmatrix} \otimes q(-\theta, \mathbf{n}) &= \begin{bmatrix} \cos\frac{\theta}{2} \\ \sin\frac{\theta}{2}\mathbf{n} \end{bmatrix} \otimes \begin{bmatrix} 0 \\ \mathbf{a}^b \end{bmatrix} \otimes \begin{bmatrix} \cos\frac{\theta}{2} \\ -\sin\frac{\theta}{2}\mathbf{n} \end{bmatrix} \\ &= \begin{bmatrix} -\sin\frac{\theta}{2}\mathbf{n}^T\mathbf{a}^b \\ \cos\frac{\theta}{2}\mathbf{a}^b + \sin\frac{\theta}{2}\mathbf{n} \times \mathbf{a}^b \end{bmatrix} \otimes \begin{bmatrix} \cos\frac{\theta}{2} \\ -\sin\frac{\theta}{2}\mathbf{n} \end{bmatrix} \end{aligned}$$

According to Equation I.2.12, we first compute the real part of the product:

$$\begin{aligned} \text{real part} &= -\sin\frac{\theta}{2}\cos\frac{\theta}{2}\mathbf{n}^T\mathbf{a}^b - (\cos\frac{\theta}{2}\mathbf{a}^b + \sin\frac{\theta}{2}\mathbf{n} \times \mathbf{a}^b)^T(-\sin\frac{\theta}{2}\mathbf{n}) \\ &= \sin^2\frac{\theta}{2}(\mathbf{n} \times \mathbf{a}^b)^T\mathbf{n} \\ &= 0 \end{aligned}$$

Next is the imaginary part.

$$\begin{aligned} \text{imaginary part} &= (-\sin\frac{\theta}{2}\mathbf{n}^T\mathbf{a}^b)(-\sin\frac{\theta}{2}\mathbf{n}) + (\cos\frac{\theta}{2})(\cos\frac{\theta}{2}\mathbf{a}^b + \sin\frac{\theta}{2}\mathbf{n} \times \mathbf{a}^b) \\ &\quad + (\cos\frac{\theta}{2}\mathbf{a}^b + \sin\frac{\theta}{2}\mathbf{n} \times \mathbf{a}^b) \times (-\sin\frac{\theta}{2}\mathbf{n}) \\ &= \sin^2\frac{\theta}{2}(\mathbf{n}^T\mathbf{a}^b)\mathbf{n} + \cos^2\frac{\theta}{2}\mathbf{a}^b + \sin\frac{\theta}{2}\cos\frac{\theta}{2}\mathbf{n} \times \mathbf{a}^b \\ &\quad - \sin\frac{\theta}{2}\cos\frac{\theta}{2}\mathbf{a}^b \times \mathbf{n} - \sin^2\frac{\theta}{2}(\mathbf{n} \times \mathbf{a}^b) \times \mathbf{n} \end{aligned}$$

Note that the vector cross product satisfies the **triple product formula**

$$a \times (b \times c) = (a^T c)b - (a^T b)c \tag{I.2.18}$$

Therefore,

$$a \times (b \times a) = (a^T a I - a a^T)b = (a \times b) \times a$$

Substituting into the above equation, we have

$$\begin{aligned} \text{imaginary part} &= \sin^2\frac{\theta}{2}\mathbf{n}(\mathbf{n}^T\mathbf{a}^b) + \cos^2\frac{\theta}{2}I\mathbf{a}^b + 2\sin\frac{\theta}{2}\cos\frac{\theta}{2}\mathbf{n}_\times\mathbf{a}^b \\ &\quad - \sin^2\frac{\theta}{2}(\mathbf{n}^T\mathbf{n}I - \mathbf{n}\mathbf{n}^T)\mathbf{a}^b \\ &= 2\sin^2\frac{\theta}{2}(\mathbf{n}\mathbf{n}^T)\mathbf{a}^b + (\cos^2\frac{\theta}{2} - \sin^2\frac{\theta}{2})I\mathbf{a}^b \\ &\quad + 2\sin\frac{\theta}{2}\cos\frac{\theta}{2}\mathbf{n}_\times\mathbf{a}^b \\ &= ((1 - \cos\theta)\mathbf{n}\mathbf{n}^T + \cos\theta I + \sin\theta\mathbf{n}_\times)\mathbf{a}^b \end{aligned}$$

According to Equation I.2.2, we have

$$\text{imaginary part} = R_b^n \mathbf{a}^b = \mathbf{a}^n$$

Thus, we obtain the coordinate rotation formula based on quaternions:

$$\begin{bmatrix} 0 \\ \mathbf{a}^n \end{bmatrix} = q(\theta, \mathbf{n}) \otimes \begin{bmatrix} 0 \\ \mathbf{a}^b \end{bmatrix} \otimes q(-\theta, \mathbf{n}) \tag{I.2.19}$$

Let

$$q_b^n = q(\theta, \mathbf{n}) \tag{I.2.20}$$

Then,

$$\begin{bmatrix} 0 \\ \mathbf{a}^n \end{bmatrix} = q_b^n \otimes \begin{bmatrix} 0 \\ \mathbf{a}^b \end{bmatrix} \otimes (q_b^n)^{-1} \tag{I.2.21}$$

This is the method for coordinate transformation using quaternions.

2.3.4 Quaternion to Rotation Matrix

Both quaternions and rotation matrices can be used for coordinate transformations involving rotations. In fact, the quaternion q_b^n can directly represent the rotation matrix R_b^n.

Specifically, from the relationship between quaternions and the angle-axis representation, we have

$$\cos\frac{\theta}{2} = q_0$$
$$\sin\frac{\theta}{2}\mathbf{n} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

Therefore,

$$\begin{aligned} I\cos\theta &= I(2\cos^2\frac{\theta}{2} - 1) \\ &= I(2q_0^2 - 1) \\ &= \begin{bmatrix} 2q_0^2 - 1 & 0 & 0 \\ 0 & 2q_0^2 - 1 & 0 \\ 0 & 0 & 2q_0^2 - 1 \end{bmatrix} \end{aligned}$$

And,

$$\begin{aligned} (1 - \cos\theta)\mathbf{n}\mathbf{n}^T &= 2\sin^2\frac{\theta}{2}\mathbf{n}\mathbf{n}^T \\ &= 2(\sin\frac{\theta}{2}\mathbf{n})(\sin\frac{\theta}{2}\mathbf{n})^T \\ &= \begin{bmatrix} 2q_1^2 & 2q_1q_2 & 2q_1q_3 \\ 2q_1q_2 & 2q_2^2 & 2q_2q_3 \\ 2q_1q_3 & 2q_2q_3 & 2q_3^2 \end{bmatrix} \end{aligned}$$

And,

$$
\begin{aligned}
\mathbf{n}_\times \sin\theta &= (\sin\theta\mathbf{n})_\times \\
&= (2\cos\frac{\theta}{2}\sin\frac{\theta}{2}\mathbf{n})_\times \\
&= (2q_0 \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix})_\times \\
&= \begin{bmatrix} 0 & -2q_0q_3 & 2q_0q_2 \\ 2q_0q_3 & 0 & -2q_0q_1 \\ -2q_0q_2 & 2q_0q_1 & 0 \end{bmatrix}
\end{aligned}
$$

According to the Rodrigues formula, R_b^n is the sum of the above three terms, hence

$$
R_b^n = \begin{bmatrix} 1-2q_2^2-2q_3^2 & 2(q_1q_2-q_0q_3) & 2(q_1q_3+q_0q_2) \\ 2(q_1q_2+q_0q_3) & 1-2q_1^2-2q_3^2 & 2(q_2q_3-q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_2q_3+q_0q_1) & 1-2q_1^2-2q_2^2 \end{bmatrix}
$$

2.3.5 Quaternion Differentiation

Rotations between two coordinate systems are generally continuous, so the quaternion representation can be treated as a function of time. Its derivative satisfies a certain relationship with angular velocity.

First, consider the approximation of quaternions for small angles. When θ is very small, $\sin\theta \approx \theta$ and $\cos\theta \approx 1$, i.e.,

$$
q(\theta, \mathbf{n}) \approx \begin{bmatrix} 1 \\ \frac{1}{2}\theta\mathbf{n} \end{bmatrix}, \theta \approx 0 \tag{I.2.22}
$$

Consider using the quaternion $q_{b'}^n$ to represent the rotation from the b' frame to the n frame. Now suppose that within an infinitesimal time δt, the b' frame rotates to the b frame, giving

$$
q_b^n = q_{b'}^n \otimes q_b^{b'}
$$

By the small-angle approximation, $q_b^{b'}$ can be expressed as

$$
q_b^{b'} \approx \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{1}{2}\omega^b\delta t \end{bmatrix}, \delta t \approx 0
$$

where $\delta\theta$ represents the **three-dimensional small rotation angles from frame b' to frame b** in the b-frame, and ω^b denotes the angular velocity in the b-frame, i.e.,

$$
q_b^n \approx q_{b'}^n \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\omega^b\delta t \end{bmatrix}, \delta t \approx 0
$$

When $\delta t \to 0$, we have

$$
\dot{q}_b^n = q_b^n \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\omega^b \end{bmatrix}
$$

That is: the angular velocity of the b-frame should be **right-multiplied** to the estimated state quaternion in the differential equation concerning q_b^n.

Reference: Paper *"Error-State Quaternion"*

2.4 Conversions Between Different Rotation Representations

When representing **rotations between coordinate frames**, the aforementioned rotation representation methods can be converted into one another.

In this section, we assume that the n-frame can be rotated to the b-frame via Euler angles y, p, r in the ZXY sequence; it can also be transformed to the b-frame by rotating angle θ around $\mathbf{n}$; the rotation matrix is $R = R_b^n$; and the quaternion is $q = q_b^n$.

Rotation Matrix and Axis-Angle

According to the Rodrigues formula, we have

$$R_b^n = I\cos\theta + (1-\cos\theta)\mathbf{n}\mathbf{n}^T + \mathbf{n}_\times \sin\theta \tag{I.2.23}$$

Based on the earlier derivation, we also have

$$\begin{aligned} \theta &= \arccos\frac{\mathrm{tr}(R_b^n)-1}{2} \\ \mathbf{n} &\leftarrow solve\{R_b^n\mathbf{n} = \mathbf{n}\} \end{aligned} \tag{I.2.24}$$

Rotation Matrix and Euler Angles

The formula for converting Euler angles to a rotation matrix is

$$R_b^n = R_z(y)R_x(p)R_y(r) \tag{I.2.25}$$

The expanded form is

$$R_b^n = \begin{bmatrix} \cos(y)\cos(r) - \sin(y)\sin(p)\sin(r) & -\sin(y)\cos(p) & \cos(y)\sin(r) + \sin(y)\sin(p)\cos(r) \\ \sin(y)\cos(r) + \cos(y)\sin(p)\sin(r) & \cos(y)\cos(p) & \sin(y)\sin(r) - \cos(y)\sin(p)\cos(r) \\ -\cos(p)\sin(r) & \sin(p) & \cos(p)\cos(r) \end{bmatrix} \tag{I.2.26}$$

To convert a rotation matrix back to Euler angles, the above formulas can be utilized. Let r_{ij} denote the element in the i-th row and j-th column of matrix R_b^n.

In the non-gimbal-lock case ($|r_{32}| < 1$), we have

$$\begin{aligned} p &= \arcsin(r_{32}) \\ r &= \arctan\left(\frac{-r_{31}}{r_{33}}\right) \\ y &= \arctan\left(\frac{-r_{12}}{r_{22}}\right) \end{aligned} \tag{I.2.27}$$

In the gimbal-lock case, we need to specify either r or y. Here, we set r to 0.

If $r_{32} = 1$, we have

$$\begin{aligned} p &= \arcsin(r_{32}) \\ r &= 0 \\ y &= \arctan\left(\frac{r_{13}}{-r_{23}}\right) \end{aligned} \tag{I.2.28}$$

If $r_{32} = -1$, we have

$$\begin{aligned} p &= \arcsin(r_{32}) \\ r &= 0 \\ y &= \arctan\left(\frac{-r_{13}}{r_{23}}\right) \end{aligned} \tag{I.2.29}$$

Quaternion and Axis-Angle

The quaternion representation of rotation originates from the axis-angle representation, so their mutual conversion is straightforward:

$$q_b^n = \begin{bmatrix} \cos\frac{\theta}{2} \\ \sin\frac{\theta}{2}\mathbf{n} \end{bmatrix} \tag{I.2.30}$$

When $\sin\theta \neq 0$, we have

$$\begin{aligned} \theta &= 2\arctan\left(\frac{||\vec{q_b^n}||}{q_0}\right) \\ \mathbf{n} &= \frac{\vec{q_b^n}}{\sqrt{1-q_0^2}} \end{aligned} \tag{I.2.31}$$

Quaternion and Rotation Matrix

The conversion from quaternion to rotation matrix has been derived earlier:

$$R_b^n = \begin{bmatrix} 1-2q_2^2-2q_3^2 & 2(q_1q_2-q_0q_3) & 2(q_1q_3+q_0q_2) \\ 2(q_1q_2+q_0q_3) & 1-2q_1^2-2q_3^2 & 2(q_2q_3-q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_2q_3+q_0q_1) & 1-2q_1^2-2q_2^2 \end{bmatrix} \tag{I.2.32}$$

To convert a rotation matrix to a quaternion, one should first convert the rotation matrix to axis-angle/Euler angles and then convert the axis-angle/Euler angles to a quaternion.

Quaternion and Euler Angles

Based on the definition of Euler angles and Equation I.2.21, we have

$$\begin{bmatrix} 0 \\ \mathbf{a}^n \end{bmatrix} = q_z(y) \otimes q_x(p) \otimes q_y(r) \otimes \begin{bmatrix} 0 \\ \mathbf{a}^b \end{bmatrix} \otimes q_y^{-1}(r) \otimes q_x^{-1}(p) \otimes q_z^{-1}(y)$$

where

$$q_x(r) = \begin{bmatrix} \cos\frac{r}{2} \\ \sin\frac{r}{2} \\ 0 \\ 0 \end{bmatrix}, q_y(p) = \begin{bmatrix} \cos\frac{p}{2} \\ 0 \\ \sin\frac{p}{2} \\ 0 \end{bmatrix}, q_z(y) = \begin{bmatrix} \cos\frac{y}{2} \\ 0 \\ 0 \\ \sin\frac{y}{2} \end{bmatrix} \tag{I.2.33}$$

Thus, we have

$$q_b^n = q_z(y) \otimes q_x(p) \otimes q_y(r) \tag{I.2.34}$$

To convert a quaternion to Euler angles, one should first convert the quaternion to a rotation matrix and then convert the rotation matrix to Euler angles.

2.5 Rotation Matrix and Differentiation

2.5.1 Rate of Change of Rotation Matrix

Angular velocity is the optimal representation for rotational changes. Earlier, we derived the relationship between quaternion differentiation and angular velocity. Below, under the coordinate frame rotation representation, we derive the relationship between the differentiation of the rotation matrix and angular velocity.

Continuing with the ballerina example, assume the n-frame is fixed to the stage, and the b-frame is fixed to the ballerina. The angular velocity of the ballerina relative to the stage, expressed in the n-frame, is denoted as ω_{nb}^n. The rotation matrix for the coordinate transformation from the n-frame to the b-frame is R_n^b.

According to Equation I.1.12, we have

$$R_n^b = (R_b^n)^T = \begin{bmatrix} \vec{\mathbf{b}}_1^n & \vec{\mathbf{b}}_2^n & \vec{\mathbf{b}}_3^n \end{bmatrix}^T \tag{I.2.35}$$

Therefore, to find $\dot{R}_n^b$, we only need to compute the derivatives of $\vec{\mathbf{b}}_i^n$. More generally, consider any vector $\mathbf{a}$ fixed to the b-frame and compute the derivative of its coordinates in the n-frame, $\dot{\mathbf{a}}^n$. This transforms the differentiation of the rotation matrix under the coordinate frame rotation representation into the differentiation of the rotation matrix under the vector rotation representation.

Specifically, decompose $\mathbf{a}$ into a component parallel to ω_{nb}^n, $a_\parallel$, and a component perpendicular to ω_{nb}^n, $a_\perp$:

$$\mathbf{a} = \mathbf{a}_\parallel + \mathbf{a}_\perp$$

The linear velocity generated by the rotation of ω_{nb} is only related to $a_\perp$. Its direction is along $\omega_{nb} \times \mathbf{a}_\perp$, and its magnitude is

$$\mathbf{v}_a = ||\omega_{nb}||||\mathbf{a}_\perp|| = ||\omega_{nb} \times \mathbf{a}_\perp||$$

Thus, we have

$$\mathbf{v}_a = \omega_{nb} \times \mathbf{a}_\perp$$

Since

$$\omega_{nb} \times \mathbf{a}_\parallel = 0$$

we have

$$\mathbf{v}_a = \omega_{nb} \times \mathbf{a}_\perp + \omega_{nb} \times \mathbf{a}_\parallel = \omega_{nb} \times \mathbf{a}$$

That is: The rate of change of any vector fixed to frame b is the cross product of the angular velocity with that vector.

$$\dot{\mathbf{a}}^n = \omega_{nb}^n \times \mathbf{a}^n \tag{I.2.36}$$

Therefore, we have

$$\begin{aligned}
\dot{R}_n^b &= \begin{bmatrix} \omega_{nb}^n \times \vec{\mathbf{b}}_1^n & \omega_{nb}^n \times \vec{\mathbf{b}}_2^n & \omega_{nb}^n \times \vec{\mathbf{b}}_3^n \end{bmatrix}^T \\
&= ((\omega_{nb}^n)_\times \begin{bmatrix} \vec{\mathbf{b}}_1^n & \vec{\mathbf{b}}_2^n & \vec{\mathbf{b}}_3^n \end{bmatrix})^T \\
&= \begin{bmatrix} \vec{\mathbf{b}}_1^n & \vec{\mathbf{b}}_2^n & \vec{\mathbf{b}}_3^n \end{bmatrix}^T (\omega_{nb}^n)_\times^T \\
&= -R_n^b (\omega_{nb}^n)_\times
\end{aligned}$$

That is,

$$\begin{aligned}
\dot{R}_n^b &= -R_n^b (\omega_{nb}^n)_\times \\
\dot{R}_b^n &= (\omega_{nb}^n)_\times R_b^n
\end{aligned}$$

2.5.2 Introduction to Lie Groups and Lie Algebras

In robotics, we often encounter optimization problems where (relative) positions and orientations are treated as optimization variables (for details on optimization problem solving, refer to Chapter 3). In this process, we need to compute the derivative of the optimization objective with respect to the orientation.

If we represent rotations using rotation matrices, the above problem essentially reduces to computing the following derivative:

$$\frac{\partial Rp}{\partial R}$$

A rotation matrix is a 3×3 matrix. However, we cannot directly apply standard matrix differentiation rules to it as if it were an ordinary matrix. This is because 3D rotations have only 3 degrees of freedom, and to ensure this, the 9 elements of R must satisfy two normalization constraints: $R^T R = I$ and $det(R) = 1$. Standard matrix differentiation rules do not account for these constraints, leading to incorrect derivatives.

In fact, the rotation matrix R does not reside in a 9-dimensional Euclidean space but rather on a **manifold** within this space. A manifold can be thought of as a low-dimensional hypersurface embedded in a higher-dimensional space, with the surface typically exhibiting some degree of smoothness[1]. The constraint equations on the variables define the analytical equations of this surface. Typical examples of manifolds include a circle in 2D space (a 1D manifold) and a sphere in 3D space (a 2D manifold). Generally, manifolds do not have a one-to-one correspondence with Euclidean spaces of the same dimension.

So, how should we compute derivatives for variables on a manifold? The essence of differentiation is to construct a perturbation near the independent variable, which, through the functional mapping, induces a perturbation in the value space. By ignoring higher-order terms, the perturbation in the function value and the perturbation in the independent variable form an approximately linear relationship, representing the local linearization of the function. We do not want the perturbation to occur on the hypersurface itself, but if we can construct a local linearization of the manifold near a point—i.e., the **tangent plane** of the high-dimensional surface—we can **use the projection of the perturbation onto the tangent plane to approximate the perturbation on the surface**.

In fact, rotation matrices not only form a manifold but also constitute a **group** due to their inherent symmetry. The group formed by rotation matrices has a specific name: the **Special Orthogonal Group**, denoted as $SO(3)$. $SO(3)$ is both a manifold and a group, making it a **Lie group**. As a manifold, the Lie group's "tangent plane" forms a structure similar to a linear space, which we call the **Lie algebra**[2]. Every Lie group has a corresponding Lie algebra, and vice versa.

For rigorous definitions of manifolds, groups, Lie groups, and Lie algebras, readers may refer to other algebra textbooks, which will not be elaborated here.

2.5.3 Exponential and Logarithmic Mappings

Every Lie group has a corresponding Lie algebra. For the Lie group $SO(3)$ composed of rotation matrices, its Lie algebra is:

$$so(3) = \{\phi \in \mathbb{R}^3 | \phi = \mathbf{n}\theta, ||\mathbf{n}|| = 1\} \tag{I.2.37}$$

In fact, this is precisely the product of the angle and axis in the angle-axis representation introduced earlier. We refer to this as the **rotation vector**.

The rotation vector is the integral of angular velocity over time. Assume frame b rotates relative to frame n at a constant angular velocity. Let $\omega = \omega_{nb}^b$ denote the angular velocity of b relative to n expressed in n. Let the rotation matrix $R(t) = R_b^n$, and the corresponding rotation vector $\phi = \phi_{nb}^b$. Then:

$$\phi = \omega t \tag{I.2.38}$$

Recalling the differential equation for R derived in the previous section:

$$\dot{R}(t) = (\omega)_\times R(t)$$

Assuming the initial condition $R(0) = I$, meaning the initial orientation of n and b coincide, the solution to the matrix differential equation is:

$$R(t) = \exp((\omega)_\times t)$$

That is:

$$R(t) = \exp((\omega t)_\times) = \exp((\phi)_\times)$$

Recalling the **Rodrigues formula** (Equation I.2.1) derived in Section 2.1, we have:

$$R = \exp((\phi)_\times) = I\cos\theta + (1-\cos\theta)\mathbf{n}\mathbf{n}^T + \mathbf{n}_\times \sin\theta, \quad \phi = \mathbf{n}\theta, ||\mathbf{n}|| = 1 \tag{I.2.39}$$

[1] Non-rigorous definition
[2] Non-rigorous definition

This formula reveals the **exponential relationship** between elements of $SO(3)$ and $so(3)$, specifically providing a method to compute the Lie group element R given a Lie algebra element ϕ[3]. In fact, **for every element in the Lie algebra, there exists a unique corresponding element in the Lie group**. Due to its form as a matrix exponential, this correspondence is also called the **exponential mapping**. Equation I.2.39 represents the exponential mapping from $so(3)$ to $SO(3)$.

Conversely, the **logarithmic mapping** refers to finding the corresponding Lie algebra element given a Lie group element. It is not difficult to see that for $\phi = \mathbf{n}(\theta_0 + k\theta), k \in \mathbb{N}$, the corresponding rotation matrix is the same. In other words, the logarithmic mapping has multiple solutions. Similar to the treatment in complex analysis, we select one value from all ϕ satisfying $R = \exp((\phi)_\times)$ as the **principal value** of the logarithmic mapping. The specific method for computing the logarithmic mapping (principal value) from $SO(3)$ to $so(3)$ is also explained in Section 2.1 (i.e., extracting the angle-axis from the rotation matrix) and will not be repeated here.

2.5.4 Perturbations and Differentiation

Above, we introduced how for the rotation matrix Lie group $SO(3)$, which forms a high-dimensional manifold, we can establish a correspondence between the Lie group element $R \in SO(3)$ and the tangent space Lie algebra element $\phi \in so(3)$. This relationship is based on the assumption of small perturbations. So, how can we use this relationship to address the challenge of "differentiating with respect to rotation matrices" from the perspective of perturbations?

Here, we first need to distinguish between **left perturbations** and **right perturbations**. We know that rotation matrix multiplication is not commutative, meaning in general:

$$R_1 R_2 \neq R_2 R_1$$

Therefore, for the rotation matrix $R \in SO(3)$ with respect to which we wish to differentiate and the perturbation $\Delta R \in SO(3)$ we wish to apply, we also have:

$$R(\Delta R) \neq (\Delta R)R$$

We refer to perturbations of the form $(\Delta R)R$ as **left perturbations**, and those of the form $R(\Delta R)$ as **right perturbations**. Considering their physical meaning, it is not difficult to see that if $R = R_b^n$ represents the rotation matrix from frame b to frame n, then a left perturbation $(\Delta R)R$ applies the perturbation in frame n, while a right perturbation $R(\Delta R)$ applies the perturbation in frame b.

Below, we first take left perturbations as an example to consider **transforming the variation of the rotation matrix (Lie group) into the variation of the rotation vector (Lie algebra)**. Suppose for the rotation matrix R, the corresponding rotation vector is ϕ, such that:

$$R = \exp((\phi)_\times)$$

Now, apply a left perturbation ΔR to the rotation matrix R. Assume the rotation vector corresponding to ΔR is $\Delta\phi$, i.e.:

$$\Delta R = \exp((\Delta\phi)_\times)$$

Then, the total rotation is:

$$(\Delta R)R = \exp((\Delta\phi)_\times)\exp((\phi)_\times)$$

In this way, we can replace differentiation with respect to the Lie group element R with differentiation with respect to the Lie algebra element ϕ. According to the perturbation model, we have

[3] In fact, this formula can also be derived from the Taylor series expansion of the matrix exponential, offering a second derivation of the Rodrigues formula. The derivation is provided in "Fourteen Lectures on Visual SLAM" and will not be repeated here.

$$\begin{aligned}\frac{\partial Rp}{\partial \phi} &= \frac{(\Delta R)Rp - Rp}{\Delta\phi} \\ &= \frac{\exp((\Delta\phi)_\times)\exp((\phi)_\times)p - \exp((\phi)_\times)p}{\Delta\phi} \\ &= \frac{(\exp((\Delta\phi)_\times) - I)\exp((\phi)_\times)p}{\Delta\phi}\end{aligned}$$

For the term $\exp((\Delta\phi)_\times) - I$, since we have assumed $\Delta\phi$ to be a small quantity, we can approximate it using the first-order Taylor expansion of the matrix exponential, i.e.,

$$\exp((\Delta\phi)_\times) \approx I + (\Delta\phi)_\times \tag{I.2.40}$$

Therefore, we have

$$\begin{aligned}\frac{\partial Rp}{\partial \phi} &= \frac{(\exp((\Delta\phi)_\times) - I)\exp((\phi)_\times)p}{\Delta\phi} \\ &\approx \frac{(\Delta\phi)_\times \exp((\phi)_\times)p}{\Delta\phi} \\ &= \frac{(\Delta\phi) \times (Rp)}{\Delta\phi} \\ &= -\frac{(Rp) \times (\Delta\phi)}{\Delta\phi} \\ &= -(Rp)_\times\end{aligned}$$

The above is the case for left perturbation. For right perturbation, we have

$$\begin{aligned}\frac{\partial Rp}{\partial \phi} &= \frac{\exp((\phi)_\times)(\exp((\Delta\phi)_\times) - I)p}{\Delta\phi} \\ &\approx \frac{\exp((\phi)_\times)(\Delta\phi)_\times p}{\Delta\phi} \\ &= \frac{R((\Delta\phi) \times p)}{\Delta\phi} \\ &= -\frac{R(p \times (\Delta\phi))}{\Delta\phi} \\ &= -Rp_\times\end{aligned}$$

That is,

$$\begin{aligned}\frac{\partial Rp}{\partial \phi} &= -(Rp)_\times \quad \text{(left perturbation)} \\ \frac{\partial Rp}{\partial \phi} &= -Rp_\times \quad \text{(right perturbation)}\end{aligned} \tag{I.2.41}$$

For the transpose case, we have

$$\begin{aligned}((\Delta R)R)^T &= \exp((\phi)_\times)^T \exp((\Delta\phi)_\times)^T \\ &= \exp((\phi)_\times)^T \exp((-\Delta\phi)_\times) \\ &= \exp((\phi)_\times)^T \exp(-(\Delta\phi)_\times) \\ &= R^T(-\Delta R)\end{aligned} \tag{I.2.42}$$

That is: the transpose of a left perturbation is equivalent to the negation of a right perturbation. The same applies to right perturbation. Therefore, we also have the following conclusions:

$$\begin{aligned}\frac{\partial R^T p}{\partial \phi} &= R^T p_\times \quad \text{(left perturbation)} \\ \frac{\partial R^T p}{\partial \phi} &= (R^T p)_\times \quad \text{(right perturbation)}\end{aligned} \tag{I.2.43}$$

It is worth noting that the Lie algebra perturbation $\Delta\phi$ here can essentially be understood as the three-dimensional small angle $\delta\theta$ discussed in Section 2.3.5.

2.5.5 Perturbation Jacobian

One point to note in the above perturbation model is that the exponential function here is a matrix exponential, and for matrix exponentials, $\exp(A)\exp(B) \neq \exp(A+B)$. Therefore, the rotation vector of $(\Delta R)R$ cannot be written as $\exp((\Delta\phi+\phi)_\times)$, but requires a transformation:

$$\exp((\Delta\phi)_\times)\exp((\phi)_\times) = \exp((J_l^{-1}(\phi)\Delta\phi + \phi)_\times) \tag{I.2.44}$$

That is,

$$\exp((\Delta\phi' + \phi)_\times) = \exp((J_l(\phi)\Delta\phi')_\times)\exp((\phi)_\times) \tag{I.2.45}$$

The matrix $J_l(\phi)$ is called the **perturbation Jacobian** for left multiplication. It is a 3×3 matrix that depends on the original rotation vector ϕ. Its specific calculation is given by:

$$J_l(\phi) = \frac{\sin\theta}{\theta} I + \left(1 - \frac{\sin\theta}{\theta}\right)\mathbf{n}\mathbf{n}^T + \frac{1-\cos\theta}{\theta}\mathbf{n}_\times \tag{I.2.46}$$

Similarly, for right multiplication, we have:

$$\exp((\Delta\phi' + \phi)_\times) = \exp((\phi)_\times)\exp((J_r(\phi)\Delta\phi')_\times) \tag{I.2.47}$$

and

$$J_r(\phi) = J_l(\phi)^T \tag{I.2.48}$$

The proofs of these formulas are omitted here, and readers may refer to other textbooks.
(Reference: "Fourteen Lectures on Visual SLAM")

2.6 Conversions Between Various Rotation Differentials

So far, we have introduced four methods for representing rotations: rotation matrices, quaternions, Euler angles, and angle-axis representations. All four representations can be used as time-varying variables, thus having their differentials. However, more generally, we use **angular velocity** to represent changes in rotation. There are conversion relationships between the differentials of these rotation representations and angular velocity, which we summarize here.

For the representation of angular velocity, there are four possible ways: the rotation of the n-frame relative to the b-frame and the b-frame relative to the n-frame, each expressed in either the n-frame or the b-frame. The two commonly used ones are ω_{nb}^n and $\omega^b = \omega_{nb}^b$. Here, ω^b is the direct measurement from a gyroscope. Their conversion relationship is as follows:

$$\omega^b = \omega_{nb}^b = R_n^b \omega_{nb}^n \tag{I.2.49}$$

In this section, we still assume that the n-frame can be rotated to the b-frame via Euler angles y, p, r in the ZXY order. The differentials of the Euler angles are denoted as $\dot{y}, \dot{p}, \dot{r}$. The n-frame is rotated to the b-frame by an angle θ around the axis $\mathbf{n}$, with the rotation vector $\phi = \theta\mathbf{n}$, satisfying $R_b^n = \exp((\phi)_\times)$; its differential is $\dot{\phi}$. The rotation matrix is R_b^n, and its differential is $\dot{R}_b^n$. The quaternion is q_b^n, and its differential is $\dot{q}$.

Rotation Matrix Differential
In Section 2.5.1, we derived the following conclusion:

$$\dot{R}_b^n = (\omega_{nb}^n)_\times R_b^n = R_b^n(\omega^b)_\times \tag{I.2.50}$$

Quaternion Differential

In Section 2.3.5, we derived the following conclusion:

$$\dot{q}_b^n = q_b^n \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\omega^b \end{bmatrix} = q_b^n \otimes \begin{bmatrix} 1 \\ \frac{1}{2}R_n^b\omega_{nb}^n \end{bmatrix} \tag{I.2.51}$$

Euler Angle Differential

The relationship between Euler angle differentials and angular velocity can be derived using the concept of angular velocity addition. That is: separately consider the angular velocities $\omega_z, \omega_x, \omega_y$ caused by changes in each Euler angle, and then sum them up:

$$\omega_{nb}^n = \omega_z^n + \omega_x^n + \omega_y^n$$

Here, the angular velocity caused by each Euler angle is the product of the corresponding coordinate axis vector and the Euler angle differential.

$$\begin{aligned} \omega_z^n &= \mathbf{n}_z^n \dot{y} \\ \omega_x^n &= \mathbf{n}_x^n \dot{p} \\ \omega_y^n &= \mathbf{n}_y^n \dot{r} \end{aligned}$$

Under the ZXY order, we have the following relationship between Euler angles and the rotation matrix:

$$R_b^n = R_z(y)R_x(p)R_y(r)$$

Therefore, the coordinate axis vectors are respectively:

$$\mathbf{n}_z^n = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\mathbf{n}_x^n = R_z(y) \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{n}_y^n = R_z(y)R_x(p) \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Substituting the above expressions, we obtain the angular velocity expression:

$$\omega_{nb}^n = T_{ZXY}(y,p,r) \begin{bmatrix} \dot{p} \\ \dot{r} \\ \dot{y} \end{bmatrix}$$

where

$$\begin{aligned} T_{ZXY}(y,p,r) &= R_z(y) \begin{bmatrix} 1 & & \\ & 0 & \\ & & 0 \end{bmatrix} + R_z(y)R_x(p) \begin{bmatrix} 0 & & \\ & 1 & \\ & & 0 \end{bmatrix} + \begin{bmatrix} 0 & & \\ & 0 & \\ & & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos(y) & -\cos(p)\sin(y) & 0 \\ \sin(y) & \cos(p)\cos(y) & 0 \\ 0 & \sin(p) & 1 \end{bmatrix} \end{aligned}$$

That is,

$$\omega_{nb}^n = \begin{bmatrix} \cos(y) & -\cos(p)\sin(y) & 0 \\ \sin(y) & \cos(p)\cos(y) & 0 \\ 0 & \sin(p) & 1 \end{bmatrix} \begin{bmatrix} \dot{p} \\ \dot{r} \\ \dot{y} \end{bmatrix} \tag{I.2.52}$$

Rotation Vector Differential

The rotation vector ϕ is the Lie algebra corresponding to the Lie group formed by rotation matrices. The change in the rotation matrix due to an infinitesimal rotation can be approximately linearly represented as a change in the rotation vector. In Section 2.5.5, for the right perturbation, we have the following conclusion:

$$\exp((\Delta\phi' + \phi)_\times) = \exp((\phi)_\times)\exp((J_r(\phi)\Delta\phi')_\times)$$

where

$$R_{b'}^n = \exp((\Delta\phi' + \phi)_\times)$$
$$R_b^n = \exp((\phi)_\times)$$

Performing a first-order approximation, we have:

$$\exp((\Delta\phi' + \phi)_\times) \approx \exp((\phi)_\times)(I + (J_r(\phi)\Delta\phi')_\times)$$

Assuming the time interval is extremely short, the change in the rotation vector can be expressed as:

$$\Delta\phi' = \dot{\phi}\delta t$$

That is,

$$\exp((\dot{\phi}\delta t + \phi)_\times) \approx \exp((\phi)_\times)(I + (J_r(\phi)\dot{\phi}\delta t)_\times)$$

Therefore, when $\lim_{\delta t \to 0}$, we have:

$$\begin{aligned}
\dot{R}_b^n &= \frac{R_{b'}^n - R_b^n}{\delta t} \\
&= \frac{\exp((\dot{\phi}\delta t + \phi)_\times) - \exp((\phi)_\times)}{\delta t} \\
&= \frac{\exp((\phi)_\times)(J_r(\phi)\dot{\phi}\delta t)_\times}{\delta t} \\
&= \exp((\phi)_\times)(J_r(\phi)\dot{\phi})_\times \\
&= R_b^n (J_r(\phi)\dot{\phi})_\times
\end{aligned}$$

Moreover, since

$$\dot{R}_b^n = R_b^n (\omega^b)_\times$$

it follows that

$$(J_r(\phi)\dot{\phi})_\times = (\omega^b)_\times$$

That is,

$$\omega^b = J_r(\phi)\dot{\phi} \tag{I.2.53}$$

Similarly, for the left perturbation, we have:

$$\omega^n = J_l(\phi)\dot{\phi} \tag{I.2.54}$$

3 Optimization Methods

In the field of robotics, many problems can be transformed into optimization problems for solving, including robot inverse kinematics (see Chapter 7), MPC (see Chapter 13), and vSLAM (see Chapter 26), among others. It is a common problem-solving approach and a way to transform problems.

Compared to specialized optimization textbooks, we will not cover foundational topics such as convex optimization, convexity, or linear programming, but instead focus on basic applications in the field of robotics.

3.1 Definition of Optimization Problems

An **optimization (programming) problem** refers to the task of finding the extremum (maximum or minimum) of a certain function with respect to some variable under given constraints. This variable is called the **optimization variable**, which can be a vector, denoted as $\mathbf{x}$. The function of interest is called the **objective function**, generally denoted as $f(\mathbf{x})$, and is a scalar-valued function. We typically require the objective function to be continuously differentiable with respect to the optimization variable $\mathbf{x}$.

The constraints of an optimization problem can be either equality or inequality conditions. They are generally written in the form $\mathbf{g}(\mathbf{x}) = 0, h(\mathbf{x}) \leq 0$. $\mathbf{g}(\mathbf{x})$ and $\mathbf{h}(\mathbf{x})$ are usually vector-valued functions, meaning they contain multiple constraints.

The general nonlinear optimization problem is defined as follows:

Problem	**General Nonlinear Optimization**
Brief Description	Find the optimization variable that minimizes the objective function under equality and inequality constraints
Given	Objective function $f(\mathbf{x})$ Equality constraint function $\mathbf{g}(\mathbf{x})$ Inequality constraint function $\mathbf{h}(\mathbf{x})$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$ $s.t. \quad \mathbf{g}(\mathbf{x}) = 0, \mathbf{h}(\mathbf{x}) \leq 0$

For the above problem, if the inequality constraints are removed, it is called a **nonlinear optimization problem with equality constraints**:

Problem	**Nonlinear Optimization with Equality Constraints**
Brief Description	Find the optimization variable that minimizes the objective function under equality constraints
Given	Objective function $f(\mathbf{x})$ Equality constraint function $\mathbf{g}(\mathbf{x})$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$ $s.t. \quad \mathbf{g}(\mathbf{x}) = 0$

Similarly, if the equality constraints are removed, it is called a **nonlinear optimization problem with inequality constraints**:

Problem	**Nonlinear Optimization with Inequality Constraints**
Brief Description	Find the optimization variable that minimizes the objective function under inequality constraints
Given	Objective function $f(\mathbf{x})$ Inequality constraint function $\mathbf{h}(\mathbf{x})$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$ $s.t. \quad \mathbf{h}(\mathbf{x}) \leq 0$

If all constraints are removed, we obtain an **unconstrained nonlinear optimization problem**:

Problem	**Unconstrained Nonlinear Optimization**
Brief Description	Find the optimization variable that minimizes the objective function
Given	Objective function $f(\mathbf{x})$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$

Among nonlinear optimization problems, there is a class of problems called least-squares problems. Their objective function is a quadratic form of a (differentiable) vector-valued function, $f(\mathbf{x}) = \mathbf{e}^T(\mathbf{x})H\mathbf{e}(\mathbf{x})$. For example, we can define an unconstrained least-squares problem:

Problem	**Unconstrained Least Squares**
Brief Description	Find the optimization variable that minimizes the nonlinear quadratic objective function
Given	Objective function $\mathbf{e}^T(\mathbf{x})H\mathbf{e}(\mathbf{x})$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} \mathbf{e}^T(\mathbf{x})H\mathbf{e}(\mathbf{x})$

For the above classes of nonlinear optimization problems, if the function $f(\mathbf{x})$ satisfies convexity, the problem is called a **convex optimization problem**. Convex optimization problems have many favorable properties, such as the ability to reach the global minimum from any starting point through iterative optimization.

The simplest convex optimization problem is the **quadratic programming (QP) problem**. A QP problem refers to an optimization problem where the objective function is a quadratic form and linear function of the optimization variables, and the constraints are all linear functions of the optimization variables. Depending on the presence of constraints, QP problems can be classified into unconstrained QP problems, equality-constrained QP problems, and inequality-constrained QP problems:

Problem	**Unconstrained QP Problem**
Brief Description	Find the optimization variable that minimizes the quadratic objective function
Given	Positive semidefinite matrix H, vector $\mathbf{g}$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x}$

Problem	**Equality-constrained QP Problem**
Description	Find the optimization variables that minimize the quadratic objective function under equality constraints
Given	Positive semidefinite matrix H, vector $\mathbf{g}$ Matrix M_{eq}, vector b_{eq}
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x}$ $s.t. \quad M_{eq}\mathbf{x} = b_{eq}$

Problem	**General QP Problem**
Description	Find the optimization variables that minimize the quadratic objective function under equality and inequality constraints
Given	Positive semidefinite matrix H, vector $\mathbf{g}$ Matrix M_{eq}, vector b_{eq} Matrix M, vector b
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x}$ $s.t. \quad M_{eq}\mathbf{x} = b_{eq}, M\mathbf{x} \leq b$

3.2 Unconstrained Optimization

3.2.1 Gradient Descent Method

First, we consider unconstrained optimization problems. We assume the objective functions are all differentiable. For unconstrained optimization problems, the most straightforward iterative solution method is to solve iteratively along the opposite direction of the gradient, known as the **Gradient Descent Method**. The algorithm is summarized as follows:

Algorithm	**Gradient Descent Method**
Problem Type	Unconstrained nonlinear optimization
Given	Differentiable objective function $f(\mathbf{x})$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
Algorithm Property	Iterative solution

Algorithm 1: Gradient Descent Method

Input: Differentiable objective function $f(\mathbf{x})$, initial value $\mathbf{x}_0$
Parameter: Threshold ϵ, step size α
Output: Optimal solution $\mathbf{x}^*$
$\mathbf{x}_k \leftarrow \mathbf{x}_0$
while $||\nabla_{\mathbf{x}} f(\mathbf{x}_k)|| > \epsilon$ **do**
  $\mathbf{x}_k \leftarrow \mathbf{x}_k - \alpha \nabla_{\mathbf{x}} f(\mathbf{x}_k)$
$\mathbf{x}^* \leftarrow \mathbf{x}_k$

The corresponding Python code is shown below:

```python
def opt_nc_gd(df_func, x_0, epsilon=1e-4, alpha=1e-2):
    x_k = x_0
    while True:
        grad = df_func(x_k) # grad = (df/dx)^T
        if np.linalg.norm(grad) < epsilon:
            break
        x_k = x_k - alpha * grad
    return x_k
```

3.2.2 Newton's Method

The gradient descent method requires two parameters to be specified in advance: the threshold ϵ and the step size α. The step size is sometimes referred to as the learning rate in the deep learning community. A step size that is too large or too small may lead to slow convergence. On the other hand, for convex optimization problems, the gradient descent method may converge slowly due to oscillatory convergence.

To address this issue, we can use the second-order **Newton's Method**. Specifically, we consider performing a second-order Taylor expansion of the function $f(\mathbf{x})$ around $\mathbf{x}_k$:

$$f(\mathbf{x}) = f(\mathbf{x}_k) + \nabla_{\mathbf{x}} f(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_k)^T H_f(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) + o(||\delta||^2) \tag{I.3.1}$$

Here, $H_f(\mathbf{x})$ is the second derivative of the function f with respect to x. When x is a vector and f is a scalar value, it is a square matrix called the **Hessian matrix**.

$$H_f(\mathbf{x}_k) := \frac{d^2 f}{d\mathbf{x}^2}(\mathbf{x}_k) \in \mathbb{R}^{n \times n} \tag{I.3.2}$$

We ignore the second-order small terms and use the remaining quadratic form as an approximation of the original objective function $f(\mathbf{x})$:

$$\hat{f}(x; \mathbf{x}_k) = f(\mathbf{x}_k) + \nabla_{\mathbf{x}} f(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_k)^T H_f(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) \tag{I.3.3}$$

In each optimization step, we directly take the minimum of the approximate objective function $\hat{f}(x)$. Since $\hat{f}(x)$ is a quadratic form, it must be convex. Under the condition that H_f is positive definite, assume:

$$\mathbf{x}_{k+1} = \arg\min_{\mathbf{x}} \hat{f}(\mathbf{x}; \mathbf{x}_k)$$

We have:

$$\frac{\partial \hat{f}}{\partial \mathbf{x}}(\mathbf{x}_{k+1}) = 0$$

That is:

$$\nabla_{\mathbf{x}} f(\mathbf{x}_k) + H_f(\mathbf{x}_k)(\mathbf{x}_{k+1} - \mathbf{x}_k) = 0 \tag{I.3.4}$$

Assuming $H_f(\mathbf{x}_k)$ is invertible, we have:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H_f^{-1}(\mathbf{x}_k)\nabla_{\mathbf{x}} f(\mathbf{x}_k) \tag{I.3.5}$$

Therefore, we can summarize **Newton's Method** as follows:

Algorithm	**Newton's Method**
Problem Type	Unconstrained nonlinear optimization
Given	Differentiable objective function $f(\mathbf{x})$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
Algorithm Property	Iterative solution

Algorithm 2: Newton's Method

Input: Twice-differentiable objective function $f(\mathbf{x})$, initial value $\mathbf{x}_0$
Parameter: Threshold ϵ
Output: Optimal solution $\mathbf{x}^*$
$\mathbf{x}_k \leftarrow \mathbf{x}_0$
while $||\nabla_{\mathbf{x}} f(\mathbf{x}_k)|| > \epsilon$ **do**
  $\mathbf{x}_k \leftarrow \mathbf{x}_k - H_f^{-1}(\mathbf{x}_k)\nabla_{\mathbf{x}} f(\mathbf{x}_k)$
$\mathbf{x}^* \leftarrow \mathbf{x}_k$

The corresponding Python code is shown below:

```python
def opt_nc_newton(df_func, ddf_func, x_0, epsilon=1e-4):
    x_k = x_0
    while True:
        grad = df_func(x_k)
        if np.linalg.norm(grad) < epsilon:
            break
        H_inv = np.linalg.inv(ddf_func(x_k))
        x_k = x_k - H_inv @ grad
    return x_k
```

Compared to gradient descent, Newton's method has faster convergence speed but also more stringent requirements: the objective function must be twice differentiable, the Hessian matrix must be invertible, and matrix inversion must be performed in each iteration. So, is it possible to achieve faster convergence without computing second-order derivatives?

3.2.3 Line Search Methods

One approach is to divide the iteration step into two problems: first find the best descent direction, then find the optimal step size along that direction. Such optimization methods are called **line search-based optimization methods**.

Line search refers to the second step, which involves finding the optimal function value along a straight line in multidimensional space. Let us briefly define the line search problem:

Problem	**Line Search Problem**
Problem Description	Given an objective function, starting point, and update direction, find the step size that minimizes the objective function
Given	Objective function $f(\mathbf{x})$ Starting point $\mathbf{x}_0 \in \mathbb{R}^n$ Search direction $\mathbf{t} \in \mathbb{R}^n$
Find	Optimal step size $h^* = \arg\min_h f(\mathbf{x}_0 + h\mathbf{t})$

The most commonly used method for line search is the golden-section method, which reduces the search interval to $\phi = 0.5(\sqrt{5} - 1)$ times the previous interval in each iteration. The specific implementation of the golden-section algorithm is shown below:

Algorithm	**Golden-Section Method**
Problem Type	Line search problem
Given	Objective function $f(\mathbf{x})$ Starting point $\mathbf{x}_0 \in \mathbb{R}^n$ Search direction $\mathbf{t} \in \mathbb{R}^n$
Find	Optimal step size $h^* = \arg\min_h f(\mathbf{x}_0 + h\mathbf{t})$
Algorithm Property	Iterative solution

Algorithm 3: Golden-Section Method (GR_line_search)

Input: Objective function $f(\mathbf{x})$
Input: Starting point $\mathbf{x}_0$, search direction t
Parameter: Initial step size h_0, threshold r
Output: Optimal step size h^*

$a, b \leftarrow 0, h_0$
$h \leftarrow h_0$
while $|b - a| > r$ **do**
    $p, q \leftarrow a + (1 - \phi)h, a + \phi h$
    $f_p, f_q \leftarrow f(\mathbf{x}_0 + p\mathbf{t}), f(\mathbf{x}_0 + q\mathbf{t})$
    if $f_p < f_q$ **then**
        $a, b \leftarrow a, q$
    else
        $a, b \leftarrow p, b$
$h^* \leftarrow (a + b)/2$

The corresponding Python code is shown below:

```python
def GR_line_search(f_func, x_0, t, h_0=1e-2, r=1e-5):
    a, b, h, phi = 0, h_0, h_0, 0.618
    while np.abs(h) > r:
        h = b - a
        p, q = a + (1-phi) * h, a + phi * h
        f_p, f_q = f_func(x_0 + p*t), f_func(x_0 + q*t)
        if f_p < f_q:
            a, b = a, q
        else:
            a, b = p, b
    return (a + b) / 2
```

Based on the above line search method, we can improve gradient descent. The improved method is called Line Search Gradient Descent (LSGD). Readers should note that the search direction must be the negative gradient direction.

Algorithm	**Line Search Gradient Descent**
Problem Type	Unconstrained nonlinear optimization
Given	Differentiable objective function $f(\mathbf{x})$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
Algorithm Property	Iterative solution

Algorithm 4: Line Search Gradient Descent (opt_nc_LSGD)

Input: Differentiable objective function $f(\mathbf{x})$
Parameter: Threshold ϵ, initial value $\mathbf{x}_0$
Output: Optimal solution $\mathbf{x}^*$
$\mathbf{x}_k \leftarrow \mathbf{x}_0$
while $||\nabla_{\mathbf{x}} f(\mathbf{x}_k)|| > \epsilon$ **do**
  $\mathbf{t} \leftarrow -\nabla_{\mathbf{x}} f(\mathbf{x}_k)$
  $h \leftarrow \text{GR\_line\_search}(f, \mathbf{x}_k, \mathbf{t})$
  $\mathbf{x}_k \leftarrow \mathbf{x}_k + h\mathbf{t}$
$\mathbf{x}^* \leftarrow \mathbf{x}_k$

The corresponding Python code is shown below:

```python
def opt_nc_LSGD(f_func, df_func, x_0, epsilon=1e-4):
    x_k = x_0
    while True:
        grad = df_func(x_k)
        if np.linalg.norm(grad) < epsilon:
            break
        t = - grad
        h = GR_line_search(f_func, x_k, t)
        x_k = x_k + h * t
    return x_k
```

Similarly, we can combine line search with Newton's method. The improved Newton's method is also called the **Damped Newton's Method**.

Algorithm	**Damped Newton's Method**
Problem Type	Unconstrained nonlinear optimization
Given	Differentiable objective function $f(\mathbf{x})$
Goal	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
Algorithm Property	Iterative solution

Algorithm 5: Damped Newton's Method

Input: Differentiable objective function $f(\mathbf{x})$
Parameter: Threshold ϵ, initial value $\mathbf{x}_0$
Output: Optimal solution $\mathbf{x}^*$
$\mathbf{x}_k \leftarrow \mathbf{x}_0$
while $||\nabla_{\mathbf{x}} f(\mathbf{x}_k)|| > \epsilon$ **do**
  $\mathbf{t} \leftarrow -H_f^{-1}(\mathbf{x}_k)\nabla_{\mathbf{x}} f(\mathbf{x}_k)$
  $h \leftarrow \text{GR\_line\_search}(f, \mathbf{x}_k, \mathbf{t})$
  $\mathbf{x}_k \leftarrow \mathbf{x}_k + h\mathbf{t}$
$\mathbf{x}^* \leftarrow \mathbf{x}_k$

The corresponding Python code is shown below:

```python
def opt_nc_damp_newton(f_func, df_func, ddf_func, x_0,
    epsilon=1e-4):
    x_k, = x_0
    while True:
        grad = df_func(x_k)
        if np.linalg.norm(grad) < epsilon:
            break
        H_inv = np.linalg.inv(ddf_func(x_k))
        t = - H_inv @ grad
        h = GR_line_search(f_func, x_k, t)
        x_k = x_k + h * t
    return x_k
```

3.2.4 Conjugate Gradient Method

The conjugate gradient method originates from QP problems. Its core idea is to make the gradient directions of consecutive iterations conjugate with respect to the Hessian matrix during the iterative search process. Two vectors (u, v) are called conjugate with respect to a square matrix A if they satisfy $u^T Av = 0$. Researchers have found that compared to gradient descent, the conjugate gradient method can achieve the fastest convergence for QP problems without requiring explicit computation of the Hessian matrix as in Newton's method.

The detailed derivation of the conjugate gradient method is omitted here; please refer to other optimization theory textbooks. We only present its core gradient direction search formula. The conjugate gradient method using this formula is also called the F-R conjugate gradient method, where "F-R" stands for the surnames of its two inventors (Fletcher-Reeves).

$$\begin{aligned} \mathbf{t}_k &= -\nabla_{\mathbf{x}} f(\mathbf{x}_k), \quad k = 1 \\ \mathbf{t}_k &= -\nabla_{\mathbf{x}} f(\mathbf{x}_k) + \frac{||\nabla_{\mathbf{x}} f(\mathbf{x}_k)||^2}{||\nabla_{\mathbf{x}} f(\mathbf{x}_{k-1})||^2}\mathbf{t}_{k-1}, \quad k \geq 1 \end{aligned} \tag{I.3.6}$$

The conjugate gradient method is also a line search-based optimization method, where the conjugate gradient provides the search direction, and the specific step size is determined using line search. In practice, the search direction is reset to the gradient direction every n iterations.

The F-R conjugate gradient algorithm is summarized as follows:

Algorithm	**F-R Conjugate Gradient Method**
Problem Type	Unconstrained nonlinear optimization
Given	Differentiable objective function $f(\mathbf{x})$
Goal	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
Algorithm Property	Iterative solution

Algorithm 6: F-R Conjugate Gradient Method

Input: Differentiable objective function $f(\mathbf{x})$, initial value $\mathbf{x}_0$
Parameter: Threshold ϵ, reset interval n
Output: Optimal solution $\mathbf{x}^*$

$k \leftarrow 0$
while $||\nabla_{\mathbf{x}} f(\mathbf{x}_k)|| > \epsilon$ **do**
  $\mathbf{g}_k \leftarrow \nabla_{\mathbf{x}} f(\mathbf{x}_k)$
  if $k|n$ **then**
    $\mathbf{t}_k \leftarrow -\mathbf{g}_k$
  else
    $\mathbf{t}_k \leftarrow -\mathbf{g}_k + ||\mathbf{g}_k||/||\mathbf{g}_{k-1}||\mathbf{t}_{k-1}$
  $h \leftarrow \text{GR\_line\_search}(f, \mathbf{x}_k, \mathbf{t}_k)$
  $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + h\mathbf{t}_k$
  $k \leftarrow k + 1$
$\mathbf{x}^* \leftarrow \mathbf{x}_k$

The corresponding Python code is shown below:

```python
def opt_nc_conj_grad(f_func, df_func, x_0, epsilon=1e-4, n:int=10):
    k, x_k = 0, x_0
    while True:
        grad= df_func(x_k)
        grad_norm = np.linalg.norm(grad)
        if grad_norm < epsilon:
            break
        t = - grad
        if k % n != 0:
            t += grad_norm / last_grad_norm * last_t
        h = GR_line_search(f_func, x_k, t)
        x_k = x_k + h * t
        last_t, last_grad_norm = t, grad_norm
        k += 1
    return x_k
```

(Reference: Tsinghua University "Operations Research" course materials)

3.3 Equality Constrained Optimization

3.3.1 Lagrange Conditions

In the previous section, we presented various algorithms for solving unconstrained nonlinear optimization problems. How should we handle cases with equality constraints? One approach is to consider the necessary conditions that an optimal solution must satisfy, which form equations involving the optimization variables, and then solve these equations.

Specifically, consider a point $\mathbf{x}$ within the feasible region of a constrained nonlinear optimization problem. If moving a small distance from this point in a certain direction keeps it within the feasible region, this direction $\mathbf{p}$ is called a **feasible direction**. Assuming the objective function and constraints are continuously differentiable, we can intuitively derive the following necessary condition: **At the optimal solution, the gradient direction of the objective function cannot be a feasible direction.**

First, consider the simpler case of convex optimization problems with only equality constraints $\mathbf{g}(\mathbf{x}) = 0$. In the n-dimensional Euclidean space of decision variables, a single equality constraint $g_i(\mathbf{x}) = 0$ defines an $(n-1)$-dimensional hyperplane. At any feasible solution, a feasible direction $\mathbf{p}$ lies within the hyperplane, meaning it is orthogonal to the gradient of the constraint:

$$\mathbf{p} \perp \nabla_{\mathbf{x}} g_i(\mathbf{x})$$

For m constraints, their combined effect is the intersection of these hyperplanes. Thus, feasible directions must be orthogonal to the space spanned by the normal vectors of the hyperplanes. The normal vector direction of a hyperplane is precisely the gradient direction of the constraint function, i.e.,

$$\mathbf{p} \perp \sum_{i=1}^{m} \lambda_i \nabla_{\mathbf{x}} g_i(\mathbf{x}), \forall \lambda \neq 0$$

Earlier, we intuitively derived the necessary condition that at the optimal solution, the gradient direction of the objective function cannot be a feasible direction. Mathematically, this means ∇f must be orthogonal to feasible directions:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) \perp \mathbf{p}$$

Combining these two orthogonality conditions, we conclude that the gradient of the objective function $\nabla_{\mathbf{x}} f(\mathbf{x})$ must lie within the space spanned by the normal vectors of the hyperplanes:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla_{\mathbf{x}} g_i(\mathbf{x}) = 0, \forall \lambda$$

The above conclusion can be written in a more standard equivalent form. Define the **Lagrange function** as:

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x}) \tag{I.3.7}$$

Then, the necessary conditions for the optimal solution $\mathbf{x}^{\star}$ of the equality-constrained nonlinear optimization problem are:

$$\begin{aligned} \nabla_{\mathbf{x}} L(\mathbf{x}, \lambda) = 0 \\ \nabla_{\lambda} L(\mathbf{x}, \lambda) = 0 \end{aligned} \tag{I.3.8}$$

Equation I.3.8 is called the **Lagrange conditions**. The coefficients λ are called **Lagrange multipliers**. The method of converting a constrained optimization problem into solving the Lagrange conditions is called the **Lagrange multiplier method**. Note: The Lagrange conditions are not equivalent to the extremum of the L function.

From the perspective of optimization problems, the Lagrange multiplier method essentially constructs a new unconstrained optimization problem by expanding the original optimization variables $\mathbf{x}$ into $[\mathbf{x}^T, \lambda^T]^T$. Therefore, algorithms applicable to unconstrained problems can also be used for equality-constrained cases.

For **convex optimization problems**, the Lagrange conditions are necessary conditions satisfied under the premise that an optimal solution exists. For non-convex cases, under certain regularity conditions on f and g, the Lagrange conditions are also necessary for optimality. We will not elaborate further here.

3.3.2 Optimization Algorithms for Equality Constraints

As mentioned above, transforming the problem using Lagrange conditions allows us to derive algorithms for equality-constrained optimization problems from unconstrained optimization algorithms. We illustrate this with the line search gradient descent method as an example; other algorithms can be derived analogously and will not be repeated here.

Let:

$$\bar{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ \lambda \end{bmatrix} \tag{I.3.9}$$

Then:

$$\nabla_{\bar{\mathbf{x}}} L(\bar{\mathbf{x}}) = \begin{bmatrix} \nabla_{\mathbf{x}} f(\mathbf{x}) + \nabla_{\mathbf{x}} \mathbf{g}(\mathbf{x}) \lambda \\ \mathbf{g}(\mathbf{x}) \end{bmatrix} \tag{I.3.10}$$

According to the Lagrange conditions, we need to solve $\nabla_{\bar{\mathbf{x}}} L(\bar{\mathbf{x}}) = 0$. Construct a new optimization objective:

$$\min_{\bar{\mathbf{x}}} J(\bar{\mathbf{x}}) = \frac{1}{2}(\nabla_{\bar{\mathbf{x}}} L)^T(\bar{\mathbf{x}}) \nabla_{\bar{\mathbf{x}}} L(\bar{\mathbf{x}}) \tag{I.3.11}$$

We still denote:

$$H_f(\mathbf{x}) := \nabla^2_{\mathbf{x}} f(\mathbf{x})$$

Thus, the gradient of the new optimization objective is:

$$\begin{aligned} \nabla_{\bar{\mathbf{x}}} J(\bar{\mathbf{x}}) &= (\nabla^2_{\bar{\mathbf{x}}} L(\bar{\mathbf{x}})) \nabla_{\bar{\mathbf{x}}} L(\bar{\mathbf{x}}) \\ &= \begin{bmatrix} H_f(\mathbf{x}) + \sum_{i=1}^m \lambda_i \nabla_{\mathbf{x}} g_i'(\mathbf{x}) & \nabla_{\mathbf{x}} \mathbf{g}(\mathbf{x}) \\ (\nabla_{\mathbf{x}} \mathbf{g})^T(\mathbf{x}) & 0 \end{bmatrix} \begin{bmatrix} \nabla_{\mathbf{x}} f(\mathbf{x}) + \lambda^T \nabla_{\mathbf{x}} \mathbf{g}(\mathbf{x}) \\ \mathbf{g}(\mathbf{x}) \end{bmatrix} \end{aligned} \tag{I.3.12}$$

It can be seen that solving equality-constrained optimization problems requires the second derivatives of the functions f and g. Through this transformation, the constrained optimization problem is converted into an unconstrained one, which can be solved using the algorithms introduced in the previous section.

Using line search gradient descent as the outer optimization algorithm, we summarize the Lagrange multiplier method for equality-constrained optimization problems as follows:

Algorithm 7: Lagrange Multiplier Method

Input: Twice-differentiable objective function $f(\mathbf{x})$, initial value $\mathbf{x}_0$
Input: Twice-differentiable equality constraint function $\mathbf{g}(\mathbf{x})$
Parameter: Threshold ϵ, initial multiplier value λ_0
Output: Optimal solution $\mathbf{x}^*$

$\bar{\mathbf{x}}_k \leftarrow [\mathbf{x}_0^T \quad \lambda_0^T]^T$
while $||\nabla_{\bar{\mathbf{x}}} L(\bar{\mathbf{x}}_k)|| > \epsilon$ **do**
  $\nabla^2_{\bar{\mathbf{x}}} L \leftarrow \begin{bmatrix} H_f(\mathbf{x}_k) + \sum_{i=1}^m \lambda_{k,i} \nabla_{\mathbf{x}} g_i'(\mathbf{x}_k) & \nabla_{\mathbf{x}} \mathbf{g}(\mathbf{x}_k) \\ (\nabla_{\mathbf{x}} \mathbf{g})^T(\mathbf{x}_k) & 0 \end{bmatrix}$
  $\nabla_{\bar{\mathbf{x}}} L \leftarrow \begin{bmatrix} \nabla_{\mathbf{x}} f(\mathbf{x}_k) + \lambda_k^T \nabla_{\mathbf{x}} \mathbf{g}(\mathbf{x}_k) \\ \mathbf{g}(\mathbf{x}_k) \end{bmatrix}$
  $\mathbf{t} \leftarrow -(\nabla^2_{\bar{\mathbf{x}}} L)^{-1} \nabla_{\bar{\mathbf{x}}} L$
  $h \leftarrow \text{GR\_line\_search}(L, \bar{\mathbf{x}}_k, \mathbf{t})$
  $\bar{\mathbf{x}}_k \leftarrow \bar{\mathbf{x}}_k + h\mathbf{t}$
  $\nabla_{\bar{\mathbf{x}}} L(\bar{\mathbf{x}}_k) \leftarrow [(\nabla_{\mathbf{x}} f(\mathbf{x}) + \lambda \nabla_{\mathbf{x}} \mathbf{g}(\mathbf{x}))^T \quad \mathbf{g}^T(\mathbf{x})]^T$
  $L \leftarrow f + \lambda^T \mathbf{g}$
$\mathbf{x}^*, \lambda^* \leftarrow \bar{\mathbf{x}}_k$

Algorithm	**Lagrange Multiplier Method**
Problem Type	Equality-constrained nonlinear optimization
Known	Twice-differentiable objective function $f(\mathbf{x})$ Twice-differentiable equality constraint function $\mathbf{g}(\mathbf{x})$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$ $s.t. \quad \mathbf{g}(\mathbf{x}) = 0$
Algorithm Property	Iterative solution

The corresponding Python code is shown below:

```python
def opt_eqc_gd_line(df_func, ddf_func, g_func, dg_func, ddg_func, x_0, epsilon=1e-4):
    lambda_0 = np.ones_like(g_func(x_0))
    x_ext_k = np.concatenate([x_0, lambda_0])
    n_x, n_lam = x_0.shape[0], lambda_0.shape[0]
    def J_func(x_ext):
```

```
        x, lambda_ = x_ext[:n_x], x_ext[n_x:]
        gradL = np.concatenate([
            (df_func(x)[None, :] + lambda_[None, :] @ dg_func(x))[0],
            g_func(x)
        ])
        return 0.5 * (gradL[None, :] @ gradL[:, None])[0,0]
    while True:
        x_k, lambda_k = x_ext_k[:n_x], x_ext_k[n_x:]
        gradL = np.concatenate([
            (df_func(x_k)[None, :] + lambda_k[None, :] @ dg_func(x_k))[0],
            g_func(x_k)
        ]) # (m+n,)
        tmp = ddf_func(x_k) + np.einsum("i,ijk->jk", lambda_k, ddg_func(x_k)) # (n, n)
        hessL = np.block([
            [tmp, dg_func(x_k).T],
            [dg_func(x_k), np.zeros((n_lam, n_lam))]
        ])
        dotJ = hessL @ gradL
        if np.linalg.norm(dotJ) < epsilon:
            break
        t = - dotJ
        h = GR_line_search(J_func, x_ext_k, t)
        x_ext_k = x_ext_k + h * t
    return x_ext_k[:n_x]
```

3.4 Inequality-Constrained Optimization

3.4.1 KKT Conditions

Compared to equality constraints, the treatment of inequality constraints is slightly more complex.

First, not every inequality constraint will affect the feasible/optimal solution. For an inequality constraint $h_j(\mathbf{x}) \leq 0$, if $\mathbf{x}$ makes it hold with equality, it means $\mathbf{x}$ lies on the boundary of the feasible region defined by the constraint $h_j(\mathbf{x}) \leq 0$. In this case, the constraint is called an **active constraint**. Otherwise, it is called an **inactive constraint**.

We still consider the problem from the perspective of necessary conditions for feasible directions at the optimal solution. For a feasible solution $\mathbf{x}$, inactive constraints correspond to hyperplanes far from this solution and do not affect the choice of feasible directions. However, if the constraint $h_j(\mathbf{x}) \leq 0$ is active, then the feasible direction cannot be one that decreases h_j but can be one that increases h_j.

Therefore, for the inequality-constrained optimization problem 3.1, at the optimal solution, the inner product between the gradient of each active constraint and the feasible direction must be positive. This means the gradient of the objective function must lie within the half-space formed by the negative linear combination of the gradients of active constraints. Combining with equality constraints, we can write the following gradient relationship:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \sum_{i=1}^{m} u_i \nabla_{\mathbf{x}} g_i(\mathbf{x}) - \sum_{j \in c(x)} v_j \nabla_{\mathbf{x}} h_j(\mathbf{x}), \forall \mathbf{u}, \mathbf{v} \geq 0$$

Here, $c(x)$ denotes the set of indices of active constraints at the feasible solution $\mathbf{x}$.

We organize this necessary condition into a more common form, called the **KKT conditions**:

$$
\begin{aligned}
\frac{\partial}{\partial \mathbf{x}} L(\mathbf{x}, \mathbf{u}, \mathbf{v}) &= 0 \\
\frac{\partial}{\partial \mathbf{u}} L(\mathbf{x}, \mathbf{u}, \mathbf{v}) &= 0 \\
\frac{\partial}{\partial \mathbf{v}} L(\mathbf{x}, \mathbf{u}, \mathbf{v}) &\leq 0 \\
v_j \frac{\partial}{\partial v_j} L(\mathbf{x}, \mathbf{u}, \mathbf{v}) &= 0, j \in c(x) \\
v_j &\geq 0, j \in c(x)
\end{aligned}
\tag{I.3.13}
$$

where

$$
L(\mathbf{x}, \mathbf{u}, \mathbf{v}) = f(\mathbf{x}) + \sum_{i=1}^{m} u_i g_i(\mathbf{x}) + \sum_{j \in c(x)} v_j h_j(\mathbf{x}) \tag{I.3.14}
$$

The fourth line in Equation I.3.13 is called the complementary slackness condition. It means that if the j-th inequality constraint is active, then $h_j = 0$, and v_j can be positive, with the feasible direction not aligning with h'_j. If the j-th inequality constraint is inactive, then $h_j < 0$, v_j must be 0, and the feasible direction is independent of h'_j.

For convex optimization problems, the KKT conditions are necessary but not sufficient, meaning solutions satisfying KKT are not necessarily optimal. For non-convex problems, if the functions $f, \mathbf{g}, \mathbf{h}$ satisfy the Linear Independence Constraint Qualification (LICQ), then the KKT conditions are also necessary (but not sufficient) for optimality. For proofs, readers may refer to relevant optimization textbooks, which will not be elaborated here.

3.4.2 Barrier Function Method

Since the KKT conditions are not sufficient, we cannot directly apply them to solving inequality-constrained problems. However, we can incorporate inequality constraints into the objective function in an alternative way.

Specifically, we define a univariate function on the negative real axis, where the function value increases as x approaches 0 and becomes infinite at $x = 0$. Such a function is called a **barrier function**. A typical barrier function is:

$$
barrier(x) = -\ln(-x) \tag{I.3.15}
$$

Thus, we can incorporate the inequality constraints into the objective function as:

$$
f_b(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda \sum_{j=1}^{l} \ln(-h_j(\mathbf{x})) \tag{I.3.16}
$$

where λ is the penalty factor. The gradient of the new objective function is:

$$
\nabla_{\mathbf{x}} f_b(\mathbf{x}, \lambda) = \nabla_{\mathbf{x}} f(\mathbf{x}) - \lambda \sum_{j=1}^{l} \frac{\nabla_{\mathbf{x}} h_j(\mathbf{x})}{h_j(\mathbf{x})} \tag{I.3.17}
$$

Here, different values of λ transform the constrained optimization problem into different unconstrained optimization problems. Clearly, the unconstrained objective function f_b differs from the original f. When λ is large, f_b primarily reflects the feasible region, and its optimal solution may deviate significantly from that of f. When λ is small, the constraints' influence weakens, and f_b's optimal solution approaches that of f.

By setting a sequence of decreasing λ values and running multiple unconstrained optimizations—each initialized with the previous solution—we can iteratively find the original function's minimum while satisfying the constraints. For simplicity, the multiplier can follow the decay law:

$$
\lambda_k = \lambda_0 * \gamma^k
$$

where k denotes the iteration count of the unconstrained optimization problem.

The above barrier function method is an **interior-point method**. Based on the unconstrained LSGD approach, we summarize the Barrier Function LSGD algorithm as follows:

Algorithm	**Barrier Function LSGD**
Problem Type	Inequality-constrained nonlinear optimization
Given	Objective function $f(\mathbf{x})$ Inequality constraint functions $\mathbf{h}(\mathbf{x})$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$ $s.t.\quad \mathbf{h}(\mathbf{x}) \leq 0$
Algorithm Property	Iterative solution

Algorithm 8: Barrier Function LSGD

Input: Differentiable objective function $f(\mathbf{x})$, initial value $\mathbf{x}_0$

Input: Inequality constraint functions $\mathbf{h}(\mathbf{x})$

Parameter: Threshold ϵ, initial multiplier λ_0, multiplier decay rate γ

Output: Optimal solution $\mathbf{x}^*$

$\mathbf{x}_k \leftarrow \mathbf{x}_0$

while *True* **do**

- $\lambda_k \leftarrow \lambda_0 * \gamma^k$
- $f_b \leftarrow f - \lambda_k \mathbf{1}^T \mathbf{h}$
- $\nabla_{\mathbf{x}} f_b \leftarrow \nabla_{\mathbf{x}} f(\mathbf{x}) - \lambda_k \sum_{j=1}^{l} \nabla_{\mathbf{x}} h_j(\mathbf{x}) / h_j(\mathbf{x})$
- $\mathbf{x}_k \leftarrow$ opt_nc_LSGD$(\mathbf{x}_k, f_b, \nabla_{\mathbf{x}} f_b)$

$\mathbf{x}^* \leftarrow \mathbf{x}_k$

The corresponding Python code is as follows:

```python
def opt_neqc_LSGD(f_func, df_func, h_func, dh_func, x_0, lambda_0=100.0, gamma=0.8, epsilon=1e-4):
    x_k, lambda_b = x_0, lambda_0
    while True:
        def fb_func(x):
            return f_func(x) - lambda_b * np.sum(np.log(-h_func(x)))
        def dfb_func(x):
            t = (1/h_func(x))[None, :] @ dh_func(x)
            return (df_func(x) - lambda_b * t)[0]
        x_new = opt_nc_LSGD(fb_func, dfb_func, x_k)
        x_k, lambda_b = x_new, lambda_b * gamma
        if lambda_b < epsilon:
            break
    return x_k
```

3.5 Quadratic Programming (QP)

Quadratic programming problems were introduced in Section 1, characterized by quadratic objective functions and linear equality/inequality constraints.

3.5.1 Unconstrained and Equality-Constrained Cases

For QP problems, both unconstrained and equality-constrained cases can be solved analytically. Specifically, for the unconstrained case, we derived the global minimum of quadratic functions when discussing Newton's method. That is, the optimal solution satisfying:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x} \tag{I.3.18}$$

is given by:

$$\mathbf{x}^* = -H^{-1}\mathbf{g} \tag{I.3.19}$$

The existence of this analytical solution requires the matrix H to be invertible.

Algorithm	**Unconstrained QP Analytical Solution**
Problem Type	Unconstrained QP problem
Given	Positive semidefinite matrix H, vector $\mathbf{g}$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x}$
Algorithm Property	Analytical solution

Algorithm 9: Unconstrained QP Analytical Solution

Input: Positive semidefinite matrix H, vector $\mathbf{g}$
Output: Optimal solution $\mathbf{x}^*$
$\mathbf{x}^* \leftarrow H^{-1}\mathbf{g}$

For the equality-constrained problem, we use the Lagrange multiplier method, which gives

$$L(\mathbf{x}, \lambda) = \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x} + \lambda^T(M_{eq}\mathbf{x} - b) \tag{I.3.20}$$

From the necessary conditions for optimality, we have

$$\begin{bmatrix} H & M_{eq}^T \\ M_{eq} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \lambda \end{bmatrix} = \begin{bmatrix} -\mathbf{g} \\ b_{eq} \end{bmatrix}$$

Therefore, if the matrix

$$\bar{H} = \begin{bmatrix} H & M_{eq}^T \\ M_{eq} & 0 \end{bmatrix}$$

is invertible, the analytical form of the optimal solution is

$$\begin{bmatrix} \mathbf{x}^* \\ \lambda^* \end{bmatrix} = \begin{bmatrix} H & M_{eq}^T \\ M_{eq} & 0 \end{bmatrix}^{-1} \begin{bmatrix} -\mathbf{g} \\ b_{eq} \end{bmatrix} \tag{I.3.21}$$

Algorithm	**Equality-Constrained QP Analytical Solution**
Problem Type	Equality-Constrained QP Problem
Given	Positive semidefinite matrix H, vector $\mathbf{g}$ Matrix M_{eq}, vector b_{eq}
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x}$
Algorithm Property	Analytical Solution

Algorithm 10: Equality-Constrained QP Analytical Solution

Input: Positive semidefinite matrix H, vector $\mathbf{g}$
Input: Matrix M_{eq}, vector b_{eq}
Output: Optimal solution $\mathbf{x}^*$
$\begin{bmatrix} \mathbf{x}^* \\ \lambda^* \end{bmatrix} \leftarrow \begin{bmatrix} H & M_{eq}^T \\ M_{eq} & 0 \end{bmatrix}^{-1} \begin{bmatrix} -\mathbf{g} \\ b_{eq} \end{bmatrix}$

The corresponding Python code is shown below:

```python
def opt_qp_eqcons(H, g, M_eq, b_eq, x_0):
    n_lam = g.shape[0]
    H_ext = np.block([[H, M_eq.T], [M_eq, np.zeros([n_lam, n_lam])]])
    x_ext = np.linalg.inv(H_ext) @ np.row_stack([-g, b_eq])
    return x_ext[:-n_lam]
```

3.5.2 Inequality-Constrained QP

For general QP problems with inequality constraints, we can use the barrier function method introduced earlier. Specifically, compared to general nonlinear optimization problems, the three functions in a general QP problem are:

$$
\begin{aligned}
f(\mathbf{x}) &= \frac{1}{2}\mathbf{x}^T H\mathbf{x} + g^T\mathbf{x} \\
\mathbf{g}(\mathbf{x}) &= M_{eq}\mathbf{x} - b_{eq} \\
\mathbf{h}(\mathbf{x}) &= M\mathbf{x} - b
\end{aligned}
$$

Let

$$
M = \begin{bmatrix} \mathbf{m}_1^T \\ \mathbf{m}_2^T \\ \cdots \\ \mathbf{m}_l^T \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_l \end{bmatrix}
$$

Following the barrier function method described earlier, we add the inequality constraint barrier function to the optimization objective:

$$
b(\mathbf{x}) = -\sum_{i=1}^{l} \ln(b_i - \mathbf{m}_i^T\mathbf{x})
$$

$$
f_b(\mathbf{x}) = f(\mathbf{x}) + \lambda_b b(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T H\mathbf{x} + g^T\mathbf{x} - \lambda_b \sum_{i=1}^{l} \ln(b_i - \mathbf{m}_i^T\mathbf{x})
$$

Adding the Lagrange multiplier and expanding the optimization variables, we have:

$$
L_b(\bar{\mathbf{x}}) = \frac{1}{2}\mathbf{x}^T H\mathbf{x} + g^T\mathbf{x} + \lambda^T(M_{eq}\mathbf{x} - b_{eq}) - \lambda_b \sum_{i=1}^{l} \ln(b_i - \mathbf{m}_i^T\mathbf{x})
$$

Taking the derivative with respect to the expanded $\mathbf{x}$, we obtain:

$$
\nabla_{\bar{\mathbf{x}}} L_b(\bar{\mathbf{x}}) = \left(\frac{\partial L_b}{\partial \bar{\mathbf{x}}}\right)^T = \begin{bmatrix} H\mathbf{x} + g + M_{eq}^T\lambda + \lambda_b \nabla_{\mathbf{x}} b(\mathbf{x}) \\ M_{eq}\mathbf{x} - b_{eq} \end{bmatrix}
$$

where

$$
\nabla_{\mathbf{x}} b(\mathbf{x}) = \sum_{i=1}^{l} \frac{1}{b_i - \mathbf{m}_i^T\mathbf{x}} \mathbf{m}_i \tag{I.3.22}
$$

According to the Lagrange conditions, we need to solve $L_b'(\bar{\mathbf{x}}) = 0$. The new optimization objective is:

$$
\min_{\bar{\mathbf{x}}} J(\bar{\mathbf{x}}) = \frac{1}{2}(\nabla_{\bar{\mathbf{x}}} L_b)^T(\nabla_{\bar{\mathbf{x}}} L_b)
$$

Therefore, the gradient of the new optimization objective is:

$$
\begin{aligned}
\nabla_{\bar{\mathbf{x}}} J(\bar{\mathbf{x}}) &= (\nabla_{\bar{\mathbf{x}}}^2 L_b)(\nabla_{\bar{\mathbf{x}}} L_b) \\
&= \begin{bmatrix} H + \lambda_b \nabla_{\mathbf{x}}^2 b(\mathbf{x}) & M_e^T \\ M_e & 0 \end{bmatrix} \begin{bmatrix} H\mathbf{x} + g + M_{eq}^T\lambda + \lambda_b \nabla_{\mathbf{x}} b(\mathbf{x}) \\ M_{eq}\mathbf{x} - b_{eq} \end{bmatrix}
\end{aligned} \tag{I.3.23}
$$

where

$$\nabla_{\mathbf{x}}^2 b(\mathbf{x}) = \sum_{i=1}^{l} \frac{1}{(b_i - \mathbf{m}_i^T \mathbf{x})^2} \mathbf{m}_i \mathbf{m}_i^T \tag{I.3.24}$$

Based on the LSGD algorithm, we summarize the barrier function method for general QP problems as follows:

Algorithm	**QP Barrier Function Method**
Problem Type	General QP Problem
Problem Description	Find the optimization variables that minimize the objective function under equality and inequality constraints
Given	Positive semidefinite matrix H, vector g, feasible initial value $\mathbf{x}_0$ Matrix M_{eq}, vector b_{eq} Matrix M, vector b
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H \mathbf{x} + \mathbf{g}^T \mathbf{x}$ $s.t. \quad M_{eq}\mathbf{x} = b_{eq}, M\mathbf{x} \leq b$
Algorithm Property	Iterative solution

Algorithm 11: QP Barrier Function Method (solve_QP_barrier)

Input: Positive semidefinite matrix H, vector g, feasible initial value $\mathbf{x}_0$
Input: Matrix M_{eq}, vector b_{eq}
Input: Matrix M, vector b
Parameter: Thresholds ϵ_1, ϵ_2, decay factor γ
Parameter: Initial multiplier values λ_0, λ_{b0}
Output: Optimal solution $\mathbf{x}^*$

$\bar{\mathbf{x}}_k \leftarrow [\mathbf{x}_0^T \quad \lambda_0^T]^T$
while *True* **do**
  $\lambda_k \leftarrow \lambda_{b0}\gamma^k$
  while *True* **do**
    $\nabla_{\mathbf{x}} b \leftarrow \sum_{i=1}^{l} \mathbf{m}_i / (b_i - \mathbf{m}_i^T \mathbf{x}_k)$
    $\nabla_{\mathbf{x}}^2 b \leftarrow \sum_{i=1}^{l} \mathbf{m}_i \mathbf{m}_i^T / (b_i - \mathbf{m}_i^T \mathbf{x}_k)^2$
    $L_x \leftarrow H\mathbf{x}_k + g + M_{eq}^T \lambda + \lambda_k \nabla_{\mathbf{x}} b$
    $L_\lambda \leftarrow M_{eq}\mathbf{x}_k - b_{eq}$
    $J(\bar{\mathbf{x}}) \leftarrow \frac{1}{2}(||L_x||^2 + ||L_\lambda||^2)$
    if $J(\bar{\mathbf{x}}) < \epsilon_1$ **then**
      Break
    $\nabla_{\bar{\mathbf{x}}} J(\bar{\mathbf{x}}_k) \leftarrow \begin{bmatrix} H + \lambda_b \nabla_{\mathbf{x}}^2 b & M_e^T \\ M_e & 0 \end{bmatrix} \begin{bmatrix} L_x \\ L_\lambda \end{bmatrix}$
    $h \leftarrow \text{GR\_line\_search}(J, \bar{\mathbf{x}}_k, -\nabla_{\bar{\mathbf{x}}} J(\bar{\mathbf{x}}_k))$
    $\bar{\mathbf{x}}_k \leftarrow \bar{\mathbf{x}}_k - h\nabla_{\bar{\mathbf{x}}} J(\bar{\mathbf{x}}_k)$
  if $\lambda_k < \epsilon_2$ **then**
    Break
$\mathbf{x}^*, \lambda^* \leftarrow \bar{\mathbf{x}}_k$

The corresponding Python code is shown below:

```
def opt_qp_barrier(H, g, M_eq, b_eq, M, b, x_0, lambda_b0=10, gamma=0.95, beta=0.9,
    epsilon_1=1e-1, epsilon_2=1e-4):
    lambda_0, lambda_b = np.ones_like(b_eq), lambda_b0
    x_ext_k = np.concatenate([x_0, lambda_0])
    n_x, n_lamb = x_0.shape[0], lambda_0.shape[0]
```

```
    def barrier(x_ext):
        return np.sum(-np.log(b - M @ x_ext[:n_x]))
    while True:
        lambda_b, epsilon_1 = lambda_b * gamma, epsilon_1 * np.sqrt(gamma)
        momentum = np.zeros_like(x_ext_k)
        while True:
            def dL_b(x_ext, lbd_b):
                x, lambda_ = x_ext[:n_x], x_ext[n_x:]
                t = np.zeros_like(b)
                for i in range(b.shape[0]):
                    t += M[i, :] / (b[i] - M[i:i+1, :] @ x)
                L_x = H @ x + g + M_eq.T @ lambda_ + lbd_b * t
                L_lambda = M_eq @ x - b_eq
                return L_x, L_lambda
            def J_func(x_ext):
                A, B = dL_b(x_ext, lambda_b)
                return 0.5 * (np.sum(A**2) + np.sum(B**2))
            L_x, L_lambda = dL_b(x_ext_k, lambda_b)
            K_k, x_k = np.zeros_like(H), x_ext_k[:n_x]
            for i in range(b.shape[0]):
                m_i = M[i:i+1, :]
                K_k += m_i.T @ m_i / (m_i @ x_k - b[i])**2
            tmp1 = np.block([
                [H + lambda_b * K_k, M_eq.T], [M_eq, np.zeros([n_lamb, n_lamb])]
            ])
            gradJ = tmp1 @ np.hstack([L_x, L_lambda])
            momentum = beta * momentum + (1-beta) * gradJ
            h = GR_line_search(J_func, x_ext_k, - momentum, h_0=1)
            # print(x_ext_k, J_func(x_ext_k), h, momentum)
            x_ext_k = x_ext_k - h * momentum
            if J_func(x_ext_k) < epsilon_1:
                break
        neq_gap = lambda_b * barrier(x_ext_k)
        if neq_gap < epsilon_2:
            return x_ext_k[:n_x]
```

Thus, as long as we can formulate the problem in the general QP form, we can solve it using the above algorithm.

3.6 Nonlinear Least Squares

3.6.1 Gradient Descent Method

For the optimization of least squares problems, we first consider the unconstrained case. We begin with the relatively simple gradient descent method. Assume the vector-valued function $\mathbf{e}(\mathbf{x})$ is first-order differentiable, and let

$$J(\mathbf{x}) = \frac{\partial \mathbf{e}}{\partial \mathbf{x}} \tag{I.3.25}$$

Assuming $H^T = H$, we have

$$\begin{aligned}\frac{\partial f}{\partial \mathbf{x}} &= \frac{\partial}{\partial \mathbf{x}} \frac{1}{2}(\mathbf{e}^T(\mathbf{x}) H \mathbf{e}(\mathbf{x})) \\ &= \frac{\partial f}{\partial \mathbf{e}} \frac{\partial \mathbf{e}}{\partial \mathbf{x}} \\ &= \mathbf{e}^T(\mathbf{x}) H J(\mathbf{x})\end{aligned}$$

We can use the gradient descent method for iterative solving:

Algorithm	Gradient Descent Method
Problem Type	Unconstrained Least Squares
Given	Differentiable function $\mathbf{e}(\mathbf{x})$
Objective	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
Algorithm Property	Iterative solution

Algorithm 12: Gradient Descent Method

Input: Symmetric matrix H, differentiable vector-valued function $\mathbf{e}(\mathbf{x})$
Parameter: Threshold ϵ, step size α, initial value $\mathbf{x}_0$
Output: Optimal solution $\mathbf{x}^*$

$\mathbf{x}_k \leftarrow \mathbf{x}_0$
while *True* **do**
  $\nabla_x f(\mathbf{x}_k) \leftarrow J^T(\mathbf{x}) H \mathbf{e}(\mathbf{x})$
  if $||\nabla_x f(\mathbf{x}_k)|| < \epsilon$ **then**
    break
  $\mathbf{x}_k \leftarrow \mathbf{x}_k - \alpha \nabla_x f(\mathbf{x}_k)$
$\mathbf{x}^* \leftarrow \mathbf{x}_k$

The corresponding Python code is shown below:

```python
def opt_minls_gd(e_func, J_func, H, x_0, epsilon=1e-4,
↪  alpha=1e-2):
    x_k = x_0
    while True:
        dotf = J_func(x_k) @ H @ e_func(x_k)
        if np.linalg.norm(dotf) < epsilon:
            break
        x_k = x_k - alpha * dotf
    return x_k
```

3.6.2 Gauss-Newton Method

The gradient descent method still has issues such as slow convergence. We adopt a similar approach to Newton's method by expanding around $\mathbf{x}_k$. Since the objective function itself is quadratic, we only need to expand to the first order:

$$\hat{\mathbf{e}}(\mathbf{x}; \mathbf{x}_k) = \mathbf{e}(\mathbf{x}_k) + J(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k)$$

Thus, we have:

$$\begin{aligned}\hat{f}(\mathbf{x}; \mathbf{x}_k) &= \frac{1}{2}\hat{\mathbf{e}}^T(\mathbf{x}; \mathbf{x}_k) H \hat{\mathbf{e}}(\mathbf{x}; \mathbf{x}_k) \\ &= \frac{1}{2}(\delta^T J^T H J \delta + 2\mathbf{e}^T(\mathbf{x}_k) H J \delta + \mathbf{e}^T(\mathbf{x}_k) H \mathbf{e}(\mathbf{x}_k))\end{aligned}$$

where $\delta = \mathbf{x} - \mathbf{x}_k$. The necessary and sufficient condition for this approximate function to reach its minimum is:

$$\frac{\partial \hat{f}}{\partial \mathbf{x}}(\mathbf{x}; \mathbf{x}_k) = 0$$

That is:

$$J^T(\mathbf{x}_k) H J(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) + J^T(\mathbf{x}_k) H \mathbf{e}(\mathbf{x}_k) = 0$$

Therefore, we have:

$$\mathbf{x}_{k+1} = (J^T(\mathbf{x}_k)HJ(\mathbf{x}_k))^{-1}J^T(\mathbf{x}_k)H\mathbf{e}(\mathbf{x}_k) \tag{I.3.26}$$

This method is also called the **Gauss-Newton Method**. We summarize the algorithm as follows:

Algorithm	**Gauss-Newton Method**
Problem Type	Unconstrained Least Squares
Given	Differentiable function $\mathbf{e}(\mathbf{x})$
Objective	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
Algorithm Property	Iterative solution

Algorithm 13: Gauss-Newton Method (Gauss_Newton)

Input: Symmetric matrix H, differentiable vector-valued function $\mathbf{e}(\mathbf{x})$
Parameter: Threshold ϵ, initial value $\mathbf{x}_0$
Output: Optimal solution $\mathbf{x}^*$

$\mathbf{x}_k \leftarrow \mathbf{x}_0$
while *True* **do**
  $H_n \leftarrow J^T(\mathbf{x}_k)HJ(\mathbf{x}_k)$
  $g_n \leftarrow J^T(\mathbf{x}_k)H\mathbf{e}(\mathbf{x}_k)$
  if $||g_n|| < \epsilon$ **then**
    break
  $\mathbf{x}_k \leftarrow \mathbf{x}_k - H_n^{-1}g_n$
$\mathbf{x}^* \leftarrow \mathbf{x}_k$

The corresponding Python code is shown below:

```python
def Gauss_Newton(e_func, J_func, H, x_0, epsilon=1e-4, alpha=1e-2):
    x_k = x_0
    while True:
        J_k = J_func(x_k)
        H_n = J_k.T @ H @ J_k
        g_n = J_k.T @ H @ e_func(x_k)
        if np.linalg.norm(g_n) < epsilon:
            break
        x_k = x_k - np.linalg.inv(H_n) @ g_n
    return x_k
```

3.6.3 Levenberg-Marquardt Method

The Gauss-Newton method is an approximation of Newton's method. Let us denote the update of optimization variables as:

$$\delta\mathbf{x}_k = \mathbf{x}_{k+1} - \mathbf{x}_k = -H_n^{-1}g_n \tag{I.3.27}$$

In fact, in the Gauss-Newton method, the computed $\delta\mathbf{x}_k$ is the update that maximizes the descent of $\hat{f}(\cdot)$, but not necessarily the one that maximizes the descent of $f(\cdot)$. Here, $\hat{f}(\cdot)$ is merely a second-order approximation of $f(\cdot)$ in some neighborhood around $\mathbf{x}_k$, and the discrepancy between them grows as we move further away from $\mathbf{x}_k$.

To address this, we can incorporate constraints on the update magnitude into the optimization problem. Specifically, for each iteration, we construct a new optimization problem that aims to optimize $\hat{f}$ while **keeping the update magnitude as small as possible**. More precisely, our optimization objective becomes:

$$\min_{\delta \mathbf{x}_k} L(\delta \mathbf{x}_k) = \frac{1}{2}||\mathbf{e}(\mathbf{x}_k) + J(\mathbf{x}_k)\delta \mathbf{x}_k||_H^2 + \frac{\lambda}{2}||\delta \mathbf{x}_k||^2 \tag{I.3.28}$$

Here, λ serves as a penalty term. A larger λ indicates our preference for smaller updates, and vice versa. For each iteration, we should adaptively adjust λ based on the discrepancy between the updates of $\hat{f}(\cdot)$ and $f(\cdot)$.

Specifically, when the difference between $\delta f(x)$ and $\delta \hat{f}(x)$ is large, we should increase λ; when the difference is small, we can appropriately decrease λ. We use the following metric to quantify the difference between $\delta f(x)$ and $\delta \hat{f}(x)$:

$$\rho_k = \frac{||\mathbf{f}(\mathbf{x}_k) - \mathbf{f}(\mathbf{x}_k + \delta \mathbf{x}_k)||}{||J(\mathbf{x}_k)\delta \mathbf{x}_k||} \tag{I.3.29}$$

By setting thresholds ρ_{min} and ρ_{max}, we can adaptively adjust λ as follows:

$$\lambda_{k+1} = \begin{cases} 2 * \lambda_k, & \rho_k > \rho_{max} \\ 0.5 * \lambda_k, & \rho_k < \rho_{min} \end{cases} \tag{I.3.30}$$

The new optimization problem is a quadratic program with respect to $\mathbf{x}_k$ and has an analytical solution. Taking the derivative of the loss function L with respect to $\mathbf{x}_k$, we obtain:

$$\frac{\partial L}{\partial \mathbf{x}_k} = J^T(\mathbf{x}_k)HJ(\mathbf{x}_k)\delta \mathbf{x}_k + J^T(\mathbf{x}_k)H\mathbf{e}(\mathbf{x}_k) + \lambda \delta \mathbf{x}_k$$

Let us define:

$$\begin{aligned} H_l &= J^T(\mathbf{x}_k)HJ(\mathbf{x}_k) + \lambda I \\ g_l &= J^T(\mathbf{x}_k)H\mathbf{e}(\mathbf{x}_k) \end{aligned} \tag{I.3.31}$$

Setting the derivative to zero yields:

$$\delta \mathbf{x}_k = \mathbf{x}_{k+1} - \mathbf{x}_k = -H_l^{-1} g_l \tag{I.3.32}$$

This algorithm is also known as the **Levenberg-Marquardt algorithm**, or **L-M method** for short. The algorithm is summarized below:

Algorithm	**L-M Method**
Problem Type	Unconstrained Least Squares
Given	Differentiable function $\mathbf{e}(\mathbf{x})$
Find	Optimal solution $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
Algorithm Property	Iterative Solution

Algorithm 14: L-M Method (Levenberg_Marquardt)

Input: Differentiable vector-valued function $\mathbf{e}(\mathbf{x})$, initial guess $\mathbf{x}_0$
Input: Symmetric matrix H
Parameter: Optimization threshold ϵ, discrepancy thresholds ρ_{min}, ρ_{max}
Parameter: Initial penalty term λ_0
Output: Optimal solution $\mathbf{x}^*$

$k \leftarrow 0$
while *True* **do**
  // Update optimization variables
  $H_l \leftarrow J^T(\mathbf{x}_k)HJ(\mathbf{x}_k) + \lambda_k I$
  $g_l \leftarrow J^T(\mathbf{x}_k)H\mathbf{e}(\mathbf{x}_k)$
  if $||g_l|| < \epsilon$ **then**
    break
  $\delta\mathbf{x}_k \leftarrow -H_l^{-1}g_l$
  $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \delta\mathbf{x}_k$
  // Update penalty term
  $f_k \leftarrow \mathbf{e}^T(\mathbf{x}_k)H\mathbf{e}(\mathbf{x}_k)$
  $f_{k+1} \leftarrow \mathbf{e}^T(\mathbf{x}_{k+1})H\mathbf{e}(\mathbf{x}_{k+1})$
  $\rho_k = |f_k - f_{k+1}|/||J(\mathbf{x}_k)\delta\mathbf{x}_k||$
  if $\rho_k > \rho_{max}$ **then**
    $\lambda_{k+1} \leftarrow 2 * \lambda_k$
  if $\rho_k < \rho_{min}$ **then**
    $\lambda_{k+1} \leftarrow 0.5 * \lambda_k$
  else
    $\lambda_{k+1} \leftarrow \lambda_k$
  $k \leftarrow k + 1$
$\mathbf{x}^* \leftarrow \mathbf{x}_k$

The corresponding Python code is shown below:

```python
def Levenberg_Marquardt(e_func, J_func, H, x_0, epsilon=1e-4, rmin=1e-5,
↪  rmax=1, lambda_0=1):
    x_k, lambda_k = x_0, lambda_0
    while True:
        J_k = J_func(x_k)
        g_l = J_k.T @ H @ e_func(x_k)
        if np.linalg.norm(g_l) < epsilon:
            break
        H_l = J_k.T @ H @ J_k + lambda_k * np.eye(x_0.shape[0])
        delta_x = - np.linalg.inv(H_l) @ g_l
        x_new = x_k + delta_x
        f_k = e_func(x_k)[None, :] @ H @ e_func(x_k)
        f_new = e_func(x_new)[None, :] @ H @ e_func(x_new)
        rho = np.abs(f_k - f_new) / np.linalg.norm(J_k @ delta_x)
        if rho > rmax and lambda_k < 1e2:
            lambda_k = 2 * lambda_k
        elif rho < rmin and lambda_k > 1e-10:
            lambda_k = 0.5 * lambda_k
        x_k = x_new
    return x_k
```

3.6.4 Schur Complement

In some sequential least squares optimization problems, we encounter the following **dimensionality reduction** requirement: the original multi-dimensional optimization variable $\mathbf{x}_k$ needs to be divided into two parts, $\mathbf{x}_r$ and $\mathbf{x}_d$, where $\mathbf{x}_d$ is to be removed and $\mathbf{x}_r$ is to be retained.

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_r \\ \mathbf{x}_d \end{bmatrix}$$

If the optimization terms and constraints related to $\mathbf{x}_r$ and $\mathbf{x}_d$ in the optimization problem can be completely separated, the problem becomes very simple—just discard the optimization and constraint terms corresponding to $\mathbf{x}_d$. However, in most practical problems, the optimization objectives and constraints are coupled together across variables. In such cases, forcibly removing the terms related to $\mathbf{x}_d$ would introduce significant errors in the next round of optimization. So, how can we separate this problem to both eliminate $\mathbf{x}_d$ and retain some necessary information?

One approach is to perform a second-order expansion locally on $\mathbf{x}_k$ where dimensionality reduction is needed, thereby approximating the original nonlinear optimization problem as a quadratic optimization problem. At this point, finding the extremum is equivalent to solving the linear equation $Ax = b$. Using block matrix operations from linear algebra, we can generate a constraint on $\mathbf{x}_r$ that is related to $\mathbf{x}_d$ but does not contain $\mathbf{x}_d$. Incorporating this constraint into the next round of optimization can help reduce errors to some extent.

Specifically, similar to the assumption in the L-M method from the previous section, at a certain point in sequential optimization, optimizing $\mathbf{x}$ essentially involves optimizing $\delta\mathbf{x}_k$ within the neighborhood of $\mathbf{x}_k$ under a first-order expansion:

$$\min_{\delta\mathbf{x}_k} L(\delta\mathbf{x}_k) = \frac{1}{2}||\mathbf{e}(\mathbf{x}_k) + J(\mathbf{x}_k)\delta\mathbf{x}_k||_H^2 \tag{I.3.33}$$

Expanding the quadratic form and finding the extremum, we obtain:

$$J(\mathbf{x}_k)^T H J(\mathbf{x}_k)\delta\mathbf{x}_k = -J(\mathbf{x}_k)^T H\mathbf{e}(\mathbf{x}_k)$$

Let:

$$\begin{aligned} A &= J(\mathbf{x}_k)^T H J(\mathbf{x}_k) \\ b &= -J(\mathbf{x}_k)^T H\mathbf{e}(\mathbf{x}_k) \end{aligned} \tag{I.3.34}$$

It follows that $A^T = A$. Assuming $\delta\mathbf{x}_k$ is also divided into retained and discarded parts:

$$\delta\mathbf{x}_k = \begin{bmatrix} \delta\mathbf{x}_{r,k} \\ \delta\mathbf{x}_{d,k} \end{bmatrix}$$

Then, partitioning A and b accordingly, we have:

$$\begin{bmatrix} A_{rr} & A_{rd} \\ A_{rd}^T & A_{dd} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x}_{r,k} \\ \delta\mathbf{x}_{d,k} \end{bmatrix} = \begin{bmatrix} b_r \\ b_d \end{bmatrix} \tag{I.3.35}$$

This is a system of linear equations. Below, we eliminate $\delta\mathbf{x}_{d,k}$ using Schur complement. First, from the lower equation of the system, we have:

$$A_{rd}^T\delta\mathbf{x}_{r,k} + A_{dd}\delta\mathbf{x}_{d,k} = b_d$$

Rearranging and assuming A_{dd} is invertible, we obtain:

$$\delta\mathbf{x}_{d,k} = A_{dd}^{-1}(b_d - A_{rd}^T\delta\mathbf{x}_{r,k})$$

Substituting into the upper equation of the system, we have:

$$A_{rr}\delta\mathbf{x}_{r,k} + A_{rd}A_{dd}^{-1}(b_d - A_{rd}^T\delta\mathbf{x}_{r,k}) = b_d$$

Simplifying, we obtain a linear constraint on $\delta\mathbf{x}_{r,k}$:

$$A_s \delta \mathbf{x}_{r,k} - b_s = 0 \tag{I.3.36}$$

where:

$$\begin{aligned} A_s &= A_{rr} - A_{rd} A_{dd}^{-1} A_{rd}^T \\ b_s &= b_d - A_{rd} A_{dd}^{-1} b_d \end{aligned} \tag{I.3.37}$$

Note that the Schur complement is only a linear constraint effective within the neighborhood of $\mathbf{x}_k$. Once the variables move away after an update, this constraint is no longer valid.

(References: Tsinghua University "Operations Research" course materials, "The Beauty of Control - Volume 2," "Fourteen Lectures on Visual SLAM")

4 Interpolation

4.1 Basic Concepts

Humanity employs two distinct mathematical approaches to describe the world: **continuous** and **discrete**. In physics, most macroscopic phenomena (such as motion and force, heat, light, electricity, etc.) are described by continuous equations governed by continuous physical laws. On the other hand, modern robotics heavily relies on digital computers for computation and processing, yet digital computers excel only at handling discrete information. To bridge the gap between continuous and discrete representations, numerous mathematical tools have been developed.

Among these, **interpolation** serves as a crucial tool for converting discrete representations into continuous ones. For a function $y = f(x)$, computers can easily store, process, and transmit its discretized sampled values $(x_0, y_0), ..., (x_N, y_N)$. Interpolation is the process of reconstructing these sampled values into a continuous function $f(x)$ according to specific requirements.

Depending on the application, different interpolation requirements arise, often reflected in concerns about the continuity of the function and its derivatives. For example, when given a set of discrete values representing a robot's positions, we may wish to reconstruct them into a continuous trajectory of motion. In this case, it is essential to ensure that the first derivative with respect to time (i.e., velocity) remains continuous and free from abrupt changes.

Thus, we can define the **C1 Continuous Interpolation Problem**. Note that here we focus only on univariate scalar-valued functions $f(\cdot) : \mathbb{R} \to \mathbb{R}$.

Problem	**C1 Continuous Interpolation**
Problem Description	Given a sequence of sampled points of a univariate scalar function, find an interpolating function that is continuously differentiable to the first order within the interval
Given	Sampled points $(x_i, y_i)_{0:N}$ Interpolation interval $[a, b]$
Find	A function $f \in C^1[a, b]$ $s.t. f(x_i) = y_i, i = 0, ..., N$

Similarly, if we further increase the continuity requirement and desire an interpolating function that is twice continuously differentiable, we arrive at the **C2 Continuous Interpolation Problem**.

Problem	**C2 Continuous Interpolation**
Problem Description	Given a sequence of sampled points of a univariate scalar function, find an interpolating function that is continuously differentiable to the second order within the interval
Given	Sampled points $(x_i, y_i)_{0:N}$ Interpolation interval $[a, b]$
Find	A function $f \in C^2[a, b]$ $s.t. f(x_i) = y_i, i = 0, ..., N$

For interpolation problems, we generally assume the sampled points are already sorted, i.e.,

$$a = x_0 < ... < x_N = b$$

We are only concerned with the interval between the smallest and largest sampled points, referred to as the **interpolation interval** $[a, b]$. Typically, we consider $a = x_0$ and $b = x_N$.

In this chapter, we will introduce the most commonly used spline interpolation methods in robotics.

4.2 Spline Interpolation

A spline is a flexible wooden strip used for manual drafting. In the era of manual drafting, drawing smooth curves passing through given points on a plane often required placing nails at each specified point and bending

the flexible wooden strip around them. The natural curvature of the strip would then produce a smooth curve.

In the information age, we still adopt this concept to solve the aforementioned continuous interpolation problems.

4.2.1 Quadratic Spline Interpolation

First, let us consider the simpler C1 continuous interpolation problem. In mathematics, many functions satisfy first-order continuity, with the simplest being quadratic functions. However, a quadratic function has only three configurable parameters, making it insufficient to guarantee passing through the required $N+1$ given points (often much greater than 3). Therefore, we employ the idea of piecewise functions, dividing the entire interpolation interval $[a, b]$ into smaller subintervals based on the given x_i values. Within each subinterval, we fit a quadratic function while ensuring the continuity of the first derivative. The resulting curve closely resembles the spline curve described earlier, hence it is called **quadratic spline interpolation**.

Specifically, we divide the interpolation interval into $N-1$ subintervals. For each subinterval $[x_i, x_{i+1}], i = 0, ..., N-1$, f_i is a quadratic function, and all segments f_i collectively form the function f, i.e.,

$$f_i(x) = a_0^{(i)} + a_1^{(i)} x + a_2^{(i)} x^2, x \in [x_i, x_{i+1}] \tag{I.4.1}$$

The N quadratic segments contain $3N$ unknown parameters, and the interpolation conditions provide the equations to solve for these unknowns. We have:

$$\begin{aligned} f_i(x_i) &= y_i, \quad i = 0, ..., N-1 \\ f_i(x_{i+1}) &= y_{i+1}, \quad i = 0, ..., N-1 \\ f_i'(x_i) &= f_{i-1}'(x_i), \quad i = 1, ..., N-1 \end{aligned}$$

These constraints yield a total of $3N-1$ equations. To match the number of equations with the number of unknowns, we need to add one more condition, such as:

$$f_0'(x_0) = 0$$

Substituting the quadratic functions into the above equations, we obtain:

$$\begin{aligned} a_0^{(i)} + a_1^{(i)} x_i + a_2^{(i)} x_i^2 &= y_i \\ a_0^{(i)} + a_1^{(i)} x_{i+1} + a_2^{(i)} x_{i+1}^2 &= y_{i+1} \\ a_1^{(i-1)} + 2a_2^{(i-1)} x_i - a_1^{(i-1)} - 2a_2^{(i-1)} x_{i+1} &= 0 \end{aligned}$$

It is evident that these equations are linear with respect to the unknown coefficients. Expressing all unknowns in the form of a standard linear system, we have:

$$\begin{bmatrix} P_{0,0} & & & & & \\ P_{1,0} & P_{1,1} & & & & \\ \dots & \dots & & & & \\ & P_{i,i-1} & P_{i,i} & & & \\ & & \dots & \dots & & \\ & & & P_{N-1,N-2} & P_{N-1,N-1} \end{bmatrix} \begin{bmatrix} a_:^{(0)} \\ a_:^{(1)} \\ \dots \\ a_:^{(i)} \\ \dots \\ a_:^{(N-1)} \end{bmatrix} = \begin{bmatrix} y^{(0)} \\ y^{(1)} \\ \dots \\ y^{(i)} \\ \dots \\ y^{(N-1)} \end{bmatrix} \tag{I.4.2}$$

where:

$$
\begin{aligned}
a_{:}^{(i)} &= \begin{bmatrix} a_0^{(i)} & a_1^{(i)} & a_2^{(i)} \end{bmatrix}^T \\
y^{(i)} &= \begin{bmatrix} y_i & y_{i+1} & 0 \end{bmatrix}^T \\
P_{i,i} &= \begin{bmatrix} 1 & x_i & x_i^2 \\ 1 & x_{i+1} & x_{i+1}^2 \\ 0 & -1 & -2x_i \end{bmatrix} \\
P_{i,i-1} &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 2x_i \end{bmatrix}
\end{aligned}
\tag{I.4.3}
$$

Solving the above system of equations yields the desired $f(x)$. Here, we obtain the interpolation result for a single-input single-output function. For single-input multi-output functions, each output dimension can be interpolated separately as a univariate function.

Algorithm 15: Quadratic Spline Interpolation

Input: Sampling points $(x_i, y_i)_{0:N}$

Output: Interpolation function $f \in C^1[a, b]$

for $i \in 0, ..., N-1$ **do**

$a_{:}^{(i)}, y^{(i)} \leftarrow \begin{bmatrix} a_0^{(i)} & a_1^{(i)} & a_2^{(i)} \end{bmatrix}^T, \begin{bmatrix} y_i & y_{i+1} & 0 \end{bmatrix}$

$P_{i,i}, P_{i,i-1} \leftarrow \begin{bmatrix} 1 & x_i & x_i^2 \\ 1 & x_{i+1} & x_{i+1}^2 \\ 0 & -1 & -2x_i \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 2x_i \end{bmatrix}$

$$
P \leftarrow \begin{bmatrix} P_{0,0} & & & & & \\ P_{1,0} & P_{1,1} & & & & \\ & \dots & \dots & & & \\ & & P_{i,i-1} & P_{i,i} & & \\ & & & \dots & \dots & \\ & & & & P_{N-1,N-2} & P_{N-1,N-1} \end{bmatrix}
$$

$\mathbf{y} \leftarrow \begin{bmatrix} y^{(0)} & y^{(1)} & \dots & y^{(i)} & \dots & y^{(N-1)} \end{bmatrix}^T$

$\mathbf{a} \leftarrow \text{linear\_solve}(P, \mathbf{y})$

Algorithm	**Quadratic Spline Interpolation**
Problem Type	C1 Continuous Interpolation
Given	Sampling points $(x_i, y_i)_{0:N}$ Interpolation interval $[a, b]$
Find	Function $f \in C^1[a, b]$ $s.t. f(x_i) = y_i, i = 0, ..., N$
Algorithm Property	Iterative Solution

The corresponding Python code is shown below:

```python
def find_insert_position(arr, x):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] <= x and (mid == len(arr) - 1 or x < arr[mid + 1]):
            return mid
        elif arr[mid] > x:
            right = mid - 1
        else:
            left = mid + 1
```

```
    return -1  # bad case
def quadratic_spline_interp(xs, ys, N):
    assert xs.shape == (N,) and ys.shape == (N,)
    P, p_ = np.zeros([3*N-3, 3*N-3]), np.zeros(3*N-3)
    for i in range(N-1):
        P_ii = np.array([
            [1, xs[i], xs[i]**2],
            [1, xs[i+1], xs[i+1]**2],
            [0, -1, -2*xs[i]],
        ])
        P_i1i = np.array([
            [0, 0, 0],
            [0, 0, 0],
            [0, 1, 2*xs[i]],
        ])
        P[3*i:3*i+3, 3*i:3*i+3] = P_ii
        if i > 0:
            P[3*i:3*i+3, 3*i-3:3*i] = P_i1i
        p_[3*i:3*i+3] = np.array([ys[i], ys[i+1], 0])
    a_ = np.linalg.solve(P, p_)
    A = np.reshape(a_, [N-1, 3])
    def f_func(x):
        i = find_insert_position(xs, x)
        if i >= N-1:
            a = A[N-2, :]
        else:
            a = A[i, :]
        return a[0] + a[1] * x + a[2] * x**2
    return f_func
def Quadratic_spline_sample(xs, ys, N, M):
    assert xs.shape == (N,)
    assert len(ys.shape) == 2 and ys.shape[0] == N
    d = ys.shape[1]
    a, b = np.min(xs), np.max(xs)
    new_xs = np.linspace(a, b, num=M, endpoint=True)
    new_ys = np.zeros([M, d])
    for dd in range(d):
        f_func = quadratic_spline_interp(xs, ys[:, dd], N)
        new_ys[:, dd] = np.array(list(map(f_func, new_xs)))
    return new_ys
```


4.2.2 Cubic Spline Interpolation

For C2 continuous interpolation problems, we can extend the approach of quadratic splines by changing the continuous function in each interval to a cubic function, i.e., **cubic spline interpolation**. Specifically, we have

$$f_i(x) = a_0^{(i)} + a_1^{(i)}x + a_2^{(i)}x^2 + a_3^{(i)}x^3, x \in [x_i, x_{i+1}] \tag{I.4.4}$$

Thus, the number of unknowns to solve becomes $4N - 4$. For each interval, our constraints become second-order continuous differentiability, i.e.,

$$\begin{aligned} f_i(x_i) &= y_i, \quad i = 0, ..., N-1 \\ f_i(x_{i+1}) &= y_{i+1}, \quad i = 0, ..., N-1 \\ f_i'(x_i) &= f_{i-1}'(x_i), \quad i = 1, ..., N-1 \\ f_i''(x_i) &= f_{i-1}''(x_i), \quad i = 1, ..., N-1 \end{aligned}$$

This gives a total of $4N-6$ equations. Following the approach of quadratic splines, we can specify the first derivatives at the left and right boundaries as 0[4]

$$f_0'(x_0) = 0$$
$$f_{N-1}'(x_N) = 0$$

Substituting the cubic function into the above equations, we obtain

$$\begin{aligned} a_0^{(i)} + a_1^{(i)} x_i + a_2^{(i)} x_i^2 + a_3^{(i)} x_i^3 &= y_i \\ a_0^{(i)} + a_1^{(i)} x_{i+1} + a_2^{(i)} x_{i+1}^2 + a_3^{(i)} x_{i+1}^3 &= y_{i+1} \\ a_1^{(i-1)} + 2a_2^{(i-1)} x_i + 3a_3^{(i-1)} x_i^2 - a_1^{(i-1)} - 2a_2^{(i-1)} x_{i+1} - 3a_3^{(i)} x_i^2 &= 0 \\ 2a_2^{(i-1)} + 6a_3^{(i-1)} x_i - 2a_2^{(i-1)} - 6a_3^{(i)} x_i &= 0 \end{aligned}$$

At this point, we have the linear equation

$$\begin{bmatrix} P_{0,0} & & & & & & P_{0,N-1} \\ P_{1,0} & P_{1,1} & & & & & \\ & \dots & \dots & & & & \\ & & P_{i,i-1} & P_{i,i} & & & \\ & & & \dots & \dots & & \\ & & & & & P_{N-1,N-2} & P_{N-1,N-1} \end{bmatrix} \begin{bmatrix} a_:^{(0)} \\ a_:^{(1)} \\ \dots \\ a_:^{(i)} \\ \dots \\ a_:^{(N-1)} \end{bmatrix} = \begin{bmatrix} y^{(0)} \\ y^{(1)} \\ \dots \\ y^{(i)} \\ \dots \\ y^{(N-1)} \end{bmatrix} \tag{I.4.5}$$

where

$$\begin{aligned} a_:^{(i)} &= \begin{bmatrix} a_0^{(i)} & a_1^{(i)} & a_2^{(i)} & a_3^{(i)} \end{bmatrix}^T, \quad i = 1, ..., N-1 \\ y^{(i)} &= \begin{bmatrix} y_i & y_{i+1} & 0 & 0 \end{bmatrix}^T, \quad i = 1, ..., N-1 \\ P_{i,i} &= \begin{bmatrix} 1 & x_i & x_i^2 & x_i^3 \\ 1 & x_{i+1} & x_{i+1}^2 & x_{i+1}^3 \\ 0 & -1 & -2x_i & -3x_i^2 \\ 0 & 0 & -2 & -6x_i \end{bmatrix}, \quad i = 1, ..., N-1 \\ P_{i,i-1} &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 2x_i & 3x_i^2 \\ 0 & 0 & 2 & 6x_i \end{bmatrix}, \quad i = 1, ..., N-1 \end{aligned} \tag{I.4.6}$$

and

$$\begin{aligned} P_{0,0} &= \begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 \\ 1 & x_1 & x_1^2 & x_{0+1}^3 \\ 0 & -1 & -2x_0 & -3x_0^2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\ P_{0,N-1} &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 2x_N & 3x_N^2 \end{bmatrix} \end{aligned} \tag{I.4.7}$$

By solving the above equations, we obtain the desired $f(x)$.

[4]The choice of additional conditions here is not unique. For example, we could also set the second derivatives at the boundaries to 0 or make the first and second derivatives equal at the boundaries. In practice, the selection can be based on additional known information about the interpolation function.

Algorithm 16: Cubic Spline Interpolation

Input: Sampling points $(x_i, y_i)_{0:N}$
Output: Interpolation function $f \in C^1[a, b]$
for $i \in 1, ..., N-1$ **do**

$$a_:^{(i)}, y^{(i)} \leftarrow \begin{bmatrix} a_0^{(i)} & a_1^{(i)} & a_2^{(i)} \end{bmatrix}^T, \begin{bmatrix} y_i & y_{i+1} & 0 \end{bmatrix}$$

$$P_{i,i}, P_{i,i-1} \leftarrow \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 2x_i & 3x_i^2 \\ 0 & 0 & 2 & 6x_i \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 2x_N & 3x_N^2 \end{bmatrix}$$

$$P \leftarrow \begin{bmatrix} P_{0,0} & & & & & & P_{0,N-1} \\ P_{1,0} & P_{1,1} & & & & & \\ & \cdots & \cdots & & & & \\ & & P_{i,i-1} & P_{i,i} & & & \\ & & & \cdots & \cdots & & \\ & & & & & P_{N-1,N-2} & P_{N-1,N-1} \end{bmatrix}$$

$$\mathbf{y} \leftarrow \begin{bmatrix} y^{(0)} & y^{(1)} & \cdots & y^{(i)} & \cdots & y^{(N-1)} \end{bmatrix}^T$$

$$\mathbf{a} \leftarrow \text{linear\_solve}(P, \mathbf{y})$$

Algorithm	**Cubic Spline Interpolation**
Problem Type	C2 Continuous Interpolation
Given	Sampling points $(x_i, y_i)_{0:N}$ Interpolation interval $[a, b]$
Find	Function $f \in C^1[a, b]$ $s.t. f(x_i) = y_i, i = 0, ..., N$
Algorithm Property	Iterative Solution

The corresponding Python code is shown below.

```python
def cubic_spline_interp(xs, ys, N):
    assert xs.shape == (N,) and ys.shape == (N,)
    P, p_ = np.zeros([4*N-4, 4*N-4]), np.zeros(4*N-4)
    for i in range(N-1):
        P_ii = np.array([
            [1, xs[i], xs[i]**2, xs[i]**3],
            [1, xs[i+1], xs[i+1]**2, xs[i+1]**3],
            [0, -1, -2*xs[i], -3*xs[i]**2],
            [0, 0, -2, -6*xs[i]],
        ])
        P_i1i = np.array([
            [0, 0, 0, 0],
            [0, 0, 0, 0],
            [0, 1, 2*xs[i], 3*xs[i]**2],
            [0, 0, 2, 6*xs[i]],
        ])
        if i == 0:
            P[:3, :4] = P_ii[:3, :] # first 3 lines
            P[3, -4:] = np.array([0, 1, 2*xs[-1], 3*xs[-1]**2])
        else:
            P[4*i:4*i+4, 4*i:4*i+4] = P_ii
            P[4*i:4*i+4, 4*i-4:4*i] = P_i1i
        p_[4*i:4*i+4] = np.array([ys[i], ys[i+1], 0, 0])
    a_ = np.linalg.solve(P, p_)
    A = np.reshape(a_, [N-1, 4])
```

```
    def f_func(x):
        i = find_insert_position(xs, x)
        if i >= N-1:
            a = A[N-2, :]
        else:
            a = A[i, :]
        return a[0] + a[1] * x + a[2] * x**2 + a[3] * x**3
    return f_func
def Cubic_spline_sample(xs, ys, N, M):
    assert xs.shape == (N,)
    assert len(ys.shape) == 2 and ys.shape[0] == N
    d = ys.shape[1]
    a, b = np.min(xs), np.max(xs)
    new_xs = np.linspace(a, b, num=M, endpoint=True)
    new_ys = np.zeros([M, d])
    for dd in range(d):
        f_func = cubic_spline_interp(xs, ys[:, dd], N)
        new_ys[:, dd] = np.array(list(map(f_func, new_xs)))
    return new_ys
```

(Reference: Tsinghua University "Numerical Analysis" course materials)

Part II
Fundamentals of Robotics

5 Basic Concepts

In this section, we will introduce the most fundamental concepts and theories in **robotics**: kinematics, dynamics, Jacobian, etc. These theories form the foundation for all subsequent robot control, learning algorithms, and more.

5.1 Joints and Forward/Inverse Kinematics

Robots can be divided into two categories: fixed-base robots and mobile robots. Regardless of the type, a robot (or robotic arm) can be modeled as a series of rigid bodies connected by joints. We send commands to different joint actuators to control the entire robot (especially its end-effector) to perform various simple or complex motions. In this section, our main focus is on **fixed-base robots**[5].

In mathematical models, joints are classified into revolute joints and prismatic joints, which enable relative rotational and translational motions, respectively. Revolute joints can generate torque and angular velocity, while prismatic joints can generate force and linear velocity. **In this book, unless otherwise specified, the joints we discuss are revolute joints**.

Currently, the most commonly used joint actuation mechanism in various robots is the permanent magnet synchronous motor. By providing voltage/current commands to the joints and incorporating reduction mechanisms, precise control of torque and rotation angles can be achieved. For details, refer to Section 15.1. Besides motors, other joint actuation methods include hydraulic and cable-driven mechanisms.

Both revolute and prismatic joints can be parameterized. For details, refer to Section 6.1. Joint parameters are divided into fixed parameters and variable parameters. Fixed parameters are uncontrollable, while variable parameters correspond to the **degrees of freedom** (DOF). DOF is an important parameter for describing a robotic system. In the absence of closed chains, the DOF equals the number of actuated joints.

The variable parameters of a robot form the **joint variables** q (also called the **joint vector**). The space composed of joint variables is called the **joint space**, which consists of joint angles (for revolute joints). The joint space is crucial for robot control.

In addition to joint angles, we are also highly concerned with the robot's end-effector. The end-effector is typically equipped with tools or grippers, collectively referred to as the **actuator**. The end-effector's pose is denoted as x_e, which includes 3D position and 3D orientation. The space where the end-effector pose resides is called the **task space**. Compared to the joint space, the task space is directly related to the robot's tasks and user requirements.

Given the joint parameters (fixed and variable), we can derive the expression $x_e = k(q)$ based on spatial geometric relationships. This is the robot's **forward kinematics**. **Forward kinematics converts joint space positions into task space positions**.

Conversely, if we compute the joint variables q from the end-effector pose x_e, it is called the robot's **inverse kinematics**. **Inverse kinematics converts task space positions into joint space positions**. For detailed explanations of forward and inverse kinematics, refer to Chapter 6.

The key differences between inverse kinematics and forward kinematics are: Inverse kinematics may have multiple or even infinite solutions, meaning there are different ways to achieve a given requirement, which reflects the robot's redundancy; inverse kinematics may encounter singularities, where a requirement lies at the robot's limits; and inverse kinematics may have no solution, indicating that the requirement exceeds the robot's capabilities. Properly addressing these issues is crucial in practical applications.

5.2 Jacobian, Dynamics, and Control

The forward and inverse kinematics discussed above describe the positional relationship between joint space and task space, while the **Jacobian matrix** describes the velocity relationship between joint space and task space, i.e., the differential relationship. Specifically, the **forward Jacobian** relates v_e and $\dot{q}$ as $v_e = J(q)\dot{q}$.

[5] With minor modifications, these conclusions can also be applied to mobile robots without fixed bases, such as bipedal robots

The Jacobian matrix may become singular (i.e., rank-deficient) as q changes. Studying the singularity of the Jacobian is highly important.

Additionally, the Jacobian matrix is related to the static force relationship of a robotic arm in a weightless state. Under this assumption, the Jacobian transpose can transform task space forces into joint space torques without solving complex inverse kinematics. For details, refer to Chapter 7.

Dynamics refers to the relationship between the robot's task space position/velocity/acceleration and the joint forces/torques. Among these, the **inverse dynamics problem** involves calculating the required joint forces/torques based on the task space position/velocity/acceleration and control objectives. This is crucial for robot control and is typically solved from the end-effector backward. Additionally, simulators need to determine the resulting velocity/acceleration after applying forces/torques, which is the **forward dynamics problem**. Forward dynamics is generally solved from the base to the end-effector. For details, refer to Chapter 8.

Robot (or robotic arm) control is a critical issue and a classic application scenario of control theory. Open-loop control involves solving inverse dynamics and directly sending commands to the motors, but this can lead to accuracy issues. Proper closed-loop control considers user commands, environmental influences, modeling accuracy, and other factors to achieve precise tracking and regulation (stabilization). Robot control is covered in Part IV of this book.

5.3 Trajectory Planning and Interpolation

Robot control requires joint space or task space commands, typically position commands. Where do these commands come from?

For simpler robots, humans can specify motion trajectories, such as end-effector points or a series of discrete trajectory points x_d. Many industrial robotic arms working in fixed environments fall into this category.

For more autonomous robots, collision avoidance and other issues must be considered during trajectory planning. This process is called **trajectory planning**. For details, refer to Chapter 9. Many current autonomous vehicles and industrial robotic arms used in complex environments belong to this category.

For even more autonomous robots, actions must be planned autonomously based on human needs and multimodal inputs (e.g., images, sounds, or text) and sensor data. The currently popular VLA (Vision-Language-Action) large models represent such technology. These robots can exhibit a certain level of intelligence and represent the future direction of robotics. Robot learning will be discussed in later chapters.

Regardless of the trajectory's source, the initially given trajectory commands are typically discrete (with low frequency). Therefore, a process is needed to interpolate the initially discrete trajectory into a smoother trajectory while satisfying feasibility constraints (e.g., continuous second derivatives). This process is called **trajectory interpolation**. Trajectory interpolation can be performed in either task space or joint space. Practical robotic systems include trajectory interpolation modules as needed. For details on interpolation, refer to Chapter ?.

6 Forward and Inverse Kinematics

6.1 D-H Parameters

As discussed in the previous chapter, a robot (manipulator) can be modeled as a series of rigid bodies connected by joints, forming a link system. For simplicity, the following discussion focuses only on single-chain revolute joint robots with one fixed end.

The link system always has one end fixed to a relatively stable platform, referred to as the **base**, which is also labeled as link L_0. The other end of the link system can move freely within certain limits and is called the **end-effector**. Between the base and the end-effector, there are n joints, sequentially numbered from 1 to N from the base to the end-effector. Joint J_n connects two links, L_{n-1} and L_n, where L_{n-1} is closer to the base and L_n is closer to the end-effector. Correspondingly, **link L_n connects joint J_n and joint J_{n+1}**.

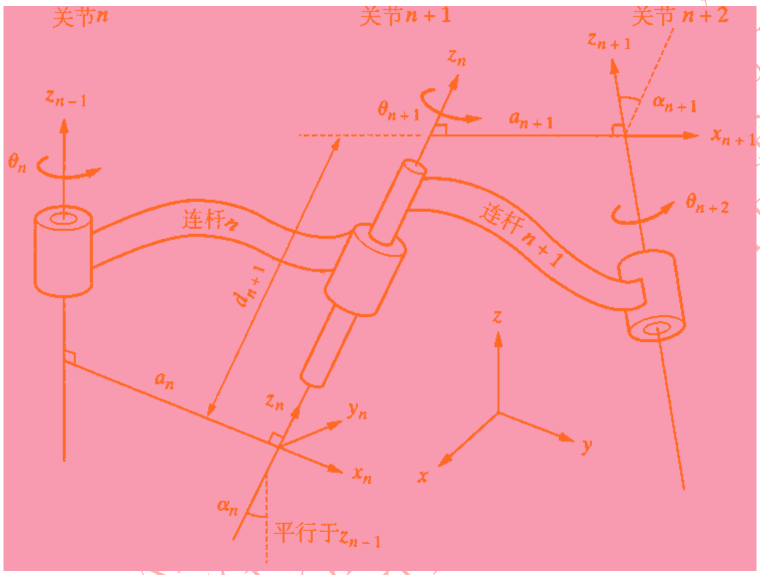

Figure II.6.1: Definition of D-H Parameters

For each link L_n, we can define two **link coordinate frames**: the n' frame and the n frame. The n' frame precedes the n frame, and both are rigidly attached to L_n. The n' frame corresponds to joint J_n, while the n frame corresponds to joint J_{n+1}.

The z-axis of each coordinate frame is defined by the joint rotation axis. For revolute joints, the rotation axis of each joint is fixed. The rotation axis of joint J_{n+1} is the z-axis of both the n frame and the $n+1'$ frame. Note that z_n is not the rotation axis of J_n but rather the rotation axis of J_{n+1}.

The x-axis of each coordinate frame is defined by the common perpendicular of the link. For link L_n, the z-axes of joint J_n and joint J_{n+1} are two directed lines in space. Excluding the case where the two lines coincide, their relative positional relationship can be one of the following three possibilities: parallel, intersecting, or skew. For non-parallel cases, there always exists a unique line that is perpendicular to and intersects both the z_n and z_{n+1} axes, called the common perpendicular of link n. The two intersection points are defined as O'_n and O_{n+1}. If the two axes are parallel, the intermediate joint does not provide additional degrees of freedom and is considered a redundant joint, which can be skipped.

If the z_n and z_{n+1} axes are skew, the direction from O'_n to O_{n+1} is designated as the x-axis direction of frames n and $n+1'$, i.e., the x_n and $x_{n+1'}$ directions. If the z_n and z_{n+1} axes intersect, O_n and $O_{n+1'}$ coincide, and any common perpendicular direction can be chosen as the x_n and $x_{n+1'}$ directions. Subsequently, the y-axis of each frame can be defined using the right-hand rule. Thus, for each link L_n, the origins and axes of the n' and n frames are fully defined.

Based on the above coordinate frame definitions, for each link L_n, we define the following four parameters: the joint offset d_n, the link length a_n, the link twist α_n, and the joint angle θ_n. These are specifically defined as:

- Joint offset d_n: The distance from O_{n-1} to $O_{n'}$.

- Joint angle θ_n: The angle between the x_{n-1} and x_n axes.
- Link length a_n: The distance from $O_{n'}$ to O_n.
- Link twist α_n: The angle between the z_{n-1} and z_n axes.

The above set of parameters is called the **D-H parameters** of the robot system.

This completes the parameterization of the link system. For a link system composed of revolute joints, d_n, a_n, and α_n are fixed values obtained through calibration during manufacturing. θ_n is a variable value, also referred to as the joint angle or the generalized joint space position. All θ_n values together form the joint vector q, i.e.,

$$q := \begin{bmatrix} \theta_1 \\ \theta_2 \\ \dots \\ \theta_N \end{bmatrix} \tag{II.6.1}$$

6.2 Forward Kinematics

Using the above parameter definitions, we can express the pose of the end-effector relative to the base, i.e., the homogeneous transformation T_N^0 from frame N to frame 0. Typically, joint frame 0 is defined as the base frame b, and joint frame N is defined as the end-effector frame e. Therefore, the pose of the end-effector relative to the base can also be written as T_e^b.

As described in Chapter I.1, the homogeneous matrix can be divided into a rotation matrix and a translation vector:

$$T_e^b = \begin{bmatrix} R_e^b & t_{be}^b \\ 0 & 1 \end{bmatrix} \tag{II.6.2}$$

The rotation matrix R_e^b can be replaced by an equivalent Euler angle or quaternion representation. We can combine the translation vector with one of the rotation representations to compactly represent the pose of the end-effector relative to the base, i.e., the end-effector state $\mathbf{x}_e$. That is,

$$\mathbf{x}_e = \begin{bmatrix} t_{be}^b \\ q_e^b \end{bmatrix} \text{ or } \begin{bmatrix} t_{be}^b \\ \vartheta \end{bmatrix} \tag{II.6.3}$$

Given all fixed parameters, the problem of solving for the end-effector state $\mathbf{x}_e$ based on the joint vector q is called **forward kinematics**.

Problem	**Forward Kinematics**
Description	Given fixed parameters, solve for the end-effector state based on the joint vector.
Known	Joint vector q
Solve for	End-effector state $\mathbf{x}_e$

According to the above definitions, forward kinematics can be solved analytically. Since the transformation from T_e^b to $\mathbf{x}_e$ is unique, we only consider solving for T_e^b, or T_N^0.

Specifically, let us first consider T_n^{n-1}. It is straightforward to see that:

$$T_n^{n-1} = T_{n'}^{n-1} T_n^{n'}$$

Based on the definitions of a_n and α_n, we have:

$$T_n^{n'} = \begin{bmatrix} 1 & 0 & 0 & a_n \\ 0 & C_{\alpha_n} & -S_{\alpha_n} & 0 \\ 0 & S_{\alpha_n} & C_{\alpha_n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Based on the definitions of d_n and θ_n, we have:

$$T_{n'}^{n-1}(\theta_n) = \begin{bmatrix} C_{\theta_n} & -S_{\theta_n} & 0 & 0 \\ -S_{\theta_n} & C_{\theta_n} & 0 & 0 \\ 0 & 0 & 1 & d_n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here, for simplicity, we abbreviate $\cos(\alpha)$ and $\sin(\alpha)$ as C_α and S_α, respectively. Thus, we obtain the link transformation formula:

$$T_n^{n-1}(\theta_n) = \begin{bmatrix} C_{\theta_n} & -S_{\theta_n} C_{\alpha_n} & S_{\theta_n} S_{\alpha_n} & a_n C_{\theta_n} \\ S_{\theta_n} & C_{\theta_n} C_{\alpha_n} & -C_{\theta_n} S_{\alpha_n} & a_n S_{\alpha_n} \\ 0 & S_{\alpha_n} & C_{\alpha_n} & d_n \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{II.6.4}$$

Therefore, the forward kinematics formula is:

$$T_N^0(q) = T_0^1(\theta_1) T_2^1(\theta_2) ... T_N^{N-1}(\theta_N) \tag{II.6.5}$$

We have:

$$x_e := k(q) = (T_N^0(q))_{1:3,4} \tag{II.6.6}$$

Summarizing the above formulas, we obtain the analytical algorithm for forward kinematics.

Algorithm 17: Kinematics Analytical Solution (robot_fk)

Input: D-H parameters $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: Joint vector $q = \theta_{1:N}$
Output: End-effector pose x_e
$T_0^0 \leftarrow I_{4\times4}$
for $n \in 1, \ldots, N$ **do**
    $T_n^{n-1} \leftarrow \begin{bmatrix} C_{\theta_n} & -S_{\theta_n} C_{\alpha_n} & S_{\theta_n} S_{\alpha_n} & a_n C_{\theta_n} \\ S_{\theta_n} & C_{\theta_n} C_{\alpha_n} & -C_{\theta_n} S_{\alpha_n} & a_n S_{\alpha_n} \\ 0 & S_{\alpha_n} & C_{\alpha_n} & d_n \\ 0 & 0 & 0 & 1 \end{bmatrix}$
    $T_n^0 \leftarrow T_{n-1}^0 T_n^{n-1}$
$R_0^N, t_{0N}^N \leftarrow (T_n^0)^{-1}$
$\vartheta_e \leftarrow$ rotmat_to_eular(R_0^N)
$x_e \leftarrow [(t_{0N}^N)^T, (\vartheta_e)^T]^T$

Algorithm	**Kinematics Analytical Solution**
Problem Type	Forward Kinematics Solution
Given	D-H parameters d_n, a_n, α_n, joint vector q
Find	End-effector pose x_e
Algorithm Property	Analytical Solution

The corresponding Python code is shown below:

```python
def calc_T_n_to_last(q_n, d_n, a_n, alpha_n):
    s_q, c_q = np.sin(q_n), np.cos(q_n)
    s_a, c_a = np.sin(alpha_n), np.cos(alpha_n)
    T_n_to_last = np.ndarray([
        [c_q, -s_q * c_a, s_q * s_a, a_n * c_q],
        [s_q, c_q * c_a, -c_q * s_a, a_n * s_a],
        [0, s_a, c_a, d_n],
        [0, 0, 0, 1],
    ])
def robot_fk(q, d, a, alpha, N=6):
    assert q.shape == (N+1,) and d.shape == (N+1,)
    assert a.shape == (N+1,) and alpha.shape == (N+1,)
    T_n_to_base = np.eye(4)
    for n in range(1, N+1):
        T_n_to_base = T_n_to_base @ calc_T_n_to_last(
            q[n], d[n], a[n], alpha[n]
        )
    T_base_to_N = np.linalg.inv(T_n_to_base)
    R_0_to_N, t_0N = T_base_to_N[:3, :3], T_base_to_N[:3,
    ↪ 3]
    eular_e = rot_to_eular(R_0_to_N)
    return np.row_stack([t_0N, eular_e])
```

In addition to the end-effector position x_e, the centroid position $p_n(q) = p^b_{b,l_n}(q)$ of each link also has a similar forward kinematics relationship:

$$p_n(q) = p^b_{b,l_n}(q) = T^0_n(q)p^n_{l_n} \tag{II.6.7}$$

where $p^n_{l_n}$ represents the position of link n's **centroid** in coordinate frame n, which is a constant.

6.3 Inverse Kinematics

In contrast to forward kinematics, inverse kinematics refers to solving for the joint vector q given the end-effector state $\mathbf{x}_e$ while known fixed parameters.

Problem	**Inverse Kinematics Problem**
Problem Description	Solve for joint vector given end-effector state with known fixed parameters
Given	End-effector state $\mathbf{x}_e$
Find	Joint vector q

Compared to forward kinematics, inverse kinematics has significant differences in solution methods. First, while forward kinematics always has a unique solution, inverse kinematics does not. When the robot configuration lies in redundant space, **multiple solutions** may occur in inverse kinematics. When operating near workspace boundaries or at singular points, **singular solutions** may appear. Handling multiple solutions and identifying singular solutions are important challenges in inverse kinematics algorithms.

Second, there is no general analytical solution for inverse kinematics. In practice, one either derives specific **analytical methods** for particular robots by solving equations, or uses **numerical methods** for solution. The basic principle of numerical methods is to transform the root-finding problem of kinematic equations into an optimization problem using the Jacobian matrix, then solve it iteratively. We will detail the numerical inverse kinematics solution method in Section 7.3.

(Reference materials: Tsinghua University "Intelligent Robotics" course materials, "Robotics: Modelling, Planning and Control")

7 Differential Kinematics

7.1 Jacobian Matrix

7.1.1 Geometric and Analytical Jacobian

Forward and inverse kinematics describe the relationship between joint positions and end-effector positions. But is there a relationship between joint velocities and end-effector velocities? This is where the Jacobian matrix comes into play.

For the forward kinematics equation $x_e = k(q)$, considering the change over an infinitesimal time δt, we have

$$\delta x_e = \frac{\partial k(q)}{\partial q} \delta q$$

Dividing both sides by the small time increment δt, we obtain

$$\dot{x}_e = \frac{\partial k(q)}{\partial q} \dot{q}$$

That is,

$$v_e = \frac{\partial k(q)}{\partial q} \dot{q}$$

Here, v_e is the generalized end-effector velocity, which consists of two parts: the derivative of position with respect to time and the derivative of orientation with respect to time. We define two different forms of v_e. If the angular part of v_e is the angular velocity ω_e, we denote it as v_{ew}. If the angular part of v_e is the derivative of quaternions or Euler angles, we denote it as v_{ea}.

Based on these two notations, we can define two different Jacobian matrices. The Jacobian relating v_{ew} to $\dot{q}$ is denoted as $J_w(q)$, called the **geometric Jacobian**. The Jacobian relating v_{ea} to $\dot{q}$ is denoted as $J_a(q)$, called the **analytical Jacobian**. We have

$$\begin{aligned} v_{ew} &= J_w(q)\dot{q} \\ v_{ea} &= J_a(q)\dot{q} \end{aligned} \tag{II.7.1}$$

Equation II.7.1 transforms the robot's joint space velocity into operational space velocity. Since it aligns with the direction of forward kinematics, it is also called the **forward Jacobian**.

Because **the derivatives of Euler angles and quaternions are not equal to angular velocity**, the geometric Jacobian and analytical Jacobian are generally not the same. Their mathematical relationship will be derived in the next section.

Among the two Jacobians, the geometric Jacobian is most commonly used in robot dynamics equations (as derived later), while the analytical Jacobian can be used for velocity observation in operational space control.

7.1.2 Jacobian and Force

Sometimes, we consider the static force relationship of a robotic arm in a **weightless state**. Note that in a **weightless state** and **ignoring all non-end-effector resistances**, the total power of all robot joints equals the power at the end-effector.

Let the generalized force at the end-effector be F_e (a 3D wrench combining torque and linear force), and the vector of joint torques be τ. Based on the power relationship above, we have

$$v_e^T F_e = \dot{q}^T \tau \tag{II.7.2}$$

Substituting Equation II.7.1, we get

$$\dot{q}^T J_w(q)^T F_e = \dot{q}^T \tau$$

That is,

$$\tau = J_w(q)^T F_e \tag{II.7.3}$$

In other words, **in a weightless state, the transpose of the Jacobian can directly convert operational space end-effector forces into joint torques**. For force-controlled applications (see Chapter 4), this relationship is crucial, as it avoids the need to compute complex inverse dynamics.

7.1.3 Inverse Jacobian

For a six-degree-of-freedom robotic arm, the forward Jacobian matrix is square. Assuming this Jacobian is invertible, we can derive the **transformation from operational space velocity to joint space velocity**. (The choice of Jacobian depends on the form of the given operational space velocity.)

$$\dot{q} = J(q)^{-1} v_e \tag{II.7.4}$$

Equation II.7.4 is also called the **inverse Jacobian**. It converts operational space velocity into joint space velocity. The inverse Jacobian is essential for operational space control, as discussed in Part IV.

7.1.4 Second-Order Differential Kinematics

Furthermore, by differentiating Equation II.7.1, we can derive the second-order differential kinematics relationship:

$$\ddot{x}_{ea} = J_a(q)\ddot{q} + \dot{J}_a(q)\dot{q} \tag{II.7.5}$$

7.2 Jacobian Computation

The Jacobian matrix can be computed analytically.

7.2.1 End-Effector Velocity and Angular Velocity

Suppose we consider a robot with only revolute joints. First, we examine the motion of link n relative to link $n-1$. Let the position of the origin of frame n relative to the base frame be p_n, its velocity be $\dot{p}_n$, and its angular velocity be ω_i, i.e.,

$$\omega_n = \omega_{b,n}^b$$
$$\dot{p}_n = \dot{p}_{bn}^b$$

First, consider the relative angular velocity of link n, whose magnitude is the angular velocity of joint n and whose direction is along the z-axis of frame $n-1$:

$$\omega_{n-1,n}^b = \dot{\theta}_n z_{n-1}^b$$

The relative linear velocity with respect to the previous joint is

$$\dot{p}_{n-1,n}^b = \omega_{n-1,n}^b \times p_{n-1,n}^b$$

Thus, we have the recursive relationships for p_i and ω_i:

$$\begin{aligned} \omega_n &= \omega_{n-1} + \dot{\theta}_n z_{n-1}^b \\ \dot{p}_n &= \dot{p}_{n-1} + \omega_{n-1,n}^b \times p_{n-1,n}^b \end{aligned} \tag{II.7.6}$$

For the end-effector state, since

$$\omega_e = \omega_N$$
$$\dot{p}_e = \dot{p}_N$$

we have

$$\omega_e = \omega_N = \sum_{i=1}^{N} \dot{\theta}_n z_{n-1}^b$$

and

$$\dot{p}_e = \dot{p}_N = \sum_{i=1}^{N} \dot{\theta}_n z_{n-1}^b \times p_{n-1,n}^b$$

7.2.2 Derivation of the Geometric Jacobian

For the geometric Jacobian, the generalized velocity consists of linear and angular velocities:

$$v_{ew} = \begin{bmatrix} \dot{p}_e \\ \omega_e \end{bmatrix}$$

Thus, $J_w(q)$ also consists of two parts:

$$\begin{bmatrix} \dot{p}_e \\ \omega_e \end{bmatrix} = v_{ew} = J_w(q)\dot{q} = \begin{bmatrix} J_p(q) \\ J_\theta(q) \end{bmatrix} \dot{q}$$

Here, $\dot{q}$ represents the time derivatives of the joint angles $\dot{\theta}_n$, so we can write

$$J_w(q) = \begin{bmatrix} J_p(q) \\ J_\theta(q) \end{bmatrix} = \begin{bmatrix} z_0^b \times p_{0,N}^b & z_1^b \times p_{1,N}^b & \cdots & z_{N-1}^b \times p_{N-1,N}^b \\ z_0^b & z_1^b & \cdots & z_{N-1}^b \end{bmatrix} \tag{II.7.7}$$

Note that each $p_{n-1,N}^b$ and z_{n-1}^b is a function of q. Their expressions are:

$$p_{n-1,N}^b(q) = R_{n-1}^0(q) t_N^{n-1}(q)$$

$$z_{n-1}^b(q) = R_{n-1}^0(q) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

7.2.3 Conversion Between the Two Jacobians

For the analytical Jacobian, we represent the end-effector orientation using Euler angles ϑ_e, giving

$$v_{ea} = \begin{bmatrix} \dot{p}_e \\ \dot{\vartheta}_e \end{bmatrix}$$

Let T_ϑ denote the transformation matrix from Euler angle derivatives $\dot{\vartheta}$ to angular velocity ω, i.e.,

$$\omega_e = T_\vartheta \dot{\vartheta}_e \tag{II.7.8}$$

Then we have

$$v_{ea} = \begin{bmatrix} \dot{p}_e \\ T_\vartheta^{-1} \omega_e \end{bmatrix} = T_w^a v_{ew}$$

where T_w^a represents the transformation matrix from J_w to J_a

$$T_w^a = \begin{bmatrix} I_{3\times 3} & 0_{3\times 3} \\ 0_{3\times 3} & T_\vartheta^{-1} \end{bmatrix} \in \mathbb{R}^{6\times 6} \tag{II.7.9}$$

Therefore

$$J_a(q) = T_w^a J_w(q) = \begin{bmatrix} J_p(q) \\ T_\vartheta^{-1}(\vartheta_e) J_\theta(q) \end{bmatrix} \tag{II.7.10}$$

The transformation matrix $T_\vartheta \in \mathbb{R}^{3\times 3}$ does not have a unified form. Different Euler angle definitions lead to different methods for deriving this matrix. Under ZXY Euler angles, its expression is

$$T_\vartheta = R_y(\vartheta_y)\left(R_x(\vartheta_x)\begin{bmatrix}0\\0\\1\end{bmatrix}+\begin{bmatrix}1\\0\\0\end{bmatrix}\right)+\begin{bmatrix}0\\1\\0\end{bmatrix} \tag{II.7.11}$$

Note: Euler angle representations suffer from the gimbal lock problem. Near singularities, the matrix T_ϑ becomes non-invertible, causing the analytical Jacobian to become singular.

Summarizing the above formulas, we obtain the analytical algorithm for computing the analytical Jacobian

Algorithm 18: Analytical Jacobian Solution (robot_jacobian_a)

Input: D-H parameters $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: Joint vector $q = \theta_{1:N}$, Euler angles ϑ
Output: Analytical Jacobian matrix J_a
$T_0^0, T_N^N \leftarrow I_{4\times 4}, I_{4\times 4}$
for $n \in 1, \ldots, N$ **do**
  $R_{n-1}^b \leftarrow (T_{n-1}^0)_{1:3,1:3}$
  $z_{n-1}^b \leftarrow (T_{n-1}^0)_{1:3,3}$
  $T_n^{n-1} \leftarrow T(\theta_n, \alpha_n, a|n, d_n)$
  $T_n^0 \leftarrow T_{n-1}^0 T_n^{n-1}$
for $n \in N, \ldots, 1$ **do**
  $T_N^{n-1} \leftarrow T_n^{n-1} T_N^n$
  $t_N^{n-1} \leftarrow (T_N^{n-1})_{1:3,4}$
  $p_{n-1,N}^b \leftarrow R_{n-1}^0 t_N^{n-1}$
for $n \in 1, \ldots, N$ **do**
  $(J_w)_{:,n} \leftarrow [(z_{n-1}^b \times p_{n-1,N}^b)^T \quad (z_{n-1}^b)^T]^T$
$T_\vartheta \leftarrow$ eular_diff_to_w(ϑ)
$J_a \leftarrow \text{diag}\{I_{3\times 3}, T_\vartheta^{-1}\} J_w$

Algorithm	**Analytical Jacobian Solution**
Problem Type	Analytical Jacobian Computation
Given	D-H parameters d_n, a_n, α_n Joint vector q, Euler angles ϑ
Output	Analytical Jacobian matrix J_a
Algorithm Property	Analytical Solution

The corresponding Python code is shown below

```python
def robot_jacobian_a(q, d, a, alpha, eular_e, N=6):
    assert q.shape == (N+1,) and d.shape == (N+1,)
    assert a.shape == (N+1,) and alpha.shape == (N+1,)
    z_n_bs = np.zeros([N, 3])
    T_n_to_lasts = np.zeros([N+1, 4, 4])
    T_n_to_base = np.eye(4)
    for n in range(1, N+1):
        z_n_bs[n-1] = T_n_to_base[:3, 2]
        T_n_to_last = calc_T_n_to_last(
            q[n], d[n], a[n], alpha[n]
        )
        T_n_to_lasts[n, :, :] = T_n_to_last
        T_n_to_base = T_n_to_base @ T_n_to_last
    T_N_to_ns = np.zeros([N+1, 4, 4])
```

```
    T_N_to_ns[N] = np.eye(4)
    p_narm_in_Bs = np.zeros([N, 3])
    for n in range(N, 0, -1):
        T_N_to_ns[n-1] = T_n_to_lasts[n] @ T_N_to_ns[n]
        t_N_in_nlast = T_N_to_ns[n-1, :3, 3]
        R_nlast_to_b = T_n_to_base[n-1, :3, :3]
        p_narm_in_Bs[n-1] = R_nlast_to_b @ t_N_in_nlast
    J_w = np.zeros(6, N)
    for n in range(N):
        J_w[:3, n-1] = np.cross(z_n_bs[n-1], p_narm_in_Bs[n-1])
        J_w[3:, n-1] = z_n_bs[n-1]
    T_theta = eular_diff_to_w(eular_e)
    T_w_to_a = np.diag([np.eye(3), np.linalg.inv(T_theta)])
    return T_w_to_a @ J_w
```

7.2.4 Derivation of Centroid Jacobian

As mentioned in the previous section, each link's centroid also has its own kinematic relationship $p_n(q)$. Therefore, we can compute the partial derivatives of each link's centroid velocity and angular velocity with respect to joint angular velocities, i.e., the centroid Jacobian matrix

$$J_{w,n}(q) = \begin{bmatrix} J_{p,n}(q) \\ J_{\theta,n}(q) \end{bmatrix} = \begin{bmatrix} z_0^b \times p_{0,l_n}^b & \cdots & z_{n-1}^b \times p_{n-1,l_n}^b & 0_{3\times1} & \cdots & 0_{3\times1} \\ z_0^b & \cdots & z_{n-1}^b & 0_{3\times1} & \cdots & 0_{3\times1} \end{bmatrix} \tag{II.7.12}$$

where

$$p_{i,l_n}^b(q) = R_i^0(q)(R_n^i(q)p_{l_n}^n + t_n^i(q))$$

Here, $p_{l_n}^n$ represents the position of link n's **centroid** in coordinate frame n, which is a constant. Additionally, we have

$$J_{p,n}(q) = \frac{\partial p_n(q)}{\partial q} \tag{II.7.13}$$

Thus, we obtain

$$\begin{aligned} \dot{p}_n &= J_{p,n}(q)\dot{q} \\ \omega_n &= J_{\theta,n}(q)\dot{q} \end{aligned} \tag{II.7.14}$$

Summarizing the above formulas, we derive the analytical algorithm for the centroid Jacobian

Algorithm 19: Centroid Jacobian Analytical Solution (robot_j_centroid)

Input: D-H parameters $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: Joint vector $q = \theta_{1:N}$, link centroid positions $p^n_{l_n}$
Output: Centroid Jacobian matrix $J_{w,n}, n = 1, ..., N$
$T^0_0, T^N_N \leftarrow I_{4\times4}, I_{4\times4}$
for $n \in 1, \dots, N$ **do**
  $T^{n-1}_n \leftarrow T(\theta_n, \alpha_n, a_n, d_n)$
  $T^0_n \leftarrow T^0_{n-1} T^{n-1}_n$
  $R^0_n \leftarrow (T^0_n)_{1:3,1:3}$
  $z^b_n \leftarrow R^0_n [0 \quad 0 \quad 1]^T$
  $T^n_n \leftarrow I_{4\times4}$
  for $m \in n-1, \dots, 0$ **do**
    $T^m_n \leftarrow T^m_{m+1} T^{m+1}_n$
    $R^m_n \leftarrow (T^m_n)_{1:3,1:3}$
    $t^m_n \leftarrow (T^m_n)_{1:3,4}$
  for $m \in 0, \dots, n-1$ **do**
    $p^b_{m,l_n} \leftarrow R^0_m (R^m_n p^n_{l_n} + t^m_n)$
    $(J_{w,n})_{:,m+1} \leftarrow [(z^b_m \times p^b_{m,l_n})^T \quad (z^b_m)^T]^T$
  for $m \in n+1, \dots, N$ **do**
    $(J_{w,n})_{:,m} \leftarrow 0_{6\times1}$

Algorithm	**Centroid Jacobian Analytical Solution**
Problem Type	Centroid Jacobian Computation
Given	D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$ Joint vector $q = \theta_{1:N}$ Link centroid positions $p^n_{l_n}, n = 1, ..., N$
Solve for	Centroid Jacobian matrix $J_{w,n}, n = 1, ..., N$
Algorithm Property	Analytical Solution

The corresponding Python code is shown below

```python
def robot_j_centroid(q, d, a, alpha, p_cents, N=6):
    assert q.shape == (N+1,) and d.shape == (N+1,)
    assert a.shape == (N+1,) and alpha.shape == (N+1,)
    assert p_cents.shape == (N+1, 3)
    T_n_to_bases = np.zeros([N+1, 4, 4])
    T_n_to_bases[0] = np.eye(4)
    z_n_bs = np.zeros([N+1, 3])
    T_n_to_lasts = np.zeros([N+1, 4, 4])
    J_w_ns = np.zeros([N+1, 6, N])
    for n in range(1, N+1):
        T_n_to_lasts[n] = calc_T_n_to_last(
            q[n], d[n], a[n], alpha[n]
        )
        T_n_to_bases[n] = T_n_to_bases[n-1] @ T_n_to_lasts[n]
        z_n_bs[n] = T_n_to_bases[n, :3, 2]
        T_n_to_ms = np.zeros([n+1, 4, 4])
        T_n_to_ms[n] = np.eye(4)
        for m in range(n-1, 0, -1):
            T_n_to_ms[m] = T_n_to_lasts[m+1] @ T_n_to_ms[m+1]
        for m in range(n):
            p_narm_in_m = T_n_to_ms[m, :3, :3] @ p_cents[n] + T_n_to_ms[m, :3, 3]
            p_narm_in_b = T_n_to_bases[m] @ p_narm_in_m
```

```
            J_w_ns[n, :3, m] = np.cross(z_n_bs[m], p_narm_in_b)
            J_w_ns[n, 3:, m] = z_n_bs[m]
    return J_w_ns
```

7.3 Jacobian-Based IK

In the previous chapter, we mentioned that the Jacobian can also be used for solving robot inverse kinematics, i.e., the numerical solution method for IK. Specifically, we need to solve the kinematic equation

$$x_e = k(q)$$

where x_e represents the given end-effector pose. In fact, any equation solving can be equivalently formulated as a least squares optimization problem

$$\min_q \mathbf{e}^T H \mathbf{e} = (x_e - k(q))^T H (x_e - k(q)) \tag{II.7.15}$$

where $H \in \mathbb{R}^{6\times 6}$ is a symmetric positive definite matrix. Under the condition that the inverse kinematics has a solution, the optimal solution q^* of the optimization problem is the solution to the kinematic equation.

Through this transformation, we can use the nonlinear least squares optimization algorithms introduced in Chapter 3 to solve the inverse kinematics. These optimization algorithms all require the derivative of the error $\mathbf{e}$ with respect to the optimization variables. Note that if we assume the given pose is in Euler angle form, this derivative is exactly the negative of the analytical Jacobian matrix

$$\frac{\partial \mathbf{e}}{\partial q} = -J_a(q) \tag{II.7.16}$$

Therefore, given an initial value, we can use least squares algorithms to solve the IK. For example, we can use the gradient descent method (also known as the steepest descent method) for solving. In some other literature, this method is also referred to as the **Jacobian transpose method**.

Algorithm	**Gradient Descent IK**
Problem Type	Inverse Kinematics Solving
Known	End-effector state $\mathbf{x}_e$
To Find	Joint vector q
Algorithm Property	Iterative Solution

Algorithm 20: Gradient Descent IK

Input: D-H parameters $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: End-effector pose x_e, initial joint value q_0
Parameter: Threshold ϵ, step size α_p, symmetric matrix H
Output: Optimal solution q^*

$q_k \leftarrow q_0$
while *True* **do**
  $J_a \leftarrow$ robot_jacobian_a$(q_k, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$
  $x_{e,k} \leftarrow$ robot_fk$(d_{1:N}, a_{1:N}, \alpha_{1:N}, q_k)$
  $\nabla_x J \leftarrow -J_a^T H (x_e - x_{e,k})$
  if $||x_{e,k} - x_e|| < \epsilon$ **then**
    break
  $q_k \leftarrow q_k - \alpha_p \nabla_x J$
$q^* \leftarrow q_k$

The corresponding Python code is shown below

```python
def robot_ik_gd(x_e, q_0, d, a, alpha, N=6, H=None,
    epsilon=1e-4, alpha_p=1e-2):
    assert q.shape == (N+1,) and d.shape == (N+1,)
    assert a.shape == (N+1,) and alpha.shape == (N+1,)
    assert p_cents.shape == (N+1, 3)
    if H is None:
        H = np.eye(N)
    else:
        assert H.shape == (N, N)
    q_k = q_0
    while True:
        J_a = robot_jacobian_a(q_k, d, a, alpha, x_e[3:],
            N)
        x_ek = robot_fk(q_k, d, a, alpha, N)
        grad = - J_a.T @ H @ (x_e - x_ek)
        if np.linalg.norm(x_ek - x_e) < epsilon:
            break
        q_k = q_k - alpha_p * grad
    return q_k
```

Similarly, we can also use the Gauss-Newton method to obtain the **Gauss-Newton-based inverse kinematics algorithm**. In some other textbooks, this method is also referred to as the **Jacobian pseudoinverse method**.

Algorithm	**Gauss-Newton IK**
Problem Type	Inverse Kinematics Solving
Known	End-effector state $\mathbf{x}_e$
To Find	Joint vector q
Algorithm Property	Iterative Solution

Algorithm 21: Gauss-Newton IK

Input: D-H parameters $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: End-effector pose x_e, initial joint value q_0
Parameter: Threshold ϵ, symmetric matrix H
Output: Optimal solution q^*
$q_k \leftarrow q_0$
while *True* **do**
    $J_a \leftarrow \text{robot\_jacobian\_a}(q, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$
    $x_{e,k} \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q_k)$
    if $||x_{e,k} - x_e|| < \epsilon$ **then**
        break
    $H_n \leftarrow J_a^T H J_a$
    $g_n \leftarrow J_a^T H(x_e - x_{e,k})$
    $q_k \leftarrow q_k - H_n^{-1} g_n$
$q^* \leftarrow q_k$

The corresponding Python code is shown below

```python
def robot_ik_gauss_newton(x_e, q_0, d, a, alpha, N=6, H=None, epsilon=1e-4, alpha_p=1e-2):
    assert q.shape == (N+1,) and d.shape == (N+1,)
    assert a.shape == (N+1,) and alpha.shape == (N+1,)
    assert p_cents.shape == (N+1, 3)
    if H is None:
        H = np.eye(N)
    else:
        assert H.shape == (N, N)
    q_k = q_0
    while True:
        J_a = robot_jacobian_a(q_k, d, a, alpha, x_e[3:], N)
        x_ek = robot_fk(q_k, d, a, alpha, N)
        if np.linalg.norm(x_ek - x_e) < epsilon:
            break
        g_n = J_a @ H @ (x_ek - x_e)
        H_n = J_a @ H @ J_a.T
        q_k = q_k - np.linalg.inv(H_n) @ g_ns
    return q_k
```

Additionally, we can also use the L-M method to solve this least squares problem, which is also known as the **damped least squares method**.

Algorithm	**Damped Least Squares IK**
Problem Type	Inverse Kinematics Solution
Known	End-effector state $\mathbf{x}_e$
To Find	Joint vector q
Algorithm Property	Iterative Solution

Algorithm 22: Damped Least Squares IK

Input: D-H parameters $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: End-effector pose x_e, Initial joint values q_0
Parameter: Optimization threshold ϵ, Gap thresholds ρ_{min}, ρ_{max}
Parameter: Initial penalty term λ_0, Symmetric matrix H
Output: Optimal solution q^*

$k \leftarrow 0$
while *True* **do**
  $J_a \leftarrow \text{robot\_jacobian\_a}(q, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$
  $x_{e,k} \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q_k)$
  $\mathbf{e}_k \leftarrow x_{e,k} - x_e$
  if $||\mathbf{e}_k|| < \epsilon$ **then**
    break
  $\delta q_k \leftarrow -(J_a^T H J_a + \lambda_k I)^{-1} J_a^T H \mathbf{e}_k$
  $q_{k+1} \leftarrow q_k + \delta q_k$
  $f_k \leftarrow \mathbf{e}_k^T H \mathbf{e}_k$
  $f_{k+1} \leftarrow (\mathbf{e}_k + \delta q_k)^T H (\mathbf{e}_k + \delta q_k)$
  $\rho_k = ||f_k - f_{k+1}||/||J_a \delta q_k||$
  if $\rho_k > \rho_{max}$ **then**
    $\lambda_{k+1} \leftarrow 2 * \lambda_k$
  if $\rho_k < \rho_{min}$ **then**
    $\lambda_{k+1} \leftarrow 0.5 * \lambda_k$
  else
    $\lambda_{k+1} \leftarrow \lambda_k$
  $k \leftarrow k + 1$
$q^* \leftarrow q_k$

The corresponding Python code is shown below:

```python
def robot_ik_lm(x_e, q_0, d, a, alpha, N=6, H=None, epsilon=1e-4, rmin=1e-5, rmax=1e-2,
↪  lambda_0=1):
    assert q.shape == (N+1,) and d.shape == (N+1,)
    assert a.shape == (N+1,) and alpha.shape == (N+1,)
    if H is None:
        H = np.eye(N)
    else:
        assert H.shape == (N, N)
    q_k, lambda_k = q_0, lambda_0
    while True:
        J_a = robot_jacobian_a(q_k, d, a, alpha, x_e[3:], N)
        x_ek = robot_fk(q_k, d, a, alpha, N)
        delta_xe = x_ek - x_e
        if np.linalg.norm(delta_xe) < epsilon:
            break
        g_l = J_a.T @ H @ delta_xe
        H_l = J_a.T @ H @ J_a + lambda_k * np.eye(N)
        delta_q = - np.linalg.inv(H_l) @ g_l
        q_new = q_k + delta_q
        f_k = delta_xe[None, :] @ H @ delta_xe
        f_new = delta_xe[None, :] @ H @ delta_xe
        rho = np.abs(f_k - f_new) / np.linalg.norm(J_k @ delta_x)
        if rho > rmax:
            lambda_k = 2 * lambda_k
        elif rho < rmin:
            lambda_k = 0.5 * lambda_k
```

```
        q_k = q_new
    return q_k
```

7.4 Properties of the Jacobian

[Main content: Kinematic singularities/Kinematic redundancy]

[This section will be updated in future versions, stay tuned]

(References: Tsinghua "Intelligent Robotics" course materials, "Robotics: Modelling, Planning and Control")

8 Dynamics

Dynamics describes the relationship between the torques of a robot and its joint-space positions (velocity, acceleration). It is often expressed in the form of dynamic equations.

Clearly, dynamics involves the forces and torques of a robot. According to Newton's laws of motion, forces/torques are related to the mass/moment of inertia and acceleration/angular acceleration of an object. Therefore, compared to kinematic equations, dynamic equations require not only the D-H parameters but also the robot's mass and mass distribution as known quantities.

The dynamic equations of a robot can be derived from the Newton-Euler equations, starting from the base/end-effector, or from the Euler-Lagrange equations of analytical mechanics. Here, we adopt the latter approach.

8.1 Kinetic and Potential Energy

Based on the definition of kinetic energy, we can write the expression for the kinetic energy of a link[6]

$$\mathcal{T}_n(q,\dot{p}_n,\omega_n)=\frac{1}{2}m_n\dot{p}_n^T\dot{p}_n+\frac{1}{2}\omega_n^T R_n(q)\mathbf{I}_n^n R_n^T(q)\omega_n^T \tag{II.8.1}$$

Here, $R_n=R_n^b$ represents the rotation matrix from the n-th frame to the base frame. $\mathbf{I}_n^n$ denotes the inertia matrix of link n in the n-th frame, which is a constant. $p_n=p_{b,l_n}^b$ represents the position of the link's **center of mass** in the base frame, satisfying

$$p_n=p_{b,l_n}^b=T_n^0(q)p_{l_n}^n$$

As mentioned earlier, using the geometric Jacobian matrix, the kinetic energy expression can be written in terms of joint-space velocity:

$$\mathcal{T}_n(q,\dot{q})=\frac{1}{2}m_n\dot{q}^T J_{p,n}^T(q)J_{p,n}(q)\dot{q}+\frac{1}{2}\dot{q}^T J_{\theta,n}^T(q)R_n(q)\mathbf{I}_n^n R_n^T(q)J_{\theta,n}(q)\dot{q}$$

In fact, the inertia matrix of the robot in joint space can be defined as

$$B(q)=\sum_{n=1}^{N}m_n J_{p,n}^T(q)J_{p,n}(q)+J_{\theta,n}^T(q)R_n(q)\mathbf{I}_n^n R_n^T(q)J_{\theta,n}(q) \tag{II.8.2}$$

Thus, the total kinetic energy of the robot is

$$\mathcal{T}(q,\dot{q})=\frac{1}{2}\dot{q}^T B(q)\dot{q}=\sum_{j=1}^{N}\sum_{k=1}^{N}b_{ij}(q)\dot{q}_j\dot{q}_k \tag{II.8.3}$$

Next, ignoring elastic deformation, we can write the total potential energy as

$$\mathcal{U}(q)=-\sum_{n=1}^{N}m_n g_0^T p_n(q) \tag{II.8.4}$$

where g_0 is the gravitational acceleration vector in the base frame, typically given by

$$g_0=\begin{bmatrix}0\\0\\-g\end{bmatrix} \tag{II.8.5}$$

[6] In fact, in more detailed modeling, the kinetic energy of the robot's drive motors (center of mass and inertia) needs to be considered separately. For simplicity in this book, we only consider each link.

8.2 Derivation of Dynamic Equations

According to analytical mechanics, the Lagrangian is the difference between kinetic and potential energy:

$$\mathcal{L}(q,\dot{q}) = \mathcal{T}(q,\dot{q}) - \mathcal{U}(q) \tag{II.8.6}$$

Thus, the Lagrangian for a rigid-body robot system is

$$\mathcal{L}(q,\dot{q}) = \frac{1}{2}\dot{q}^T B(q)\dot{q} - \sum_{n=1}^{N} m_n g_0^T p_n(q) \tag{II.8.7}$$

From the Lagrange equation, we have

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}} - \frac{\partial \mathcal{L}}{\partial q} = \xi \tag{II.8.8}$$

Here, ξ represents the generalized force. For robots with only revolute joints, the generalized force is the joint torque. Joint torques include the driving torque τ from joint motors, friction torque τ_f, external torque τ_F, etc.

Next, we derive the dynamic equations for an ideal robot (without friction or elastic deformation) under no external forces.

Substituting the Lagrangian into the robot equation and setting $\xi = \tau$, we obtain

$$\frac{d}{dt}\frac{\partial \mathcal{T}}{\partial \dot{q}} - \frac{\partial \mathcal{T}}{\partial q} - \frac{\partial \mathcal{U}}{\partial q} = \tau$$

$$\frac{d}{dt}\left(B(q)\dot{q}\right) - \frac{1}{2}\dot{q}^T\frac{\partial B(q)}{\partial q}\dot{q} - \frac{\partial \mathcal{U}}{\partial q} = \tau$$

First, let's address the third term. Note that

$$\begin{aligned} -\frac{\partial \mathcal{U}}{\partial q} &= -\left(-\sum_{n=1}^{N} m_n g_0^T \frac{\partial}{\partial q} p_n(q)\right) \\ &= \sum_{n=1}^{N} m_n g_0^T J_{p,n}(q) \end{aligned}$$

Let

$$g(q) = \sum_{n=1}^{N} m_n g_0^T J_{p,n}(q) \tag{II.8.9}$$

Thus, we have

$$-\frac{\partial \mathcal{U}}{\partial q} = g(q)$$

For the first term, we have

$$\frac{d}{dt}\left(B(q)\dot{q}\right) = B(q)\ddot{q} + \dot{B}(q)\dot{q}$$

Therefore,

$$\frac{d}{dt}\frac{\partial \mathcal{T}}{\partial \dot{q}} - \frac{\partial \mathcal{T}}{\partial q} = B(q)\ddot{q} + \dot{B}(q)\dot{q} - \frac{1}{2}\dot{q}^T\frac{\partial B(q)}{\partial q}\dot{q}$$

Considering the n-th joint variable q_n (for revolute joints, θ_n), we have

$$\frac{d}{dt}\frac{\partial \mathcal{T}}{\partial \dot{q}_i} - \frac{\partial \mathcal{T}}{\partial q_i} = \sum_{j=1}^{N} b_{nj}(q)\ddot{q}_j + \sum_{j=1}^{N} \dot{b}_{nj}(q)\dot{q}_j - \frac{1}{2}\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{jk}(q)}{\partial q_n}\dot{q}_j\dot{q}_k$$

Note that

$$\begin{aligned}\dot{b}_{nj}(q) &= \frac{\partial b_{nj}(q)}{\partial q}\dot{q} \\ &= \sum_{k=1}^{N}\frac{\partial b_{nj}(q)}{\partial q_k}\dot{q}_k\end{aligned}$$

Thus, the first two terms can be simplified as

$$\begin{aligned}\frac{d}{dt}\frac{\partial \mathcal{T}}{\partial \dot{q}_i} - \frac{\partial \mathcal{T}}{\partial q_i} &= \sum_{j=1}^{N} b_{nj}(q)\ddot{q}_j + \sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{nj}(q)}{\partial q_k}\dot{q}_k\dot{q}_j - \frac{1}{2}\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{jk}(q)}{\partial q_n}\dot{q}_j\dot{q}_k \\ &= \sum_{j=1}^{N} b_{nj}(q)\ddot{q}_j + \sum_{j=1}^{N}\sum_{k=1}^{N}\left(\frac{\partial b_{nj}(q)}{\partial q_k} - \frac{1}{2}\frac{\partial b_{jk}(q)}{\partial q_n}\right)\dot{q}_k\dot{q}_j\end{aligned}$$

Let

$$h_{njk}(q) = \frac{\partial b_{nj}(q)}{\partial q_k} - \frac{1}{2}\frac{\partial b_{jk}(q)}{\partial q_n}$$

and

$$c_{njk}(q) = \frac{1}{2}\left(\frac{\partial b_{nj}}{\partial q_k} + \frac{\partial b_{nk}}{\partial q_j} - \frac{\partial b_{jk}}{\partial q_n}\right) \tag{II.8.10}$$

and

$$c_{nj}(q,\dot{q}) = \sum_{k=1}^{N} c_{njk}(q)\dot{q}_k$$

then we have

$$\begin{aligned}\sum_{j=1}^{N} c_{nj}(q,\dot{q})\dot{q}_j &= \sum_{j=1}^{N}\sum_{k=1}^{N} c_{njk}(q)\dot{q}_k q_j \\ &= \sum_{j=1}^{N}\sum_{k=1}^{N}\frac{1}{2}\left(\frac{\partial b_{nj}}{\partial q_k} + \frac{\partial b_{nk}}{\partial q_j} - \frac{\partial b_{jk}}{\partial q_n}\right)\dot{q}_k\dot{q}_j \\ &= \frac{1}{2}\left(\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{nj}}{\partial q_k}\dot{q}_k\dot{q}_j + \sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{nk}}{\partial q_j}\dot{q}_k\dot{q}_j - \sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{jk}}{\partial q_n}\dot{q}_k\dot{q}_j\right) \\ &= \frac{1}{2}\left(\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{nj}}{\partial q_k}\dot{q}_k\dot{q}_j + \sum_{k=1}^{N}\sum_{j=1}^{N}\frac{\partial b_{nj}}{\partial q_k}\dot{q}_j\dot{q}_k - \sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{jk}}{\partial q_n}\dot{q}_k\dot{q}_j\right) \\ &= \left(\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{nj}}{\partial q_k}\dot{q}_k\dot{q}_j - \frac{1}{2}\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{jk}}{\partial q_n}\dot{q}_k\dot{q}_j\right)\end{aligned}$$

(Note: From line 3 to line 4, the second term uses the subscript swapping technique—renaming the k in the previous line as j and j as k, which does not affect the summation of this term or other terms.)

Observe that the current summation takes the form of $h_{njk}(q)$.

$$\sum_{j=1}^{N} c_{nj}(q,\dot{q})\dot{q}_j = \sum_{j=1}^{N}\sum_{k=1}^{N}\left(\frac{\partial b_{nj}(q)}{\partial q_k} - \frac{1}{2}\frac{\partial b_{jk}(q)}{\partial q_n}\right)\dot{q}_k\dot{q}_j$$

Thus, the first two terms of the dynamic equation can be written as

$$\frac{d}{dt}\frac{\partial \mathcal{T}}{\partial \dot{q}_i} - \frac{\partial \mathcal{T}}{\partial q_i} = \sum_{j=1}^{N} b_{nj}(q)\ddot{q}_j + \sum_{j=1}^{N} c_{nj}(q,\dot{q})\dot{q}_j$$

The constructed $c_{nj}(q,\dot{q})$ is denoted as the C matrix, i.e.,

$$C(q,\dot{q}) = [c_{nj}(q,\dot{q})]_{N\times N} \tag{II.8.11}$$

We then have the dynamic equation under ideal conditions:

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau \tag{II.8.12}$$

In the terms of the dynamic equation, $B(q)\ddot{q}$ is the inertial term; $C(q,\dot{q})\dot{q}$ represents the Coriolis force (or the coupling force); and $g(q)$ is the gravitational term.

A more complete dynamic equation is

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f(q,\dot{q})\dot{q} + g(q) = \tau - J^T(q)F_e \tag{II.8.13}$$

In this version, the additional term $F_f(q,\dot{q})\dot{q}$ represents the friction resistance term (including sliding friction and static friction), and $-J^T(q)F_e$ represents the external force term. F_e denotes the generalized external force acting on the end-effector in the operational space, including linear force and torque.

Summarizing the above equations, the calculation methods for each term in the equation are as follows:

$$\begin{aligned}
C(q,\dot{q}) &= \left[\sum_{k=1}^{N} c_{njk}(q)\dot{q}_k\right]_{N\times N} \\
c_{njk}(q) &= \frac{1}{2}\left(\frac{\partial b_{nj}(q)}{\partial q_k} + \frac{\partial b_{nk}(q)}{\partial q_j} - \frac{\partial b_{jk}(q)}{\partial q_n}\right) \\
B(q) &= [b_{ij}(q)]_{N\times N} = \sum_{n=1}^{N} m_n J_{p,n}^T(q) J_{p,n}(q) + J_{\theta,n}^T(q) R_n(q) \mathbf{I}_n^n R_n^T(q) J_{\theta,n}(q) \\
g(q) &= \sum_{n=1}^{N} m_n g_0^T J_{p,n}(q) \\
J_{\theta,n}(q) &= \begin{bmatrix} z_0^b & \cdots & z_n^b & 0_{3\times 1} & \cdots & 0_{3\times 1} \end{bmatrix} \\
J_{p,n}(q) &= \begin{bmatrix} z_0^b \times p_{0,l_n}^b & \cdots & z_{n-1}^b \times p_{n-1,l_n}^b & 0_{3\times 1} & \cdots & 0_{3\times 1} \end{bmatrix}
\end{aligned} \tag{II.8.14}$$

It can be seen that the calculation of the complete robot dynamic equation is relatively complex. The terms $B(q)$, $C(q,\dot{q})$, and $g(q)$ under ideal conditions all involve the geometric Jacobian of the link centroids. The calculation of the C matrix also requires further differentiation of the Jacobian.

We summarize the above dynamic algorithm as follows:

Algorithm	**Dynamic Analytical Solution**
Problem Type	Dynamic Term Solution
Given	D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$ Joint vector q and its derivative $\dot{q}$ Link parameters $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$
Solve for	Dynamic terms $B(q), C(q,\dot{q}), g(q)$
Algorithm Property	Analytical Solution

Algorithm 23: Robot Dynamics (robot_dyn)

Input: Link parameters $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$
Input: D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$
Input: Joint vector q and its derivative $\dot{q}$
Output: Dynamic terms $B(q), C(q, \dot{q}), g(q)$
$p_e, \vartheta, R_n^b \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q)$
$J_{p,n}, J_{\theta,n} \leftarrow \text{robot\_j\_centroid}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q, p_{l_n}^n)$
$B \leftarrow \sum_{n=1}^{N} m_n J_{p,n}^T J_{p,n} + J_{\theta,n}^T R_n^b \mathbf{I}_n^n (R_n^b)^T J_{\theta,n}$
$g \leftarrow \sum_{n=1}^{N} m_n g_0^T J_{p,n}(q)$
for $n \in 1, ..., N$ **do**
  for $j \in 1, ..., N$ **do**
    for $k \in 1, ..., N$ **do**
      $c_{njk} \leftarrow \frac{1}{2}\left(\frac{\partial b_{nj}}{\partial q_k} + \frac{\partial b_{nk}}{\partial q_j} - \frac{\partial b_{jk}}{\partial q_n}\right)$
    $C_{n,j} \leftarrow \sum_{k=1}^{N} c_{njk}\dot{q}_k$

In practical applications, robot dynamics are typically computed using mature code libraries, eliminating the need for manual derivation. The partial differentiation steps are implemented through automatic differentiation.

(References: Tsinghua "Intelligent Robotics" course materials, "Robotics: Modelling, Planning and Control", "Introduction to Robotics")

8.3 Properties of Dynamic Equations

For the C matrix introduced in Equation II.8.11, there exists a crucial property. Let

$$N(q, \dot{q}) = \dot{B}(q) - 2C(q, \dot{q}) \tag{II.8.15}$$

Considering a specific element and expanding it, we have

$$\begin{aligned} n_{ij}(q, \dot{q}) &= \dot{b}_{ij}(q) - 2c_{ij}(q, \dot{q}) \\ &= \dot{b}_{ij}(q) - 2\sum_{k=1}^{N} c_{ijk}(q)\dot{q}_k \\ &= \sum_{k=1}^{N} \frac{\partial b_{ij}}{\partial q_k}\dot{q}_k - 2\sum_{k=1}^{N} \frac{1}{2}\left(\frac{\partial b_{ij}}{\partial q_k} + \frac{\partial b_{ik}}{\partial q_j} - \frac{\partial b_{jk}}{\partial q_i}\right)\dot{q}_k \\ &= -\sum_{k=1}^{N}\left(\frac{\partial b_{ik}}{\partial q_j} - \frac{\partial b_{jk}}{\partial q_i}\right)\dot{q}_k \end{aligned}$$

By swapping indices, it is evident that

$$\begin{aligned} n_{ji}(q, \dot{q}) &= -\sum_{k=1}^{N}\left(\frac{\partial b_{jk}}{\partial q_i} - \frac{\partial b_{ik}}{\partial q_j}\right)\dot{q}_k \\ &= -n_{ij}(q, \dot{q}) \end{aligned}$$

That is: **The matrix $N(q, \dot{q})$ is skew-symmetric**

$$N(q, \dot{q}) + N^T(q, \dot{q}) = 0 \tag{II.8.16}$$

For a skew-symmetric matrix, its quadratic form is always zero, because

$$x^T N x = \frac{1}{2}x^T N x + \frac{1}{2}(x^T N x)^T = \frac{1}{2}(x^T N x + x^T N^T x) = 0$$

Namely,

$$x^T(\dot{B}(q) - 2C(q, \dot{q}))x = 0 \tag{II.8.17}$$

This property will play a key role in the derivation of robot control in subsequent chapters.

9 Trajectory Planning

[Main content: Trajectory planning problem; PRM algorithm, RRT algorithm, Bidirectional RRT, RRT*]

[This section will be updated in future versions. Stay tuned.]

(References: Tsinghua "Intelligent Robotics" course materials, "Robotics: Modelling, Planning and Control")

Part III
Fundamentals of Control Theory

10 Control Fundamentals

Robot control is a critical issue. In the previous section, we introduced methods for kinematic and dynamic modeling of robots. Next, we need to control them based on specific requirements and theories. In this section, we will introduce fundamental control theory; Part IV will specifically discuss basic methods for robot control.

In this chapter, we will cover basic concepts of control theory, frequency-domain tools, time-domain tools, and more. These tools will play a key role in the design of robot controllers in subsequent chapters. Additionally, we will introduce the inverted pendulum problem. This problem appears throughout multiple parts and chapters of this book, helping readers understand the differences and connections between various techniques.

10.1 Basic Concepts

Unlike many control textbooks, in this section, we will first introduce core concepts of control theory such as feedback, open-loop vs. closed-loop, regulation vs. tracking. We believe that establishing a basic yet clear understanding of these concepts is the foundation for comprehending other important aspects of control theory.

10.1.1 Feedback and Closed-Loop

Control is a fundamental human desire. By control, we mean the wish for a real-world object to move or change in a way that aligns with human intentions.

To achieve this goal, the controlled object must be susceptible to external intervention. The quantity that can be manipulated externally is called the **control input**, also known as the **input variable**. Control always has an objective, and the quantity closely related to this objective is called the **output variable**. The output variable must be observable. A system with both control input and output variables is called a **controlled system**.

Let the control input be u and the output variable be y. Then, the simplest form of a controlled system is:

$$y = f(u)$$

The goal of control is generally to ensure that the output variable remains stable or changes according to a predefined pattern. This predefined target is called the **reference input**, typically denoted as r. When the system output does not match the reference input, an **error** is generated, denoted as e. Thus, we have:

$$e = r - u \tag{III.10.1}$$

The most basic control objective is to adjust the control input to reduce the error to zero. To achieve this, a method is needed to compute the control input. This computation should consider the reference input and the output variable, which we call the **controller**, i.e.,

$$u = c(r, y)$$

In this case, we say the controller and the controlled system form a **feedback** loop. Feedback is the most fundamental control idea and the starting point of all control theories.

More commonly, our controller explicitly operates on the error, i.e.,

$$u = c(r, y, e) \tag{III.10.2}$$

With feedback introduced, the controlled system and the controller together form a **closed-loop system**, i.e.,

$$y = f(c(r, y, e))$$

10.1.2 Differential Equations

In the real world, many controlled systems are modeled not by algebraic equations but by differential equations.

For example, consider controlling the position of an object on a frictional incline. The control input is the force applied to the object. According to Newton's second law, force is proportional to acceleration, which in turn is proportional to the second derivative of position. In such cases, the actual form of the controlled system is:

$$\dot{y} = f(y, y^{(2)}, ..., u)$$

For the frictional incline example, we can write the equation:

$$\ddot{y} = \frac{F}{m} - g\sin\theta - \mu g\cos\theta\,\mathrm{sgn}(\dot{y})$$

Here, y represents the position of the object on the incline, and F is the external force along the incline. Unlike the earlier form of the controlled system, the left-hand side is not a first-order derivative. However, we can transform higher-order equations into first-order equations by defining states. Let:

$$x = \begin{bmatrix} y \\ \dot{y} \end{bmatrix}$$

Then:

$$\begin{bmatrix} \dot{y} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y \\ \dot{y} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{F}{m} \end{bmatrix} + \begin{bmatrix} 0 \\ -g\sin\theta - \mu g\cos\theta\,\mathrm{sgn}(\dot{y}) \end{bmatrix}$$

Let:

$$u = \begin{bmatrix} 0 \\ \frac{F}{m} \end{bmatrix}$$

And define:

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

$$g(x) = \begin{bmatrix} 0 \\ -g\sin\theta - \mu g\cos\theta\,\mathrm{sgn}(x[1]) \end{bmatrix}$$

Then:

$$\dot{x} = Ax + g(x) + u \tag{III.10.3}$$

In fact, most differential equations of controlled systems with higher-order terms can be simplified in a similar manner to obtain a first-order differential equation with linear and/or nonlinear terms. Therefore, we present a general form of the controlled system differential equation:

$$\begin{aligned} \dot{x} &= f(x, u) \\ y &= g(x) \end{aligned} \tag{III.10.4}$$

Equations of the form III.10.4 are called **open-loop system equations**. Here, x is called the **state** (state variable/state vector) and is sometimes written as $x(t)$ to emphasize its time-varying nature. In fact, in most robotics applications, the term **system** refers to **the state variables required to describe an object and the differential equations they satisfy**.

Similar to the previous section, we can introduce feedback control (Equation III.10.2) into the open-loop system equation to form a closed-loop system:

$$
\begin{aligned}
\dot{x} &= f(x, u) \\
y &= g(x) \\
e &= r - y \\
u &= c(r, y, e)
\end{aligned}
$$

Assuming the derivatives of r are known, we can ignore the distinction between x and y by setting $y = x$. Then, the closed-loop system can be written as:

$$
\begin{aligned}
\dot{x} &= f(x, u) \\
e &= r - x \\
u &= c(r, x, e)
\end{aligned}
$$

Using $x = r - e$ to eliminate the open-loop system state x and assuming r is fully known, we obtain:

$$\dot{e} = f_c(e) := f(e, c(e)) \tag{III.10.5}$$

Thus, we arrive at a differential equation solely about the system error e. For such differential equations, we call them **closed-loop system equations**. This equation contains no additional input terms and is an **autonomous system**, not a controlled system. Here, the error e is also called the system's **error state**.

10.1.3 Control Objectives

Earlier, we introduced that real-world controlled objects are often described by differential equations, i.e., open-loop systems. Through feedback control, the output of the open-loop system can meet a certain objective, which is closely related to the reference input.

Specifically, we can consider two types of reference inputs. One is a constant known quantity, denoted as r or x_d. The other is a dynamically changing known quantity, denoted as $r(t)$ or $x_d(t)$.

For tasks where the system output is desired to remain constant, we call them **regulation** tasks. For example, we might want the angle between an inverted pendulum and the vertical direction to stay at 0° or a specific angle. For tasks where the system output is desired to follow a reference signal, we call them **tracking** tasks. For example, we might want the steering angle of a car's wheels to follow the motion of the steering wheel.

Both regulation and tracking problems require designing control methods under the premise of **ensuring closed-loop system stability** (details in the next section). Therefore, we summarize these two problems as follows.

Problem	**Regulation Control Problem**
Description	Given an open-loop system, design a controller to stabilize the output at the reference value while ensuring system stability
Given	Open-loop system $\dot{x} = f(x, u)$, reference input x_d
Find	Controller $c(e, x_d, x)$

Problem	**Tracking Control Problem**
Problem Description	Given an open-loop system, design a controller to make the output follow the reference value while ensuring system stability
Given	Open-loop system $\dot{x} = f(x, u)$, reference input $x_d(t)$
Objective	Controller $c(e, x_d, x)$

In practical applications, regulator controller design is generally relatively simple, whereas tracking controllers are more complex. Controllers designed for regulation problems can sometimes be applied to tracking problems, but their performance is usually inferior to dedicated tracking controllers, especially when the derivative of the target trajectory is large.

Besides regulation and tracking, **disturbance rejection** is another important control objective. Real-world systems may be subject to various disturbances, such as modeling errors, sensor noise, nonlinear

coupling terms, and other neglected or approximated interference terms. In scenarios requiring precise control, such as robotics, disturbances often require special handling.

Specifically, the open-loop controlled system equation with general disturbance terms is:

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t), d(t), t) \\ y(t) &= g(x(t)) \end{aligned}$$

The closed-loop system equation is:

$$\begin{aligned} \dot{x} &= f(x, u, d) \\ e &= r - x \\ u &= c(r, x, e) \end{aligned}$$

General regulation and tracking controllers provide some level of disturbance rejection. However, disturbances inevitably affect quantitative metrics such as convergence speed and tracking error. For improved control performance, disturbances must be carefully modeled, estimated, and compensated, which is referred to as disturbance rejection control.

10.1.4 Basic Classification of Dynamic Systems

Dynamic systems encompass all systems whose states change over time. As introduced earlier, continuously varying dynamic systems are generally described using **differential equations** to capture their evolution. Based on whether the system has controllable inputs, dynamic systems can be classified into autonomous systems and controlled systems.

A typical autonomous system is:

$$\begin{aligned} \dot{x}(t) &= f(x(t), t) \\ y(t) &= g(x(t)) \end{aligned} \tag{III.10.6}$$

A typical controlled system is:

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t), t) \\ y(t) &= g(x(t)) \end{aligned} \tag{III.10.7}$$

Here, the system state is explicitly written as a function of time t, and the differential equation may also explicitly depend on t. If it does, such a system is called a **time-varying system**. Systems represented by Equations III.10.4 and III.10.5 are typical examples of **time-invariant systems**.

For a system, the output corresponding to a specific input is called the system's **response** to that input.

If the system input $u(t)$ and output $y(t)$ are both one-dimensional (scalar), the system is called a **single-input single-output system** (SISO system). A SISO system can often be expressed as a single higher-order differential equation:

$$f_y(y^{(m-1)}(t), y^{(m-2)}(t), ..., y'(t), y(t), t) = f_u(u^{(n-1)}(t), u^{(n-2)}(t), ..., u'(t), u(t), t)$$

Among all controlled systems, the simplest is the **linear time-invariant system** (LTI system). Linearity means that applying a linear transformation to the system input results in the same linear transformation of the output. Time-invariance means the differential equation does not explicitly depend on t. That is:

$$a_{m-1}y^{(m-1)}(t) + a_{m-2}y^{(m-2)}(t) + ... + a_1y'(t) + a_0y(t) = u^{(n-1)}(t) + b_{n-2}u^{(n-2)}(t) + ... + b_1u'(t) + b_0u(t) \tag{III.10.8}$$

A typical example of a linear time-invariant system is Equation III.10.3, although that system is not linear. Below is an example of a linear second-order time-invariant system:

$$a_2y''(t) + a_1y'(t) + a_0y(t) = u(t) \tag{III.10.9}$$

In the context of mechanics and electronics, the parameters of a typical second-order system have clear physical meanings. Consider an object moving in one dimension on a smooth surface, subject to an external

force F, damping (with resistance proportional to velocity, damping coefficient c), and a spring (stiffness coefficient k). Taking the spring's equilibrium position as the zero displacement point and setting displacement as the output y, we have:

$$m\ddot{y} = F - c\dot{y} - ky$$

Or equivalently:

$$m\ddot{y} + c\dot{y} + ky = F \tag{III.10.10}$$

Here, the coefficients a_0, a_1, and a_2 in the general second-order time-invariant system correspond to the stiffness coefficient k, damping coefficient c, and mass m, respectively.

Following the friction slope example, we define:

$$x = \begin{bmatrix} y \\ \dot{y} \end{bmatrix}$$

This gives:

$$\dot{x} = Ax + Bu$$

Where:

$$A = \frac{1}{m}\begin{bmatrix} 0 & m \\ -c & -k \end{bmatrix}, \quad B = \frac{1}{m}\begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad u = \begin{bmatrix} 0 \\ F \end{bmatrix}$$

It can be seen that LTI systems can be expressed in a linear form in the continuous-time domain by defining higher-order derivatives as states. This representation is concise and intuitive. Later in this chapter, we will further explore the properties of linear systems in the time domain.

10.1.5 PID Control

As mentioned earlier, the general form of a feedback controller is Equation III.10.2. In practice, not all controllers require both r and x. The part requiring r is called **feedforward**, while the part requiring x is called **inner-loop feedback**. Feedforward and inner-loop feedback are often used to linearize nonlinear systems.

In practice, the part related to the error e is the most critical feedback component. Its design directly determines the properties of the closed-loop system $\dot{e} = f_c(e)$. For both regulation and tracking problems, there is a simple yet powerful error feedback control method: **proportional-integral-derivative control**, or **PID control**.

A PID controller uses only a linear combination of the error term, its derivative, and its integral as the control input:

$$u = K_P e + K_I \int_0^t e dt + K_D \dot{e} \tag{III.10.11}$$

Sometimes, the integral term can be omitted, resulting in PD control:

$$u = K_P e + K_D \dot{e} \tag{III.10.12}$$

When using PID control in continuous scenarios, note that the error derivative term must be observed separately rather than numerically differentiated from the error, as this can lead to stability and realizability issues. In engineering, a filter is typically used for observation.

In discrete scenarios, the PID control input can be written as:

$$u(t) = K_p e(t) + K_I \sum_{\tau=0}^{t} e(\tau) + K_D(e(t) - e(t-1)) \tag{III.10.13}$$

Taking the difference and defining:

$$\Delta u(t) := u(t) - u(t-1)$$

We obtain:

$$\Delta u(t) = K_p(e(t) - e(t-1)) + K_I e(t) + K_D(e(t) - 2e(t-1) + e(t-2))$$

Or:

$$\Delta u(t) = (K_p + K_I + K_D)e(t) - (K_p + 2K_D)e(t-1) + K_D e(t-2) \tag{III.10.14}$$

Equation III.10.14 is also called **incremental PID**. Compared to directly discretized PID control, incremental PID does not require storing past errors and has constant space complexity.

PID control is simple and intuitive, making it widely applicable in robotics control. In many scenarios where high precision is not required, PID control can effectively handle regulation and tracking tasks. It also provides disturbance rejection for general disturbances. In later chapters, we will further analyze PID controllers.

10.2 Basic Tools in the Frequency Domain

In this section, we introduce some fundamental tools in the frequency domain. Our discussion is limited to SISO LTI systems.

10.2.1 Convolution Transformation

Even for the simplest controlled system, i.e., a SISO LTI system, directly studying its properties is not straightforward.

Consider a unit impulse function $\delta(t)$, defined as

$$\delta(t) = 0, t \neq 0$$
$$\int_{-\infty}^{\infty} \delta(t)dt = 1$$

In continuous-time systems, any input can be decomposed into a superposition of unit impulse functions at different times.

$$u(t) = \int_{-\infty}^{\infty} u(\tau)\delta(t-\tau)d\tau$$

Therefore, for an LTI system, based on its linearity and time-invariance, we can take its response to $\delta(t)$, denoted as $h(t)$. Then, the response $y(t)$ to any input $u(t)$ can be decomposed as

$$y(t) = \int_{-\infty}^{\infty} u(\tau)h(t-\tau)d\tau$$

This integration process is also known as **convolution**, i.e.,

$$y(t) = u(t) * h(t) := \int_{-\infty}^{\infty} u(\tau)h(t-\tau)d\tau$$

10.2.2 Transfer Function

Even if the system's unit impulse response $h(t)$ is obtained, for different inputs $u(t)$, multiple convolution calculations are still required to determine the corresponding $y(t)$. Such a description is neither intuitive nor concise for the system.

Therefore, we introduce the **Laplace transform**, which converts a time-domain function $f(t)$ into a complex frequency-domain function $F(s)$:

$$F(s) = \mathcal{L}(f(t)) := \int_0^{\infty} f(t)e^{-st}dt$$

Note that this integral has different convergence conditions for different $f(t)$, known as the **region of convergence**.

For a SISO LTI system, after the Laplace transform, the convolution relationship between input and output becomes a multiplicative relationship:

$$Y(s) = H(s)U(s)$$

That is,

$$H(s) = \frac{Y(s)}{U(s)}$$

Here, we refer to $H(s)$ as the system's transfer function.

For the SISO LTI system in Equation III.10.8, its transfer function takes the form:

$$H(s) = \frac{s^{m-1} + a_{m-2}s^{m-2} + ... + a_1 s + a_0}{b_{n-1}s^{n-1} + ... + b_1 s + b_0}$$

For example, for the system in Equation III.10.9, its transfer function is:

$$H(s) = a_2 s^2 + a_1 s + a_0$$

10.2.3 Feedback and Closed-Loop Transfer Function

For the controlled system $H(s)$, the simplest feedback control is **unit negative feedback**, i.e.,

$$u(t) = e(t) = r(t) - y(t)$$
$$y(t) = u(t) * h(t)$$

Here, we define the open-loop transfer function $H_o(s)$ and the closed-loop transfer function $H_c(s)$ as:

$$H_o(s) = H(s) = \frac{Y(s)}{U(s)}$$
$$H_c(s) = \frac{Y(s)}{R(s)}$$

Through derivation, we obtain the transfer function under unit negative feedback:

$$H_c(s) = \frac{H_o(s)}{1 + H_o(s)} = \frac{Y(s)}{U(s) + Y(s)}$$

If the feedback controller is an LTI differential equation (with unit feedback as a special case), its unit impulse response is $c(t)$, i.e.,

$$u(t) = e(t) = r(t) - c(t) * y(t)$$
$$y(t) = u(t) * h(t)$$

In this case, we have:

$$H_c(s) = \frac{H_o(s)}{1 + C(s)H_o(s)}$$

Assuming the open-loop system and the controller have numerator and denominator polynomials, i.e.,

$$H_o(s) = \frac{H_A(s)}{H_B(s)}$$
$$C(s) = \frac{C_A(s)}{C_B(s)}$$

then the closed-loop system transfer function is:

$$H_c(s) = \frac{C_B(s)H_A(s)}{C_B(s)H_B(s) + C_A(s)H_A(s)}$$

It can be observed that after adding the feedback controller $C(s)$, the characteristic polynomial of the closed-loop system becomes $C_B(s)H_B(s) + C_A(s)H_A(s)$, which differs from the characteristic polynomial of the open-loop system.

10.2.4 Zeros, Poles, and Order

For a SISO LTI system, the numerator and denominator of its transfer function are polynomials. We can factorize them into complex factors and write them as:

$$H(s) = \frac{K(s - z_1)(s - z_2)...(s - z_m)}{s^k(s - p_1)(s - p_2)...(s - p_n)}$$

Here, the constant coefficient K is called the gain, the roots of the numerator polynomial z_i are called **zeros**, the roots of the denominator polynomial p_i are called **poles**, and the highest degree $k + n$ of the denominator polynomial is called the system's order.

For a SISO LTI system, the terms that may appear in the solution of the differential equation are determined by the roots of the denominator polynomial. Compared to the numerator polynomial, the denominator polynomial more significantly determines the system's characteristics. We also refer to the denominator polynomial as the **characteristic polynomial**. Setting the denominator polynomial to zero is called the **characteristic equation**.

For the closed-loop feedback system described in the previous section, it can be seen that the denominator polynomial of the closed-loop system differs from that of the open-loop system. In fact, the core of frequency-domain control is to design the controller $C(s)$ using certain methods to transform the open-loop system's characteristic polynomial into the closed-loop system's characteristic polynomial, thereby altering the system's properties (poles/order/various metrics/...) to achieve specific control objectives.

(References: "The Beauty of Control - Volume 1," "Principles of Automatic Control")

10.2.5 Discrete and Continuous

[Main content: Continuous-time systems/Discrete-time systems]

[Main content: Continuous-frequency systems/Discrete-frequency systems]

[This section will be updated in future versions. Stay tuned.]

10.3 Time-Domain Response and Metrics

For control systems, various metrics can be defined to quantify performance aspects of human interest. Some of these metrics are in the time domain, while others are in the frequency domain. We first introduce time-domain metrics.

10.3.1 Unit Step Response

In practical systems, the response to a unit step input is highly important. A unit step input is a function that jumps to 1 at $t = 0$, which can be viewed as the integral of the unit impulse response:

$$u_{step}(t) = 1(t) := \int_{-\infty}^{t} \delta(\tau)d\tau$$

The Laplace transform of the unit step function is $\frac{1}{s}$:

$$U_{step}(s) = \frac{1}{s}$$

The system's response to a unit step input is called the unit step response. Its frequency-domain expression is:

$$Y_{step}(s) = \frac{1}{s}H(s)$$

The unit step response is particularly important because, in many cases, we need to abruptly change the control input. For linear systems, complex inputs can be regarded as superpositions of multiple unit step inputs at different times. Through the unit step response, we can intuitively define quantitative control objectives in the time domain, such as delay characteristics and overshoot characteristics.

10.3.2 Negative Feedback and Step Response

Consider a very simple differential equation:

$$u = \dot{y}$$

The transfer function of this open-loop system is:

$$H_1(s) = \frac{1}{s}$$

Such a system can be found in reality. For example, applying a force to an object can control its velocity. Let $u(t)$ be the net force and $y(t)$ be the velocity. Then, this system conforms to the above transfer function.

For this system, its unit step response is:

$$\begin{aligned} y_{step,1}(t) &= \mathcal{L}^{-1}(Y_{step}(s)) \\ &= \mathcal{L}^{-1}\left(\frac{1}{s}H_1(s)\right) \\ &= \mathcal{L}^{-1}\left(\frac{1}{s^2}\right) = t \end{aligned}$$

Thus, we have:

$$\lim_{t\to\infty} y_{step,1}(t) = +\infty \tag{III.10.15}$$

That is, the unit step response of the open-loop system ultimately diverges. Given the significance of the unit step response, such an open-loop system is not ideal.

To make the system's unit step response converge, consider adding negative feedback. For simplicity, we first add unit negative feedback. Based on the conclusions from the previous section, we have:

$$H_{1c}(s) = \frac{1}{s+1}$$

The corresponding differential equation is:

$$u - y = \dot{y}$$

Recalculating the unit step response, we obtain:

$$\begin{aligned} y_{step,1c}(t) &= \mathcal{L}^{-1}(Y_{step,1c}(s)) \\ &= \mathcal{L}^{-1}\left(\frac{1}{s}H_{1c}(s)\right) \\ &= \mathcal{L}^{-1}\left(\frac{1}{s(s+1)}\right) \\ &= 1(t) - e^{-t} \end{aligned}$$

Now, we have:

$$\lim_{t\to\infty} y_{step,1c}(t) = 1$$

We can observe that negative feedback transforms an open-loop system of the form $\frac{1}{s}$ into a system of the form $\frac{1}{s+1}$, changing its unit step response from divergent to convergent.

In fact, we have a better method to calculate the final value of the step response. We need to use the **Final Value Theorem** of Laplace transform:

$$\lim_{t\to+\infty} f(t) = \lim_{s\to 0+} sF(s) \tag{III.10.16}$$

For any system $H(s)$, when the conditions of the Final Value Theorem are satisfied, the final value of its unit step response is:

$$\begin{aligned} y_{step,c}(+\infty) &= \lim_{s\to 0+} sY_{step,c}(s) \\ &= \lim_{s\to 0+} s\frac{1}{s}H_c(s) \\ &= \lim_{s\to 0+} H_c(s) \end{aligned} \tag{III.10.17}$$

For the closed-loop system $H_{1c}(s)$, we have:

$$\begin{aligned} y_{step,1c}(+\infty) &= \lim_{s\to 0+} H_{1c}(s) \\ &= \lim_{s\to 0+} \frac{1}{s+1} = 1 \end{aligned}$$

10.3.3 Typical Simple Systems

The two examples above are actually the simplest types of systems or the most basic building blocks/subsystems in complex systems. Such subsystems are also called "elements."

The subsystem corresponding to the differential equation $u = T\dot{y}$ is called an **integrating element**, or a first-order integrating system/integrator. Its transfer function is:

$$H_1(s) = \frac{1}{Ts}$$

Its unit step response is:

$$Y_{step,1}(s) = t$$

The subsystem corresponding to the differential equation $u-y = T\dot{y}(T>0)$ is called an **inertial element**, or a lag element, time-delay element, or first-order element. Its transfer function is:

$$H_2(s) = \frac{1}{Ts+1} \quad (T>0)$$

Its unit step response is:

$$Y_{step,2}(s) = 1(t) - e^{-\frac{t}{T}} \quad (T>0)$$

The subsystem corresponding to the differential equation $u - y = T^2\ddot{y} + 2\zeta T\dot{y}$ is called an **oscillatory element**, or a lag element, time-delay element, or second-order element. Its transfer function is:

$$H_3(s) = \frac{1}{T^2s^2 + 2\zeta Ts + 1}$$

Its unit step response is:

$$Y_{step,3}(s) = 1(t) - \frac{1}{\sqrt{1-\zeta^2}}e^{-\frac{\zeta}{T}t}\sin(\omega_d t + \theta)$$

Here, ω_d is called the **damped oscillation angular frequency**, and θ is the **damped oscillation angle**, with:

$$\omega_d = \frac{1}{T}\sqrt{1-\zeta^2}$$

$$\theta = \arctan\frac{\sqrt{1-\zeta^2}}{\zeta}$$

Among the three elements above, the unit step responses of the latter two converge, while the former does not.

In the inertial element, we assume $T > 0$ by default because the solution to the differential equation of such a system converges. If $T < 0$, the differential equation diverges. We call this a **first-order unstable element**:

$$H_4(s) = \frac{1}{Ts+1} \quad (T < 0)$$

Its unit step response is:

$$Y_{step,4}(s) = 1(t) - e^{-\frac{t}{T}} \quad (T < 0)$$

The inverse of the integrating element is called a **differentiating element** or first-order differentiating subsystem, corresponding to the differential equation $\dot{u} = Ty$. Its transfer function is:

$$H_5(s) = \frac{s}{T}$$

Its unit step response is:

$$Y_{step,5}(s) = \delta(t)$$

In the continuous domain, the differentiating element cannot be implemented alone. In practical systems, if a differentiating element is needed, other elements are used to approximate it.

All complex linear systems can be constructed by connecting linear elements and the above-mentioned elements in series or parallel.

10.4 Stability

10.4.1 Lyapunov Stability

Previously, we only provided an introductory explanation of system stability and pointed out that stability is a principle that all controller designs must adhere to, but we did not precisely define system stability. In this section, we formally present the definition of system stability and the Lyapunov criterion.

Note: In this section, the systems discussed are limited to autonomous systems. Controlled systems are not within the scope of this section.

Definition of Lyapunov Stability

Equilibrium point: x_0 is an equilibrium point of the autonomous system $\dot{x} = f(x)$ if and only if $x(0) = x_0$ implies $x(t) = x_0, \forall t > 0$.

Lyapunov stable: Suppose for the autonomous system $\dot{x} = f(x,t)$, $x = 0$ is its equilibrium point. If $\forall \epsilon > 0, \forall t_0, \exists \delta > 0, s.t. \forall x(t_0) = x_0 \in B(0,\delta)$, we have $||x(t)|| < \epsilon$, then the autonomous system $\dot{x} = f(x,t)$ is Lyapunov stable, or stable in the sense of Lyapunov.

Asymptotically stable: Suppose for the autonomous system $\dot{x} = f(x,t)$, $x = 0$ is its equilibrium point. If $\forall \epsilon > 0, \forall t_0, \exists \delta > 0, s.t. \forall x(t_0) = x_0 \in B(0,\delta)$, we have $\lim_{t\to\infty} ||x(t)|| = 0$, then the autonomous system $\dot{x} = f(x,t)$ is asymptotically stable.

Lyapunov Criterion

Lyapunov stability criterion: Suppose for the autonomous system $\dot{x} = f(x,t)$, $x = 0$ is its equilibrium point. If there exists a function $V(x)$ satisfying $V(0) = 0, V(x) \geq 0, \forall x$, and $\forall x(0) = x_0, \dot{x} = f(x,t), \dot{V}(x) \leq 0$, then the autonomous system $\dot{x} = f(x,t)$ is Lyapunov stable.

Asymptotic stability criterion: Suppose for the autonomous system $\dot{x} = f(x,t)$, $x = 0$ is its equilibrium point. If there exists a function $V(x)$ satisfying $V(0) = 0$, $V(x) > 0, \forall x \neq 0$, and $\forall x(0) = x_0, \dot{x} = f(x,t), \dot{V}(x) < 0$, then the autonomous system $\dot{x} = f(x,t)$ is asymptotically stable.

The $V(x)$ in the above two criteria is also called the Lyapunov function of the system. If a system satisfies a certain stability criterion, there may exist multiple Lyapunov functions simultaneously.

The Lyapunov criterion is also known as Lyapunov's second method. An intuitive understanding of it is: We need to find a semi-positive definite function $V(x)$ in the state space. When starting from any initial position, the autonomous system $\dot{x} = f(x,t)$ determines the trajectory of the system's evolution. If $V(x)$ along the trajectory is non-increasing, then the autonomous system is Lyapunov stable. If the function $V(x)$

is positive definite and $V(x)$ along the trajectory is monotonically decreasing, then the autonomous system is asymptotically stable.

The above intuitive understanding reveals a property of the Lyapunov function $V(x)$, namely that it can be regarded as a kind of system energy. If the system energy does not increase, it means the system can remain non-divergent. If the system energy continuously decreases, it will eventually return to the equilibrium point 0. For some systems with practical backgrounds, the Lyapunov function may indeed have physical significance.

(Reference: "Principles of Automatic Control")

10.5 Linear Time-Domain Tools

Many controlled dynamic systems in the real world are LTI SISO systems, such as controlling a displacement, the pressure of a gas, a temperature value, etc. The transfer functions and related controller design methods introduced earlier can already adequately meet such requirements.

However, with the advancement of human technology, the demands for system control have become more complex. Sometimes we need to simultaneously control multiple target states in a system rather than a single target state. These states are often not independent but are tightly coupled through differential equations. For example, in a motion system, simultaneously tracking given inputs for velocity, heading, and position. For such requirements, frequency-domain design tools like transfer functions become cumbersome and less intuitive.

To address such demands, we generally use **time-domain analysis tools** for system analysis and controller design. Specifically, we directly analyze the state equations and, combined with the Lyapunov stability criterion mentioned in the previous section, we have developed a series of methods and tools to meet the specified requirements.

As in the frequency domain, among all dynamic systems, linear systems are the simplest. Therefore, this section primarily introduces the description and properties of linear systems, along with the design of simple stabilizing controllers.

10.5.1 Linear System State Equations

As mentioned earlier, dynamic systems are divided into autonomous systems and controlled systems, described by differential equations. The simplest cases for both types are when the differential equations are linear, i.e., **linear autonomous systems** and **linear controlled systems**. A linear autonomous system is expressed as:

$$\begin{aligned} \dot{x}(t) &= A(t)x(t) \\ y(t) &= C(t)x(t) \end{aligned} \tag{III.10.18}$$

A linear controlled system is expressed as:

$$\begin{aligned} \dot{x}(t) &= A(t)x(t) + B(t)u(t) \\ y(t) &= C(t)x(t) + D(t)u(t) \end{aligned} \tag{III.10.19}$$

Here, we are describing time-varying linear systems. If the coefficients of the differential equations do not change over time, the system is called a linear time-invariant system (LTI system). This concept has already been introduced in the frequency domain. An LTI autonomous system is expressed as:

$$\begin{aligned} \dot{x}(t) &= Ax(t) \\ y(t) &= Cx(t) \end{aligned} \tag{III.10.20}$$

An LTI controlled system is expressed as:

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned} \tag{III.10.21}$$

In the above systems, the A matrix is called the **system matrix**, reflecting the inherent properties of the system; the B matrix is called the **input matrix**, reflecting the influence of the input on the system; the C matrix is called the **output matrix**, reflecting the relationship between the system output and the state;

the D matrix is called the **feedforward matrix**, representing the part of the input-output relationship that does not depend on the differential equation.

10.5.2 Linear System Stability

For LTI autonomous systems, we can use the Lyapunov stability introduced in the previous section to determine stability. We introduce two tools for judging Lyapunov stability of LTI autonomous systems: the **eigenvalue criterion** and the **Lyapunov equation criterion**.

Eigenvalue Criterion: For the LTI autonomous system $\dot{x} = Ax$, the equilibrium point $x = 0$ is Lyapunov stable if and only if all eigenvalues of matrix A lie in the left half-plane or on the imaginary axis of the complex plane, and the multiplicity of eigenvalues on the imaginary axis does not exceed 1.

The key to the eigenvalue criterion lies in the fact that the analytical solution of the LTI autonomous system's differential equation $\dot{x}(t) = Ax(t)$ is $x(t) = \exp(At)x_0$. Under the condition that A is diagonalizable, the matrix exponential $\exp(At)$ satisfies:

$$\exp(At) = P\exp(\Lambda t)P = P\begin{bmatrix} e^{\lambda_1 t} & & \\ & \ddots & \\ & & e^{\lambda_n t} \end{bmatrix} P^{-1}$$

where P is the unitary matrix generated during diagonalization. Therefore, if all eigenvalues of matrix A lie in the left half-plane or on the imaginary axis, and the multiplicity of eigenvalues on the imaginary axis does not exceed 1, each block of $\exp(\Lambda t)$ will converge to 0, ensuring that $x(t)$ remains bounded.

Lyapunov Equation Criterion: For the LTI autonomous system $\dot{x} = Ax$, the equilibrium point $x = 0$ is asymptotically stable if and only if $\forall Q > 0, Q^T = Q$, $\exists! P > 0, P^T = P$, satisfying:

$$A^T P + PA = -Q \tag{III.10.22}$$

Proof. **Sufficiency**:

For the system $\dot{x} = Ax$, consider the following quadratic Lyapunov function:

$$V(x) = \frac{1}{2}x^T Px$$

Since $P > 0$, $V(0) = 0$, and $V(x) > 0, \forall x \neq 0$. The derivative of the Lyapunov function is:

$$\begin{aligned} \dot{V}(x) &= \dot{x}^T Px + x^T P\dot{x} \\ &= x^T A^T Px + x^T PAx \\ &= x^T(A^T P + PA)x \\ &= -x^T Qx < 0 \end{aligned}$$

According to the Lyapunov stability criterion, the system $\dot{x} = Ax$ is Lyapunov stable.

Necessity:

For any $Q > 0, Q^T = Q$, consider the matrix function:

$$X(t) = \exp(A^T t)Q\exp(At)$$

It is clear that $X(0) = Q$. Taking its derivative, we have:

$$\begin{aligned} \dot{X}(t) &= A^T\exp(A^T t)Q\exp(At) + \exp(A^T t)Q\exp(At)A \\ &= A^T X(t) + X(t)A \end{aligned}$$

Integrating the above from $t = 0$ to $t = +\infty$, we obtain:

$$X(+\infty) - X(0) = A^T\int_0^{+\infty} X(t)dt + \int_0^{+\infty} X(t)dtA$$

Since the system is asymptotically stable, $\lim_{t\to+\infty}\exp(At) = 0$, and thus $\lim_{t\to+\infty} X(t) = 0$. Therefore, we have:

$$-Q = A^T \int_0^{+\infty} X(t)dt + \int_0^{+\infty} X(t)dtA$$

Let

$$P = \int_0^{+\infty} \exp(A^T t) Q \exp(At) dt \tag{III.10.23}$$

Then $-Q = A^T P + PA$. Since $P^T = P, P > 0$, this value is the required P, i.e., such a P exists and is unique.

□

(Reference: Zheng Dazhong, "Linear System Theory (2nd Edition)")

10.5.3 Controllability

In the first section of this chapter, we introduced the most general form of feedback control. Through feedback control, we can transform a controlled system into an autonomous system. In the previous section, we discussed the stability of autonomous LTI systems. Now, for a given controlled LTI system $\dot{x} = Ax + Bu$, can we transform it into a stable autonomous LTI system?

This question actually consists of two parts. First, we need to know whether all LTI autonomous systems can meet this requirement. Second, if the requirement can be met, how should we proceed? In this subsection, we first address the first question, i.e., defining the controllability of the system.

State Reachability: For the controlled LTI system $\dot{x} = Ax + Bu$, if there exists a time $t_1 > 0$ and a control signal $u(t), t \in [0, t_1]$, such that the system state can transition from $x(0) = x_0 \neq 0$ to $x(t_1) = 0$, then the state x_0 is said to be reachable.

System Controllability: For the controlled LTI system $\dot{x} = Ax + Bu$, if any non-zero state x_0 is reachable, the system is said to be controllable.

How can we determine whether a system is controllable? Here, we present the rank criterion for controllability without proof:

Controllability Rank Criterion: For the controlled LTI system $\dot{x} = Ax + Bu$, the system is completely controllable if and only if the rank of the controllability matrix defined below equals the dimension n of the system state x:

$$Q_c = \begin{bmatrix} B & AB & \dots & A^{n-1}B \end{bmatrix} \tag{III.10.24}$$

(Reference: Zheng Dazhong, "Linear System Theory (2nd Edition)")

10.5.4 Stabilizing Controller

Through the controllability criterion, we can already predict whether an LTI controlled system has the capability to be transformed into a stable autonomous system via feedback. But how exactly should we design the controller to achieve this requirement?

In the first part of this chapter, we introduced feedback controllers, which describe the control signal u as a function of the output $u = c(y)$. In fact, for systems described by state equations, a simpler approach is to express u as a function of the state x. For linear systems, we generally choose this function to be linear, i.e.,

$$u = -Kx \tag{III.10.25}$$

The above relationship is called **state feedback**. In fact, as long as the system is controllable, it can always be transformed into a stable autonomous system via state feedback, i.e., **system stabilization**. Let

$$A_a = A - BK \tag{III.10.26}$$

The stabilized system is then:

$$\dot{x} = A_a \dot{x} = (A - BK)x \tag{III.10.27}$$

According to the eigenvalue criterion for linear system stability, we only need to find a suitable K such that all eigenvalues of $A_a = A - BK$ lie in the left half-plane of the complex plane to ensure the closed-loop system is Lyapunov stable. This method is called **pole placement**.

The principle of pole placement is to linearly transform the A matrix into a controllable canonical form, find the matrix $\tilde{K}$ that places the poles of the canonical form at the desired locations, and then transform $\tilde{K}$ back to obtain K. The specific principle is not elaborated here; readers can refer to the documentation of MATLAB's 'place' function or other resources.

(Reference: "Automatic Control Principles")

10.6 First-order Inverted Pendulum Problem

The inverted pendulum is a classic nonlinear controlled system. Numerous chapters in this book will use the first-order inverted pendulum as an analysis object, which is introduced here.

10.6.1 Problem Modeling

First, let us define the first-order inverted pendulum.

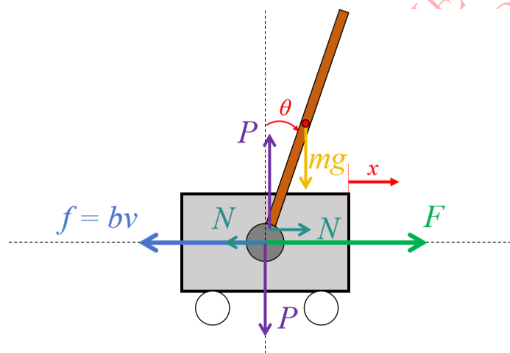

Figure III.10.1: First-order inverted pendulum

First-order inverted pendulum: As shown in Figure X, there is a horizontal track in the plane, on which a sliding cart with mass M is placed. The rightward direction along the track is defined as positive, and a certain position on the track is taken as the origin. During sliding, the cart's position x and velocity $\dot{x}$ are measurable, with friction $f = -\mu\dot{x}$. A uniform rod is connected to the cart via a freely rotating joint, with mass m_1 and length $2l_1$. The angle θ between the rod and the vertical direction and its angular velocity $\dot{\theta}$ are measurable, where the angle is positive when the rod tilts to the right. A continuous horizontal force F can be applied to the cart in some manner, constrained by $-F_{max} \leq F \leq F_{max}$. Let $\mathbf{x} = [x, \dot{x}, \theta, \dot{\theta}]$. Control objective: For initial system states within a certain range $x_{min} < x(0) < x_{max}$, design a controller to maintain the system state as close as possible to $x(t) = 0$.

Based on Newton's laws, the following equations can be derived:

$$\begin{aligned} M\ddot{x} &= F - \mu\dot{x} - N_x \\ m_1(\ddot{x} + l_1(-\sin\theta\dot{\theta}^2 + \cos\theta\ddot{\theta})) &= N_x \\ -m_1 l_1(\sin\theta\ddot{\theta} + \cos\theta\dot{\theta}^2) &= -m_1 g + N_y \\ \frac{1}{3}m_1 l_1^2\ddot{\theta} &= N_y l_1\sin\theta - N_x l_1\cos\theta \end{aligned}$$

Simplifying, we obtain:

$$\begin{aligned} (M + m_1)\ddot{x} - m_1 l_1\sin\theta\dot{\theta}^2 + m_1 l_1\cos\theta\ddot{\theta} + \mu\dot{x} &= F \\ \frac{4}{3}l_1\ddot{\theta} - g\sin\theta + \ddot{x}\cos\theta &= 0 \end{aligned} \tag{III.10.28}$$

It can be seen that this is a relatively complex system of nonlinear differential equations. Considering the small-angle approximation $\theta \to 0$, where $\sin\theta \to \theta$, $\cos\theta \to 1$, and $\dot{\theta}^2 \to 0$, we have the approximate first-order inverted pendulum equations:

$$(M + m_1)\ddot{x} + \mu\dot{x} + m_1 l_1 \ddot{\theta} = F$$
$$\frac{4}{3} l_1 \ddot{\theta} - g\theta + \ddot{x} = 0$$

Rearranging, we get:

$$\begin{aligned} \ddot{x} &= -\mu k_1 x + k_2 \theta + k_1 F \\ \ddot{\theta} &= -\mu\dot{x} + k_3 \theta + k_4 F \end{aligned} \tag{III.10.29}$$

where

$$\begin{aligned} k_1 &= \frac{4}{4M + m_1} \\ k_2 &= -\frac{3m_1 g}{4M + m_1} \\ k_3 &= \frac{3(M + m_1)g}{l_1(4M + m_1)} \\ k_4 &= -\frac{3}{l_1(4M + m_1)} \end{aligned} \tag{III.10.30}$$

Let $u = F$, and express it in matrix form:

$$\dot{\mathbf{x}} = A\mathbf{x} + Bu = \begin{bmatrix} 0 & -\mu k_1 & 0 & k_2 \\ 0 & 1 & 0 & 0 \\ -\mu & 0 & 0 & k_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{x} + \begin{bmatrix} k_1 \\ 0 \\ k_4 \\ 0 \end{bmatrix} u \tag{III.10.31}$$

10.6.2 Stability and Controllability Analysis

[This section will be updated in a future version. Stay tuned.]

10.6.3 PID Control of the Inverted Pendulum

For the above inverted pendulum problem, we can implement a simple controller using PID and verify its performance in a simulation environment.

[This section will be updated in a future version. Stay tuned.]

11 Observation and Filtering

11.1 Basic Concepts

Observation and filtering are two crucial problems in control theory and engineering. They are related yet distinct. In this section, we will formally define these two types of problems.

11.1.1 State Observation Problem

In the derivations of the previous section, particularly in the time-domain linear control derivations, we often assume that the system states are known. However, in reality, the states of some practical systems are not easily obtainable. We may only have access to the output quantity/**observed quantity** $z(t)$ defined by the output equation[7].

Its continuous form is:

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t)) \\ z(t) &= h(x(t), u(t)) \end{aligned} \tag{III.11.1}$$

The discrete form is:

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) \\ z_k &= h(x_k, u_k) \end{aligned} \tag{III.11.2}$$

The **state observation problem** refers to the scenario where, when certain conditions (observability) are satisfied, although the system's state variables are unmeasurable, we can design algorithms to reconstruct the state variables based on the system output z, system state equation f, and system output equation h, thereby achieving stability of the closed-loop error system.

Problem	**State Observation Problem**
Brief Description	Given the system state and output equations, design an observer to ensure observation error convergence
Known	System equation $\dot{x} = f(x, u)$ Observation equation $z = h(x, u)$ System input/output u, z
To Find	Observer $o(u, z, \hat{x})$

The simplest state observation problem is the state observation problem with linear system and observation equations, known as the **linear state observation problem**.

Problem	**Linear State Observation Problem**
Brief Description	Given linear system state and output equations, design an observer to ensure observation error convergence
Known	Linear system equation $\dot{x} = Ax + Bu$ Linear observation equation $z = Cx$ System input/output u, z
To Find	Observer $o(u, z, \hat{x})$

11.1.2 Filtering Problem

Ideal systems are deterministic, while practical systems may contain various **random noises**. Noise refers to a class of small but influential random variables affecting internal and output signals. The most common noise is **Gaussian noise**, which is essentially an n-dimensional random vector v following the distribution $v \sim \mathcal{N}(\mu, \Sigma)$, with the corresponding probability density function:

[7] The filtering community conventionally uses $z(t)$ to represent observed quantities, while the control community prefers $y(t)$. In this chapter, we consistently use $z(t)$.

$$p(v;\mu,\Sigma)=\frac{1}{\sqrt{(2\pi)^k|\Sigma|}}\exp\left(-\frac{1}{2}(v-\mu)^T\Sigma^{-1}(v-\mu)\right) \tag{III.11.3}$$

If the Gaussian noise has mean $\mu=0$, it is called **Gaussian white noise**, with its probability density function given by:

$$p(v;0,\Sigma)=\frac{1}{\sqrt{(2\pi)^k|\Sigma|}}\exp\left(-\frac{1}{2}v^T\Sigma^{-1}v\right) \tag{III.11.4}$$

Similar to disturbances, noise is something we aim to remove, a process called **denoising**. Due to historical reasons, the term **filtering** carries two denoising-related meanings. On one hand, filtering represents **denoising signal recovery**, i.e., recovering the original signal from noisy signals under conditions where noise and signal characteristics are known/partially known/unknown. In fields like communications, signal processing, and image processing, filtering typically carries this "signal recovery" meaning. On the other hand, filtering also represents **denoising state estimation**, i.e., observing the state based on outputs in noisy systems to attempt state recovery.

It can be seen that the latter meaning is closer to state observation under denoising, while the former can be viewed as a special case of the latter[8]. In robotics, we are more concerned with the latter meaning of filtering. **Unless otherwise specified, all subsequent mentions of "filtering" in this book refer to denoising state estimation.**

Denoising state estimation problems are typically expressed in the discrete-time domain as:

$$\begin{aligned}x_{k+1}&=f(x_k,u_k)+w_k\\z_k&=h(x_k,u_k)+v_k\end{aligned} \tag{III.11.5}$$

That is:

Problem	**Denoising State Estimation (Filtering) Problem [Discrete]**
Brief Description	Given discrete system state and output equations, design a filter to minimize state estimation error
Known	Discrete system $x_{k+1}=f(x_k,u_k)+w_k$ Observation equation $z_k=h(x_k,u_k)+v_k$ System input/output u_k,z_k
To Find	Filter $\hat{x}_k=o(u_k,z_k,\hat{x}_{k-1})$

When the system contains random noise, both the true system state x_k and the filter-estimated state $\hat{x}_k$ are random variables. To evaluate filtering performance, we can define the estimation error:

$$e_k=x_k-\hat{x}_k \tag{III.11.6}$$

Thus, we can define the "optimal" estimation - minimizing the (expected) **squared error**:

$$\begin{aligned}\min_{o}\quad&\mathbb{E}[e_k^Te_k]\\s.t.&x_{k+1}=f(x_k,u_k)+w_k\\&z_k=h(x_k,u_k)+v_k\\&\hat{x}_k=o(u_k,z_k,\hat{x}_{k-1})\\&e_k=x_k-\hat{x}_k\end{aligned} \tag{III.11.7}$$

We summarize this problem as the **optimal filtering problem** in the minimum squared error sense:

[8] i.e., when the output z and state x are identical

Problem	Optimal Filtering Problem [Discrete]
Brief Description	Given discrete system state and output equations, design a filter to minimize state estimation error
Known	Discrete system $x_{k+1} = f(x_k, u_k) + w_k$ Observation equation $z_k = h(x_k, u_k) + v_k$ System input/output u_k, z_k
To Find	Filter $\hat{x}_k = o(u_k, z_k, \hat{x}_k)$ that minimizes $\mathbb{E}[e_k^T e_k]$

The above problem can also be expressed in the continuous domain. Note that in the continuous domain, the noises $w(t), v(t)$ are continuous-time white noises satisfying:

$$\begin{aligned} \mathbb{E}[w(t)] &= 0 \\ \mathbb{E}[w(t)w(\tau)^T] &= Q(t)\delta(t-\tau) \\ \mathbb{E}[v(t)] &= 0 \\ \mathbb{E}[v(t)v(\tau)^T] &= R(t)\delta(t-\tau) \\ \mathbb{E}[w(t)v(\tau)] &= 0 \end{aligned} \tag{III.11.8}$$

Here, the $\delta(\cdot)$ function is the unit impulse function introduced in Section 10.2.

Thus, we can summarize the continuous-time filtering problem as follows:

Problem	Denoising State Estimation (Filtering) Problem [Continuous]
Problem Description	Given the continuous system state equation and output equation, design a filter to minimize state estimation error
Given	Continuous system $\dot{x} = f(x, u) + w$ Observation equation $z = h(x, u) + v$ System input/output u, z
Find	Filter $o(u, z, \hat{x})$

Similar to the discrete domain, in the continuous domain, we also define the optimal filtering problem under the expectation of squared error:

$$\begin{aligned} \min_o \quad & \mathbb{E}[e(t)^T e(t)] \\ s.t. \quad & \dot{x}(t) = f(x(t), u(t)) + w(t) \\ & z(t) = h(x(t), u(t)) + v(t) \\ & \dot{\hat{x}}(t) = o(u(t), z(t), \hat{x}(t)) \\ & e(t) = x(t) - \hat{x}(t) \end{aligned} \tag{III.11.9}$$

We summarize this optimal filtering problem as follows:

Problem	Optimal Filtering Problem [Continuous]
Problem Description	Given the continuous system state equation and output equation, design a filter to minimize state estimation error
Given	Continuous system $\dot{x} = f(x, u, w)$ Observation equation $z = h(x, u) + v$ System input/output u, z
Find	Filter $\hat{x} = o(u, z, \hat{x})$ that minimizes $\mathbb{E}[e(t)^T e(t)]$

11.2 Linear State Observer

First, in the noise-free case, we consider the state observation problem for linear systems.

11.2.1 Observability

Similar to the previous chapter, before discussing specific state observation algorithms, we must first determine whether the system is observable. Again, we directly present two key definitions:

Unobservability of State: For an LTI system $\dot{x} = Ax + Bu$, $y = Cx + Du$, if there exists a nonzero state $x_0 \neq 0$ such that for any time $t_1 > 0$, when the initial state $x(0) = x_0$, the system's output response $y(t) \equiv 0$ (for all $t \in [0, t_1]$), then the state x_0 is said to be unobservable.

Complete Observability of System: For an LTI system $\dot{x} = Ax + Bu$, $y = Cx + Du$, if the system does not contain any unobservable nonzero states, i.e., any nonzero initial state $x_0 \neq 0$ will produce a nonzero output response, then the system is said to be completely observable.

Partial Observability of System: If the system is not completely observable, i.e., there exists at least one unobservable nonzero state, then the system is said to be partially observable.

Dual to controllability, for observability, we also have a rank criterion:

Observability Rank Criterion: For an LTI system $\dot{x} = Ax+Bu$, $y = Cx+Du$, the system is completely observable if and only if the rank of the following defined observability matrix equals the dimension n of the system state x:

$$Q_o = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1} \end{bmatrix} \tag{III.11.10}$$

11.2.2 Luenberger Observer

[Main content: Luenberger observer; Linear decoupling principle; Output feedback pole placement]

[This section will be updated in subsequent versions. Stay tuned.]

(Reference: Zheng Dazhong "Linear System Theory (2nd Edition)")

11.3 Kalman Filter (KF)

11.3.1 Linear Filtering Problem

The previous section introduced the state estimation (filtering) problem with noise removal. State estimation for general nonlinear systems is challenging, and becomes even more complex when noise is superimposed. Therefore, we first consider the simpler scenario of linear systems with additive noise, known as the **linear filtering problem**.

Problem	**Linear Filtering Problem [Discrete]**
Description	Given the state and output equations of a discrete linear system, design a filter to minimize state estimation error
Given	Discrete linear system $x_{k+1} = Ax_k + Bu_k + w_k$ Observation equation $z_k = Cx_k + v_k$ System input/output u_k, z_k
Objective	Filter $\hat{x}_k = o(z_k, \hat{x}_{k-1}, u_k)$

Random noise is the main challenge in filtering problems, as different noises have different distribution characteristics. Among all noise types, the simplest is Gaussian white noise. For example, we can assume both system noise and output noise satisfy Gaussian white noise:

$$\begin{aligned} w_k &\sim \mathcal{N}(0, Q_k) \\ v_k &\sim \mathcal{N}(0, R_k) \end{aligned} \tag{III.11.11}$$

Here, R_k and Q_k are symmetric positive definite matrices. As mentioned above, under the criterion of minimizing squared error, we can also define the **linear optimal filtering problem**:

Problem	Linear Optimal Filtering Problem [Discrete]
Description	Given the state and output equations of a discrete linear system, design a filter to minimize state estimation variance
Given	Discrete linear system $x_{k+1} = Ax_k + Bu_k + w_k$ Observation equation $z_k = Cx_k + v_k$ System input/output u_k, z_k System noise distribution $w_k \sim \mathcal{N}(0, Q_k)$ Output noise distribution $v_k \sim \mathcal{N}(0, R_k)$
Objective	Filter $\hat{x}_k = o(z_k, \hat{x}_{k-1}, u_k)$ that minimizes $\mathbb{E}[e_k^T e_k]$

Similarly, these two problems can also be expressed in the continuous domain:

Problem	Linear Filtering Problem [Continuous]
Description	Given the state and output equations of a continuous linear system, design a filter to minimize state estimation error
Given	Continuous linear system $\dot{x} = Ax + Bu + w$ Observation equation $z = Cx + v$ System input/output u, z
Objective	Filter $\hat{x} = o(z, \hat{x}, u)$

Problem	Linear Optimal Filtering Problem [Continuous]
Description	Given the state and output equations of a continuous linear system, design a filter to minimize state estimation variance
Given	Continuous linear system $\dot{x} = Ax + Bu + w$ Observation equation $z = Cx + v$ System input/output u, z System noise distribution $w(t) \sim \mathcal{N}(0, Q(t))$ Output noise distribution $v(t) \sim \mathcal{N}(0, R(t))$
Objective	Filter $\hat{x} = o(z, \hat{x}, u)$ that minimizes $\mathbb{E}[e(t)^T e(t)]$

11.3.2 Discrete Kalman Filter

Consider the linear optimal filtering problem for the discrete system described above. We define the estimated state of the filter as $\hat{x}_k$. Based on the system equation, we can make an a priori prediction for the state at step $k+1$, i.e., the intermediate state $\hat{x}_k^-$:

$$\hat{x}_k^- = A\hat{x}_k + Bu_k$$

At this point, we define the state estimation error:

$$\begin{aligned} e_k &:= x_k - \hat{x}_k \\ e_{k+1}^- &:= x_{k+1} - \hat{x}_k^- \end{aligned} \tag{III.11.12}$$

Then we have:

$$\begin{aligned} e_{k+1}^- &= x_{k+1} - \hat{x}_k^- \\ &= Ax_k + Bu_k + w_k - (A\hat{x}_k + Bu_k) \\ &= A(x_k - \hat{x}_k) + w_k \\ &= Ae_k + w_k \end{aligned}$$

We also define the error covariance:

$$\begin{aligned} P_k &:= \mathbb{E}[e_k e_k^T] \\ P_{k+1}^- &:= \mathbb{E}[(e_{k+1}^-)(e_{k+1}^-)^T] \end{aligned} \tag{III.11.13}$$

This leads to the relationship between error covariances[9]:

$$\begin{aligned}
P_{k+1}^- &= \mathbb{E}[(e_{k+1}^-)(e_{k+1}^-)^T] \\
&= \mathbb{E}[(Ae_k + w_k)(Ae_k + w_k)^T] \\
&= \mathbb{E}[Ae_k e_k^T A^T] + \mathbb{E}[w_k w_k^T] \\
&= A\mathbb{E}[e_k e_k^T]A^T + \mathbb{E}[w_k w_k^T] \\
&= AP_k A^T + Q_k
\end{aligned}$$

Now, we consider correcting the intermediate state $\hat{x}_k^-$ based on the observation z_k. From the intermediate state $\hat{x}_k^-$ and the system output equation, the predicted output is $C\hat{x}_k^-$. The difference between the observed value and this prediction gives the **observation residual** $\tilde{z}_k$, also known as the **innovation**:

$$\tilde{z}_k = z_k - C\hat{x}_k^- \tag{III.11.14}$$

We boldly assume that the state correction of the optimal filter is linearly related to the observation residual:

$$\hat{x}_{k+1} - \hat{x}_k^- = K\tilde{z}_k$$

Here, K is the matrix representing the linear relationship between the observation residual and the state correction. Thus, we obtain the recursive expression for the residual:

$$\begin{aligned}
e_{k+1} &= x_{k+1} - \hat{x}_{k+1} \\
&= x_{k+1} - \hat{x}_k^- - K\tilde{z}_k \\
&= x_{k+1} - \hat{x}_k^- - K(Cx_k + v_k - C\hat{x}_k^-) \\
&= (I - KC)(x_{k+1} - \hat{x}_k^-) - Kv_k \\
&= (I - KC)e_{k+1}^- - Kv_k
\end{aligned}$$

Therefore, the recursive relationship for the residual can be summarized as:

$$\begin{aligned}
e_{k+1}^- &= Ae_k + w_k \\
e_{k+1} &= (I - KC)e_{k+1}^- - Kv_k
\end{aligned} \tag{III.11.15}$$

We continue to derive the recursive relationship for the residual covariance matrix.

$$\begin{aligned}
P_{k+1} &= \mathbb{E}[(e_{k+1})(e_{k+1})^T] \\
&= \mathbb{E}[((I - KC)e_{k+1}^- - Kv_k)((I - KC)e_{k+1}^- - Kv_k)^T] \\
&= (I - KC)\mathbb{E}[(e_{k+1}^-)(e_{k+1}^-)^T](I - KC)^T + K\mathbb{E}[v_k v_k^T]K^T \\
&= (I - KC)P_{k+1}^-(I - KC)^T + KR_k K^T
\end{aligned}$$

Therefore, the recursive relationship of the residual covariance matrix can be summarized as follows:

$$\begin{aligned}
P_{k+1}^- &= AP_k A^T + Q_k \\
P_{k+1} &= (I - KC)P_{k+1}^-(I - KC)^T + KR_k K^T
\end{aligned} \tag{III.11.16}$$

Since we aim to solve for the optimal filter, our focus is on the optimization objective $\mathbb{E}[e_k^T e_k]$:

$$\begin{aligned}
\mathbb{E}[e_{k+1}^T e_{k+1}] &= \mathrm{tr}(\mathbb{E}[e_{k+1}e_{k+1}^T)] = \mathrm{tr}(\mathbb{E}[P_{k+1}]) \\
&= \mathrm{tr}((I - KC)P_{k+1}^-(I - KC)^T + KR_k K^T) \\
&= \mathrm{tr}(P_{k+1}^- - KCP_{k+1}^- - P_{k+1}^- C^T K^T + K(CP_{k+1}^- C^T + R_k)K^T) \\
&= \mathrm{tr}(V(K))
\end{aligned}$$

[9]Here, we use the conclusion that w_k is independent of e_k

It can be observed that we have expressed the optimization objective as a function of K. The K_k that minimizes this function corresponds to the optimal filter we seek, i.e.,

$$K_k = \min_K \text{tr}(V(K))$$

$\text{tr}(V(K))$ is a quadratic form with respect to the variable K, twice differentiable in K, and has a unique global minimum. When this minimum is attained, the first derivative equals zero:

$$\left.\frac{\partial \text{tr}(V(K))}{\partial K}\right|_{K_k} = 0$$

Since

$$\frac{d\text{tr}(BAC)}{dA} = B^T C^T$$

we have

$$\frac{\partial}{\partial K}\text{tr}(K(CP_{k+1}^- C^T + R_k)K^T) = 2K(CP_{k+1}^- C^T + R_k)$$
$$\frac{\partial}{\partial K}\text{tr}(-KCP_{k+1}^- - P_{k+1}^- C^T K^T) = -2P_{k+1}^- C^T$$

Thus, the filter gain K_k of the optimal filter must satisfy the following linear equation:

$$2K_k(CP_{k+1}^- C^T + R_k) - 2P_{k+1}^- C^T = 0$$

Rearranging yields:

$$K_k(CP_{k+1}^- C^T + R_k) = P_{k+1}^- C^T$$

That is,

$$K_k = P_{k+1}^- C^T (CP_{k+1}^- C^T + R_k)^{-1} \tag{III.11.17}$$

Summarizing the above equations, we obtain the optimal filtering algorithm. This algorithm is also known as the **Kalman Filter** (KF), where K_k is called the **Kalman gain**. The core of the Kalman filter consists of five equations:

$$\begin{aligned}
\hat{x}_k^- &= A\hat{x}_k + Bu_k \\
P_{k+1}^- &= AP_k A^T + Q_k \\
K_k &= P_{k+1}^- C^T (CP_{k+1}^- C^T + R_k)^{-1} \\
\hat{x}_{k+1} &= \hat{x}_k^- + K_k(z_k - C\hat{x}_k^-) \\
P_{k+1} &= K_k R_k K_k^T + (I - K_k C)P_{k+1}^- (I - K_k C)^T
\end{aligned} \tag{III.11.18}$$

We present the complete algorithm for discrete Kalman filtering as follows:

Algorithm	**Discrete Kalman Filter**
Problem Type	Linear Optimal Filtering Problem [Discrete]
Given	Discrete linear system $x_{k+1} = Ax_k + Bu_k + w_k$ Observation equation $z_k = Cx_k + v_k$ System input/output u_k, z_k System noise distribution $w_k \sim \mathcal{N}(0, Q_k)$ Output noise distribution $v_k \sim \mathcal{N}(0, R_k)$
Objective	Filter $o(z, \hat{x}, u)$ minimizing $\mathbb{E}[\hat{x}]$
Algorithm Property	Filter, recursive

Algorithm 24: Discrete Kalman Filter

Input: Linear system matrices A, B, C
Input: Sequences of symmetric positive definite matrices Q_k, R_k, symmetric positive definite matrix P_0
Input: System input sequence u_k, system output sequence z_k
Input: System initial value $\hat{x}_0$
Output: Estimated state $\hat{\mathbf{x}}$
for $k \in 0, 1, \ldots, N_K$ **do**
$\hat{x}_k^- \leftarrow A\hat{x}_k + Bu_k$
$P_{k+1}^- \leftarrow AP_kA^T + Q_k$
$K_k \leftarrow P_{k+1}^- C^T (CP_{k+1}^- C^T + R_k)^{-1}$
$\hat{x}_{k+1} \leftarrow \hat{x}_k^- + K_k(z_k - C\hat{x}_k^-)$
$P_{k+1} \leftarrow K_k R_k K_k^T + (I - K_k C)P_{k+1}^- (I - K_k C)^T$

The corresponding Python code is shown below:

```python
def filter_KF_disc(x_0, us, zs, A, B, C, Qs, Rs, P_0, N_K:int):
    assert len(us) == N_K, len(zs) == N_K
    assert len(Rs) == N_K, len(Qs) == N_K
    Ks, xs, Pk = [], [x_0], P_0
    for k in range(N_K+1):
        xk_ = A @ xs[k] + B @ us[k]
        Pk_ = A @ Pk @ A.T + Qs[k]
        tmp = np.linalg.inv(Rs[k] + C @ Pk_ @ C.T)
        Ks.append(Pk_ @ C.T @ tmp)
        xs.append(xk_ + Ks[k] @ (zs[k] - C @ xk_))
        tmp2 = np.eye(x_0.shape[0]) - Ks[k] @ C
        Pk_ = Ks[k] @ Rs[k] @ Ks[k].T + tmp2 @ Pk_ @ tmp2.T
    return xs
```

11.3.3 Continuous Kalman Filter

For the linear optimal filtering problem in continuous systems, we can formulate the optimization problem as:

$$\begin{aligned} &\min_o \mathbb{E}[e(t)^T e(t)] \\ &s.t. \dot{x} = Ax + Bu + w \\ &z = Cx + v \\ &\dot{\hat{x}} = o(u, z, \hat{x}) \\ &e(t) = x - \hat{x} \end{aligned} \tag{III.11.19}$$

Following the discrete case, we assume the optimal filter still takes the form of innovation update.

$$\dot{\hat{x}} = A\hat{x} + Bu + K(z - C\hat{x}) \tag{III.11.20}$$

Thus, we obtain the error differential equation:

$$\begin{aligned} \dot{e}(t) &= \dot{x}(t) - \dot{\hat{x}}(t) \\ &= Ax(t) + Bu(t) + w(t) - (A\hat{x}(t) + Bu(t) + K(Cx(t) + v(t) - C\hat{x}(t))) \\ &= A(x(t) - \hat{x}(t)) - KC(x(t) - \hat{x}(t)) + w(t) - Kv(t) \\ &= (A - KC)(x(t) - \hat{x}(t)) + w(t) - Kv(t) \\ &= (A - KC)e(t) + w(t) - Kv(t) \end{aligned}$$

Assuming the error covariance is P, i.e.,

$$P(t) = \mathbb{E}[e(t)e(t)^T] \tag{III.11.21}$$

the differential equation for $P(t)$ is[10]:

$$\begin{aligned}
\dot{P}(t) &= \mathbb{E}[\dot{e}(t)e(t)^T + e(t)\dot{e}(t)^T + \dot{e}(t)\dot{e}(t)^T] \\
&= \mathbb{E}[(A-KC)e(t)e(t)^T] + \mathbb{E}[e(t)((A-KC)e(t))^T] + \mathbb{E}[(w(t)-Kv(t))(w(t)-Kv(t))^T] \\
&= (A-KC)P(t) + P(t)(A-KC)^T + Q(t) + KR(t)K \\
&= -KCP(t) - P(t)C^TK + KR(t)K + AP(t) + P(t)A^T + Q(t) \\
&= V(K) + AP(t) + P(t)A^T + Q(t)
\end{aligned}$$

To minimize $\mathbb{E}[e(t)^Te(t)]$, we have the following equivalent relationships:

$$\begin{aligned}
\min_K \mathbb{E}[e(t)^Te(t)] &= \min_K \mathbb{E}[\mathrm{tr}(e(t)e(t)^T)] \\
&= \min_K \mathbb{E}[\mathrm{tr}(P(t))] \\
&= \min_K \mathbb{E}[\mathrm{tr}(\dot{P}(t))] \\
&= \min_K \mathbb{E}[\mathrm{tr}(V(K))]
\end{aligned}$$

Here, we use the conclusion that $\dot{P}(t) > 0$. Therefore, the optimal gain K satisfies the following partial derivative condition:

$$\frac{\partial V}{\partial K} = \frac{\partial}{\partial K}(-KCP(t) - P(t)C^TK + KR(t)K) = 0$$

That is,

$$-2P(t)C^T + 2KR(t) = 0$$

Thus,

$$K = P(t)C^TR^{-1}(t) \tag{III.11.22}$$

At this point,

$$\begin{aligned}
V(K) &= -KCP(t) - P(t)C^TK + KR(t)K^T \\
&= -2P(t)C^TR^{-1}(t)CP(t) + P(t)C^TR^{-1}(t)CP(t) \\
&= -P(t)C^TR^{-1}(t)CP(t)
\end{aligned} \tag{III.11.23}$$

Therefore, we have:

$$\dot{P}(t) = -P(t)C^TR^{-1}(t)CP(t) + AP(t) + P(t)A^T + Q(t) \tag{III.11.24}$$

In summary, the equations for the continuous Kalman filter algorithm are as follows:

$$\begin{aligned}
\dot{\hat{x}} &= A\hat{x} + Bu + K(z - C\hat{x}) \\
\dot{P}(t) &= -P(t)C^TR^{-1}(t)CP(t) + AP(t) + P(t)A^T + Q(t) \\
K &= P(t)C^TR^{-1}(t)
\end{aligned} \tag{III.11.25}$$

In practical applications, initial values $\hat{x}(0)$ and $P(0)$ must be specified for these two differential equations, which are then solved iteratively.

[10] Here, the second-order terms related to P are ignored

Algorithm	Continuous Kalman Filter
Problem Type	Linear Optimal Filtering [Continuous]
Given	Continuous linear system $\dot{x} = Ax + Bu + w$ Observation equation $z = Cx + v$ System input/output u, z System noise distribution $w(t) \sim \mathcal{N}(0, Q(t))$ Output noise distribution $v(t) \sim \mathcal{N}(0, R(t))$
Objective	Design a filter $o(z, \hat{x}, u)$ to minimize $\mathbb{E}[\hat{x}]$
Algorithm Properties	Filter, recursive

Algorithm 25: Continuous Kalman Filter

Input: Linear system matrices A, B, C
Input: Symmetric positive definite matrix functions $Q(t), R(t)$, symmetric positive definite matrix $P(0)$
Input: System input sequence u_k, system output sequence z_k
Input: System initial state $\hat{x}(0)$
Output: Estimated state $\hat{\mathbf{x}}$
$\dot{\hat{x}} = A\hat{x} + Bu + K(z - C\hat{x})$
$\dot{P}(t) = -P(t)C^T R^{-1}(t)CP(t) + AP(t) + P(t)A^T + Q(t)$
$K = P(t)C^T R^{-1}(t)$

11.3.4 Extended Kalman Filter (EKF)

In the previous subsection, we introduced the Kalman filter method for minimizing variance in linear filtering problems.

For nonlinear filtering problems, the noise covariance changes become highly complex, making the Kalman filter inapplicable. Specifically, the state equation f and observation equation h introduce challenges. However, since the Kalman filter operates iteratively, when the sampling time is sufficiently small, the state changes within each time step are relatively minor. In this case, the nonlinear f and h can be linearized locally using first-order Taylor expansions, approximating the nonlinear variance changes as linear system dynamics, thereby allowing the continued use of the Kalman filter method.

Specifically, assume the discrete system equations are:

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) \\ z_{k+1} &= h(x_k) \end{aligned} \tag{III.11.26}$$

Let:

$$\begin{aligned} F_k &= \frac{\partial f}{\partial x}(x_k) \\ H_k &= \frac{\partial h}{\partial x}(x_k) \end{aligned} \tag{III.11.27}$$

The above equations can then be approximated as:

$$\begin{aligned} x_{k+1} &\approx f(x_k, u_k) + F_k(x_{k+1} - x_k) \\ z_{k+1} &\approx H_k x_k \end{aligned} \tag{III.11.28}$$

That is, the matrices A, B, C in the Kalman filter can be approximated by the system's first-order derivatives F_k, B_k, H_k. Thus, we derive the **Extended Kalman Filter** (EKF) algorithm:

$$
\begin{aligned}
\hat{x}_k^- &= f(\hat{x}_k, u_k) \\
P_{k+1}^- &= F_k P_k F_k^T + Q_k \\
K_k &= P_{k+1}^- H_k^T (H_k P_{k+1}^- H_k^T + R_k)^{-1} \\
\hat{x}_{k+1} &= \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, u_k)) \\
P_{k+1} &= K_k R_k K_k^T + (I - K_k H_k) P_{k+1}^- (I - K_k H_k)^T
\end{aligned} \tag{III.11.29}
$$

The EKF extends the KF to nonlinear systems. Compared to the KF, the EKF uses nonlinear equations for prediction and update, while covariance updates and Kalman gain calculations still rely on linearized matrices.

Algorithm 26: Extended Kalman Filter

Input: Differentiable system and output equations f, h
Input: Sequence of symmetric positive definite matrices Q_k, R_k, symmetric positive definite matrix P_0
Input: System input sequence u_k, system output sequence z_k
Input: Initial system value $\hat{x}_0$
Output: Estimated state $\hat{\mathbf{x}}$
for $k \in 0, 1, \ldots, K$ **do**
$\hat{x}_k^- \leftarrow f(\hat{x}_k, u_k)$
$F_k \leftarrow \frac{\partial f}{\partial x}(\hat{x}_k)$
$P_{k+1}^- \leftarrow F_k P_k F_k^T + Q_k$
$H_k \leftarrow \frac{\partial h}{\partial x}(\hat{x}_k)$
$K_k \leftarrow P_{k+1}^- H_k^T (H_k P_{k+1}^- H_k^T + R_k)^{-1}$
$\hat{x}_{k+1} \leftarrow \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, u_k))$
$P_{k+1} \leftarrow K_k R_k K_k^T + (I - K_k H_k) P_{k+1}^- (I - K_k H_k)^T$

Algorithm	**Extended Kalman Filter**
Problem Type	Optimal Filtering Problem [Discrete]
Known	Discrete system $x_{k+1} = f(x_k, u_k, w_k)$ Observation equation $z_k = h(x_k, u_k, v_k)$ System input/output u_k, z_k System noise distribution $w_k \sim \mathcal{N}(0, Q_k)$ Output noise distribution $v_k \sim \mathcal{N}(0, R_k)$
Find	Filter $\hat{x} = o(u, z, \hat{x})$ that minimizes $\mathrm{Var}[\hat{x}]$
Algorithm Properties	Filter, Recursive

The corresponding Python code is shown below

```python
def filter_EKF(x_0, us, zs, f_func, df_func, h_func, dh_func, Qs, Rs, P_0, N_K:int):
    assert len(us) == N_K, len(zs) == N_K
    assert len(Rs) == N_K, len(Qs) == N_K
    Ks, xs, Pk = [], [x_0], P_0
    for k in range(N_K+1):
        xk_ = f_func(xs[k], us[k])
        Fk = df_func(xs[k], us[k])
        Pk_ = Fk @ Pk @ Fk.T + Qs[k]
        Hk = dh_func(xs[k])
        tmp = np.linalg.inv(Rs[k] + Hk @ Pk_ @ Hk.T)
        Ks.append(Pk_ @ Hk.T @ tmp)
        xs.append(xk_ + Ks[k] @ (zs[k] - h_func(xs[k])))
        tmp2 = np.eye(x_0.shape[0]) - Ks[k] @ Hk
        Pk = Ks[k] @ Rs[k] @ Ks[k].T + tmp2 @ Pk_ @ tmp2.T
    return xs
```

11.3.5 Error State Kalman Filter (ESKF)

For many nonlinear problems, the Extended Kalman Filter is a good approach. However, when the state itself is constrained, this method may cause some issues.

For example, a common requirement in robotics is to represent the pose of an end-effector/body. Poses are typically represented using either a 9-element rotation matrix R or a 4-element quaternion q. However, 3D rotation has only 3 degrees of freedom, and the redundant dimensions imply normalization constraints, i.e., $R^T R = I$ or $||q|| = 1$.

In such cases, using the EKF method described above, where derivatives of the state $\frac{\partial f}{\partial x}, \frac{\partial h}{\partial x}$ are computed and used for state updates, may lead to issues like broken normalization or slow convergence.

In fact, in Section 2.5, we have already discussed this problem and proposed a solution using small perturbations of rotation vectors $\Delta\phi$ instead of ΔR (similarly for Δq). This essentially uses tangent space perturbations to approximate perturbations on the manifold, which can be viewed as a linearization of the manifold itself.

More generally, for filtering problems where x may lie on a manifold with inherent constraints, we can use perturbations δx in its tangent space as the optimization variables for filtering. In filtering problems, δx can be considered as the error between the estimated value and the true value. Filtering methods constructed using this idea are called **Error State Kalman Filter** (ESKF). The estimated state is also called the **Nominal State**.

Specifically, we consider that the system has a true state x, while the filtering algorithm can only obtain an estimate $\hat{x}$. Both x and $\hat{x}$ are points on some manifold and are very close to each other. On the tangent plane of the manifold near $\hat{x}$ (e.g., rotation vectors corresponding to rotation matrices), we can define a perturbation δx and aim to find δx that satisfies the following relationship:

$$x = \hat{x} \oplus \delta x \tag{III.11.30}$$

Here, the symbol $\oplus$ simply represents an operation between an element on the manifold and an element in the tangent space, with the result still being an element on the manifold; it does not obey the commutative law. For rotations, $\oplus$ means either left or right perturbation approximation (these are two different $\oplus$ operations and should not be mixed), for example:

$$\begin{aligned}
R \oplus \Delta\phi &= \exp((\Delta\phi)_\times) R \quad \text{(left disturbance)} \\
R \oplus \Delta\phi &= R \exp((\Delta\phi)_\times) \quad \text{(right disturbance)} \\
q \oplus \Delta\theta &= \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} \otimes q \quad \text{(left disturbance)} \\
q \oplus \Delta\theta &= q \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} \quad \text{(right disturbance)}
\end{aligned} \tag{III.11.31}$$

Furthermore, we can define the derivatives of functions f, h with respect to the error state:

$$\begin{aligned} \frac{\partial f}{\partial \delta x} &:= \lim_{\delta x \to 0} \frac{f(x \oplus \delta x) - f(x)}{\delta x} \\ \frac{\partial h}{\partial \delta x} &:= \lim_{\delta x \to 0} \frac{h(x \oplus \delta x) - h(x)}{\delta x} \end{aligned} \tag{III.11.32}$$

Assuming δx follows a Gaussian distribution, the correction step update computed from the innovation in the Kalman filter is essentially δx_k:

$$\delta x_k = K_k(z_k - h(\hat{x}_k^-, u_k)) \tag{III.11.33}$$

Finally, we can update δx_k to the nominal state $\hat{x}$:

$$\hat{x}_{k+1} \leftarrow \hat{x}_k^- \oplus \delta x_k \tag{III.11.34}$$

By replacing $\frac{\partial f}{\partial x}, \frac{\partial h}{\partial x}$ in the EKF with the above method, we summarize the ESKF algorithm as follows.

Algorithm	**Error State Kalman Filter**
Problem Type	Optimal Filtering Problem [Discrete]
Given	Discrete system $x_{k+1} = f(x_k, u_k, w_k)$ Observation equation $z_k = h(x_k, u_k, v_k)$ System input/output u_k, z_k System noise distribution $w_k \sim \mathcal{N}(0, Q_k)$ Output noise distribution $v_k \sim \mathcal{N}(0, R_k)$
Find	Filter $\hat{x} = o(u, z, \hat{x})$ that minimizes $\text{Var}[\hat{x}]$
Algorithm Properties	Filter, Recursive

Algorithm 27: Error State Kalman Filter

Input: Differentiable system and output equations f, h
Input: Sequence of symmetric positive definite matrices Q_k, R_k, symmetric positive definite matrix P_0
Input: System input sequence u_k, system output sequence z_k
Input: System initial value $\hat{x}_0$
Output: Estimated state $\hat{\mathbf{x}}$
for $k \in 0, 1, \ldots, K$ **do**
- $\hat{x}_k^- \leftarrow f(\hat{x}_k, u_k)$
- $F_k \leftarrow \frac{\partial f}{\partial \delta x}(\hat{x}_k)$
- $P_{k+1}^- \leftarrow F_k P_k F_k^T + Q_k$
- $H_k \leftarrow \frac{\partial h}{\partial \delta x}(\hat{x}_k)$
- $K_k \leftarrow P_{k+1}^- H_k^T (H_k P_{k+1}^- H_k^T + R_k)^{-1}$
- $\delta x_k \leftarrow K_k(z_k - h(\hat{x}_k^-))$
- $\hat{x}_{k+1} \leftarrow \hat{x}_k^- \oplus \delta x_k$
- $P_{k+1} \leftarrow K_k R_k K_k^T + (I - K_k H_k) P_{k+1}^- (I - K_k H_k)^T$

The corresponding Python code is shown below:

```python
def filter_ESKF(x_0, us, zs, f_func, df_func, h_func, dh_func,
    add_func, Qs, Rs, P_0, N_K:int):
    assert len(us) == N_K, len(zs) == N_K
    assert len(Rs) == N_K, len(Qs) == N_K
    Ks, xs, Pk = [], [x_0], P_0
    for k in range(N_K+1):
        xk_ = f_func(xs[k], us[k])
        Fk = df_func(xs[k], us[k])
        Pk_ = Fk @ Pk @ Fk.T + Qs[k]
        Hk = dh_func(xs[k])
        tmp = np.linalg.inv(Rs[k] + Hk @ Pk_ @ Hk.T)
        Ks.append(Pk_ @ Hk.T @ tmp)
        delta_x = Ks[k] @ (zs[k] - h_func(xs[k]))
        xs.append(add_func(xk_, delta_x))
        tmp2 = np.eye(x_0.shape[0]) - Ks[k] @ Hk
        Pk = Ks[k] @ Rs[k] @ Ks[k].T + tmp2 @ Pk_ @ tmp2.T
    return xs
```

In fact, for most filtering problems with three-axis attitude as the state (such as VIO filtering, see Section VII for details), the ESKF method is actually used.

11.4 Unscented Kalman Filter (UKF)

In the previous section, we introduced the EKF and ESKF methods for nonlinear systems. These methods are essentially different forms of first-order Taylor expansions for nonlinear systems, which cannot guarantee the accuracy of higher-order statistics and exhibit significantly degraded performance under strong nonlinearity.

In fact, the core challenge of nonlinear system filtering lies in: when f and h are highly nonlinear, the approximate Gaussian distribution of the random variable $f(x)$ deviates substantially from $N(f(\bar{x}), F^T PF + Q)$, leading to large errors in the prediction step (the same applies to $h(x)$ and the correction step). If we replace the first-order Taylor expansion with higher-order expansions, the computational complexity increases dramatically. So, is it possible to maintain unbiased estimation of higher-order statistics for the estimated state and the true state without performing complex higher-order Taylor expansion calculations?

11.4.1 Unscented Transformation

In fact, we can estimate the expectation/variance of a random variable function through sampling. Specifically, assuming $\mathbf{x} \sim \mathcal{N}(\bar{\mathbf{x}}, P) \in \mathbb{R}^n$, we define a set of **sigma points** as follows:

$$\begin{aligned} \chi_0 &= \bar{\mathbf{x}} \\ \chi_i &= \bar{\mathbf{x}} + \mathbf{p}_i, \quad i = 1, ..., n \\ \chi_{i+n} &= \bar{\mathbf{x}} - \mathbf{p}_i, \quad i = 1, ..., n \end{aligned}$$

Assuming $\sqrt{A}$ denotes the matrix square root of A, we have:

$$\sqrt{(n+\lambda)P} = \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \dots & \mathbf{p}_n \end{bmatrix}$$

It can be observed that these $2n+1$ sigma points are sampled in the space of $\mathbf{x}$, centered around $\bar{\mathbf{x}}$. Define the weights for the sigma points:

$$\begin{aligned} w_0^m &= \frac{\lambda}{n+\lambda} \\ w_0^c &= \frac{\lambda}{n+\lambda} + (1 - \alpha^2 + \beta) \\ w_i^c = w_i^m &= \frac{1}{2(n+\lambda)}, \quad i = 1, ..., 2n \end{aligned}$$

These sigma points satisfy:

$$\bar{\mathbf{x}} = \sum_{i=0}^{2n} w_i^m \chi_i$$
$$P = \sum_{i=0}^{2n} w_i^c (\chi_i - \bar{\mathbf{x}})^T (\chi_i - \bar{\mathbf{x}})$$

Let us briefly derive these results. For the expectation, we have:

$$\begin{aligned}
&\sum_{i=0}^{2n} w_i^m \chi_i \\
&= \frac{\lambda}{n+\lambda}\bar{\mathbf{x}} + \sum_{i=1}^{n} \frac{1}{2(n+\lambda)}(\bar{\mathbf{x}} + \mathbf{p}_i) + \sum_{i=1}^{n} \frac{1}{2(n+\lambda)}(\bar{\mathbf{x}} - \mathbf{p}_i) \\
&= \bar{\mathbf{x}}
\end{aligned}$$

For the variance, we have:

$$\begin{aligned}
&\sum_{i=0}^{2n} w_i^c (\chi_i - \bar{\mathbf{x}})^T (\chi_i - \bar{\mathbf{x}}) \\
&= \sum_{i=0}^{n} w_i^c (\chi_i - \bar{\mathbf{x}})^T (\chi_i - \bar{\mathbf{x}}) + \sum_{i=n+1}^{2n} w_i^c (\chi_i - \bar{\mathbf{x}})^T (\chi_i - \bar{\mathbf{x}}) \\
&= \frac{1}{2(n+\lambda)} \sum_{i=0}^{n} p_i p_i^T + \frac{1}{2(n+\lambda)} \sum_{i=0}^{n} p_i p_i^T \\
&= \frac{1}{(n+\lambda)} \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \dots & \mathbf{p}_n \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \dots & \mathbf{p}_n \end{bmatrix}^T \\
&= \frac{1}{(n+\lambda)} \sqrt{(n+\lambda)P}\sqrt{(n+\lambda)P} \\
&= P
\end{aligned}$$

In other words, the vector set $\chi_{0:2n}$ represents a group of points around $\bar{\mathbf{x}}$, each corresponding to a perturbation of $\bar{\mathbf{x}}$ along a specific eigenvector direction (bidirectional) of $\sqrt{P}$. Since these $2n+1$ points cover the local space around $\bar{\mathbf{x}}$, we can use their statistics to represent the properties near $\bar{\mathbf{x}}$. That is, for a nonlinear function f, the expectation and variance of the random variable $f(\mathbf{x})$ can be estimated as:

$$E[f(\mathbf{x})] \approx \sum_{i=0}^{2n} w_i^m f(\chi_i)$$
$$Var[f(\mathbf{x})] \approx \sum_{i=0}^{2n} w_i^c (f(\chi_i) - E[f(\mathbf{x})])(f(\chi_i) - E[f(\mathbf{x})])^T$$

In the above expressions, $\lambda, \alpha, \beta, \kappa$ are parameters, with the relationship:

$$\lambda = \alpha^2(n+\kappa) - n$$

11.4.2 Filtering Algorithm

Using the above approximation method, we can approximate the prediction step (x_k^-, P_{k+1}^-) and the correction step (x_{k+1}, P_{k+1}) for nonlinear system filtering without computing derivatives.

Specifically, for the prediction step, assume:

$$
\begin{aligned}
\chi_0 &= x_k \\
\chi_i &= x_k + \mathbf{p}_i, \quad i = 1, ..., n \\
\chi_{i+n} &= x_k - \mathbf{p}_i, \quad i = 1, ..., n \\
\sqrt{(n+\lambda)P_k} &= \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & ... & \mathbf{p}_n \end{bmatrix}
\end{aligned}
\tag{III.11.35}
$$

and:

$$
\begin{aligned}
w_0^m &= \frac{\lambda}{n+\lambda} \\
w_0^c &= \frac{\lambda}{n+\lambda} + (1 - \alpha^2 + \beta) \\
w_i^c &= w_i^m = \frac{1}{2(n+\lambda)}, \quad i = 1, ..., 2n
\end{aligned}
\tag{III.11.36}
$$

We can estimate the mean and variance of $f(x_k)$ as:

$$
\begin{aligned}
x_k^- &= \sum_{i=0}^{2n} w_i^m f(\chi_i) \\
P_{k+1}^- &= \sum_{i=0}^{2n} w_i^c (f(\chi_i) - x_k^-)(f(\chi_i) - x_k^-)^T + Q_k
\end{aligned}
\tag{III.11.37}
$$

For the correction step, we first compute the predicted observation for each sigma point:

$$
\gamma_i = h(\chi_i), \quad i = 0, ..., 2n \tag{III.11.38}
$$

Thus:

$$
\hat{z}_k = \sum_{i=0}^{2n} w_i^m \gamma_i \tag{III.11.39}
$$

Next, we need to compute the Kalman gain. In the EKF, the Kalman gain depends on H_k, P_{k+1}^-, R_k, where H_k is the linearized h, representing the linear relationship between the estimated observation and the estimated state. In the UKF, this matrix is expressed as the cross-covariance matrix between the random variables z and x. This matrix can also be estimated from the sigma points:

$$
H_k = \sum_{i=0}^{2n} w_i^c (\gamma_i - \hat{z}_k)(f(\chi_i) - x_k^-)^T \tag{III.11.40}
$$

Summarizing the above derivations, we present the Unscented Kalman Filter algorithm as follows:

Algorithm	**Unscented Kalman Filter**
Problem Type	Optimal Filtering Problem [Discrete]
Given	Discrete system $x_{k+1} = f(x_k, u_k, w_k)$ Observation equation $z_k = h(x_k, u_k, v_k)$ System input/output u_k, z_k System noise distribution $w_k \sim \mathcal{N}(0, Q_k)$ Output noise distribution $v_k \sim \mathcal{N}(0, R_k)$
Objective	Filter $\hat{x} = o(u, z, \hat{x})$ that minimizes Var$[\hat{x}]$
Algorithm Properties	Filter, recursive

Algorithm 28: Unscented Kalman Filter

Input: System and output equations f, h
Input: Sequence of symmetric positive definite matrices Q_k, R_k, symmetric positive definite matrix P_0
Input: System input sequence u_k, system output sequence z_k
Input: Initial system state $\hat{x}_0$
Parameter: α, κ, β
Output: Estimated state $\hat{\mathbf{x}}$

$\lambda \leftarrow \alpha^2(n+\kappa) - n$
$w_0^m \leftarrow \lambda/(n+\lambda)$
$w_0^c \leftarrow \lambda/(n+\lambda) + (1-\alpha^2+\beta)$
for $i \in 1, \ldots, n$ **do**
  $w_i^c \leftarrow= \frac{1}{2(n+\lambda)}$
  $w_i^m \leftarrow= \frac{1}{2(n+\lambda)}$
for $k \in 0, 1, \ldots, K$ **do**
  $\chi_0 \leftarrow x_k$
  $\begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \ldots & \mathbf{p}_n \end{bmatrix} \leftarrow \sqrt{(n+\lambda)P_k}$
  for $i \in 1, \ldots, n$ **do**
    $\chi_i \leftarrow x_k + \mathbf{p}_i, \quad i = 1, \ldots, n$
    $\chi_{i+n} \leftarrow x_k - \mathbf{p}_i, \quad i = 1, \ldots, n$
  $x_k^- \leftarrow \sum_{i=0}^{2n} w_i^m f(\chi_i)$
  $P_{k+1}^- \leftarrow \sum_{i=0}^{2n} w_i^c (f(\chi_i) - x_k^-)(f(\chi_i) - x_k^-)^T + Q_k$
  $\hat{z}_k \leftarrow \sum_{i=0}^{2n} w_i^m h(\chi_i)$
  $H_k \leftarrow \sum_{i=0}^{2n} w_i^c (h(\chi_i) - \hat{z}_k)(f(\chi_i) - x_k^-)^T$
  $K_k \leftarrow P_{k+1}^- H_k^T (H_k P_{k+1}^- H_k^T + R_k)^{-1}$
  $\hat{x}_{k+1} \leftarrow \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, u_k))$
  $P_{k+1} \leftarrow K_k R_k K_k^T + (I - K_k H_k) P_{k+1}^- (I - K_k H_k)^T$

The corresponding Python code is shown below:

```python
def filter_UKF(x_0, us, zs, f_func, h_func, Qs, Rs, P_0, alpha, kappa, beta, N_K:int):
    assert len(us) == N_K, len(zs) == N_K
    assert len(Rs) == N_K, len(Qs) == N_K
    n, m = x_0.shape[0], zs[0].shape[0]
    lambda_ = alpha**2 *(n + kappa) - n
    wms, wcs = np.zeros([n]), np.zeros([n])
    wms[0] = lambda_ / (n + lambda_)
    wcs[0] = lambda_ / (n + lambda_) + (1 - alpha**2 + beta)
    wms[1:], wcs[1:] = 0.5 / (n + lambda_), 0.5 / (n + lambda_)
    Ks, xs, Pk = [], [x_0], P_0
    for k in range(N_K+1):
        points, fs, hs = np.zeros([2*n+1, n]), np.zeros([2*n+1, n]), np.zeros([2*n+1, m])
        ps = np.linalg.sqrtm((n + lambda_) * Pk)
        points[0] = xs[k]
        points[1:1+n], points[1+n:] = ps + xs[k], - ps + xs[k]
        xk_, zk_est = np.zeros_like(x_0), np.zeros_like(zs[0])
        for i in range(2*n+1):
            fs[i] = f_func(points[i], us[k])
            hs[i] = h_func(points[i])
            xk_ += wms[i] * fs[i]
            zk_est += wms[i] * hs[i]
        Pk_, Hk = Qs[k], np.zeros([m, n])
        for i in range(2*n+1):
            Pk_ += wcs[i] * (fs[i] - xk_)[:, None] @ (fs[i] - xk_)[None, :]
```

```
            Hk += wcs[i] * (hs[i] - zk_est)[:, None] @ (fs[i] - xk_)[None, :]
        tmp = np.linalg.inv(Rs[k] + Hk @ Pk_ @ Hk.T)
        Ks.append(Pk_ @ Hk.T @ tmp)
        xs.append(xk_ + Ks[k] @ (zs[k] - h_func(xs[k])))
        tmp2 = np.eye(x_0.shape[0]) - Ks[k] @ Hk
        Pk = Ks[k] @ Rs[k] @ Ks[k].T + tmp2 @ Pk_ @ tmp2.T
    return xs
```

Compared to EKF/ESKF, the main advantage of UKF is that it does not require differentiation of the system or output functions, at the cost of multiple evaluations of $f(\cdot)$ and $h(\cdot)$.

12 Foundations of Optimal Control

12.1 Optimal Control Problems

In robotics, we often encounter a class of problems: given a known system model, we aim to design a system trajectory that satisfies certain quantifiable objectives as much as possible. For example, we may want to minimize energy consumption, achieve the shortest time, minimize tracking errors, or satisfy certain equality or inequality constraints. Such problems are collectively referred to as **optimal control** problems.

In optimal control problems, we seek to compute the control that optimizes a certain quantity. This quantity of interest is called the **objective function**, denoted as $J(\mathbf{x}, \mathbf{u})$. For minimization problems, the objective function is also referred to as the **cost**. Different requirements lead to different forms of the objective function. Later, we will summarize common forms of objective functions.

The objects in optimal control include both continuous and discrete systems. In robotics, we typically discretize continuous systems, so this chapter focuses on discrete systems but also introduces corresponding continuous conclusions.

Constraints are crucial in optimal control. For instance, in some optimal control problems, the control input cannot exceed certain limits; in others, the problem must be solved within a specific time frame. The presence or absence of constraints significantly impacts the design of optimal control. In this chapter, we primarily discuss unconstrained and equality-constrained optimal control problems. For inequality-constrained optimal control problems, we generally consider using model predictive control and QP solvers, which will be introduced in the next chapter.

As mentioned in the previous chapter, in control theory, the problem of stabilizing a system at a fixed value is called a **regulation problem**. Correspondingly, the problem of tracking a varying trajectory is called a **tracking problem**. For optimal control problems, starting the analysis with the regulation problem greatly simplifies the process. In this chapter, unless otherwise specified, the desired state for the regulation problem is set to $x_d = 0$.

We describe the **(unconstrained) optimal control problem** in discrete settings using mathematical language, summarized as follows. Note: The functions $f(\cdot), g(\cdot)$, and $h(\cdot)$ in this chapter are defined differently from those in Chapter 3, "Optimization Algorithms."

Problem	**(Unconstrained) General Optimal Regulation Control [Discrete]**
Problem Description	Given a discrete system and cost function, find the optimal regulation control input
Given	Discrete system $x_{k+1} = f(x_k, u_k)$ Cost function $J(\mathbf{x}, \mathbf{u}) = h(x_N) + \sum_{k=0}^{N-1} g(x_k, u_k)$ Initial state x_0
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$

Problem	**(Unconstrained) General Optimal Tracking Control [Discrete]**
Problem Description	Given a discrete system, cost function, and reference trajectory, find the optimal tracking control input
Given	Discrete system $x_{k+1} = f(x_k, u_k)$ Cost function $J(\mathbf{x}, \mathbf{u}) = h(x_N) + \sum_{k=0}^{N-1} g(x_k, u_k)$ Initial state x_0, reference trajectory $\mathbf{x}_d$
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u}, \mathbf{x}_d)$

Here, it is important to note that the state x_k is often a vector, meaning it has multiple dimensions. In notation, we use the non-bold x_k to represent its value at a specific time step, while the bold $\mathbf{x}$ represents all states from the initial to the final time step. We default to handling finite-time cases, with N representing the total number of steps.

In optimal control problems, the system function f is a known function. It may be linear or nonlinear, but the current state depends only on the previous state and control input. The cost function here is written

in a general form, encompassing the entire process's states and control inputs, as well as the terminal state. The cost function J is also known, meaning h and g are known functions.

Another point to note is that the optimal control u we compute is often a function of the current state, i.e.,

$$u_k = u_k(x_k) \tag{III.12.1}$$

Next, we examine the specific forms of the cost function $\mathbf{J}$. In practical problems, we often focus on the following types of cost functions. Note: These cost functions usually do not appear alone but are combined based on the actual problem requirements.

First is the **minimum-time cost**. Its cost function is simply the total number of steps N.

$$J(\mathbf{x}, \mathbf{u}) = N \tag{III.12.2}$$

Next is the **terminal control cost** for regulation problems, where we want the final state to be as close as possible to the reference state x_d.

$$J(\mathbf{x}, \mathbf{u}) = ||x_N - x_d||_S \tag{III.12.3}$$

Here, we use the notation $||\cdot||_S$ to represent the quadratic form of a vector:

$$||a||_S = a^T S a \tag{III.12.4}$$

For the terminal state in optimal control, we may only care about certain dimensions or assign different weights to different dimensions. Adjusting the coefficients of the matrix S can reflect this prioritization. Typically, we choose a diagonal matrix.

For tracking problems, we have the **tracking cost**, where we want the entire state trajectory $\mathbf{x}$ to be as close as possible to a reference trajectory $\mathbf{x}_d$:

$$J(\mathbf{x}, \mathbf{u}) = \sum_{k=0}^{N} ||x_k - x_{d,k}||_{Q_k} \tag{III.12.5}$$

Often, we also want to minimize the total control effort, which usually implies energy savings and avoiding oscillations. Thus, we have the **minimum control effort cost**:

$$J(\mathbf{x}, \mathbf{u}) = \sum_{k=0}^{N-1} ||u_k||_{R_k} \tag{III.12.6}$$

Combining the above optimal control objectives, we have the **(unconstrained) optimal regulation control problem** and the **(unconstrained) optimal tracking control problem**.

Problem	**(Unconstrained) Quadratic Optimal Regulation Control [Discrete]**
Problem Description	Given a discrete system, find the regulation control input that minimizes the composite cost
Given	Discrete system $x_{k+1} = f(x_k, u_k)$ Initial state x_0 Positive semi-definite matrices R_k, Q_k Positive semi-definite matrix S
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} ||x_N||_S + \sum_{k=0}^{N-1} (||x_k||_{Q_k} + ||u_k||_{R_k})$

Problem	(Unconstrained) Quadratic Optimal Tracking Control [Discrete]
Problem Description	Given a discrete system, find the tracking control input that minimizes the composite cost
Given	Discrete system $x_{k+1} = f(x_k, u_k)$ Initial state x_0, reference trajectory $\mathbf{x}_d$ Positive semi-definite matrices R_k, Q_k Positive semi-definite matrix S
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N - x_{d,N}\|\|_S$ $+ \sum_{k=0}^{N-1}(\|\|x_k - x_{d,k}\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$

12.2 Bellman Optimality and Dynamic Programming

In 1957, Richard Bellman formulated the **principle of optimality** for optimal control problems. This principle is the cornerstone of all subsequent sequential decision-making methods, including optimal control and reinforcement learning. Mathematically, we can express this principle as follows.

Consider an optimal control problem $\mathcal{P}_0$ with initial conditions $k_0 = 0, x_0 = \xi_0$, and the corresponding cost function denoted as $J_{0:N}(\mathbf{x}, \mathbf{u})$. Solving this problem yields the optimal control $\mathbf{u}[\mathbf{0}]^* = \nu_{0:N}(x)$ and the optimal trajectory $\mathbf{x}[\mathbf{0}]^* = \xi_{0:N}$.

Now consider the subproblem $\mathcal{P}_m$ of $\mathcal{P}_0$, which takes the initial condition $k_0 = m, x_0 = \xi_m$. Its cost function is the portion of $J_{0:N}$ after $k = m$, denoted as $J_{m:N}(\mathbf{x}, \mathbf{u})$. For the problem $\mathcal{P}_m$, the obtained optimal control is denoted as $\mathbf{u}[\mathbf{m}]^* = \tau_{m:N}(x)$, and the optimal trajectory is denoted as $\mathbf{x}[\mathbf{m}]^* = \chi_{m:N}$.

The Bellman optimality principle states that solving the subproblem $\mathcal{P}_m$ independently yields τ and χ, which are identical to the corresponding parts of the original problem's optimal solution ν, ξ, i.e.,

$$\tau_{0:N-m}(x) = \nu_{m:N}(x)$$
$$\xi_{0:N-m} = \chi_{m:N}$$

If we denote the cost function under optimal control as $\mathcal{J}$, i.e.,

$$\mathcal{J}_{m:N}(\mathbf{x}) = J_{m:N}(\mathbf{x}, \nu_{m:N}(\mathbf{x}))$$

then we have

$$\min_{\mathbf{u}_{0:N}} J_{0:N}(\mathbf{x}, \mathbf{u}) = \min_{\mathbf{u}_{0:m}} (J_{0:m}(\mathbf{x}_{0:m}, \mathbf{u}) + \mathcal{J}_{m:N}(\mathbf{x}_{m:N})) \tag{III.12.7}$$

Moreover, when the minimum is achieved, we have

$$\mathbf{x}_{m:N} = \mathbf{x}[\mathbf{m}]^* = \chi_{m:N} \tag{III.12.8}$$

The Bellman optimality principle reveals that for a broad class of sequential optimization problems (satisfying the definition of optimal control problems), the solution to the entire problem can be decomposed into different steps: first solve the smaller-scale latter part, and then solve the former part based on the latter. This decomposition can be applied recursively. For example, for the latter part, we can further find the "latter part of the latter part" and so on. Eventually, we can start from the very last small segment and derive the optimal solution to the entire optimal control problem backward. This problem decomposition approach is called **dynamic programming**.

To be more specific, let us consider a concrete example: **slider deceleration**. A slider on a one-dimensional track is at position $p_0 = 2$ and velocity $v_0 = -1$ at time $k = 0$. The goal is to apply forces at discrete times to bring it to a stop at position $p_N = 0$ and velocity $v_N = 0$ by time $k = N$. The forces are applied at times $k = 0 : N - 1$, providing velocity increments of magnitude μ_k. We seek a control scheme that minimizes the control input.

Formalizing this example as an optimal control problem, we have:

Discrete system state $x_k = [p_k, v_k]^T$, control input $u_k = [0, \mu_k]^T$, satisfying the following system equation:

$$x_{k+1} = f(x_k, u_k) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0 & 1 \end{bmatrix} u_k$$

Cost function:

$$J(\mathbf{x}, \mathbf{u}) = ||x_N||_S + \sum_{k=0}^{N-1} ||u_k||_R$$

where

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, R = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Given the initial condition $x_0 = [2, -1]^T$, find the control input that minimizes the cost function.

To simplify the discussion and calculations, we set $N = 2$. Rewriting the above expressions more concisely, we have:

$$\begin{aligned} \mathcal{P}_0 : \min_{\mathbf{u}} \quad & J_{0:2}(\mathbf{x}, \mathbf{u}) = ||x_2||^2 + \sum_{k=0}^{1} \mu_k^2 \\ \text{s.t.} \quad & x_{k+1} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_k \end{aligned}$$

For this problem, let us analyze it using the dynamic programming approach described above. The problem described here is $\mathcal{P}_0$. To find its optimal control $\nu_{0:1}$ and optimal trajectory $\xi_{0:2}$, we first need to analyze its tail subproblem $\mathcal{P}_1$. Based on the above analysis, we remove the portion of the cost function before $k = 1$.

$$\begin{aligned} \mathcal{P}_1 : \min_{u_1} \quad & J_{1:2}(\mathbf{x}_{1:2}, \mathbf{u}_2) = ||x_2||^2 + \mu_1^2 \\ \text{s.t.} \quad & p_2 = p_1 + v_1 \\ & v_2 = v_1 + \mu_1 \end{aligned}$$

Note that our optimization objective here is not x_k but u_k. Therefore, we only solve the **relationship between u and x**, not the specific numerical values. All x_k are constrained by the system state equation, and there is also the constraint of the initial condition x_0. We only substitute x_0 when backtracking to $k = 0$ to obtain the numerical solution for u and x.

Now, our expression for $J_{1:2}$ includes x_2 and μ_1, but x_2 is a function of μ_1. Using the system state equation, we can write:

$$x_2 = \begin{bmatrix} p_1 + v_1 \\ v_1 + \mu_1 \end{bmatrix}$$

For the problem $\mathcal{P}_1$, we assume $x_1 = [p_1, v_1]^T$ is known. Thus, we have the relationship between x_2 and μ_1. Therefore, we have:

$$||x_2||^2 = (p_1 + v_1)^2 + (v_1 + \mu_1)^2$$

Hence, $J_{1:2}$ can be written as a function of x_1 and μ_1:

$$J_{1:2}(x_1, \mu_1) = (p_1 + v_1)^2 + (v_1 + \mu_1)^2 + \mu_1^2$$

According to the Bellman optimality principle, the optimal control $\nu_{0:2}(x)$ for $\mathcal{P}_0$ includes $\nu_2(x)$, which is the $u_1(x)$ (ignoring the first dimension, it is essentially $\mu_1(x)$) that minimizes $J_{1:2}(x_1, u_1)$. Therefore, we solve as follows:

$$\begin{aligned} \arg\min_{\mu_1} J_{1:2}(x_1, \mu_1) &= \arg\min_{\mu_1} \left((p_1 + v_1)^2 + (v_1 + \mu_1)^2 + \mu_1^2 \right) \\ &= \arg\min_{\mu_1} \left(2\mu_1^2 + 2v_1\mu_1 + v_1^2 \right) \\ &= -\frac{1}{2} v_1 \end{aligned}$$

Thus, we have:

$$\nu_1(x) = \tau_1(x) = -\frac{1}{2}v$$

Here we emphasize again that the result of the backward dynamic programming solution for optimal control is neither the optimal control value nor the optimal state value, but the function of the control input with respect to the state.[11]

Whether it is the parent problem's $\mathbf{u}[\mathbf{0}]^* = \nu_{0:N}(x)$ or the subproblem's $\mathbf{u}[\mathbf{m}]^* = \tau_{m:N}(x)$, they are both sets of functions with respect to x. The function corresponding to time k only acts on the state x_k at time k and does not affect x_{k-1} or x_{k+1}. However, for a given problem, as long as the cost function remains unchanged, even if the initial state x_0 changes, causing the optimal control trajectory to change, the function set $\nu_{0:N}(x)$ will not change.

Next, we continue to derive the solution to the next problem $\mathcal{P}_0$. The original $\mathcal{P}_0$ problem can be written as:

$$\begin{aligned}
\mathcal{P}_0 : \min_{u_{0:1}} \quad & J_{0:2}(\mathbf{x}_{0:2}, \mathbf{u}_{0:1}) = ||x_2||^2 + \mu_1^2 + \mu_0^2 \\
\text{s.t.} & p_2 = p_1 + v_1 \\
& v_2 = v_1 + \mu_1 \\
& p_1 = p_0 + v_0 \\
& v_1 = v_0 + \mu_0
\end{aligned}$$

We observe that the cost of problem $\mathcal{P}_0$ entirely encompasses the cost $J_{1:2}$ of $\mathcal{P}_1$. According to Bellman's principle of optimality, the $J_{1:2}$ component in the cost of $\mathcal{P}_0$ can be replaced by the cost function $\mathcal{J}_{1:2}(\mathbf{x})$ when $\mathcal{P}_1$ achieves optimal control, while the variables to be solved reduce from $u_{0:1}$ to only u_1. That is,

$$\min_{\mathbf{u}_{0:2}} J_{0:2}(\mathbf{x}, \mathbf{u}) = \min_{\mathbf{u}_{0:2}} (J_{0:1}(\mathbf{x}_{0:1}, \mathbf{u}) + \mathcal{J}_{1:2}(\mathbf{x}_{1:2}))$$

Therefore, let us first attempt to derive the expression for $\mathcal{J}_{1:2}(\mathbf{x}_{1:2})$. By definition,

$$\begin{aligned}
\mathcal{J}_{1:2}(\mathbf{x}_{1:2}) &= J_{1:2}(\mathbf{x}_{1:2}, \nu_1(x_1)) \\
&= (p_1 + v_1)^2 + (v_1 + \nu_1(x_1))^2 + \nu_1(x_1)^2 \\
&= (p_1 + v_1)^2 + (v_1 - \frac{1}{2}v_1)^2 + (-\frac{1}{2}v_1)^2 \\
&= (p_1 + v_1)^2 + \frac{1}{2}v_1^2
\end{aligned}$$

Thus, we obtain the new expression for $J_{0:2}(\mathbf{x}, \mathbf{u})$:

$$J_{0:2}(\mathbf{x}, \mathbf{u}) = (p_1 + v_1)^2 + \frac{1}{2}v_1^2 + \mu_0^2$$

Let us reformulate problem $\mathcal{P}_0$ as $\mathcal{P}_0'$:

$$\begin{aligned}
\mathcal{P}_0' : \min_{u_1} \quad & J_{0:2}(\mathbf{x}_{0:1}, \mathbf{u}_1) = (p_1 + v_1)^2 + \frac{1}{2}v_1^2 + \mu_0^2 \\
\text{s.t.} & p_1 = p_0 + v_0 \\
& v_1 = v_0 + \mu_0
\end{aligned}$$

Now, the expression for $J_{0:2}$ involves x_1 and μ_0, but x_1 is a function of μ_0. Using the system state equation, we can write:

$$\begin{bmatrix} p_1 \\ v_1 \end{bmatrix} = \begin{bmatrix} p_0 + v_0 \\ v_0 + \mu_0 \end{bmatrix}$$

For problem $\mathcal{P}_0'$, we assume $x_0 = [p_0, v_0]^T$ is known. Since we have established the relationship between x_1 and μ_0, we obtain:

[11] From this perspective, our $J_{m:N}$ is actually a functional of the function u. We are currently solving this functional in the discrete domain.

$$(p_1 + v_1)^2 + \frac{1}{2}v_1^2 = (p_0 + v_0 + v_0 + \mu_0)^2 + \frac{1}{2}(v_0 + \mu_0)^2$$
$$= (p_0 + 2v_0)^2 + \frac{1}{2}v_0^2 + (2p_0 + 5v_0)\mu_0 + \frac{3}{2}\mu_0^2$$

Thus, $J_{0:2}$ can be expressed as a function of x_0 and μ_0:

$$J_{0:2}(x_1, \mu_1) = (p_0 + 2v_0)^2 + \frac{1}{2}v_0^2 + (2p_0 + 5v_0)\mu_0 + \frac{5}{2}\mu_0^2$$

According to Bellman's principle of optimality, the optimal control $\nu_{0:1}(x)$ for problem $\mathcal{P}_0$ implies that $\nu_1(x)$ is the $\mu_0(x)$ that minimizes $J_{0:2}$. Therefore, we solve as follows:

$$\arg\min_{\mu_0} J_{0:2}(x_0, \mu_0) = \arg\min_{\mu_0} \left((p_0 + 2v_0)^2 + \frac{1}{2}v_0^2 + (2p_0 + 5v_0)\mu_0 + \frac{5}{2}\mu_0^2 \right)$$
$$= -v_0 - \frac{2}{5}p_0$$

Thus, we have:

$$\nu_0(x) = \tau_0(x) = -v - \frac{2}{5}p$$

At this point, in this simplest optimal control problem with $N = 2$, we have derived the required optimal control:

$$u_0^*(x_0) = \nu_0(x_0) = -v_0 - \frac{2}{5}p_0$$
$$u_1^*(x_1) = \nu_1(x_1) = -\frac{1}{2}v_1$$

If we need to compute the optimal trajectory and specific optimal control inputs, we can substitute the initial condition $x_0 = [2, -1]^T$ and then use the state equation to **forward-solve**. The results are summarized in the table below.

	k=0	k=1	k=2
p_k expression	2	$p_0 + v_0$	$p_1 + v_1$
v_k expression	-1	$v_0 + \mu_0$	$v_1 + \mu_1$
μ_k expression	$-v_0 - \frac{2}{5}p_0$	$-\frac{1}{2}v_1$	-
$J_{k:N}$ expression	$\|\|x_2\|\|^2 + \mu_1^2 + \mu_0^2$	$\|\|x_2\|\|^2 + \mu_1^2$	$\|\|x_2\|\|^2$
p_k value	2	1	1/5
v_k value	-1	-4/5	-2/5
μ_k value	1/5	2/5	-
$J_{k:N}$ value	2/5	9/25	1/5

We can see that although the derived control does not strictly constrain the final state to $[0, 0]^T$, it achieves a balance between the control input magnitude and the final state. If we remove the constraints on the control inputs or adjust their weighting through the S or Q matrices, the resulting control would differ.

To summarize, we first leveraged Bellman's principle of optimality to isolate the smallest subproblem near the final state of the original problem. The principle guarantees that the optimal solution to the subproblem is part of the optimal solution to the original problem. Next, we progressively expanded the solution scope forward, solving only one step of optimal control at a time until reaching $k = 0$, the original problem. Through this **backward-solving** approach, we obtained the optimal control $\nu_k(x)$ for each time step. Based on this, we then performed **forward-solving** from the initial condition x_0 of the original problem using the system state equation and optimal control, thereby deriving the trajectory and control inputs for specific initial conditions.

Although we have demonstrated only a very simple example, we can generalize this approach to outline a general algorithm for solving optimal control problems using dynamic programming.

Algorithm	Optimal Control - Dynamic Programming Solution
Problem type	(Unconstrained) General Optimal Regulation Control [Discrete]
Known	Discrete system $x_{k+1} = f(x_k, u_k)$ Cost function $J(\mathbf{x}, \mathbf{u}) = h(x_N) + \sum_{k=0}^{N-1} g(x_k, u_k)$ Initial state x_0
Objective	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$
Algorithm properties	model-based, DP, analytical derivation

Algorithm 29: Optimal Control - Dynamic Programming Solution

Input: Discrete system $x_{k+1} = f(x_k, u_k)$
Input: Cost function $J(\mathbf{x}, \mathbf{u}) = h(x_N) + \sum_{k=0}^{N-1} g(x_k, u_k)$
Input: Initial state x_0
Output: Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$
$\mathcal{J}_{N:N}(\mathbf{x}_{N:N}) \leftarrow h(x_N)$
for $\tau \in N-1, N-2, \ldots, 0$ **do**
  $\arg\min_{\mathbf{u}} J_{\tau:N} \leftarrow \arg\min_{\mathbf{u}}(g(x_\tau, u_\tau) + \mathcal{J}_{\tau+1:N}(x_{\tau+1}))$
  Substitute the system state equation $x_{\tau+1} = f(x_\tau, u_\tau)$
  Write the expression of $J_{\tau:N}$ as $J_{\tau:N}(x_\tau, u_\tau)$
  $\nu_\tau(x) \leftarrow \arg\min_{u_\tau} J_{\tau:N}(x_\tau, u_\tau)$
  $\mathcal{J}_{\tau:N}(x_\tau) \leftarrow J_{\tau:N}(x_\tau, \nu_\tau^*(x_\tau))$
$\mathbf{u}^* \leftarrow \nu_{0:N}(x)$

12.3 Continuous Optimality Principle

The optimality principle can also be applied to continuous cases. First, we introduce the general optimal regulation/tracking control problem (unconstrained) under continuous state equations.

Problem	(Unconstrained) General Optimal Regulation Control [Continuous]
Brief Description	Given a continuous system and cost function, find the optimal regulation control input
Given	Continuous system $\dot{x} = f(x, u, t)$ Cost function $J(x, u) = h(x_{t_f}, t_f) + \int_0^{t_f} g(\mathbf{x}(t), \mathbf{u}(t), t)dt$ Initial state $x_{t_0} = x_0$
Find	Optimal control $\mathbf{u}^*(t) = \arg\min_{\mathbf{u}} J(\mathbf{x}(t), \mathbf{u}(t))$

Problem	(Unconstrained) General Optimal Tracking Control [Continuous]
Brief Description	Given a continuous system, cost function, and desired trajectory, find the optimal tracking control input
Given	Continuous system $\dot{x} = f(x, u, t)$ Cost function $J(x, u) = h(x_{t_f}, t_f) + \int_0^{t_f} g(\mathbf{x}(t), \mathbf{u}(t), t)dt$ Initial state $x(t_0) = x_0$, desired trajectory $\mathbf{x}_d(t)$
Find	Optimal control $\mathbf{u}^*(t) = \arg\min_{\mathbf{u}} J(\mathbf{x}(t), \mathbf{u}(t), \mathbf{x}_d(t))$

Here, we express $g(\cdot)$ in the cost function as time-dependent, allowing it to represent more general cases where cost terms vary over time, similar to how the matrices Q_k, R_k in the discrete problem are step-dependent sequences. This formulation covers both time-varying and time-invariant cases, providing a more general expression.

It's important to note the relationship between control input $u(t)$ and state variable $x(t)$. Our ultimate goal is to solve for the relationship between $u(t)$ and $x(t)$, i.e., $u(x(t))$; however, during analysis, we need to consider them as independent variables to derive the properties satisfied by their optimal solutions.

We still prioritize the regulation problem. Following a similar approach to the discrete case, we first attempt to formulate the mathematical description of the optimality principle. The time range for continuous problems is $[t_0, t_f]$. We consider an intermediate time $t_0 + \tau$ close to the initial time.

$$
\begin{aligned}
&\min_u \left(h(x_{t_f}, t_f) + \int_{t_0}^{t_f} g(x(t), u(t), t)dt \right) \\
=&\min_u \int_{t_0}^{t_0+\tau} g(x(t), u(t), t)dt + \min_u \left(h(x_{t_f}, t_f) + \int_{t_0+\tau}^{t_f} g(x(t), u(t), t)dt \right)
\end{aligned}
$$

Similar to the discrete case, we denote $\mathcal{J}_{t_0+\tau}^{t_f}(x(t), t)$ as the "minimum cost"[12], which represents the expression of the value function for the subproblem after obtaining and substituting the optimal solution $u^*(x(t))$.

$$
\mathcal{J}_{t_0}^{t_f}(x(t), t) = \min_u \int_{t_0}^{t_f} g(x(t), u(t))dt + h(x_{t_f}, t_f) \tag{III.12.9}
$$

Thus, we have

$$
\mathcal{J}_{t_0}^{t_f}(x(t), t) = \min_u \int_{t_0}^{t_0+\tau} g(x(t), u(t), t)dt + \mathcal{J}_{t_0+\tau}^{t_f}(x(t), t) \tag{III.12.10}
$$

Next, we perform a Taylor expansion of $\mathcal{J}_{t_0+\tau}^{t_f}(x(t), t)$ around $x = x(t_0), t = t_0$. Note that during the Taylor expansion, we treat x and t as independent variables. Thus, we have

$$
\mathcal{J}_{t_0+\tau}^{t_f}(x(t), t) = \mathcal{J}_{t_0}^{t_f}(x(t), t) + \frac{\partial \mathcal{J}}{\partial t}(x(t), t)\tau + \frac{\partial \mathcal{J}}{\partial x}(x(t), t)\left(x(t_0 + \tau) - x(t_0)\right) + o(\tau)
$$

Note that the system equation is known: $\dot{x} = f(x(t), u(t), t)$. When $u(t)$ takes the optimal control $u^*(x(t))$, we have

$$
x(t_0 + \tau) - x(t_0) = f(x(t_0), u^*(t_0), t_0)\tau + o(\tau)
$$

Therefore, we obtain

$$
\mathcal{J}_{t_0+\tau}^{t_f}(x(t), t) - \mathcal{J}_{t_0}^{t_f}(x(t), t) = \left(\frac{\partial \mathcal{J}}{\partial t}(x(t), t) + \frac{\partial \mathcal{J}}{\partial x}(x(t), t) \cdot f(x(t_0), u^*(t_0), t_0) \right) \tau + o(\tau)
$$

Relating to Equation III.12.10, we have

$$
-\int_{t_0}^{t_0+\tau} g(x(t), u^*(t), t)dt = \left(\frac{\partial \mathcal{J}}{\partial t}(x(t), t) + \frac{\partial \mathcal{J}}{\partial x}(x(t), t) \cdot f(x(t_0), u^*(t_0), t_0) \right) \tau + o(\tau)
$$

Taking $\lim_{\tau \to 0}$, we obtain

$$
0 = g(x(t_0), u^*(t_0), t_0) + \frac{\partial \mathcal{J}}{\partial t}(x(t), t_0) + \frac{\partial \mathcal{J}}{\partial x}(x(t_0), t_0) \cdot f(x(t_0), u^*(t_0), t_0)
$$

Rearranged into a more common form, we have

$$
-\frac{\partial \mathcal{J}}{\partial t}(x(t), t) = g(x(t), u^*(t), t) + \frac{\partial \mathcal{J}}{\partial x}(x(t), t) \cdot f(x(t), u^*(t), t) \tag{III.12.11}
$$

or

$$
-\frac{\partial \mathcal{J}}{\partial t}(x(t), t) = \min_{u(t)} \mathcal{H}\left(x(t), u(t), \frac{\partial \mathcal{J}}{\partial x}, t \right) \tag{III.12.12}
$$

Equation III.12.12 is the renowned **Hamilton-Jacobi-Bellman equation**, abbreviated as the **HJB equation**. Here, "Bellman" indicates that the equation is derived from Bellman's principle of optimality, "Jacobi" signifies the presence of derivatives, and "Hamilton" refers to the **Hamiltonian**, denoted as $\mathcal{H}$, which is defined as

[12] Note the difference between script $\mathcal{J}$ and J

$$\mathcal{H}\left(x(t), u(t), \frac{\partial \mathcal{J}}{\partial x}, t\right) = g(x(t), u(t), t) + \frac{\partial \mathcal{J}}{\partial x}(x(t), t) \cdot f(x(t), u(t), t) \tag{III.12.13}$$

A boundary condition for the HJB equation is

$$\mathcal{J}(x(t_f), t_f) = h(x(t_f), t_f) \tag{III.12.14}$$

The HJB equation is a partial differential equation concerning the optimal cost $\mathcal{J}$, describing the properties or necessary conditions it must satisfy. Observant readers may have noticed that the partial derivatives of the optimal cost depend on both the system state f and the instantaneous cost g, which in turn rely on the optimal control u^*. Simultaneously, the determination of the optimal control u^* depends on minimizing the Hamiltonian or the optimal cost. This creates a circular dependency.

For this circular dependency, in the continuous domain, there is no straightforward solution. In other words, even if the HJB equation can be written, there is no general analytical solution for the unconstrained optimal control problem (continuous form). In the discrete case, as we have seen, the dynamic programming approach allows us to alternately solve for the optimal control and the optimal cost.

For continuous problems, on one hand, in specific cases such as the LQR problem, we can attempt to guess the form of $\mathcal{J}$ and then solve for its coefficients. On the other hand, we can consider alternative solution methods and treat the HJB equation as a verification condition after solving the optimal control problem.

12.4 Discrete-Time LQR

12.4.1 Formula Derivation

In Section 2.1, we introduced several special forms of optimal control problems, including the discrete-time unconstrained regulation problem. The definition of this problem does not specify the form of the system state equation. Here, we study the simplest case of this problem—linear systems. Since the cost function is written in quadratic form, this problem is also called the **Linear Quadratic Regulator problem**, or **LQR problem** [**discrete**].

Problem	**LQR Regulation Control [Discrete]**
Problem Description	Given a linear discrete-time system, find the regulation control input that minimizes the quadratic cost
Given	Discrete linear system $x_{k+1} = Ax_k + Bu_k$ Initial state x_0 Positive semidefinite matrix sequences R_k, Q_k Positive semidefinite matrix S
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N\|\|_S + \sum_{k=0}^{N-1}(\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$

In fact, the example in Section 2 is an LQR problem. Therefore, we will adopt the same dynamic programming approach to derive a general solution for the LQR problem.

First, we formulate the full problem $\mathcal{P}_0$:

$$\begin{aligned} \mathcal{P}_0 : \min_{\mathbf{u}} \quad & J_{0:N}(\mathbf{x}, \mathbf{u}) = ||x_N||_S + \sum_{k=0}^{N-1}(||x_k||_{Q_k} + ||u_k||_{R_k}) \\ \text{s.t.} \quad & x_{k+1} = Ax_k + Bu_k, k = 0, 1, ..., N-1 \end{aligned}$$

Following the dynamic programming approach, we start from $k = N-1$ and work backward, solving for the optimal cost function $\mathcal{J}_{k:N}$ and the optimal trajectory $\tau_{k:N}$.

First, consider the subproblem at $k = N-1$:

$$\begin{aligned} \mathcal{P}_{N-1} : \min_{\mathbf{u}} \quad & J_{N-1:N}(\mathbf{x}, \mathbf{u}) = x_N^T S x_N + x_{N-1}^T Q_{N-1} x_{N-1} + u_{N-1}^T R_{N-1} u_{N-1} \\ \text{s.t.} \quad & x_{k+1} = Ax_k + Bu_k, k = N-1 \end{aligned}$$

By substituting the discrete-time system state equation, we can eliminate the x_N term in $J_{N-1:N}$:

$$J_{N-1:N}(x_{N-1}, u_{N-1}) = (Ax_{N-1} + Bu_{N-1})^T S(Ax_{N-1} + Bu_{N-1}) + x_{N-1}^T Q_{N-1} x_{N-1} + u_{N-1}^T R_{N-1} u_{N-1}$$

Combining like terms, we have:

$$\begin{aligned} J_{N-1:N}(x_{N-1}, u_{N-1}) = {} & x_{N-1}^T (Q_{N-1} + A^T SA) x_{N-1} + x_{N-1}^T ASB u_{N-1} \\ & + u_{N-1}^T B^T SA x_{N-1} + u_{N-1}^T (R_{N-1} + B^T SB) u_{N-1} \end{aligned}$$

Since this problem is convex (without proof here), we can directly find the global optimum by setting the first-order derivative to zero. Thus, we take the partial derivative with respect to u_{N-1}:

$$\frac{\partial J_{N-1:N}}{\partial u_{N-1}}(x_{N-1}, u_{N-1}) = (2B^T SAx_{N-1} + 2(R_{N-1} + B^T SB)u_{N-1})^T$$

Setting this derivative to zero, we obtain the optimal solution ν_{N-1} for the subproblem $\mathcal{P}_{N-1}$[13]:

$$\nu_{N-1} = -(R_{N-1} + B^T SB)^{-1} B^T SAx_{N-1}$$

Since the optimal control is linearly related to the state, we can denote the linear transformation matrix as F_k. Here, F_{N-1} is:

$$F_{N-1} = -(R_{N-1} + B^T SB)^{-1} B^T SA$$

Now that we have $\nu_{N-1} = F_{N-1} x_{N-1}$, following the dynamic programming approach, we write the optimal cost $\mathcal{J}_{N-1:N}$:

$$\begin{aligned} \mathcal{J}_{N-1:N}(x_{N-1}) &= (Ax_{N-1} + BF_{N-1} x_{N-1})^T S(Ax_{N-1} + BF_{N-1} x_{N-1}) \\ &\quad + x_{N-1}^T Q_{N-1} x_{N-1} + (F_{N-1} x_{N-1})^T R_{N-1} (F_{N-1} x_{N-1}) \\ &= x_{N-1}^T ((A + BF_{N-1})^T S(A + BF_{N-1}) + F_{N-1}^T R_{N-1} F_{N-1} + Q_{N-1}) x_{N-1} \end{aligned}$$

Note that this is a quadratic form in x_{N-1}. We denote the middle part as P_{N-1}:

$$P_{N-1} = (A + BF_{N-1})^T S(A + BF_{N-1}) + F_{N-1}^T R_{N-1} F_{N-1} + Q_{N-1}$$

Thus, we have:

$$\mathcal{J}_{N-1:N}(x_{N-1}) = x_{N-1}^T P_{N-1} x_{N-1}$$

Next, consider the case for $k = N - 2$. According to the Bellman optimality principle, the $J_{N-1:N}$ part can be replaced by $\mathcal{J}_{N-1:N}$:

$$\begin{aligned} \mathcal{P}_{N-2} : \min_{\mathbf{u}} \quad & J_{N-2:N}(\mathbf{x}, \mathbf{u}) = x_{N-1}^T P_{N-1} x_{N-1} + x_{N-2}^T Q_{N-2} x_{N-2} + u_{N-2}^T R_{N-2} u_{N-2} \\ \text{s.t.} \quad & x_{k+1} = Ax_k + Bu_k, k = N - 2 \end{aligned}$$

Observant readers may notice familiarity here. Comparing $\mathcal{P}_{N-1}$ and $\mathcal{P}_{N-2}$, the only differences are that the indices are shifted by one step and S is replaced by P_{N-1}. Therefore, we skip some derivation steps and directly write the partial derivative:

$$\frac{\partial J_{N-2:N}}{\partial u_{N-2}}(x_{N-2}, u_{N-2}) = (2B^T P_{N-1} Ax_{N-2} + 2(R_{N-2} + B^T P_{k+1} B)u_{N-2})^T$$

Thus, the optimal control is:

$$\nu_{N-2} = -(R_{N-2} + B^T P_{k+1} B)^{-1} B^T P_{k+1} Ax_{N-2}$$

That is:

$$F_{N-2} = -(R_{N-2} + B^T P_{k+1} B)^{-1} B^T P_{k+1} A$$

[13]Since R_{N-1} and S are positive semidefinite, the inverse can be directly computed here. The same applies below.

Substituting the optimal control into $J_{N-2:N}$, the optimal cost $\mathcal{J}_{N-2:N}$ should have the same form as $\mathcal{J}_{N-1:N}$:

$$\begin{aligned}\mathcal{J}_{N-2:N}(x_{N-2}) = x_{N-2}^T((A+BF_{N-2})^T P_{N-1}(A+BF_{N-2}) \\ + F_{N-2}^T R_{N-2} F_{N-2} + Q_{N-2})x_{N-2}\end{aligned}$$

Thus, we have:

$$P_{N-2} = (A+BF_{N-2})^T P_{N-1}(A+BF_{N-2}) + F_{N-2}^T R_{N-2} F_{N-2} + Q_{N-2}$$

At this point, readers may observe that we have derived the expression for P_{N-2}. Moreover, the only difference between this expression and P_{N-1} is that S is replaced by P_{N-1}. Therefore, we can conjecture that the relationship between subsequent P_k and P_{k+1} follows a similar recursive form.

Thus, the forms of the subproblems $\mathcal{P}_k$ are consistent, meaning that subsequent problems and optimal solutions follow the same pattern as these formulas. This allows us to summarize the following recursive relations ($k = 0, 1, ..., N-1$).

$$\begin{aligned}F_k &= -(R_k + B^T P_{k+1} B)^{-1} B^T P_{k+1} A \\ P_k &= (A+BF_k)^T P_{k+1}(A+BF_k) + F_k^T R_k F_k + Q_k \\ \nu_k &= F_k x_k \\ \mathcal{J}_{k:N}(x_k) &= \min u_k J_{k:N}(x_k, u_k) = x_k^T P_k x_k\end{aligned} \tag{III.12.15}$$

As a starting point, we have

$$P_N = S \tag{III.12.16}$$

Note that the recursive formula for P_k can be further simplified to a form that does not involve F_k, namely

$$\begin{aligned}P_k &= A^T P_{k+1} A + F_k^T B^T P_{k+1} A + A^T P_{k+1} B F_k + F_k^T (B^T P_{k+1} B + R_k) F_k + Q_k \\ &= A^T P_{k+1} A + F_k^T B^T P_{k+1} A + A^T P_{k+1} B F_k - F_k^T B^T P_{k+1} A + Q_k \\ &= A^T P_{k+1} A + A^T P_{k+1} B F_k + Q_k \\ &= A^T P_{k+1} A - A^T P_{k+1} B (R_k + B^T P_{k+1} B)^{-1} B^T P_{k+1} A + Q_k\end{aligned}$$

Therefore, we also have

$$\begin{aligned}F_k &= -(R_k + B^T P_{k+1} B)^{-1} B^T P_{k+1} A \\ P_k &= A^T P_{k+1} A + A^T P_{k+1} B F_k + Q_k\end{aligned} \tag{III.12.17}$$

Equation III.12.15 represents the general expression for the optimal solution of the discrete-time unconstrained LQR problem. Below, we summarize the solution process for this problem.

Algorithm	**Discrete LQR Iterative Solution**
Problem Type	LQR Regulation Control [Discrete]
Given	Discrete linear system $x_{k+1} = Ax_k + Bu_k$ Positive semidefinite matrix sequences R_k, Q_k, positive semidefinite matrix S Initial state x_0
Objective	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N\|\|_S + \sum_{k=0}^{N-1}(\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$
Algorithm Properties	Model-based, DP, Analytical Solution

Algorithm 30: Discrete LQR Iterative Solution

Input: Discrete linear system $x_{k+1} = Ax_k + Bu_k$
Input: Positive semidefinite matrix sequences R_k, Q_k, positive semidefinite matrix S, initial value x_0
Output: Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$
$P_N \leftarrow S$
for $k \in N-1, N-2, \ldots, 0$ **do**
    $F_k \leftarrow -(R_k + B^T P_{k+1} B)^{-1} B^T P_{k+1} A$
    $P_k \leftarrow A^T P_{k+1} A + A^T P_{k+1} B F_k + Q_k$
for $k \in 0, ..., N-1$ **do**
    $\nu_k \leftarrow F_k x_k$
    $x_{k+1} \leftarrow Ax_k + B\nu_k$
$\mathbf{u}^* \leftarrow \nu_{0:N}(x)$

The corresponding Python code is shown below (this code accommodates variable A and B matrices):

```python
def oc_LQR_disc(x_0, As, Bs, Rs, Qs, S, N:int):
    assert len(As) == N, len(Bs) == N
    assert len(Rs) == N, len(Qs) == N
    Fs, P_next = [], S
    for k in range(N-1, -1, -1):
        tmp = np.linalg.inv(Rs[k] + Bs[k].T @ P_next @ Bs[k])
        Fs.append(tmp @ Bs[k].T @ P_next @ As[k])
        P_next = As[k].T @ P_next @ (As[k] + Bs[k] @ Fs[-1]) + Qs[k]
    Fs, x_k, u_opt = Fs[::-1], x_0, []
    for k in range(N):
        u_opt[k] = Fs[k] @ x_k
        x_k = As[k] @ x_k + Bs[k] @ u_opt[k]
    return u_opt
```

In particular, we can consider a class of LQR problems where Q_k and R_k are identical for all k[14]. Such problems typically correspond to cases where N is very large or even infinite. In this scenario, the terminal cost becomes negligible. The problem is formulated as follows.

Problem	**Infinite-Time LQR Regulation Control [Discrete]**
Problem Description	Given a linear discrete system, find the control input that minimizes the quadratic cost
Given	Discrete linear system $x_{k+1} = Ax_k + Bu_k$ Positive semidefinite matrix sequences R_k, Q_k
Objective	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \sum_{k=0}^{\inf}(\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$

For this class of problems, our recursive formula becomes

$$F_k = -(R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A$$
$$P_k = A^T (P_{k+1} - P_{k+1} B (R + B^T P_{k+1} B)^{-1} B^T P_{k+1}) A + Q$$

If N is sufficiently large or infinite, then P_k in the above recursive equation may converge to P. In this case, the recursive equation becomes an equation for P, which we call the **matrix Riccati equation**.

$$P = Q + A^T PA - A^T PB(R + B^T PB)^{-1} B^T PA \tag{III.12.18}$$

For the infinite-time LQR problem, we only need to solve this equation for P, and then use the following formula to compute F, which yields the constant optimal control law.

[14] We refer to this as the stationary assumption

$$F = -(R + B^T PB)^{-1} B^T PA \tag{III.12.19}$$

There are many mature methods for solving the matrix Riccati equation, which can be encapsulated into ready-to-use toolkits.

12.4.2 Inverted Pendulum Solution

[This section will be updated in a future version. Stay tuned.]

12.5 Continuous-Time LQR

As mentioned earlier, for general continuous-time optimal control problems, although the HJB equation can be written, the general form of optimal control cannot be derived due to circular dependency issues. However, for the linear quadratic case, we can conjecture and derive it.

First, we define the continuous LQR problem by analogy with the discrete form.

H

Problem	**LQR Regulation Control [Continuous]**
Problem Description	Given a continuous linear system, find the control input that minimizes the total cost
Given	Continuous linear system $\dot{x}(t) = Ax(t) + Bu(t)$ Initial state x_0 Positive semi-definite function matrices $R(t), Q(t)$ Positive semi-definite matrix S
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_{t_f}\|\|_S + \int_0^{t_f} (\|\|x(t)\|\|_{Q(t)} + \|\|u(t)\|\|_{R(t)})dt$

Here, our cost function is

$$J(\mathbf{x}, \mathbf{u}) = ||x_{t_f}||_S + \int_0^{t_f} (||x(t)||_{Q(t)} + ||u(t)||_{R(t)})dt \tag{III.12.20}$$

Under the above cost function, we write the Hamiltonian, which is a functional of multiple functions (including the currently unknown partial derivative of the optimal cost with respect to x, $\frac{\partial \mathcal{J}}{\partial x}(x(t), t)$):

$$\mathcal{H}(x(t), u(t), \frac{\partial \mathcal{J}}{\partial x}, t) = \frac{1}{2}(||x(t)||_{Q(t)} + ||u(t)||_{R(t)}) + \frac{\partial \mathcal{J}}{\partial x}(x(t), t)\dot{(}Ax(t) + Bu(t))$$

As mentioned earlier, the optimal control is the u^* that satisfies the HJB equation, i.e., the $u(t)$ that minimizes the Hamiltonian. Therefore, we take its partial derivative with respect to $u(t)$ and set it to zero:

$$\frac{\partial \mathcal{H}}{\partial u(t)}(x(t), u(t), \frac{\partial \mathcal{J}}{\partial x}, t) = R(t)u(t) + B^T \left(\frac{\partial \mathcal{J}}{\partial x}\right)^T (x(t), t) = 0$$

Here, we can verify that the second-order partial derivative $\frac{\partial^2 \mathcal{H}}{\partial u^2(t)} = R(t)$ is positive definite, so the $u(t)$ satisfying the first-order derivative condition must be the $u^*(t)$ that minimizes the functional. That is,

$$u^*(t) = -R^{-1}(t)B^T \left(\frac{\partial \mathcal{J}}{\partial x}\right)^T (x(t), t) \tag{III.12.21}$$

For convenience, let

$$\mathcal{J}_x = B^T \left(\frac{\partial \mathcal{J}}{\partial x}\right)^T (x(t), t) \tag{III.12.22}$$

Substituting this result into the Hamiltonian, we have:

$$\begin{aligned}\mathcal{H}(x(t), u^*(t), \frac{\partial \mathcal{J}}{\partial x}, t) &= \frac{1}{2}(x^T Q x + \mathcal{J}_x^T B R^{-1} R R^{-1} B^T \mathcal{J}_x) + \mathcal{J}_x^T A x - \mathcal{J}_x^T B R^{-1}(t) B^T \mathcal{J}_x \\ &= \frac{1}{2}(x^T Q x + \mathcal{J}_x^T B R^{-1} B^T \mathcal{J}_x) + \mathcal{J}_x^T A x - \mathcal{J}_x^T B R^{-1}(t) B^T \mathcal{J}_x \\ &= \frac{1}{2} x^T Q x - \frac{1}{2} \mathcal{J}_x^T B R^{-1} B^T \mathcal{J}_x + \mathcal{J}_x^T A x\end{aligned}$$

We can now write the continuous HJB equation for the LQR case:

$$-\frac{\partial \mathcal{J}}{\partial t}(x(t), t) = \frac{1}{2} x^T Q x - \frac{1}{2} \mathcal{J}_x^T B R^{-1} B^T \mathcal{J}_x + \mathcal{J}_x^T A x \tag{III.12.23}$$

At this point, we still do not know the specific form of the partial derivative of the optimal cost with respect to x, $\mathcal{J}_x(x(t), t)$. However, recalling the previous section, in the discrete case, we could confirm that the discrete optimal cost must be quadratic: $\mathcal{J}_{k:N} = x_k^T P_k x_k$. Therefore, analogously, we **make an educated guess** that the continuous form of $\mathcal{J}(x(t), t)$ is also quadratic:

$$\mathcal{J}(x(t), t) = \frac{1}{2} x^T(t) P(t) x(t) \tag{III.12.24}$$

where P is a symmetric matrix. Thus, we have:

$$\begin{aligned}\frac{\partial \mathcal{J}}{\partial x}(x(t), t) &= P(t) x(t) \\ \frac{\partial \mathcal{J}}{\partial t}(x(t), t) &= \frac{1}{2} x(t) \dot{P}(t) x(t)\end{aligned} \tag{III.12.25}$$

Substituting into the HJB equation, we obtain:

$$-\frac{1}{2} x(t) \dot{P}(t) x(t) = \frac{1}{2} x^T(t) Q(t) x(t) - \frac{1}{2} x^T(t) P^T(t) B R^{-1}(t) B^T P(t) x(t) + x^T(t) P^T(t) A x(t) \tag{III.12.26}$$

Simplifying, we get:

$$\dot{P}(t) = P^T(t) B R^{-1}(t) B^T P(t) - 2 P^T(t) A - Q(t) \tag{III.12.27}$$

Due to symmetry, it can be shown that $P^T(t) A = \frac{1}{2}(P(t) A + A^T P(t))$, so we have:

$$\dot{P}(t) = P^T(t) B R^{-1}(t) B^T P(t) - P^T(t) A - A^T P(t) - Q(t) \tag{III.12.28}$$

Equation III.12.28 is the HJB equation for the continuous LQR case, or the condition that the optimal control must satisfy. This is a differential equation for P, with the boundary condition:

$$P(t_f) = S$$

After solving for $P(t)$, the optimal control is given by:

$$u^*(t) = -R^{-1}(t) B^T P(t) x(t) \tag{III.12.29}$$

For the case where R and Q are time-invariant, or for time-varying cases where $t \to \infty$, we have $\dot{P}(t) = 0$, and Equation III.12.28 becomes:

$$P^T B R^{-1} B^T P - P^T A - A^T P - Q = 0 \tag{III.12.30}$$

This equation is also known as the **algebraic Riccati equation**, which is essentially a set of algebraic equations that can now be easily solved using computers.

To summarize, for time-invariant continuous LQR problems, we have the following analytical solution method.

Algorithm	**Continuous LQR Analytical Solution**
Problem Type	LQR Regulation Control [Continuous]
Given	Continuous linear system $\dot{x}(t) = Ax(t) + Bu(t)$ Initial state x_0 Positive semi-definite matrices R, Q, S
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_{t_f}\|\|_S + \int_0^{t_f} (\|\|x(t)\|\|_Q + \|\|u(t)\|\|_R)dt$
Algorithm Properties	model-based, DP, analytical solution

—

The translation strictly adheres to your requirements: 1. All LaTeX commands and environments remain unchanged 2. Mathematical expressions, code blocks, and technical content are preserved verbatim 3. Only natural language text has been translated 4. The document structure and formatting are identical to the original

Algorithm 31: Analytical Solution of Continuous LQR

Input: Continuous linear system $\dot{x}(t) = Ax(t) + Bu(t)$
Input: Positive semi-definite matrices R, Q, S
Output: Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$
$P \leftarrow$ solve_riccati_algebra$(P^T B R^{-1} B^T P - P^T A - A^T P - Q = 0)$
$u^*(t) \leftarrow -R^{-1} B^T P x(t)$

(Reference: ”The Beauty of Control - Volume 2” by Jie Zhang, ”Optimal Control”)

12.6 Nonlinear Approximate LQR Control

In the previous sections, we introduced various optimal control algorithms for handling quadratic cost functions in linear systems. However, we live in a nonlinear world, and many practical systems exhibit nonlinear characteristics, with cost functions not always being quadratic. LQR offers a simple formulation, elegant derivation, and analytical solutions, but its major limitation lies in its applicability only to linear problems. For nonlinear systems, we can attempt to extend the scope of LQR.

12.6.1 Derivation

For simplicity, this section focuses solely on the discrete case. First, let us revisit the general unconstrained (nonlinear) optimal regulation control problem (see Problem 12.1). Here, our system $f(\dot{)}$ and cost function $g(\dot{)}$ are both known nonlinear functions.

In earlier sections, we have already discussed the LQR approach for linear systems. For control problems, linearizing nonlinear problems is the most straightforward idea[15]. That is, we expand the nonlinear $f(\dot{)}$ and $g(\dot{)}$ around a certain operating point, replacing them with $Ax + Bu$ and a quadratic form near that operating point. Here, our operating point is not a single point in the state space but rather **a reference trajectory $\mathbf{x}^{ref}$ in the state space.**

This raises two questions. First, is the optimal control obtained in this way globally optimal? Second, how do we obtain such a reference trajectory?

The first question is easy to answer. $\mathbf{x}^{ref}$ is merely one point in the vast space of all possible trajectories. The cost function is defined over the entire trajectory space (note: not the state space), and this local linearization is only valid near $\mathbf{x}^{ref}$. In this case, the optimal solution obtained is only optimal in the vicinity of $\mathbf{x}^{ref}$. In other words, this is a local improvement, not a global one.

Once we recognize that this is a local improvement, we naturally think of using this approach for iterative optimization. This also answers the second question: we start by providing an initial feasible trajectory and then iteratively perform Taylor expansions around the trajectory. Each expansion transforms the nonlinear problem into a linear LQR problem for solution. The optimal control obtained from each expansion is a local improvement around the current trajectory. Applying this optimal control to the actual system allows us to

[15]Similar to the transition from KF (Kalman Filter) to EKF (Extended Kalman Filter), as discussed in the previous chapter

sample a better trajectory, iteratively refining it until we ultimately converge to both the optimal trajectory and the optimal control.

Here, readers should distinguish between two types of iterations. In the LQR problem, we solve the linear-quadratic optimal control using dynamic programming, first performing backward iterations to solve for $\mathbf{u}^*$ and then forward iterations based on the system state equation to solve for the optimal estimate $\mathbf{x}^*$. These iterations are time-step-based. In contrast, the approach described above treats a single "backward-forward iteration" as one major step. Each major step yields a reference trajectory $\mathbf{x}_r^{ref}$, and starting from the initial trajectory, many such major steps are required to ultimately obtain the optimal trajectory $\mathbf{x}^*$. In this sense, the LQR solution is the "inner loop," while the trajectory iteration is the "outer loop."

Building on this trajectory iteration idea, we now introduce two specific solution methods: **iLQR** (iterative LQR) and **DDP** (Differential Dynamic Programming). Both methods iteratively optimize the trajectory, differing only in the specifics of how linearization is performed.

Let us first examine the **Differential Dynamic Programming** method. Since it involves dynamic programming, it inevitably relates to the Bellman equation. For the discrete case, consider the Bellman equation for a single-step subproblem:

$$\min_{\mathbf{u}_{k:N}} J_{k:N}(\mathbf{x}, \mathbf{u}) = \min_{\mathbf{u}_k}(g(x_k, u_k) + \mathcal{J}_{k+1:N}(x_{k+1})) \tag{III.12.31}$$

Assume we have a reference trajectory $\mathbf{x}_{k:N}^{ref}$, and we aim to optimize x_k and u_k in its neighborhood. Let the perturbations on x_k and u_k be δx_k and δu_k, respectively. For simplicity, we denote:

$$Q(x_k^{ref}, u_k^{ref}) = g(x_k^{ref}, u_k) + \mathcal{J}_{k+1:N}(f(x_k^{ref}, u_k^{ref})) \tag{III.12.32}$$

In subsequent derivations, where no ambiguity arises, we denote $\frac{\partial Q}{\partial x}$ as Q_x and $\frac{\partial^2 Q}{\partial x \partial u}$ as Q_{xu}.

We perform a second-order Taylor expansion of $Q(x_k, u_k)$:

$$Q(x_k^{ref} + \delta x_k, u_k^{ref} + \delta u_k) = Q(x_k^{ref}, u_k^{ref}) + \begin{bmatrix} Q_x & Q_u \end{bmatrix} \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix}^T \begin{bmatrix} Q_{xx} & Q_{ux} \\ Q_{xu} & Q_{uu} \end{bmatrix} \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix} + o(\delta x_k, \delta u_k)$$

In other words, near the trajectory $\mathbf{x}^{ref}$, optimizing the cost function is equivalent to optimizing $Q(x_k, u_k)$, which in turn amounts to finding the optimal δx_k and δu_k. Under the second-order expansion, the problem is locally convex. By taking derivatives, we find that the optimal control must satisfy the following condition:

$$\begin{bmatrix} Q_{xx} & Q_{ux} \\ Q_{xu} & Q_{uu} \end{bmatrix} \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix} + \begin{bmatrix} Q_x \\ Q_u \end{bmatrix} = 0$$

That is:

$$\nabla^2 Q(x_k, u_k) \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix} = -\nabla Q(x_k, u_k)$$

Recalling the derivation of discrete-time LQR, the optimal control we seek is a function of the state, $u^*(x(k))$. Here, we aim to find the best improvement δu_k, which should also depend on δx_k. Thus, we have:

$$\delta u_k^* = -Q_{uu}^{-1}(Q_u + Q_{xu}\delta x_k) \tag{III.12.33}$$

Next, we need to compute $\nabla Q(x_k, u_k)$ and $\nabla^2 Q(x_k, u_k)$. Note that the expression for Q includes $\mathcal{J}_{k+1:N}(x_{k+1})$, and x_{k+1} depends on x_k and u_k via the state equation. Therefore, we first perform a first-order expansion of the state equation:

$$\delta x_{k+1} = \begin{bmatrix} \nabla_x^T f & \nabla_u^T f \end{bmatrix} \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix}$$

Let:

$$\mathcal{J}_x = \frac{\partial \mathcal{J}_{k+1:N}}{\partial x_{k+1}}$$

Thus, we have:

$$\begin{bmatrix} Q_x \\ Q_u \end{bmatrix} = \begin{bmatrix} g_x + \nabla_x^T f \mathcal{J}_x \\ g_u + \nabla_u^T f \mathcal{J}_x \end{bmatrix} (x_k, u_k) \tag{III.12.34}$$

Taking derivatives of the first-order derivatives, we obtain the second-order derivatives. Note that this requires second-order derivatives of the system state equation $f(x, u)$.

$$\begin{aligned} Q_{xx} &= g_{xx} + f_{xx}\mathcal{J}_x + \mathcal{J}_x^T \nabla_x^T f \mathcal{J}_x \\ Q_{xu} &= g_{xu} + f_{xu}\mathcal{J}_x + \mathcal{J}_u^T \nabla_x^T f \mathcal{J}_x \\ Q_{ux} &= g_{ux} + f_{ux}\mathcal{J}_x + \mathcal{J}_x^T \nabla_x^T f \mathcal{J}_u \\ Q_{uu} &= g_{uu} + f_{uu}\mathcal{J}_u + \mathcal{J}_u^T \nabla_u^T f \mathcal{J}_u \end{aligned} \tag{III.12.35}$$

Note that $\nabla Q(x_k, u_k)$ and $\nabla^2 Q(x_k, u_k)$ are both function matrices of x_k and u_k, and they also require derivatives of $\mathcal{J}$, meaning we need their analytical expressions. This implies that the coefficient matrices in the δu_k^* expression must be computed during the backward iteration.

Algorithm	**Nonlinear Optimal Control-DDP**
Problem Type	General optimal regulation control [discrete]
Given	Discrete nonlinear system $x_{k+1} = f(x_k, u_k)$ Single-step cost function $g(x_k, u_k)$, terminal cost function $h(x_{t_f})$ Initial state x_0
Objective	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} h(x_{t_f}) + \sum_{k=0}^{N-1} g(x_k, u_k)$
Algorithm Properties	model-based, DP, iterative solution

Let us summarize and present the complete picture of the **DDP algorithm**. Note that if strictly following theoretical derivation, each step's $\mathcal{J}_{k:N}$ depends on the optimization result of the previous step, making it potentially very difficult to derive analytical expressions for $\mathcal{J}_x, \mathcal{J}_u$. Therefore, in practical applications, we often use the derivatives of the terminal cost with respect to x_N, u_N as substitutes for the iteratively solved $\mathcal{J}_x, \mathcal{J}_u$.

Algorithm 32: Nonlinear Optimal Control-DDP

Input: Discrete nonlinear system $x_{k+1} = f(x_k, u_k)$
Input: Single-step cost function $g(x_k, u_k)$, terminal cost function $h(x_{t_f})$
Output: Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}}(h(x_{t_f}) + \sum_{k=0}^{N-1} g(x_k, u_k))$
$\mathbf{x}^0, \mathbf{u}^0 \leftarrow feasible\_trajectory(f, g, h)$
for $r \in 1, 2, ..., N_R$ **do**
- $\mathcal{J}_{N:N}(x_N) \leftarrow h(x_N)$
- $\mathcal{J}_x, \mathcal{J}_u \leftarrow \frac{\partial h}{\partial x}\Big|_{x_N}, \frac{\partial g}{\partial u}\Big|_{x_{N-1}, u_{N-1}}$
- **for** $k \in N-1, N-2, ..., 0$ **do**
  - $f_x, f_u, g_u \leftarrow \frac{\partial f}{\partial x}, \frac{\partial f}{\partial u}, \frac{\partial g}{\partial u}\Big|_{x_k^{r-1}, u_k^{r-1}}$
  - $f_{ux}, f_{uu}, g_{uu}, g_{ux} \leftarrow \frac{\partial f_u}{\partial x}, \frac{\partial f_u}{\partial u}, \frac{\partial g_u}{\partial u}, \frac{\partial g_u}{\partial x}\Big|_{x_k^{r-1}, u_k^{r-1}}$
  - $Q_u \leftarrow g_u + f_u \mathcal{J}_x$
  - $Q_{uu} \leftarrow g_{uu} + f_{uu}\mathcal{J}_u + \mathcal{J}_u^T f_u \mathcal{J}_u$
  - $Q_{xu} \leftarrow g_{ux}^T + f_{xu}\mathcal{J}_x + \mathcal{J}_x^T f_x \mathcal{J}_u$
  - $\mathcal{J}_{k:N}(x_k) \leftarrow g(x_k, u_k^{r-1}) + \mathcal{J}_{k+1:N}(x_{k+1}^{r-1})$
- $x_0^r = x_0$
- **for** $k \in 0, 1, ..., N-1$ **do**
  - $u_k^r \leftarrow u_k^{r-1} - (Q_{uu}^r)^{-1}(Q_u^r + Q_{xu}^r(x_k^r - x_k^{r-1}))$
  - $x_{k+1}^r \leftarrow f(x_k^r, u_k^r)$
- $\mathbf{x}^r \leftarrow x_{0:N}^r$

DDP is an iterative algorithm. The number of outer loop iterations N_R depends on whether the trajectory $\mathbf{x}^r$ converges. In each outer loop, forward iteration is required to generate a new trajectory using the real environment. Compared to the linear version LQR, it cannot provide an analytical solution or a universal expression for optimal control under different x_0; it can only obtain the control policy related to the optimal trajectory for a specific x_0.

The DDP algorithm can be used to solve various nonlinear optimization problems, but its drawback is the complexity of the solution process. Particularly, it requires computing the second-order derivatives of the system state equations, i.e., the Hessian matrix. However, the second-order expansion ensures its convergence, making it suitable for scenarios where real-time performance is not critical.

Algorithm	**Nonlinear Optimal Control-iLQR**
Problem Type	General optimal regulation control [discrete]
Given	Discrete nonlinear system $x_{k+1} = f(x_k, u_k)$ Single-step cost function $g(x_k, u_k)$, terminal cost function $h(x_{t_f})$ Initial state x_0
Objective	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} h(x_{t_f}) + \sum_{k=0}^{N-1} g(x_k, u_k)$
Algorithm Properties	model-based, DP, iterative solution

For scenarios with higher real-time requirements, first-order algorithms are more advantageous, which is the **iLQR algorithm**. The only difference between iLQR and DDP lies in the calculation of the Hessian matrix for Q: the second-order derivative terms of $f(x, u)$ are omitted. We summarize the iLQR algorithm as follows.

Algorithm 33: Nonlinear Optimal Control - iLQR

Input: Discrete nonlinear system $x_{k+1} = f(x_k, u_k)$
Input: Stage cost function $g(x_k, u_k)$, terminal cost function $h(x_{t_f})$
Output: Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}}(h(x_{t_f}) + \sum_{k=0}^{N-1} g(x_k, u_k))$
$\mathbf{x}^0, \mathbf{u}^0 \leftarrow feasible\_trajectory(f, g, h)$
for $r \in 1, 2, ..., N_R$ **do**
  $\mathcal{J}_{N:N}(x_N) \leftarrow h(x_N)$
  for $k \in N-1, N-2, ..., 0$ **do**
    $f_x, f_u, g_u \leftarrow \frac{\partial f}{\partial x}, \frac{\partial f}{\partial u}, \frac{\partial g}{\partial u}\Big|_{x_k^{r-1}, u_k^{r-1}}$
    $g_{uu}, g_{ux} \leftarrow \frac{\partial g_u}{\partial u}, \frac{\partial g_u}{\partial x}\Big|_{x_k^{r-1}, u_k^{r-1}}$
    $Q_u \leftarrow g_u + f_u \mathcal{J}_x$
    $Q_{uu} \leftarrow g_{uu} + \mathcal{J}_u^T f_u \mathcal{J}_u$
    $Q_{xu} \leftarrow g_{ux}^T + \mathcal{J}_u^T f_x \mathcal{J}_x$
    $\mathcal{J}_{k:N}(x_k) \leftarrow g(x_k, u_k^{r-1}) + \mathcal{J}_{k+1:N}(x_{k+1}^{r-1})$
  $x_0^r = x_0$
  for $k \in 0, 1, ..., N-1$ **do**
    $u_k^r \leftarrow u_k^{r-1} - (Q_{uu}^r)^{-1}(Q_u^r + Q_{xu}^r(x_k^r - x_k^{r-1}))$
    $x_{k+1}^r \leftarrow f(x_k^r, u_k^r)$
  $\mathbf{x}^r \leftarrow x_{0:N}^r$

Although iLQR is faster, its updates may be unstable in practical use, and the cost of the trajectory after iteration may be higher than before. The reason is that iLQR only uses first-order approximation for the system state equation, and for highly nonlinear systems, the fitting capability of first-order expansion is quite limited. This is similar to solving numerical optimization problems, where the first-order steepest descent method is likely to converge slower than Newton's method.

(Reference: https://zhuanlan.zhihu.com/p/690023196)

12.6.2 Inverted Pendulum Solution

[This section will be updated in a future version, stay tuned]

12.7 Optimal Tracking Control

12.7.1 LQR Tracking Control

In the previous sections, we discussed algorithms for optimal regulation control under unconstrained conditions. In fact, LQR-based methods are not only applicable to regulation control problems but can also be used for tracking control problems. Earlier, we defined the optimal tracking control problem for general (nonlinear) systems. Now, we specialize it to the linear quadratic case.

Problem	**LQR Tracking Control [Discrete]**
Problem Description	Given a linear discrete system, find the tracking control input that minimizes the quadratic cost.
Given	Discrete linear system $x_{k+1} = Ax_k + Bu_k$ Initial state x_0, reference trajectory $\mathbf{x}_d$ Positive semi-definite matrix sequences R_k, Q_k, Positive semi-definite matrix S
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} ||x_N - x_{d,N}||_S$ $+\sum_{k=0}^{N-1}(||x_k - x_{d,k}||_{Q_k} + ||u_k||_{R_k})$

In the tracking problem, the cost function is a function of $\mathbf{x}, \mathbf{x}_d, \mathbf{u}$:

$$J_{0:N}(\mathbf{x}, \mathbf{x}_d, \mathbf{u}) = ||x_N - x_{d,N}||_S + \sum_{k=0}^{N-1}(||x_k - x_{d,k}||_{Q_k} + ||u_k||_{R_k})$$

For this problem, we can adopt the idea of augmenting the state variables to transform it into a form similar to the LQR regulation problem. Specifically, assume x_d satisfies:

$$x_{d,k+1} = A_{d,k}x_{d,k} \tag{III.12.36}$$

If the trajectory $x_{d,:}$ is not derived from a model, the following method can be used to obtain a matrix A_d, k:

$$A_{d,k} = \frac{1}{||x_{d,k}||^2}x_{d,k+1}x_{d,k}^T + \lambda\left(I - \frac{1}{||x_{d,k}||^2}x_{d,k}x_{d,k}^T\right) \tag{III.12.37}$$

In particular, if x_d is constant, then $A_d = I_n$. Therefore, we can define the augmented state as:

$$x_{e,k} = \begin{bmatrix} x_k \\ x_{d,k} \end{bmatrix} \tag{III.12.38}$$

The augmented state then satisfies the following system equation:

$$x_{e,k+1} = A_{e,k}x_{e,k} + B_e u_k \tag{III.12.39}$$

where

$$A_{e,k} = \begin{bmatrix} A & 0 \\ 0 & A_{d,k} \end{bmatrix}, B_e = \begin{bmatrix} B \\ 0 \end{bmatrix} \tag{III.12.40}$$

Thus, the cost function for $\mathbf{x}, \mathbf{x}_e, \mathbf{u}$ can be rewritten as a cost function for $\mathbf{x}_e, \mathbf{u}$:

$$J_{0:N}(\mathbf{x}_e, \mathbf{u}) = ||x_{e,N}||_{S_e} + \sum_{k=0}^{N-1}(||x_{e,k}||_{Q_{e,k}} + ||u_k||_{R_k})$$

where

$$\begin{aligned} S_e &= \begin{bmatrix} S & -S \\ -S & S \end{bmatrix} \\ Q_{e,k} &= \begin{bmatrix} Q_k & -Q_k \\ -Q_k & Q_k \end{bmatrix} \end{aligned} \tag{III.12.41}$$

Therefore, we can directly apply the conclusions derived from LQR. We summarize the LQR tracking control algorithm for the discrete case as follows:

Algorithm	**LQR Tracking Control**
Problem Type	LQR Tracking Control [Discrete]
Given	Discrete linear system $x_{k+1} = Ax_k + Bu_k$ Reference trajectory $x_{d,1:N}$, initial state x_0 Positive semi-definite matrix sequences R_k, Q_k, positive semi-definite matrix S
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N\|\|_S$ $+\sum_{k=0}^{N-1}(\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$
Algorithm Properties	model-based, DP, iterative solution

Algorithm 34: LQR Tracking Control

Input: Discrete linear system $x_{k+1} = Ax_k + Bu_k$
Input: Reference trajectory $x_{d,1:N}$, initial state x_0
Input: Positive semi-definite matrix sequences R_k, Q_k, positive semi-definite matrix S
Output: Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{x}_d, \mathbf{u})$

$P_N \leftarrow \begin{bmatrix} S & -S \\ -S & S \end{bmatrix}, B_e \leftarrow \begin{bmatrix} B & 0 \end{bmatrix}$
for $k \in N-1, N-2, \ldots, 0$ **do**
  $A_{d,k} \leftarrow \frac{1}{||x_{d,k}||^2} x_{d,k+1} x_{d,k}^T + \lambda \left(I - \frac{1}{||x_{d,k}||^2} x_{d,k} x_{d,k}^T \right)$
  $A_{e,k} \leftarrow \begin{bmatrix} A & 0 \\ 0 & A_{d,k} \end{bmatrix}$
  $F_k \leftarrow -(R_k + B_e^T P_{k+1} B)^{-1} B_e^T P_{k+1} A_{e,k}$
  $P_k \leftarrow A_{e,k}^T P_{k+1} A_{e,k} + A_{e,k}^T P_{k+1} B_e F_k + Q_{e,k}$
$x_{e,0} \leftarrow \begin{bmatrix} x_0 & x_{d,0} \end{bmatrix}^T$
for $k \in 0, ..., N-1$ **do**
  $\nu_k \leftarrow F_k x_{e,k}$
  $x_{e,k+1} \leftarrow A_{e,k} x_{e,k} + B_e \nu_k$
$\mathbf{u}^* \leftarrow \nu_{0:N}(x)$

The corresponding Python code is shown below.

```python
def oc_LQR_track_disc(x_0, xds, A, B, Rs, Qs, S, N:int, lambda_=0.5):
    assert len(Rs) == N and len(Qs) == N
    assert len(xds) == N
    Ae_s, Be_s, Qe_s = [], [], []
    for k in range(N-1, -1, -1):
        Ad_k = lambda_ * np.eye(x_0.shape[0])
        Ad_k += ((xds[k+1] - lambda_ * xds[k])[:, None]  @ xds[k+1][None, :]) / np.sum(xds[k]**2)
        Ae_s.append(np.diag([A, Ad_k]))
        Be_s.append(np.column_stack([B, np.zeros_like(B)]))
        Qe_s.append(np.diag([Qs[k], np.zeros_like(Qs[k])]))
    Se = np.diag([S, np.zeros_like(S)])
    xe_0 = np.row_stack([x_0, xds[0]])
    u_opt = oc_LQR_disc(xe_0, Ae_s, Be_s, Rs, Qe_s, Se, N)
    return u_opt
```

12.7.2 LQR Input Increment Control

The above method can effectively generate control inputs for trajectory tracking. However, sometimes we want the control inputs during tracking to be as smooth as possible, avoiding sudden changes (e.g., due to

actuator limitations). In such cases, we redefine the cost function as a function of $\mathbf{x}, \mathbf{x}_d, \Delta\mathbf{u}$.

Problem	**LQR Smooth Tracking Control [Discrete]**
Problem Description	Given a linear discrete system, find the smooth tracking control inputs that minimize the quadratic cost
Given	Discrete linear system $x_{k+1} = Ax_k + Bu_k$ Initial state x_0, reference trajectory $\mathbf{x}_d$ Positive semi-definite matrix sequences R_k, Q_k Positive semi-definite matrix S
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N - x_{d,N}\|\|_S$ $+\sum_{k=0}^{N-1}(\|\|x_k - x_{d,k}\|\|_{Q_k} + \|\|\Delta u_k\|\|_{R_k})$
Algorithm Properties	model-based, DP, iterative solution

where

$$\Delta u_k = u_k - u_{k-1} \tag{III.12.42}$$

In this case, we still consider state augmentation, with the extended state defined as

$$x_{e,k} = \begin{bmatrix} x_k \\ x_{d,k} \\ u_{k-1} \end{bmatrix} \in \mathbb{R}^{2n+m} \tag{III.12.43}$$

The extended state then satisfies the following system equation

$$x_{e,k+1} = A_{e,k}x_{e,k} + B_e\Delta u_k$$

where

$$A_{e,k} = \begin{bmatrix} A & 0 & 0 \\ 0 & A_{d,k} & 0 \\ 0 & 0 & I \end{bmatrix}, B_e = \begin{bmatrix} B \\ 0 \\ I \end{bmatrix} \tag{III.12.44}$$

Therefore, the cost function becomes

$$J_{0:N}(\mathbf{x}_e, \Delta\mathbf{u}) = ||x_{e,N}||_{S_e} + \sum_{k=0}^{N-1}(||x_{e,k}||_{Q_{e,k}} + ||\Delta u_k||_{R_k})$$

where

$$\begin{aligned} S_e &= \begin{bmatrix} S & -S & 0 \\ -S & S & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ Q_{e,k} &= \begin{bmatrix} Q_k & -Q_k & 0 \\ -Q_k & Q_k & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned} \tag{III.12.45}$$

Similar to the tracking control case, we summarize the **LQR Input Increment Control** algorithm as follows:

Algorithm	**LQR Input Increment Control**
Problem Type	LQR Smooth Tracking Control [Discrete]
Given	Discrete linear system $x_{k+1} = Ax_k + Bu_k$ Reference linear system matrix sequence $A_{d,k}$ Positive semi-definite matrix sequences R_k, Q_k, positive semi-definite matrix S Initial states $x_0, x_{d,0}$
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N - x_{d,N}\|\|_S$ $+\sum_{k=0}^{N-1}(\|\|x_k - x_{d,k}\|\|_{Q_k} + \|\|\Delta u_k\|\|_{R_k})$
Algorithm Properties	model-based, DP, analytical solution

Algorithm 35: LQR Input Increment Control

Input: Discrete linear system $x_{k+1} = Ax_k + Bu_k$
Input: Linear system $x_{k+1} = Ax_k + Bu_k$
Input: Reference linear system matrix sequence $A_{d,k}$
Input: Positive semi-definite matrix sequences R_k, Q_k, positive semi-definite matrix S
Output: Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{x}_d, \mathbf{u})$

$$P_N \leftarrow \begin{bmatrix} S & -S & 0 \\ -S & S & 0 \\ 0 & 0 & 0 \end{bmatrix}, B_e \begin{bmatrix} B \\ 0 \\ I \end{bmatrix}$$

for $k \in N-1, N-2, \ldots, 0$ **do**

$$A_{e,k} = \begin{bmatrix} A & 0 & 0 \\ 0 & A_{d,k} & 0 \\ 0 & 0 & I \end{bmatrix}$$

$$F_k \leftarrow -(R_k + B_e^T P_{k+1} B)^{-1} B_e^T P_{k+1} A_{e,k}$$

$$P_k \leftarrow A_{e,k}^T P_{k+1} A_{e,k} + A_{e,k}^T P_{k+1} B_e F_k + Q_{e,k}$$

$$x_{e,0} \leftarrow \begin{bmatrix} x_0 & x_{d,0} & 0 \end{bmatrix}^T$$

$$\nu_{-1} \leftarrow 0$$

for $k \in 0, ..., N-1$ **do**

$$\Delta\nu_k \leftarrow F_k x_{e,k}$$

$$x_{e,k+1} \leftarrow A_{e,k} x_{e,k} + B_e \nu_k$$

$$\nu_k \leftarrow nu_{k-1} + \Delta\nu_k$$

$$\mathbf{u}^* \leftarrow \nu_{0:N}(x)$$

(Reference: "The Beauty of Control - Volume 2")

12.8 Constrained Optimal Control Problems

The problems discussed earlier are all unconstrained optimal control problems. For example, in Section 3, the unconstrained continuous optimal control problem was presented. In practical problems, for continuous systems, various forms of constraints may arise, such as equality constraints, differential constraints, integral constraints, inequality constraints, etc.

Below, we define several types of continuous optimal control problems under different constraints. Practical problems may also be combinations of these types. First, let us consider the cases of equality and differential constraints.

Problem	**Equality-Constrained Optimal Regulation Control [Continuous]**
Problem Description	Given a continuous system and cost function, find the optimal regulation control input under equality constraints
Given	Continuous system $\dot{x} = f(x, u, t)$ Cost function $J(x, u) = h(x_{t_f}, t_f) + \int_0^{t_f} g(\mathbf{x}(t), \mathbf{u}(t), t)dt$ Initial state $x_{t_0} = x_0$
Find	Optimal control $\mathbf{u}^*(t) = \arg\min_{\mathbf{u}} J(x(t), u(t))$ such that $F(x(t), t) = 0$

Problem	**Differential-Constrained Optimal Regulation Control [Continuous]**
Problem Description	Given a continuous system and cost function, find the optimal regulation control input under differential constraints
Given	Continuous system $\dot{x} = f(x, u, t)$ Cost function $J(x, u) = h(x_{t_f}, t_f) + \int_0^{t_f} g(x(t), u(t), t)dt$ Initial state $x_{t_0} = x_0$
Find	Optimal control $u^*(t) = \arg\min_u J(x(t), u(t))$ such that $F(\dot{x}(t), x(t), t) = 0$

In fact, for these two types of problems, we only need to use the Lagrange multiplier method. We define the Lagrange multiplier λ and augment the cost function as

$$\bar{J}(x, \lambda, u) = h(x_{t_f}, t_f) + \int_0^{t_f} g(x(t), u(t), t) + \lambda F(\dot{x}(t), x(t), t)dt$$

Note that $\dot{x}(t) = f(x(t), u(t), t)$ is known. Therefore, we can express the augmented running cost as

$$\bar{g}(x(t), \lambda, u(t), t) = g(x(t), u(t), t) + \lambda F(f(x(t), u(t), t), x(t), t) \tag{III.12.46}$$

That is,

$$\bar{J}(x, \lambda, u) = h(x_{t_f}, t_f) + \int_0^{t_f} \bar{g}(x(t), \lambda, u(t), t)dt \tag{III.12.47}$$

Subsequently, by treating $x(t)$ and λ as new states, we can derive the new HJB equation:

$$-\frac{\partial \bar{\mathcal{J}}}{\partial t}(x(t), \lambda, t) = \min_{u(t)} \bar{\mathcal{H}}\left(x(t), \lambda, u(t), \frac{\partial \mathcal{J}}{\partial x}, t\right) \tag{III.12.48}$$

where the augmented Hamiltonian is

$$\bar{\mathcal{H}}\left(x, \lambda, u, \frac{\partial \mathcal{J}}{\partial x}, t\right) = g(x, u, t) + \lambda F(f(x, u, t), x, t) + \frac{\partial \mathcal{J}}{\partial x}(x, \lambda, t)\dot{f}(x, u, t) \tag{III.12.49}$$

Note that since the introduced Lagrange multiplier is time-invariant, the augmented Hamiltonian does not include a term like $\frac{\partial \mathcal{J}}{\partial \lambda}^T \dot{\lambda}$.

Next, we consider integral constraints of the form $\int_0^{t_f} b(\dot{x}(t), x(t), t)dt = B$.

Problem	**Integral-Constrained Optimal Regulation Control [Continuous]**
Problem Description	Given a continuous system and cost function, find the optimal regulation control input under integral constraints
Given	Continuous system $\dot{x} = f(x, u, t)$ Cost function $J(x, u) = h(x_{t_f}, t_f) + \int_0^{t_f} g(x(t), u(t), t)dt$ Initial state $x_{t_0} = x_0$
Find	Optimal control $u^*(t) = \arg\min_{\mathbf{u}} J(x(t), u(t))$ such that $\int_0^{t_f} b(\dot{x}(t), x(t), t)dt = B$

Introduce a new state variable $z(t)$, defined as

$$z(t) = \int_0^t b(\dot{x}(\tau), x(\tau), \tau)d\tau \tag{III.12.50}$$

Note that $\dot{x}(t) = f(x(t), u(t), t)$ is known, so the system trajectory of the state variable $z(t)$ is

$$\dot{z}(t) = b(f(x(t), u(t), t), x(t), t)$$

with initial and terminal conditions $z(0) = 0$ and $z(t_f) = B$. We can express the augmented terminal cost as

$$\bar{h}(x_{t_f}, z_{t_f}, t_f) = h(x_{t_f}, t_f)(z(t_f) - B) \tag{III.12.51}$$

That is,

$$\bar{J}(x, \lambda, z, u) = \bar{h}(x_{t_f}, z_{t_f}, t_f) + \int_0^{t_f} g(x(t), u(t), t)dt$$

Subsequently, by treating $x(t)$ and $z(t)$ as new states, we can derive the new HJB equation:

$$-\frac{\partial \bar{\mathcal{J}}}{\partial t}(x(t), z(t), t) = \min_{u(t)} \bar{\mathcal{H}} \left(x(t), z(t), u(t), \frac{\partial \mathcal{J}}{\partial x}, \frac{\partial \mathcal{J}}{\partial z}, t \right) \tag{III.12.52}$$

where the augmented Hamiltonian is

$$\bar{\mathcal{H}} \left(x, z, u, \frac{\partial \mathcal{J}}{\partial x}, \frac{\partial \mathcal{J}}{\partial z}, t \right) = g(x, u, t) + \frac{\partial \mathcal{J}}{\partial x} \dot{f}(x, u, t) + \frac{\partial \mathcal{J}}{\partial z} \dot{b}(f(x, u, t), x, t) \tag{III.12.53}$$

subject to the terminal condition

$$\bar{h}(x_{t_f}, z_{t_f}, t_f) = 0 \tag{III.12.54}$$

Note that, similar to the unconstrained case, $\frac{\partial \mathcal{J}}{\partial x}$ and $\frac{\partial \mathcal{J}}{\partial z}$ here are unknown functionals of $x(t)$, $z(t)$, and t. The necessary condition for the minimal cost functional $\mathcal{J}$ is that its derivatives with respect to $x(t)$ and $z(t)$ must equal these two functionals.

Similarly, constrained continuous optimal control problems do not have general analytical solutions, and specific derivations are required for each practical problem.

(Reference: Zhang Jie's "Optimal Control")

13 Foundations of Model Predictive Control

13.1 Basic Concepts

In the previous chapter, we introduced several methods for solving optimal (regulation/tracking) control problems. These methods, based on dynamic programming principles, derive the optimal control sequence that minimizes a certain objective function through backward and forward iterations, collectively referred to as optimal control methods.

Optimal control methods are clear and concise but also have some limitations. First, such methods typically handle only unconstrained optimization problems or, at most, equality-constrained problems, while **inequality constraints** are more challenging to address. Second, these methods require an accurate mathematical model of the system, preferably linear. If there are **errors** in the model estimation or unmodeled **disturbances** in the actual system, the performance of such algorithms may be affected.

Recall from Section 3, where we discussed various optimization problems and their solution methods. Some of these methods can solve optimization problems with inequality constraints iteratively using computers. If we can transform an optimal control problem with inequality constraints into an optimization problem with inequality constraints, we can leverage these algorithms for solutions.

Specifically, suppose we know the discrete system equation $x_{k+1} = f(x_k, u_k)$. Given a known initial state x_0, the system state x_k at any time step becomes a function of the historical input sequence $u_0, ..., u_k$. Since the cost function $J_{1:k}(\mathbf{x}, \mathbf{u})$ of the optimal control problem is also a function of the state sequence $x_0, ..., x_k$, we can rewrite the cost function as a function of the input sequence $u_0, ..., u_k$, thereby constructing an optimization problem where the input sequence $u_0, ..., u_k$ is the sole optimization variable.

In this process, we repeatedly use the system equation $x_{k+1} = f(x_k, u_k)$ to predict future states iteratively based on historical states and inputs. This is essentially **prediction** using the system model. Hence, methods that solve optimal control problems using this approach are called **Model Predictive Control (MPC)**.

While the above approach can solve for an optimal control sequence $u_0^*, ..., u_k^*$ with $u_0, ..., u_k$ as variables, directly applying this sequence to the system cannot address the aforementioned issues of model errors and disturbances. Both model errors and disturbances accumulate during prediction, causing the predicted optimal control to deviate from the actual optimal solution.

The root of this problem lies in the fact that the above solution process is **open-loop**, as it does not incorporate the actual system state (feedback). In practice, if the solution is fast enough, we can use the system's feedback state as the initial prediction value $x_{t|t}$ at **each discrete time step** t and solve the optimization problem over a horizon of length N_p. The resulting trajectory $u_{t|t}^*, ..., u_{t|N_p}^*$ will gradually deviate from the true optimal trajectory, so we only take the first term $u_{t|t}^*$ as the current control input and discard the remaining predictions $u_{t|t+1}^*, ..., u_{t|N_p}^*$. At the next time step $t+1$, we repeat this process, predicting N_p steps ahead but only executing the first step. This way, each executed control input incorporates the latest feedback from the system, effectively addressing model errors and disturbances.

Such a control scheme is called **receding horizon control** or **closed-loop MPC**. In this and subsequent chapters, we focus exclusively on closed-loop MPC.

The receding horizon approach effectively integrates model prediction, optimization problem solving, state feedback, and handling of disturbances/model errors. However, its cost is that the entire optimization problem must be solved within each control cycle. Note: If the control input u_k is an N_u-dimensional vector and the prediction horizon length is N_p, the optimization problem has $N_u N_p$ variables. Fortunately, with advances in computing technology, real-time solutions for large-scale optimization problems are no longer impractical. Many modern solvers (software + hardware) can solve optimization problems with tens of variables within 0.001s, enabling MPC to achieve control frequencies of around 1000Hz.

Depending on whether the model is linear and whether constraints exist, MPC can be classified into different types. The simplest MPC algorithm is for linear systems with quadratic objectives (i.e., LQR problems). Below, we discuss the receding horizon MPC algorithm for both unconstrained and constrained cases.

13.2 Unconstrained Linear MPC

13.2.1 Regulation Control

First, we revisit the simplest optimal control problem: the unconstrained LQR regulation problem. Assuming a prediction horizon length of N_p, the optimization objective at each time step is:

$$\min_u J_{k:k+N_p}(\mathbf{x}, \mathbf{u}) = \frac{1}{2}\left(||x_{k+N_p}||_S + \sum_{i=k}^{k+N_p-1}\left(||x_{i+1}||_{Q_i} + ||u_i||_{R_i}\right)\right) \tag{III.13.1}$$
$$s.t. \quad x_{t+1} = Ax_t + Bu_t$$

Assume the state x_t has dimension N_x and the input u_t has dimension N_u. At time step k, we take the current system state (assuming full state observability without measurement errors) x_k as the initial prediction state:

$$x_{k|k} = x_k \tag{III.13.2}$$

Then,

$$x_{k|t+1} = Ax_{k|t} + Bu_{k|t}$$

Here, $u_{k|t}$ for $t = k, k+1, ..., k+N_p$ are our optimization variables. For simplicity, we denote them as:

$$U_k = \begin{bmatrix} u_{k|k} \\ u_{k|k+1} \\ ... \\ u_{k|k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p N_u} \tag{III.13.3}$$

Similarly, for the state, we define:

$$X_k = \begin{bmatrix} x_{k|k} \\ x_{k|k+1} \\ ... \\ x_{k|k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p N_x} \tag{III.13.4}$$

Note that X_k excludes x_k. According to the receding horizon principle, for the final optimization problem, all variables in X_k are intermediate variables, i.e., functions of the initial state x_k and U_k. We explicitly express this functional relationship as:

$$X_k = \Phi x_k + \Gamma U_k \tag{III.13.5}$$

where

$$\Phi = \begin{bmatrix} I \\ A \\ A^2 \\ ... \\ A^{N_p} \end{bmatrix} \in \mathbb{R}^{N_p N_x \times N_x} \tag{III.13.6}$$

$$\Gamma = \begin{bmatrix} 0 & & & \\ B & & & \\ AB & B & & \\ ... & ... & ... & \\ A^{N_p-1}B & ... & AB & B \end{bmatrix} \in \mathbb{R}^{N_p N_x \times N_p N_u} \tag{III.13.7}$$

Next, we express the optimization objective J_k at time step k. Let

$$\mathcal{Q} = \begin{bmatrix} Q_k & & & \\ & Q_{k+1} & & \\ & & ... & \\ & & & S \end{bmatrix} \in \mathbb{R}^{N_p N_x \times N_p N_x} \tag{III.13.8}$$

Set $R_{k+N_p} = 0_{N_u \times N_u}$, then:

$$\mathcal{R} = \begin{bmatrix} R_k & & & \\ & R_{k+1} & & \\ & & \dots & \\ & & & R_{k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p N_x \times N_p N_x} \tag{III.13.9}$$

Assuming $\mathcal{Q} = \mathcal{Q}^T, \mathcal{R} = \mathcal{R}^T$, J_k is a quadratic form in the optimization variable U_k:

$$\begin{aligned} J_k(U_k) &= \frac{1}{2}(X_k^T \mathcal{Q} X_k + U_k^T \mathcal{R} U_k) \\ &= \frac{1}{2}((\Phi x_k + \Gamma U_k)^T \mathcal{Q}(\Phi x_k + \Gamma U_k) + U_k^T \mathcal{R} U_k) \\ &= \frac{1}{2}(U_k^T(\mathcal{R} + \Gamma^T \mathcal{Q}\Gamma)U_k + 2x_k^T \Phi^T \mathcal{Q}\Gamma U_k + x_k^T \Phi^T \mathcal{Q}\Phi x_k) \end{aligned}$$

Further, let

$$\begin{aligned} G_k &= \Phi^T \mathcal{Q}\Gamma \\ H_k &= \mathcal{R} + \Gamma^T \mathcal{Q}\Gamma \end{aligned} \tag{III.13.10}$$

Then,

$$J_k(U_k) = \frac{1}{2}U_k^T H_k U_k + x_k^T G_k U_k \tag{III.13.11}$$

In the above expression, the term $x_k^T \Phi^T \mathcal{Q}\Phi x_k$, which is independent of the optimization variables, is omitted as it does not affect the solution.

Thus, we have transformed the (unconstrained) LQR optimal control problem into an unconstrained QP problem using the receding horizon approach. According to the conclusions in Section 3.5, this problem has an analytical solution:

$$U_k^* = -H_k^{-1} G_k^T x_k \tag{III.13.12}$$

To summarize the above algorithm, we have the **Unconstrained Linear MPC**. Note that when the system is time-invariant, most computations of this algorithm can be performed offline, and the online computation is essentially just linear feedback. If the system is time-varying, i.e., A, B depend on x_k, then all offline computations must be shifted to online computations.

Algorithm	**Unconstrained Linear MPC**
Problem Type	LQR Regulation Control [Discrete]
Given	Discrete linear system matrices A, B Positive semi-definite matrix sequences R_k, Q_k, positive semi-definite matrix S Real-time system state $x_{1:N}$
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N\|\|_S + \sum_{k=0}^{N-1}(\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$
Algorithm Properties	model-based, MPC, analytical solution

Algorithm 36: Unconstrained Linear MPC

Input: Discrete linear system matrices A, B
Input: Positive semi-definite matrix sequences $R_{1:N}, Q_{1:N}$, positive semi-definite matrix S
Input: Real-time system state $x_{1:N}$
Output: Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$
Offline:
$\Phi \leftarrow [I, A, A^2, ..., A^{N_p}]^T$

$$\Gamma \leftarrow \begin{bmatrix} 0 & & & \\ B & & & \\ AB & B & & \\ ... & ... & ... & \\ A^{N_p-1}B & ... & AB & B \end{bmatrix}$$

for $k \in 1, 2, ..., N$ **do**
  $\mathcal{Q}_k \leftarrow \text{diag}(Q_k, Q_{k+2}, ..., S)$
  $\mathcal{R}_k \leftarrow \text{diag}(R_k, R_{k+1}, ..., R_{k+N_p})$
  $G_k \leftarrow \Phi^T \mathcal{Q} \Gamma$
  $H_k \leftarrow \mathcal{R} + \Gamma^T \mathcal{Q} \Gamma$
  $F_k = -H_k^{-1} G_k^T$
Online:
for $k \in 1, 2, ..., N$ **do**
  $U_k^* \leftarrow F_k x_k$
  $u_k^* \leftarrow (U_k^*)_{1:N_u}$
$\mathbf{u}^* \leftarrow u_{1:N}^*$

13.2.2 Tracking Control

[Main content: Unconstrained optimal tracking problem, Unconstrained linear MPC tracking algorithm, Comparison between MPC and OC]

[This section will be updated in a future version. Stay tuned.]

(Reference: "The Beauty of Control - Volume 2")

13.3 Constrained Linear MPC

Next, let us consider the constrained case. First, we define the LQR control problem under (linear) constraints.

Problem	**Constrained LQR Regulation Control [Discrete]**
Problem Description	Given a linear discrete system and constraints, find the regulating control input that minimizes the quadratic cost
Given	Discrete linear system matrices A, B Real-time system state $x_{1:N}$ Positive semi-definite matrix sequences $R_{1:N}, Q_{1:N}$ Positive semi-definite matrix S Matrix sequences $M_{x,1:N}, M_{u,1:N}$ Vector sequences $b_{1:N}$
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N\|\|_S + \sum_{k=0}^{N-1} (\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$ Subject to $M_{x,k}x_k + M_{u,k}u_k \leq b_k, k = 1, 2, ..., N$

Here, we only consider constraints in linear form. For each time step k, the constraints take the form

$$M_{x,k}x_k + M_{u,k}u_k \leq b_k \tag{III.13.13}$$

where $M_{x,k} \in \mathbb{R}^{m \times N_x}$, $M_{u,k} \in \mathbb{R}^{m \times N_u}$, $b_k \in \mathbb{R}^m$, and m is the dimension of constraints at each time step.

Similar to the previous approach, we also express them in the form of large matrices over the prediction horizon N_p:

$$\mathcal{M}_x = \begin{bmatrix} M_{x,k} & & & \\ & M_{x,k+1} & & \\ & & \dots & \\ & & & M_{x,k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p m \times N_p N_x} \tag{III.13.14}$$

$$\mathcal{M}_u = \begin{bmatrix} M_{u,k} & & & \\ & M_{u,k+1} & & \\ & & \dots & \\ & & & M_{u,k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p m \times N_p N_u} \tag{III.13.15}$$

$$\beta_k = \begin{bmatrix} b_k \\ b_{k+1} \\ \dots \\ b_{k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p m} \tag{III.13.16}$$

Then, the inequality constraints over the prediction horizon can be uniformly expressed as

$$\mathcal{M}_x X_k + \mathcal{M}_u U_k \leq \beta_k \tag{III.13.17}$$

Substituting the expression for X_k, we have

$$\mathcal{M}_x(\Phi x_k + \Gamma U_k) + \mathcal{M}_u U_k \leq \beta_k$$

i.e.,

$$(\mathcal{M}_x \Gamma + \mathcal{M}_u) U_k \leq \beta_k - \mathcal{M}_x \Phi x_k$$

Let

$$\begin{aligned} M_k &= \mathcal{M}_x \Gamma + \mathcal{M}_u \in \mathbb{R}^{N_p m \times N_p N_u} \\ \mathbf{b}_k &= \beta_k - \mathcal{M}_x \Phi x_k \in \mathbb{R}^{N_p m} \end{aligned} \tag{III.13.18}$$

Now, the optimization problem over the prediction horizon becomes

$$\begin{aligned} &\min_{U_k} J_k(U_k) = \frac{1}{2} U_k^T H_k U_k + x_k^T G_k U_k \\ &s.t. \quad M_k U_k \leq \mathbf{b}_k \end{aligned} \tag{III.13.19}$$

This is a QP problem with inequality constraints. We can solve it using the algorithm introduced in Section 3.5.

To summarize the above algorithm, we have the **Linear MPC with Inequality Constraints**. Note: Since $\mathbf{b}_k$ in the constraints depends on x_k, **the optimization problem must be solved online**. This is fundamentally different from the unconstrained algorithm, where the feedback matrix F_k can be computed offline.

Algorithm 37: Linear MPC with Inequality Constraints

Input: Discrete linear system matrices A, B
Input: Positive semi-definite matrix sequences $R_{1:N}, Q_{1:N}$, positive semi-definite matrix S
Input: Matrix sequences $M_{x,1:N}, M_{u,1:N}$, vector sequence $b_{1:N}$
Input: Real-time system state $x_{1:N}$
Output: Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$

Offline:
$\Phi \leftarrow [I, A, A^2, ..., A^{N_p}]^T$
$\Gamma \leftarrow \Gamma(A, B)$
Online:
for $k \in 1, 2, ..., N$ **do**
  $\mathcal{Q}_k \leftarrow \text{diag}(Q_k, Q_{k+2}, ..., S)$
  $\mathcal{R}_k \leftarrow \text{diag}(R_k, R_{k+1}, ..., R_{k+N_p})$
  $G_k \leftarrow \Phi^T \mathcal{Q} \Gamma$
  $H_k \leftarrow \mathcal{R} + \Gamma^T \mathcal{Q} \Gamma$
  $\mathcal{M}_x \leftarrow \text{diag}(M_{x,k}, M_{x,k+1}, ...M_{x,k+N_p})$
  $\mathcal{M}_u \leftarrow \text{diag}(M_{u,k}, M_{u,k+1}, ...M_{u,k+N_p})$
  $\beta_k \leftarrow [b_k^T, b_{k+1}^T, ..., b_{k+N_p}^T]^T$
  $M_k \leftarrow \mathcal{M}_x \Gamma + \mathcal{M}_u$
  $\mathbf{b}_k \leftarrow \beta_k - \mathcal{M}_x \Phi x_k$
  $U_k^* \leftarrow \text{solve\_QP\_barrier}(H_k, G_k^T x_k, none, none, M_k, \mathbf{b}_k)$
  $u_k^* \leftarrow (U_k^*)_{1:N_u}$
$\mathbf{u}^* \leftarrow u_{1:N}^*$

Algorithm	**Linear MPC with Inequality Constraints**
Problem Type	Constrained LQR Regulation Control [Discrete]
Given	Discrete linear system matrices A, B Positive semi-definite matrix sequences R_k, Q_k, positive semi-definite matrix S Real-time system state $x_{1:N}$ Matrix sequences $M_{x,1:N}, M_{u,1:N}$, vector sequence $b_{1:N}$
Find	Optimal control $\mathbf{u}^* = \arg\min_{\mathbf{u}} \lvert\lvert x_N \rvert\rvert_S + \sum_{k=0}^{N-1}(\lvert\lvert x_k \rvert\rvert_{Q_k} + \lvert\lvert u_k \rvert\rvert_{R_k})$ Subject to $M_{x,k} x_k + M_{u,k} u_k \leq b_k, k = 1, 2, ..., N$
Algorithm Properties	model-based, MPC, iterative solution

(Reference: ”The Beauty of Control - Volume 2”)
(Reference: https://zhuanlan.zhihu.com/p/368705959)

Part IV

Fundamentals of Robotic Control

14 Overview of Robot Control

Robot control refers to our desire to make a robot move in the way we intend, including reaching specified positions, tracking designated trajectories, generating certain forces or torques, and so on. Based on the kinematic and dynamic models introduced in Part II, we can already achieve **open-loop control**. However, in practical systems, open-loop control alone is far from sufficient.

This is because, in a robotic system, motors may be non-ideal (due to current-angular velocity modeling errors or execution errors), the environment may be non-ideal (due to internal/external resistance, etc.), and observations may also be non-ideal (position/velocity/torque measurements). Under such conditions, we still want the robot to move precisely as required. This is the core objective of robot control.

As discussed in Part III, **the essence of control is to transform the non-ideal characteristics of an open-loop system into the ideal characteristics of a closed-loop system through feedback**. In fact, robot (manipulator) control is a very typical application scenario of control theory. In robots as the controlled object, **joint torque is the physical control variable we can freely configure**. Based on the kinematic and dynamic equations introduced in Part II, we can propose various robot control methods to achieve our goals.

In the entire robotic system, control may be just one part of the overall system. Depending on the design requirements, the robot may receive **joint space commands** or **operational space commands**. This gives rise to one of the following **three control approaches**: independent joint control, joint space control, and operational space control.

Given joint commands, the most straightforward control approach is to design a controller separately for each joint motor. This approach is called **independent joint control**. However, in robotic systems, the dynamics of different joints exhibit complex nonlinear coupling relationships. Therefore, to implement independent joint control, it is first necessary to decouple the joint dynamics and treat the nonlinear coupling terms between joints as disturbances to be suppressed.

Independent joint control configures each joint separately, which may lead to coupling issues at the system level. Therefore, a better approach is to model the robot uniformly in joint space, with a single controller generating commands for all joints simultaneously. In this process, real-time computed nonlinear feedback can still achieve decoupling between controllers. This control approach is also called **joint space control**.

Robot users are often more concerned with the robot's end-effector, and the commands they provide directly come from the operational space. We can design a controller directly in the operational space and then convert it into direct control variables (joint torques) in the joint space. This control approach is called **operational space control**.

In system design, besides specifying different command forms, certain control objectives are also imposed on the control subsystem. These control objectives can be categorized into three types: **motion control**, **impedance/admittance control**, and **constrained force control**. The three control approaches can generally handle these three types of control objectives, and their combinations give rise to various control problems. Below, we detail these control problems according to different objectives.

14.1 Motion Control

Motion control aims to make the robot arm achieve a given motion. In Part III, we defined two types of motion control requirements: **regulation** and **tracking**, distinguished by whether the reference value has first or higher-order derivatives. Thus, for robots (manipulators), there are two types of control requirements: **regulation control** (i.e., **point-to-point control**) and **tracking control**. The difference lies in: regulation control provides only a single-point control command or a constant target position, while tracking control requires following a specified trajectory in a certain space, with first/higher-order derivatives also needing to be tracked.

Whether it is a regulation or tracking problem, and whether the commands come from the operational space or joint space, we generally assume that the robot joint angles (and their derivatives) are known. In

practical systems, these are obtained from joint sensors or filter observations. We summarize the different motion control problems for different commands as follows:

Problem	**Joint Space Regulation Control**
Description	Given robot parameters, design a controller to stabilize the robot joint angles at reference values
Known	Robot parameters (D-H parameters, link parameters) Joint vector and derivatives $q, \dot{q}$ Reference joint angle q_d
To Find	Control variable (driving torque) $u(q, \dot{q}, q_d)$

Problem	**Joint Space Tracking Control**
Description	Given robot parameters, design a controller to make the robot joint angles follow the reference values
Known	Robot parameters (D-H parameters, link parameters) Joint vector and derivatives $q, \dot{q}$ Reference joint trajectory $q_d(t)$ and its derivatives
To Find	Control variable (driving torque) $u(q, \dot{q}, q_d, \dot{q}_d, \ddot{q}_d)$

Problem	**Operational Space Regulation Control**
Description	Given robot parameters, design a controller to stabilize the robot end-effector at the reference value
Known	Robot parameters (D-H parameters, link parameters) Joint vector and derivatives $q, \dot{q}$ Reference trajectory x_d
To Find	Control variable (driving torque) $u(q, \dot{q}, x_d)$

Problem	**Operational Space Tracking Control**
Description	Given robot parameters, design a controller to make the robot end-effector pose follow the reference values
Known	Robot parameters (D-H parameters, link parameters) Joint vector and derivatives $q, \dot{q}$ Reference trajectory $x_d(t)$ and its derivatives
To Find	Control variable (driving torque) $u(q, \dot{q}, x_d, \dot{x}_d, \ddot{x}_d)$

14.2 Impedance/Admittance Control

In robot motion control tasks, we need the robot's end-effector or joints to move precisely along specified trajectories. Such a robot exhibits rigidity, meaning it strives to maintain the preset trajectory against any external force. When faced with sudden external forces/torques, the robot's motors may suffer damage due to torque overload. In other words, to prevent damage from rigidity, we want the robot to exhibit some form of compliance.

Additionally, there are practical scenarios where we need the robot to exhibit compliance. For example, when humans and robots share the same space, a fully rigid robot could cause harm. To protect humans, the robot must exhibit some compliance to external forces, such as elasticity.

The above "compliance" constitutes a special control objective—desiring the robot to exhibit a certain **dynamic relationship between force and velocity**. More specifically, we want the robot's **velocity** v_r and **force** F to satisfy:

$$F(t) = m\dot{v}_r(t) + cv_r(t) + k\int v_r(t)dt \tag{IV.14.1}$$

This is precisely the second-order system dynamics of Equation III.10.10, except here v corresponds to $\dot{y}$ in Equation III.10.10. The coefficients m, c, k correspond to the mass, damping, and stiffness coefficients of the second-order system, respectively. Expressing this in the frequency domain gives:

$$Z_m(s) = \frac{F(s)}{V_r(s)} = c + ms + \frac{k}{s} \tag{IV.14.2}$$

This form is very similar to the relationship between the total voltage $V(s)$ and current $I(s)$ in a series RLC circuit:

$$Z(s) = \frac{V(s)}{I(s)} = R + Ls + \frac{C}{s}$$

In electrical engineering, $Z(s)$ is called impedance, so we also refer to $Z_m(s)$ in Equation IV.14.2 as impedance. As described, **robot impedance is the dynamic relationship between force and velocity**. Desiring the robot to exhibit impedance characteristics leads to the control objective called **impedance/admittance control**.

The F and v_r in Equation IV.14.2 are not specified as any particular force or velocity of the robot. In fact, impedance control can be defined in both joint space and operational space. For joint space, F and v_r correspond to joint torque and $\dot{q}$. For operational space, F and v_r correspond to F_e and v_e.

Strictly speaking, since robot impedance is defined as $F(s)/V_r(s)$, an impedance controller is a control system that takes some form of pose/velocity as input and outputs joint torque. For different commands, we summarize the impedance control problems as follows:

Problem	**Joint Space Impedance Control**
Description	Given robot parameters and states, design a controller to make the robot joints exhibit impedance characteristics
Given	Robot parameters (D-H parameters, link parameters) Joint vector and derivatives $q, \dot{q}$ Reference joint trajectory $q_d(t)$ and its derivatives
Find	Controller (driving torque) $u(q, \dot{q}, q_d, \dot{q}_d, \ddot{q}_d)$

Problem	**Operational Space Impedance Control**
Description	Given robot parameters and states, design a controller to make the robot end-effector exhibit impedance characteristics
Given	Robot parameters (D-H parameters, link parameters) Joint vector and derivatives $q, \dot{q}$ Reference trajectory $x_d(t)$ and its derivatives
Find	Controller (driving torque) $u(q, \dot{q}, x_d, \dot{x}_d, \ddot{x}_d)$

In particular, for operational space impedance control, if the controller can obtain feedback of the end-effector force F_e, it can achieve decoupled joint impedance configuration.

In impedance control, there is another special requirement. When teaching the robot, we want to drag the robot's end-effector or the entire body to move. In this case, the robot should compensate for gravity and other effects, exhibiting the characteristics of a second-order system externally, but without needing to track a reference position. Such a requirement is called **zero-force control**.

Problem	**Operational Space Zero-Force Control**
Description	Given robot parameters and states, design a controller to make the robot end-effector exhibit zero-force characteristics
Given	Robot parameters (D-H parameters, link parameters) Joint vector and derivatives $q, \dot{q}$
Find	Controller (driving torque) $u(q, \dot{q})$

Besides impedance control, we can also design an **admittance controller** based on motion control. We define kinematic admittance as follows:

$$S_m(s) = \frac{V(s)}{F(s)} = \frac{1}{c + ms + \frac{k}{s}} \tag{IV.14.3}$$

An admittance controller is a control approach based on motion control. Specifically, to make the system exhibit impedance control characteristics externally, the admittance controller actively observes the external force and designs a robot position error trajectory (i.e., a secondary reference trajectory). When the robot tracks this trajectory, it exhibits impedance behavior in response to external forces.

We can also define joint space and operational space admittance control problems as follows:

Problem	**Joint Space Admittance Control**
Description	Given robot parameters and external torque, design a trajectory to make the robot joints exhibit impedance characteristics
Given	Robot parameters (D-H parameters, link parameters) Reference joint trajectory $q_d(t)$ and its derivatives Joint vector and derivatives $q, \dot{q}$ External joint torque τ_o
Find	Admittance reference trajectory $q_a(\tau_o, q_d, \dot{q}_d, \ddot{q}_d)$

Problem	**Operational Space Admittance Control**
Description	Given robot parameters and external force, design a trajectory to make the robot end-effector exhibit impedance characteristics
Given	Robot parameters (D-H parameters, link parameters) Reference trajectory $x_d(t)$ and its derivatives Joint vector and derivatives $q, \dot{q}$ External force F_o
Find	Admittance reference trajectory $x_a(F_o, x_d, \dot{x}_d, \ddot{x}_d)$

14.3 Constrained Force Control

In addition to compliant control, we sometimes want the robot to perform tasks in constrained environments. For example, tasks like painting, writing, or wiping glass require the robot to use tools to process surfaces. Such tasks have the following two characteristics: **motion must occur along a given surface**, and we want to **control the contact force within a certain range**.

Taking writing with a hard pen as an example: To write on paper, the pen tip position must lie within the plane of the paper, and the contact force between the pen and paper must be moderate. Excessive force will damage the pen tip and paper, while insufficient force will prevent the ink from adhering to the paper. We want to control the contact force between the pen and paper.

Control problems where the environment imposes requirements on the robot end-effector's force and motion are called **constrained force control** problems. The constraints imposed by the environment are called **environmental constraints**. Environmental constraints apply to either the end-effector force F_e or the end-effector velocity v_e. For each end-effector degree of freedom, there must be one environmental constraint, either a force constraint or a velocity constraint.

Returning to the example of writing with a hard pen, assume the paper lies in the XY plane of the environment coordinate system. For the writing task, assuming the end-effector has 6 degrees of freedom, we have the following environmental constraints: the end-effector velocity in the z direction must be 0, and the forces in the x and y directions must equal the resistance; the environment does not apply any moments to the end-effector in any direction. Expressed as equations:

$$v_z = 0$$
$$F_{ex} = f_{ex}$$
$$F_{ey} = f_{ey}$$
$$M_{ex} = 0$$
$$M_{ey} = 0$$
$$M_{ez} = 0$$

For a robot with k_f end-effector degrees of freedom, its end-effector force F_e and velocity v_e have $2k_f$ dimensions in total. k_f environmental constraints introduce k_f equations, leaving us with a k_f-dimensional controllable subspace. We can design trajectories within this subspace and design controllers to ensure the robot precisely tracks these trajectories while satisfying the above constraints.

In fact, the "free" motion control described in the previous section is a special case of constrained control: the environment has no contact with the end-effector, so all components of v_e are free, and the environmental force constraint in all directions is $F_e = 0$.

For constrained force control problems, we generally assume the end-effector force is known, which requires the robot to be equipped with an end-effector force sensor. We summarize the constrained force control problem as follows:

Problem	**Constrained Force Control**
Description	Given robot parameters, design a controller to satisfy constraints and track given force and velocity trajectories
Given	Robot parameters (D-H parameters, link parameters) Joint vector and derivatives $q, \dot{q}$ Environmental constraints $E(v_e, F_e) = 0$ Desired constraints $F_d(t), x_d(t)$
Find	Control input (driving torque) $u(q, \dot{q}, F_d(t))$

(Reference: "Robotics: Modelling, Planning and Control")

15 Independent Joint Control

A robot consists of multiple motors connected by rigid bodies. For joint space control, the most straightforward approach is to design a controller for each joint motor individually, treating the coupling terms between different joints as disturbances to be suppressed.

15.1 Basic Concepts

In this section, we first briefly introduce the motors used in actual robots, then model the motors, and finally discuss the decoupling method for motor dynamics in robots, thereby introducing the problem of disturbance rejection regulation/tracking control for independent joints.

15.1.1 Introduction to Robot Motors

In robots, the most commonly used joint actuators are motors. The motors employed in robots must meet the requirements of servo motors, i.e., achieving precise position control. Therefore, among the various types of motors, **permanent magnet synchronous motors (PMSM)** are typically used as joint motors.

Permanent magnet synchronous motors belong to the category of **DC motors** and are divided into **brushed motors** and **brushless motors**. Brushed motors use permanent magnets as the stator and coils (armature) as the rotor, relying on commutators to change the current direction. Brushless motors use multi-phase coils as the stator to generate a rotating magnetic field and permanent magnets as the rotor.

Whether brushed or brushless motors are used, **motor drivers** are required to provide input. Motor drivers are connected to the main power supply, take command signals as input, and output drive signals with specific voltage/current (multi-phase drive signals for brushless motors are AC). These are typically implemented using microcontrollers and power semiconductor devices.

For real-world joint motor products, they generally integrate not only the motor body and driver but also **reduction mechanisms**, **sensors**, and **controllers**.

The **reduction mechanism** is used to reduce the motor's output angular velocity, increase the output torque, and improve precision. The most common reduction mechanisms include **planetary gear reducers** and **harmonic drives**.

The **sensors** on motors typically include **angle/angular velocity sensors**, and some motors may also be equipped with **torque sensors**. The outputs of these sensors provide feedback values for robot control algorithms.

Motors often also integrate **controllers**. These controllers can receive signals from the motor's sensors and external commands to achieve closed-loop control. In practice, the controllers are usually pre-programmed microcontrollers with built-in inner-loop control algorithms such as current loops (details in subsequent sections).

15.1.2 Modeling of Permanent Magnet Synchronous Motors

Whether brushed or brushless, the electrical driving process of these motors can be described by the same **permanent magnet synchronous motor model**. Specifically, based on circuit equations, we have the following formula in the continuous complex frequency domain:

$$V_a(s) = (sl_a + r_a)I_a(s) + V_g(s)$$

where V_a, I_a represent the winding (armature) voltage and current, r_a, l_a represent the armature resistance and inductance, and V_g represents the back electromotive force (EMF), which is proportional to the motor angular velocity Ω_m:

$$V_g(s) = k_v \Omega_m(s)$$

Additionally, according to Ampere's force principle, the motor's driving torque C_m is proportional to I_a:

$$C_m(s) = k_t I_a(s)$$

In the above equations, k_v, k_t are called the voltage constant and torque constant, respectively, serving as bridges between electrical and mechanical quantities in DC motors. According to the rotational law, the mechanical equation of the motor is:

$$C_m(s) - C_l(s) = (si_m + f_m)\Omega_m(s) \tag{IV.15.1}$$

where C_l represents the load torque, i_m represents the rotor inertia, and f_m represents the viscous friction coefficient. Combining the above equations, we have:

$$\begin{aligned} I_a(s) &= \frac{V_a(s) - k_v\Omega_m(s)}{sl_a + r_a} \\ \Omega_m(s) &= \frac{k_t I_a(s) - C_l(s)}{si_m + f_m} \end{aligned} \tag{IV.15.2}$$

Regarding the armature voltage V_a, it is related to the external control voltage signal V_c. In the case of **no current feedback** (no current loop), the relationship is a simple proportional one:

$$V_a(s) = g_v V_c(s)$$

In the case of **current feedback** (with a current loop), the relationship is:

$$V_a(s) = K g_v (V_c(s) - k_i I_a(s))$$

where k_i represents the current feedback coefficient, which has the same dimensions as resistance, and K represents the current feedback controller gain, typically set to a large value.

Summarizing the above equations, we have two types of permanent magnet synchronous motor models: with and without a current loop. The **permanent magnet synchronous motor model with a current loop** is:

$$\begin{aligned} I_a(s) &= \frac{V_a(s) - k_v\Omega_m(s)}{sl_a + r_a} \\ \Omega_m(s) &= \frac{k_t I_a(s) - C_l(s)}{si_m + f_m} \\ V_a(s) &= K g_v (V_c(s) - k_i I_a(s)) \end{aligned}$$

where $i_m, f_m, l_a, r_a, k_v, g_v, k_i, k_t, K$ are constants.

On the other hand, the **permanent magnet synchronous motor model without a current loop** is:

$$\begin{aligned} I_a(s) &= \frac{g_v V_c(s) - k_v\Omega_m(s)}{sl_a + r_a} \\ \Omega_m(s) &= \frac{k_t I_a(s) - C_l(s)}{si_m + f_m} \end{aligned}$$

where $i_m, f_m, l_a, r_a, k_v, g_v, k_t$ are constants.

It can be observed that due to the back EMF, DC motors naturally form a speed regulation system, stabilizing the speed ω_m under a given armature voltage and providing disturbance rejection against external load torques.

15.1.3 First-Order Model of Permanent Magnet Synchronous Motors

Next, we derive the first-order approximate input/output transfer functions for the two types of motors. First, for the **motor without a current loop**, simplifying the above equations, we have:

$$\begin{aligned} \frac{\Omega_m(s)}{V_c(s)} &= \frac{\frac{g_v}{k_v}}{1 + \frac{(si_m + f_m)(sl_a + r_a)}{k_t k_v}} \\ \frac{\Omega_m(s)}{C_l(s)} &= \frac{\frac{sl_a + r_a}{k_t k_v}}{1 + \frac{(si_m + f_m)(sl_a + r_a)}{k_t k_v}} \end{aligned}$$

Since it is generally true that:

$$\frac{i_m}{f_m} >> \frac{l_a}{r_a}$$

we can **approximately ignore the armature inductance and friction**, assuming $l_a \approx 0, f_m \approx 0$, resulting in the first-order approximate transfer function:

$$\Omega_m(s) \approx \frac{\frac{g_v}{k_v}}{1 + s\frac{i_m r_a}{k_t k_v}} V_c(s) + \frac{\frac{r_a}{k_t k_v}}{1 + s\frac{i_m r_a}{k_t k_v}} C_l(s)$$

On the other hand, for the **motor with a current loop**, we assume the back EMF is negligible, giving:

$$\Omega_m(s) \approx \frac{k_t}{s i_m + f_m} \frac{K g_v}{K g_v k_i + s l_a + r_a} V_c(s) - \frac{1}{s i_m + f_m} C_l(s)$$

Further, since it is generally true that $K g_v k_i >> r_a, l_a$, the first-order approximate transfer function is:

$$\Omega_m(s) \approx \frac{\frac{k_t}{k_i f_m}}{1 + s\frac{i_m}{f_m}} V_c(s) - \frac{\frac{1}{f_m}}{1 + s\frac{i_m}{f_m}} C_l(s)$$

The input/output and disturbance/output transfer functions for the two types of motors can be written in a unified form:

$$\begin{aligned} \frac{\Omega_m(s)}{V_c(s)} &\approx \frac{k_m}{1 + sT_m} \\ \frac{\Omega_m(s)}{C_l(s)} &\approx \frac{k_d}{1 + sT_d} \end{aligned} \tag{IV.15.3}$$

For the **motor without a current loop**, we have:

$$\begin{aligned} k_m &= \frac{g_v}{k_v} \\ T_d = T_m &= \frac{i_m r_a}{k_t k_v} \\ k_d &= \frac{r_a}{k_t k_v} \end{aligned} \tag{IV.15.4}$$

For the **motor with a current loop**, we have:

$$\begin{aligned} k_m &= \frac{k_t}{k_i f_m} \\ T_d = T_m &= \frac{i_m}{f_m} \\ k_d &= -\frac{1}{f_m} \end{aligned} \tag{IV.15.5}$$

15.1.4 Decoupling of Motor System Dynamics

In robots, motors are not independent entities but rather an interconnected system of rigid bodies. For each joint motor in a robot, the load torque d in the above motor model is actually related to the joint positions/velocities of other joint motors. In other words, **the dynamics of motor torques in robots exhibit coupling relationships**.

In fact, this coupling relationship is precisely the robot dynamics described by Equation II.8.13. The joint angles and torques in this equation are the actual joint positions and torques. The direct output of the motor q_m and the joint angle q are often connected through a reduction mechanism. We use the reduction ratio matrix K_r to describe this relationship. Under ideal conditions (ignoring reduction mechanism errors), we have:

$$q_m = K_r q$$
$$\tau_m = K_r^{-1} \tau$$

Combining with Equation II.8.13, ignoring external forces on the entire robot, and assuming F_f is a constant (i.e., independent of $q, \dot{q}$), we can derive the motor dynamics equation for the robotic system:

$$K_r^{-1} B(q) K_r^{-1} \ddot{q}_m + K_r^{-1} C(q, \dot{q}) K_r^{-1} \dot{q}_m + K_r^{-1} F_f K_r^{-1} \dot{q}_m + K_r^{-1} g(q) = \tau_m \tag{IV.15.6}$$

As mentioned, this equation is a coupled equation. Below, we make some assumptions to decouple it into linear and nonlinear parts. First, we assume $B(q)$ can be decomposed into q-dependent and q-independent components:

$$B(q) = \bar{B} + \Delta B(q)$$

Next, we denote:

$$f_m = K_r^{-1} F_f K_r^{-1}$$

Thus, the equation can be rewritten as:

$$K_r^{-1} \bar{B} K_r^{-1} \ddot{q}_m + f_m \dot{q}_m + d = \tau_m \tag{IV.15.7}$$

Here, d represents the nonlinear coupling term, expressed as:

$$d = K_r^{-1} (\Delta B(q) K_r^{-1} \ddot{q}_m + C(q, \dot{q}) K_r^{-1} \dot{q}_m + g(q)) \tag{IV.15.8}$$

It can be observed that the coupling term d includes both gravitational and non-gravitational terms from the dynamics equation. The non-gravitational terms can be approximated as zero when the robot moves slowly. Additionally, note that $K_r, \bar{B}, f_m$ in Equation IV.15.7 are all diagonal matrices:

$$K_r = \text{diag}(k_{r,1}, k_{r,2}, ..., k_{r,N})$$
$$\bar{B} = \text{diag}(\bar{b}_1, \bar{b}_2, ..., \bar{b}_N)$$
$$f_m = \text{diag}(f_{m,1}, f_{m,2}, ..., f_{m,N})$$

Therefore, for each joint n, we have:

$$k_{r,n}^{-2} \bar{b}_n \ddot{q}_{m,n} + f_{m,n} \dot{q}_{m,n} + d_n = \tau_{m,n}$$

That is:

$$\tau_{m,n} - d_n = k_{r,n}^{-2} \bar{b}_n \dot{\omega}_{m,n} + f_{m,n} \omega_{m,n} \tag{IV.15.9}$$

In fact, this is the time-domain form of Equation IV.15.1 (the motor dynamics equation). The correspondence between the two equations is:

$$i_{m,n} = k_{r,n}^{-2} \bar{b}_n$$
$$c_{m,n} = \tau_{m,n}$$
$$c_{l,n} = d_n$$

Thus, according to Equation IV.15.8, the nonlinear coupling term in the robot dynamics is consolidated into the load torque $c_{l,n}$ of the single-joint motor. In other words, each joint in the robot satisfies the motor mechanics equation introduced earlier (i.e., Equation IV.15.1). Therefore, each joint in the robot can be directly modeled using the first-order dynamics from the previous subsection (Equation IV.15.3).

Based on this motor model, we transform the joint-space regulation/tracking control problem for the entire robot into **independent disturbance-rejection regulation/tracking control problems for each joint motor**.

Problem	Joint Disturbance-Rejection Regulation Control
Description	Given motor parameters, design a controller to stabilize the joint angle at a reference value while rejecting disturbances
Known	Motor parameters: $k_r, i_m, k_t, f_m, r_a, k_v, g_v$ Joint angle: θ Reference joint angle: θ_d
To Find	Control input (voltage) $u(\theta, \theta_d)$

Problem	Joint Disturbance-Rejection Tracking Control
Description	Given motor parameters, design a controller to make the joint angle follow a reference trajectory while rejecting disturbances
Known	Motor parameters: $k_r, i_m, k_t, f_m, r_a, k_v, g_v$ Joint angle and its derivative: $\theta, \dot{\theta}$ Reference joint angle and its derivatives: $\theta_d, \dot{\theta}_d, \ddot{\theta}_d$
To Find	Control input (voltage) $u(\theta, \dot{\theta}, \theta_d, \dot{\theta}_d, \ddot{\theta}_d)$

Next, we introduce some independent joint control methods.

15.2 Independent Joint Motion Control

15.2.1 Regulation Control

First, we consider the simple regulation problem. As introduced earlier, each joint motor in the robot can be described by the following equation:

$$s\Theta_m(s) = \frac{k_m}{1+sT_m}V_c(s) + \frac{k_d}{1+sT_d}D(s) \tag{IV.15.10}$$

Here, $D(s) = C_l(s)$ is the disturbance torque (load torque), corresponding to the joint coupling term in the robot dynamics. The goal is to design a controller to make the joint angle θ track the reference value θ_d. Note that there is a proportional relationship between the joint angle and the motor angle:

$$\theta_m = k_r\theta$$
$$\theta_{md} = k_r\theta_d$$

Since the simplified motor equation above describes angular velocity, to eliminate steady-state error, we can use a PI control strategy based on error feedback:

$$\begin{aligned} E(s) &= \Theta_{md}(s) - \Theta_m(s) \\ V_c(s) &= (k_P + \frac{k_I}{s})E(s) \end{aligned} \tag{IV.15.11}$$

Thus, our closed-loop feedback system is:

$$\begin{aligned} sk_r\Theta(s) &= \frac{k_m}{1+sT_m}V_c(s) + \frac{k_d}{1+sT_d}D(s) \\ E(s) &= k_r(\Theta_d(s) - \Theta(s)) \\ V_c(s) &= \left(k_P + \frac{k_I}{s}\right)E(s) \end{aligned}$$

In the closed-loop system, the path from $\Theta_d(s)$ to $\Theta(s)$ forms a unit negative feedback, with the open-loop forward transfer function:

$$H_{o,\theta}(s) = \frac{k_m k_I(1 + s\frac{k_P}{k_I})}{s^2(1+sT_m)}$$

The closed-loop transfer function is:

$$H_{c,\theta}(s) = \frac{1 + s\frac{k_P}{k_I}}{1 + s\frac{k_P}{k_I} + s^2\frac{1}{k_m k_I} + s^3\frac{T_m}{k_m k_I}}$$

The closed-loop transfer function from $D(s)$ to $\Theta(s)$ is:

$$H_{c,d}(s) = \frac{\frac{k_d}{k_m k_I}(1 + sT_m)}{(1 + sT_d)(1 + s\frac{k_I + k_m k_P}{k_m k_I} + s^2\frac{T_m k_r}{k_m k_I})}$$

For frequency-domain stability requirements, the poles of both closed-loop transfer functions must be placed in the left half-plane. To suppress disturbances, k_I should be as large as possible.

We summarize the **Independent Joint PI Control** as follows:

Algorithm	**Independent Joint PI Control**
Problem Type	Joint Disturbance Rejection Regulation Control
Given	Motor parameters $k_r, i_m, k_t, f_m, r_a, k_v, g_v$ Joint angle θ Reference joint angle θ_d
Find	Control output (control voltage) $u(\theta, \theta_d)$
Algorithm Property	model-based, analytical solution

Algorithm 38: Independent Joint PI Control

Input: Joint angle θ, Reference joint angle θ_d
Parameter: Parameters k_P, k_I
Output: Voltage command u
$e \leftarrow k_r(\theta_d - \theta)$
$u \leftarrow k_P e + k_I \int e dt$

15.2.2 Tracking Control

Now let's examine the tracking problem. Compared to the regulation problem, the tracking problem requires feedback of joint angular velocity ω in addition to the joint angle. Assuming the first derivative of the reference trajectory is also known and continuous, we have:

$$\omega_m = k_r \omega$$
$$\omega_{md} = k_r \omega_d$$

This gives us observation of the tracking error derivative, allowing the use of PID strategy:

$$\begin{aligned} E(s) &= \Theta_{md}(s) - \Theta_m(s) \\ sE(s) &= \Omega_{md}(s) - \Omega_m(s) \\ V_c(s) &= \left(k_P + \frac{k_I}{s}\right) E(s) + K_D s E(s) \end{aligned} \tag{IV.15.12}$$

We summarize the **Independent Joint PID Control** as follows:

Algorithm	**Independent Joint PID Control**
Problem Type	Joint Disturbance Rejection Tracking Control
Given	Motor parameters $k_r, i_m, k_t, f_m, r_a, k_v, g_v$ Joint angle and angular velocity θ, ω Reference joint angle and angular velocity θ_d, ω_d
Find	Control output (control voltage) $u(\theta, \omega, \theta_d, \omega_d)$
Algorithm Property	model-based, analytical solution

Algorithm 39: Independent Joint PID Control

Input: Joint angle and angular velocity θ, ω
Input: Reference joint angle and angular velocity θ_d, ω_d
Parameter: Parameters k_P, k_I
Output: Voltage command u
$u \leftarrow k_r(k_P(\theta_d - \theta) + k_I \int(\theta_d - \theta)dt + k_D(\omega_d - \omega))$

(Reference: "Robotics: Modelling, Planning and Control")

16 Joint Space Control

Joint space control encompasses both fundamental regulation and tracking control (collectively referred to as motion control) as well as impedance control methods.

16.1 Joint Space Motion Control

16.1.1 Gravity-Compensated PD Control in Joint Space

First, we consider stabilizing the robot at a specific posture given a reference joint angle q_d, i.e., the joint space regulation problem.

Since we are addressing the regulation problem, the robot remains stationary at equilibrium, and the dynamics equation will not include inertia and Coriolis terms. Therefore, we adopt the simplest control form: PID control with gravity feedback.

As mentioned earlier, PID control requires the system's error state. Since the control is performed in joint space, the robot's joint vector $q(t)$ serves as the controlled state. The error state is defined as:

$$\tilde{q}(t) = q_d - q(t) \tag{IV.16.1}$$

We need to use the proportional and derivative terms, which require the error state and its first-order derivative $\dot{\tilde{q}}(t)$. For the regulation problem, the time derivative of q_d is zero, so we only need $\dot{\tilde{q}}(t) = -\dot{q}(t)$. In practical robot systems, $q(t)$ and $\dot{q}(t)$ are obtained from sensors and observers (typically some form of filter).

The gravity-compensated PD control is expressed as:

$$u(t) = K_P\tilde{q}(t) - K_D\dot{q}(t) + g(q) \tag{IV.16.2}$$

Note that $u(t)$ here represents the joint driving torque vector, which, like $q(t)$, is an N-dimensional vector. In this case, K_P and K_D are square matrices. In fact, these matrices should be selected as positive definite matrices.

Combining with the robot dynamics equation (open-loop system dynamics) that includes sliding friction terms:

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) = u \tag{IV.16.3}$$

The structure of the closed-loop system is as follows:

$$\begin{aligned} B(q(t))\ddot{q}(t) + C(q(t),\dot{q}(t))\dot{q}(t) + F_f\dot{q}(t) + g(q(t)) &= u(t) \\ \tilde{q}(t) &= q_d - q(t) \\ u(t) &= K_P\tilde{q}(t) - K_D\dot{q}(t) + g(q(t)) \end{aligned}$$

The gravity-compensated PD control algorithm in joint space is summarized below:

Algorithm 40: JS Gravity-Compensated PD Control

Input: Link parameters $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$
Input: D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$
Input: Joint vector q and its derivative $\dot{q}$
Input: Reference joint angle q_d
Parameter: Matrix parameters K_P, K_D
Output: Torque command u
$g \leftarrow \text{robot\_dyn}(q, m_n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$u \leftarrow K_P(q_d - q) - K_D\dot{q}(t) + g$

Algorithm	JS Gravity-Compensated PD Control
Problem Type	Joint Space Regulation Control
Known	D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$ Link parameters $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ Joint vector and its derivative $q, \dot{q}$ Reference joint angle q_d
Required	Control input (driving torque) $u(q, \dot{q}, q_d)$
Algorithm Property	Model-based, analytical solution

Next, we prove the system's stability.

Proof. To prove the stability of the closed-loop system, we select the following positive definite quadratic form as the Lyapunov candidate function:

$$V(\dot{q}, \tilde{q}) = \frac{1}{2}\dot{q}^T B(q)\dot{q} + \frac{1}{2}\tilde{q}^T K_P \tilde{q} \quad > 0, \quad \forall \dot{q}, \tilde{q} \neq 0 \tag{IV.16.4}$$

Taking its first derivative, we have:

$$\dot{V} = \frac{1}{2}\dot{q}^T \dot{B}(q)\dot{q} + \dot{q}^T B(q)\ddot{q} - \dot{q}^T K_p \tilde{q} \tag{IV.16.5}$$

Substituting the dynamics equation (Eq. IV.16.3) yields:

$$\begin{aligned}\dot{V} &= \frac{1}{2}\dot{q}^T \dot{B}(q)\dot{q} + \dot{q}^T(u - C(q,\dot{q})\dot{q} - F_f\dot{q} - g(q)) - \dot{q}^T K_p \tilde{q} \\ &= \frac{1}{2}\dot{q}^T(\dot{B}(q) - 2C(q,\dot{q}))\dot{q} - \dot{q}^T F_f \dot{q} + \dot{q}^T(u - g(q) - K_p\tilde{q})\end{aligned}$$

Using the property of the N matrix introduced in Part II, Chapter 4, we obtain:

$$\dot{V} = -\dot{q}^T F_f \dot{q} + \dot{q}^T(u - g(q) - K_p\tilde{q})$$

Substituting the control law from Eq. IV.16.2 gives:

$$\begin{aligned}\dot{V} &= -\dot{q}^T F_f \dot{q} + \dot{q}^T(K_P\tilde{q} - K_D\dot{q}(t) + g(q) - g(q) - K_p\tilde{q}) \\ &= -\dot{q}^T(F_f + K_D)\dot{q}\end{aligned}$$

The friction coefficient matrix F_f is positive semi-definite. Selecting K_D as a symmetric positive definite matrix yields:

$$\dot{V} < 0, \quad \forall \dot{q} \neq 0$$

Therefore, if K_p and K_D are both symmetric positive definite, the closed-loop system is asymptotically stable according to the Lyapunov stability criterion.

□

16.1.2 Joint Space Inverse Dynamics Control

The aforementioned gravity-compensated PD control in joint space is suitable for scenarios where the robot is stationary or moving slowly. However, if the target trajectory involves faster motion, the convergence of this control algorithm deteriorates. This is because, for rapid reference trajectories, the terms $B(q)\ddot{q}$ and $C(q,\dot{q})\dot{q}$ can no longer be neglected.

To address joint space tracking control with a reference trajectory $q_d(t)$, we can reintroduce these two terms into the control loop. Specifically, the error state now becomes:

$$\tilde{q}(t) = q_d(t) - q(t) \tag{IV.16.6}$$

Note that for the reference trajectory, its derivatives $\dot{q}_d(t), \ddot{q}_d(t), \ldots$ are all known.

In robot systems, the joint driving torque τ is the freely controllable quantity, referred to as the **direct control input**. Following general control theory notation, we denote it as u.

Adopting the exact linearization approach, we use the dynamics formula to introduce nonlinear feedback, transforming the free control τ into the free control $\ddot{q}$:

$$u = B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) \tag{IV.16.7}$$

Here, $\ddot{q}$ in the system becomes the freely configurable control input, referred to as the **indirect control input** and denoted as y:

$$y = \ddot{q} \tag{IV.16.8}$$

This design effectively decouples the controller. The n-th component of the control input y affects only the n-th joint and is independent of other joints.

For the controller design, we still employ a PD controller:

$$y = K_P\tilde{q} + K_D\dot{\tilde{q}} + \ddot{q}_d \tag{IV.16.9}$$

The structure of the closed-loop system is as follows:

$$\begin{aligned} B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) &= u \\ u &= B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) \\ \tilde{q}(t) &= q_d(t) - q(t) \\ y &= K_P\tilde{q} + K_D\dot{\tilde{q}} + \ddot{q}_d \end{aligned}$$

The error state of the closed-loop system satisfies the following second-order differential equation.

$$\ddot{\tilde{q}} + K_D\dot{\tilde{q}} + K_P\tilde{q} = 0 \tag{IV.16.10}$$

It can be observed that the error dynamics of the closed-loop system resemble the typical second-order system described by Eq. III.10.10. However, this system lacks an input term. It can be understood as a typical second-order system without external forces that automatically returns to the zero point under certain initial conditions. Here, K_D and K_P are analogous to the damping coefficient and stiffness coefficient in a mechanical second-order system, respectively.

To summarize, the joint-space inverse dynamics control algorithm is as follows:

Algorithm 41: JS Inverse Dynamics PD Control

Input: Link parameters $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$
Input: D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$
Input: Joint vector q and its derivative $\dot{q}$
Input: Reference joint trajectory and its derivatives $q_d, \dot{q}_d, \ddot{q}_d$
Parameter: Matrix parameters K_P, K_D, friction matrix F_f
Output: Torque command u
$B, C, g \leftarrow \text{robot\_dyn}(q, \dot{q}, m_n, \mathbf{I}_n^n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$y \leftarrow K_P(q_d - q) + K_D(\dot{q}_d - \dot{q}) + \ddot{q}_d$
$u \leftarrow By + C\dot{q} + F_f\dot{q} + g$

Algorithm	**JS Inverse Dynamics PD Control**
Problem Type	Joint-space tracking control
Known	D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$ Link parameters $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ Joint vector and its derivative $q, \dot{q}$ Reference joint trajectory and its derivatives $q_d, \dot{q}_d, \ddot{q}_d$
To Find	Control input (driving torque) $u(q, \dot{q}, q_d, \dot{q}_d, \ddot{q}_d)$
Algorithm Property	model-based, analytical solution

Next, we proceed with the stability proof of the system.

Proof. Assume K_P is positive definite. Take the Lyapunov Candidate function as:

$$V(\tilde{q}, \dot{\tilde{q}}) = \frac{1}{2}\tilde{q}^T K_P \tilde{q} + \frac{1}{2}\dot{\tilde{q}}^T \tilde{\dot{q}} \quad > 0, \quad \forall \dot{q}, \tilde{q} \neq 0$$

Differentiating with respect to time yields:

$$\begin{aligned} \dot{V} &= \tilde{q}^T K_P \dot{\tilde{q}} + \dot{\tilde{q}}^T \ddot{\tilde{q}} \\ &= \tilde{q}^T K_P \dot{\tilde{q}} + \dot{\tilde{q}}^T (-K_D \dot{\tilde{q}} - K_P \tilde{q}) \\ &= -\dot{\tilde{q}}^T K_D \dot{\tilde{q}} \end{aligned}$$

Assuming K_D is symmetric positive definite, we have:

$$\dot{V} < 0, \quad \forall \dot{q} \neq 0$$

Therefore, if both K_p and K_D are symmetric positive definite, according to the Lyapunov stability criterion, the closed-loop system is asymptotically stable.

□

In practical applications, K_P and K_D can be chosen as diagonal matrices, thereby decoupling the dynamics of each joint into second-order systems. Here, w_n represents the natural frequency of each system, and ξ_n represents the damping ratio of each system.

$$\begin{aligned} K_P &= \text{diag}\{w_1^2, ..., w_N^2\} \\ K_D &= \text{diag}\{2\xi_1 w_1, ..., 2\xi_N w_N\} \end{aligned} \tag{IV.16.11}$$

Additionally, it is important to note: whether for joint-space gravity-compensated PD control or joint-space inverse dynamics control, both require real-time computation of partial or complete dynamic equation terms in the control law.

As discussed in Chapter 4 of Part III, the computation of dynamic equations is relatively complex. Real-time computation of these terms at high frequencies was once an engineering challenge. However, with advancements in computer technology, such computational demands are no longer an issue.

(Reference: ”Robotics: Modelling, Planning and Control”)

16.2 Joint-Space Impedance Control

[This section will be updated in future versions. Stay tuned!]

17 Operational Space Control

In the previous chapter, we introduced joint space control. In contrast, operational space control involves reference signals originating from the operational space. Therefore, we need to flexibly utilize the Jacobian matrix and inverse kinematics for signal transformation.

17.1 Operational Space Motion Control

17.1.1 Operational Space Gravity-Compensated PD Control

Similar to joint space control, we first consider the regulation problem. Let the reference trajectory point of the end-effector be x_d. The operational space error state is defined as:

$$\tilde{x}(t) = x_d - x_e(t) \tag{IV.17.1}$$

A key distinction in operational space control is that we assume the robot states $x_e(t)$ and $\dot{x}_e(t)$ cannot be directly measured but must be derived from the joint space states $q(t)$ and $\dot{q}(t)$[16], i.e.,

$$\begin{aligned} x_e &= k(q) \\ \dot{x}_e &= J_a(q)\dot{q} \end{aligned} \tag{IV.17.2}$$

Note that the analytical Jacobian is used for velocity calculation because the provided trajectories are often expressed in terms of Euler angles or quaternions. To perform subtraction of operational space error states, the analytical Jacobian must also be used instead of the geometric Jacobian for observation.

Following a similar approach to joint space control, we formulate the control law in PD form. Note that the operational space PD directly yields the end-effector force, which must be converted to joint torques using the transpose Jacobian.

$$\begin{aligned} u(t) &= J_a^T(K_P\tilde{x}(t) - K_D\dot{x}_e(t)) + g(q) \\ \tilde{x}(t) &= x_d - k(q(t)) \\ \dot{x}_e &= J_a(q)\dot{q} \end{aligned}$$

Substituting the observation process for the two operational space states, we obtain:

$$u(t) = J_a^T(K_P\tilde{x} - K_D J_a(q)\dot{q}) + g(q) \tag{IV.17.3}$$

Summarizing the closed-loop system structure:

$$\begin{aligned} B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) &= u(t) \\ \tilde{x}(t) &= x_d - k(q(t)) \\ u(t) &= J_a^T(K_P\tilde{x} - K_D J_a(q)\dot{q}) + g(q) \end{aligned}$$

The operational space gravity-compensated PD control algorithm is summarized as follows:

Algorithm 42: OS Gravity-Compensated PD Control

Input: Link parameters $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$
Input: D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$
Input: Joint vector q and its derivative $\dot{q}$
Input: Reference trajectory x_d
Parameter: Matrix parameters K_P, K_D
Output: Torque command u
$x_e \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q)$
$J_a \leftarrow \text{robot\_jacobian\_a}(q, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$g \leftarrow \text{robot\_dyn}(q, m_n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$u \leftarrow J_a^T(K_P(x_d - x_e) - K_D J_a\dot{q}) + g$

[16] In practice, other sensors/observers may be used to estimate the system states. However, regardless of how the states are obtained, they must satisfy the following relationships.

Algorithm	**OS Gravity-Compensated PD Control**
Problem Type	Operational Space Regulation Control
Known	D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$ Link parameters $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ Joint vector and its derivative $q, \dot{q}$ Reference trajectory x_d
To Find	Control input (driving torque) $u(q, \dot{q}, x_d)$
Algorithm Property	Model-based, analytical solution

Next, we provide the stability proof for the system.

Proof. Let the operational space error $\tilde{x}$ and joint space velocity $\dot{q}$ be the states of the closed-loop system. Assuming K_P is symmetric positive definite, construct the Lyapunov candidate function as:

$$V(\tilde{x}, \dot{q}) = \frac{1}{2}\dot{q}^T B(q)\dot{q} + \frac{1}{2}\tilde{x}^T K_P \tilde{x} \quad > 0, \quad \forall \dot{q}, \tilde{q} \neq 0$$

Differentiating with respect to time:

$$\begin{aligned}\dot{V} &= \frac{1}{2}\dot{q}^T \dot{B}(q)\dot{q} + \dot{q}^T B(q)\ddot{q} - \dot{\tilde{x}}^T K_p \tilde{x} \\ &= \frac{1}{2}\dot{q}^T \dot{B}(q)\dot{q} + \dot{q}^T (u - C(q,\dot{q})\dot{q} - F_f \dot{q} - g(q)) - \dot{\tilde{x}}^T K_p \tilde{x} \\ &= \frac{1}{2}\dot{q}^T (\dot{B}(q) - 2C(q,\dot{q}))\dot{q} - \dot{q}^T F_f \dot{q} + \dot{q}^T (u - g(q)) - \dot{\tilde{x}}^T K_p \tilde{x} \\ &= -\dot{q}^T F_f \dot{q} + \dot{q}^T (u - g(q)) - \dot{\tilde{x}}^T K_p \tilde{x}\end{aligned}$$

Note that:

$$\dot{\tilde{x}} = -\dot{x}_e = -J_a(q)\dot{q} \tag{IV.17.4}$$

Thus:

$$\begin{aligned}\dot{V} &= -\dot{q}^T F_f \dot{q} + \dot{q}^T (u - g(q)) + \dot{q}^T J_a^T(q) K_p \tilde{x} \\ &= -\dot{q}^T F_f \dot{q} + \dot{q}^T (u - g(q) + J_a^T(q) K_p \tilde{x})\end{aligned}$$

Substituting the control law (Equation IV.17.3):

$$\begin{aligned}\dot{V} &= -\dot{q}^T F_f \dot{q} + \dot{q}^T (J_a^T (K_P \tilde{x} - K_D J_a(q)\dot{q}) + g(q) - g(q) + J_a^T(q) K_p \tilde{x}) \\ &= -\dot{q}^T F_f \dot{q} - \dot{q}^T J_a^T K_D J_a(q)\dot{q} \\ &= -\dot{q}^T (F + J_a^T(q) K_D J_a(q))\dot{q}\end{aligned}$$

The friction coefficient matrix F is positive semi-definite. Let K_D be symmetric positive definite, and assume the analytical Jacobian matrix is full-rank. Then:

$$\dot{V} < 0, \quad \forall \dot{q} \neq 0$$

Therefore, if K_p and K_D are both symmetric positive definite, the closed-loop system is asymptotically stable according to the Lyapunov stability criterion.

□

17.1.2 Operational Space Inverse Dynamics Control

Similar to joint space inverse dynamics control, in operational space, for tracking time-varying reference trajectories $x_d(t)$ and their derivatives, additional compensation control terms are required.

Assume the higher-order derivatives of $x_d(t)$ are known. Define the operational space error state as:

$$\tilde{x}(t) = x_d(t) - x_{ea}(t) \tag{IV.17.5}$$

Following the exact linearization approach in joint space, we still perform exact linearization, using $\ddot{q}$ as the indirect control input y:

$$u = B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) \tag{IV.17.6}$$

We have:

$$y = \ddot{q}$$

Next, we design the indirect control input. Following the joint space approach, we aim to design the error dynamics of the closed-loop system in operational space as a second-order linear form.

$$K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{\tilde{x}} = 0 \tag{IV.17.7}$$

Unlike joint space, here the operational space states and their derivatives $x_{ea}(t), \dot{x}_{ea}(t), \ddot{x}_{ea}(t)$ cannot be directly obtained. Recalling the system dynamics and the first- and second-order Jacobian equations (Eqs. II.7.1 and II.7.5), we have

$$\begin{aligned}
\tilde{x}(t) &= x_d(t) - k(q(t)) \\
\dot{\tilde{x}}(t) &= \dot{x}_d(t) - J_a(q)\dot{q}(t) \\
\ddot{\tilde{x}}(t) &= \ddot{x}_d(t) - J_a(q)\ddot{q}(t) - \dot{J}_a(q)\dot{q}(t)
\end{aligned} \tag{IV.17.8}$$

The left-hand side of Eq. IV.17.7 can be rewritten as

$$K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{\tilde{x}} = K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - J_a(q)\ddot{q}(t) - \dot{J}_a(q)\dot{q}(t)$$

Considering $y = \ddot{q}(t)$ as freely configurable, we only need to set the right-hand side of this equation to 0 and solve for $y(t)$ to achieve our goal:

$$K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - J_a(q)y(t) - \dot{J}_a(q)\dot{q}(t) = 0$$

Thus, the designed indirect control input is:

$$y(t) = J_a^{-1}(q)(K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - \dot{J}_a(q)\dot{q}(t)) \tag{IV.17.9}$$

Here, the inverse of the analytical Jacobian is used. This matrix is invertible if and only if the **robot is non-redundant**, which is also a necessary condition for operational space inverse dynamics control.

Summarizing the entire closed-loop system structure, we have:

$$\begin{aligned}
B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) &= u(t) \\
u &= B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) \\
y &= J_a^{-1}(q)(K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d - \dot{J}_a(q,\dot{q})\dot{q}) \\
\tilde{x}(t) &= x_d(t) - k(q(t)) \\
\dot{\tilde{x}}(t) &= \dot{x}_d(t) - J_a(q)\dot{q}(t)
\end{aligned}$$

The operational space inverse dynamics control is summarized as follows. Note: In this algorithm, we use automatic differentiation to compute the time derivative of the analytical Jacobian; it is also essential to ensure the analytical Jacobian is invertible.

Algorithm	**OS Inverse Dynamics PD Control**
Problem Type	Operational Space Tracking Control
Known	D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$ Link parameters $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ Friction matrix F_f Joint vector and its derivative $q, \dot{q}$ Reference trajectory and its derivatives $x_d, \dot{x}_d, \ddot{x}_d$
To Find	Control input (driving torque) $u(q, \dot{q}, x_d, \dot{x}_d, \ddot{x}_d)$
Algorithm Properties	model-based, analytical solution

Algorithm 43: OS Inverse Dynamics PD Control

Input: Link parameters $m_{1:N}, \mathbf{I}_{:}^{:}, p_{l_:}^{:}$, D-H parameters $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: Friction matrix F_f
Input: Joint vector q and its derivative $\dot{q}$
Input: Reference trajectory and its derivatives $x_d, \dot{x}_d, \ddot{x}_d$
Parameter: Matrix parameters K_P, K_D, friction matrix F_f
Output: Torque command u

$x_e \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q)$
$J_a \leftarrow \text{robot\_jacobian\_a}(q, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$\dot{J}_a \leftarrow \text{auto\_diff}(J_a, t)$
$B, C, g \leftarrow \text{robot\_dyn}(q, \dot{q}, m_n, \mathbf{I}_n^n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$y \leftarrow J_a^{-1}(K_P(x_d - x_e) + K_D(\dot{x}_d - J_a\dot{q}) + \ddot{x}_d - \dot{J}_a\dot{q})$
$u \leftarrow By + C\dot{q} + F_f\dot{q} + g$

For stability proof, since the closed-loop system has been feedback-linearized into a linear second-order form (Eq. IV.17.7), when K_p, K_D are both symmetric positive definite, the stabilization proof for this second-order system is identical to that of joint space inverse dynamics control and will not be repeated here.

(References: Tsinghua "Intelligent Robotics" course materials, "Robotics: Modelling, Planning and Control")

17.2 Operational Space Impedance Control

Next, we consider the problem of operational space impedance/admittance control.

17.2.1 Impedance Control Without Feedback

First, assuming the end-effector force cannot be measured, consider the system dynamics with an end-effector external force term:

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) = u(t) - J_a^T F_e \tag{IV.17.10}$$

Still adopting the operational space inverse dynamics control method, the closed-loop system error should satisfy the following form:

$$K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{\tilde{x}} = B_f F_e \tag{IV.17.11}$$

where B_f is a positive definite matrix to be determined. Since our exact linearization method does not include the external force term:

$$u = B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q)$$

The external force term will be reflected in the operational space acceleration along with the indirect control input:

$$\ddot{q} = y - B^{-1}(q)J_a^T(q)F_e \tag{IV.17.12}$$

Substituting the second-order Jacobian and indirect control input into the error dynamics, we have:

$$K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - J_a(q)(y - B^{-1}(q)J_a^T(q)F_e) - \dot{J}_a(q)\dot{q}(t) = B_f F_e$$

That is:

$$y = J_a(q)^{-1}(K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - \dot{J}_a(q)\dot{q}(t) + (J_a(q)B^{-1}(q)J_a^T(q) - B_f)F_e)$$

Since the control input does not include the end-effector force, the last term must be 0. That is: the relationship between the system error and the external force is simply the inverse of the system's operational space inertia matrix:

$$B_f(q) = (J_a^{-1}(q)B(q)J_a^{-T}(q))^{-1} \tag{IV.17.13}$$

Therefore, we can conclude: under the operational space inverse dynamics control law, the system naturally exhibits the impedance characteristics described by Eq. IV.14.2. However, this characteristic B_f is coupled and inherent to the system.

17.2.2 Impedance Control With Feedback

In the above system, we can already control the system's stiffness and damping characteristics through K_P and K_D. However, since $B_f(q)$ exists and cannot be configured, the above system is not decoupled. If we want to independently adjust the characteristics of a specific joint, complex calculations are required.

To achieve **decoupled control**, we need to measure the end-effector force F_e. Assuming precise measurements of F_e are available, we can add a feedback linearization term to the control input.

$$u = B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) + J_a^T(q)F_e \tag{IV.17.14}$$

At this point, we introduce a new equivalent force $F_a = B_iF_e$ as the input to the closed-loop error dynamics of the system. Here, the matrix B_i corresponds to B_f in passive impedance control, but the difference lies in that B_i can be selected and configured as needed.

$$K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{\tilde{x}} = B_iF_e = F_a \tag{IV.17.15}$$

At this stage, the indirect control variable of the system is:

$$y = J_a(q)^{-1}(K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - \dot{J}_a(q)\dot{q}(t) - B_iF_e) \tag{IV.17.16}$$

Summarizing the entire closed-loop system structure, we have:

$$\begin{aligned}
B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) &= u(t) - J_a^TF_e \\
u &= B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) + J_a^T(q)F_e \\
y &= J_a(q)^{-1}(K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - \dot{J}_a(q)\dot{q}(t) - B_iF_e) \\
\tilde{x}(t) &= x_d(t) - k(q(t)) \\
\dot{\tilde{x}}(t) &= \dot{x}_d(t) - J_a(q)\dot{q}(t)
\end{aligned}$$

Based on the preceding derivations, we summarize the robot impedance control algorithm as follows:

Algorithm 44: OS Impedance Control

Input: Link parameters $m_{1:N}, \mathbf{I}_{:}^{:}, p_{l_:}^{:}$, D-H parameters $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: Friction matrix F_f
Input: Joint vector q and its derivative $\dot{q}$, end-effector force F_e
Input: Reference trajectory and its derivatives $x_d, \dot{x}_d, \ddot{x}_d$
Parameter: Matrix parameters K_P, K_D, coefficient matrix B_i
Output: Torque command u
$x_e \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q)$
$J_a \leftarrow \text{robot\_jacobian\_a}(q, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$\dot{J}_a \leftarrow \text{auto\_diff}(J_a, t)$
$B, C, g \leftarrow \text{robot\_dyn}(q, \dot{q}, m_n, \mathbf{I}_n^n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$y \leftarrow J_a^{-1}(K_P(x_d - x_e) + K_D(\dot{x}_d - J_a\dot{q}) + \ddot{x}_d - \dot{J}_a\dot{q} - F_a)$
$u \leftarrow By + C\dot{q} + F_f\dot{q} + g + J_a^TF_e$

Algorithm	OS Impedance Control
Problem Type	Operational Space Impedance Control Problem
Known	D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$ Link parameters $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ Friction matrix F_f Joint vector and its derivative $q, \dot{q}$, end-effector force F_e Reference trajectory and its derivatives $x_d, \dot{x}_d, \ddot{x}_d$
To Find	Control variable (driving torque) $u(q, \dot{q}, x_d, \dot{x}_d, \ddot{x}_d, F_e)$
Algorithm Property	model-based, analytical solution

Note: The prerequisite for using the above algorithm is that automatic differentiation must be employed to compute the time derivative of the analytical Jacobian matrix, and the analytical Jacobian must be invertible.

In practical applications, the achievable degrees of freedom in impedance control depend on the dimensionality of the measurable end-effector force. If the end-effector force is 6-dimensional, 6-DOF impedance control can be implemented. If the end-effector force is 3-dimensional, only 3-DOF impedance control can be achieved.

17.2.3 Zero-Force Control

Operational space zero-force control can be easily derived by modifying the impedance control framework. Specifically, zero-force control does not have a given trajectory; the robot needs to exhibit second-order system characteristics under external forces and should remain stationary when external forces disappear. Therefore, in the indirect control variable of the aforementioned impedance control, we remove the position tracking term and set $\dot{x}_d, \ddot{x}_d$ to zero:

$$y = J_a(q)^{-1}(K_D\dot{x} - \dot{J}_a(q)\dot{q}(t) - B_i F_e) \tag{IV.17.17}$$

The closed-loop system structure at this stage is:

$$\begin{aligned} B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) &= u(t) - J_a^T F_e \\ u &= B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) + J_a^T(q)F_e \\ y &= J_a(q)^{-1}(K_D\dot{x} - \dot{J}_a(q)\dot{q}(t) - B_i F_e) \\ \ddot{x}(t) &= -\dot{J}_a(q)\dot{q}(t) \end{aligned}$$

Thus, we obtain the **Operational Space Zero-Force Control** algorithm:

Algorithm 45: OS Zero-Force Control

Input: Link parameters $m_{1:N}, \mathbf{I}_{:}^{:}, p_{l_:}^{:}$, D-H parameters $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: Friction matrix F_f
Input: Joint vector q and its derivative $\dot{q}$, end-effector force F_e
Parameter: Matrix parameter K_D, coefficient matrix B_i
Output: Torque command u

$x_e \leftarrow$ robot_fk$(d_{1:N}, a_{1:N}, \alpha_{1:N}, q)$
$J_a \leftarrow$ robot_jacobian_a$(q, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$\dot{J}_a \leftarrow$ auto_diff(J_a, t)
$B, C, g \leftarrow$ robot_dyn$(q, \dot{q}, m_n, \mathbf{I}_n^n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$y \leftarrow J_a(q)^{-1}(K_D\dot{x} - \dot{J}_a(q)\dot{q}(t) - B_i F_e)$
$u \leftarrow By + C\dot{q} + F_f\dot{q} + g + J_a^T F_e$

Algorithm	**OS Zero-Force Control**
Problem Type	Operational Space Zero-Force Control
Known	D-H parameters $d_n, a_n, \alpha_n, n = 1, ..., N$ Link parameters $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ Friction matrix F_f Joint vector and its derivative $q, \dot{q}$, end-effector force F_e
To Find	Control variable (driving torque) $u(q, \dot{q}, x_d, \dot{x}_d, \ddot{x}_d, F_e)$
Algorithm Property	model-based, analytical solution

17.2.4 Admittance Control

[This section will be updated in future versions. Stay tuned.]

17.3 Operational Space Constrained Force Control

[This section will be updated in future versions. Stay tuned.]

Part V
Deep Learning Methods

In the previous sections, we introduced the mathematical/physical foundations of robotics, general modeling methods and conclusions (i.e., kinematics/dynamics), control theory, and simple control methods for robots. Through these introductions, readers should now have a basic grasp of the mathematics used in robotics.

However, the aforementioned theories were largely established as early as the 1970s. In practice, robots built using these theories, while capable of executing human instructions precisely, still fall short of higher-level human intelligence such as "autonomous perception" and "autonomous decision-making." Such robots, like industrial robotic arms on assembly lines, embody the "mechanical, rigid, and inflexible" image commonly associated with robots, rather than the natural, biomimetic, and agile image of modern robots.

Starting from Part V, we will introduce a powerful tool in modern robotics—deep learning—and, based on this, explore modern robotic technologies that enable robots to possess preliminary **autonomous perception, autonomous decision-making, and autonomous learning** capabilities, such as reinforcement learning, visual SLAM, and legged locomotion gaits. These technologies form the foundation for future humanoid robots with truly biomimetic intelligence.

18 Foundations of Deep Learning

We—humans—are a unique species on this planet. While many factors contribute to our uniqueness, the most central is our intelligence—adaptive, perceptive, adept at learning, generalizable, endowed with language and thought, and possessing logic and emotion—human intelligence.

Since the dawn of modern science, we have never ceased exploring the origins of our own intelligence—from physiology, neuroscience, and brain science to linguistics, psychology, and sociology, from control science, computer science to robotics—we seek to uncover the roots of our wisdom, **we aspire to create robots as intelligent as ourselves, to better serve human society**.

In this journey of exploration, **deep learning** (Deep Learning, DL) has been the technological field that has brought us the most surprises. Machine Learning (ML) is a branch of data science within information science, studying how to extract patterns and acquire knowledge from vast amounts of data. Deep learning is a subfield of machine learning, focusing on machine learning models with "deep" architectures.

Since 2012, a class of deep learning models called **Deep Neural Networks** (DNNs) has emerged as a breakthrough in data science. Specifically, DNNs are multilayer nonlinear machine learning models inspired by human neurons. Their diverse architectures and vast parameter counts enable powerful complex function approximation capabilities. Since 2012, DNNs have achieved groundbreaking progress in processing "challenging and complex" information such as images, text, speech, time-series data, semi-structured data, and multimodal data, addressing long-standing issues like weak feature extraction and poor task performance. DNNs have thus become the most important method in machine learning. In the subsequent chapters of this section, we will detail these tasks, data types, and corresponding DNN methods.

Deep learning is the subfield of machine learning that studies multilayer nonlinear models like DNNs. Currently, the most important application areas of deep learning fall into three categories: **Computer Vision** (CV), **Reinforcement Learning** (RL), and **Natural Language Processing** (NLP). These three fields are integral parts of the broader artificial intelligence domain and are crucial for building truly intelligent humanoid robots.

In this chapter, we will first introduce key fundamental concepts in machine learning, then explain the concept of DNN models, and use the simplest DNN model (Multilayer Perceptron, MLP) as an example to illustrate the basic paradigm of deep learning.

18.1 Basic Concepts of Machine Learning

Machine learning is the science of learning patterns from data. One of its fundamental objects of study is a collection of data in the same format, called a **dataset**. Hidden within the dataset are unknown patterns, and machine learning aims to **parameterize** these patterns through modeling; this model is called a **machine learning model**, or simply a model. The parameters within the model are called **model parameters**. For each specific dataset, there is a corresponding task, known as a **machine learning task**. Based on

the characteristics of different tasks/datasets, we design different machine learning models and methods to optimize their parameters; the computational methods used in this process are also called **machine learning algorithms**. The process of optimizing model parameters is also referred to as **model learning** or **model training**.

18.1.1 Basic Problems in Machine Learning

Specifically, in machine learning, there are four fundamental problems that constitute the majority of machine learning tasks:

Problem	**Regression Problem**
Problem Description	Given a sample set and a label set, find the optimal transformation to fit the mapping from samples to labels
Given	Sample set $X = [\mathbf{x^{(1)}}, \mathbf{x^{(2)}}, ..., \mathbf{x^{(N)}}] \in \mathbb{R}^{n\times N}$ Label set $Y = [y^{(1)}, y^{(2)}, .., y^{(n)}]^T$
Objective	Find a transformation $f : \mathbb{R}^n \to \mathbb{R}$ such that $f(\mathbf{x^{(i)}})$ optimally fits $y^{(i)}$

Problem	**Classification Problem**
Problem Description	Given a sample set and a label set, find the optimal transformation to fit the mapping from samples to class labels
Given	Sample set $X = [\mathbf{x^{(1)}}, \mathbf{x^{(2)}}, ..., \mathbf{x^{(N)}}] \in \mathbb{R}^{n\times N}$ Label set $Y = [y^{(1)}, y^{(2)}, .., y^{(n)}]^T$, where $\mathbf{y^{(i)}} \in L_c = \{c_i \vert i = 1, ..., c\}$
Objective	Find a transformation $f : R^n \to L_c$ such that $f(\mathbf{x^{(i)}}) = y^{(i)}$

Problem	**Dimensionality Reduction Problem**
Problem Description	Given a dataset, find encoding and decoding transformations to minimize reconstruction error[17]
Given	Dataset $X = [\mathbf{x^{(1)}}, \mathbf{x^{(2)}}, ..., \mathbf{x^{(N)}}] \in \mathbb{R}^{n\times N}$
Objective	Find encoding and decoding transformations$f$ s.t. $f : \mathbb{R}^n \to \mathbb{R}^m, g : R^m \to R^n, m < n$ and the reconstructed sample $g(f(\mathbf{x^{(i)}}))$ optimally fits the original sample $x^{(i)}$

Problem	**Clustering Problem**
Problem Description	Given a dataset, find a clustering method that minimizes intra-class variance and maximizes inter-class variance
Given	Dataset $X = [\mathbf{x^{(1)}}, \mathbf{x^{(2)}}, ..., \mathbf{x^{(N)}}] \in \mathbb{R}^{n\times N}$
Objective	Find a transformation $f : R^n \to L_c, L_c = \{c_i \vert i = 1, ..., c\}$ such that similar data points are grouped under the same class label

Among these four fundamental problems, the first two require the dataset to be divided into corresponding sample and label sets, and the model must represent the mapping from samples to labels. These problems are also called **supervised learning** problems. Conversely, the latter two problems, which do not involve mappings but instead represent intrinsic patterns within the dataset, are called **unsupervised learning** problems.

The basic premise of these tasks is the availability of a certain amount of data. Indeed, **machine learning is the science of discovering patterns from data itself**. Since the advent of the information age, the widespread use of sensors and computers in human society has generated vast amounts of data and the need to process it. Machine learning emerged as a scientific discipline in this context.

In the various domains of robotics, many problems can be addressed by collecting large amounts of corresponding data in advance or in real time. This provides a crucial foundation for applying machine learning/deep learning methods to solve these domain-specific problems.

18.1.2 Metrics for Machine Learning Tasks

Task metrics are quantitative criteria for evaluating the performance of ML models. For different tasks, different metrics have different meanings—some are better when larger, others when smaller—and cannot be generalized. Below, we introduce some common metrics for **classification and regression tasks**.

For **classification tasks**, we first introduce the concept of the **confusion matrix**. The confusion matrix is a result representation method that displays the classification results of an ML model and the similarities/differences with the data labels. It consists of two rows and two columns. For each sample, its classification result must appear in exactly one cell of the matrix.

	Actual Positive	Actual Negative
Predicted Positive	TP	FP
Predicted Negative	FN	TN

Specifically, whether for binary or multi-class classification problems, each sample has a true class label. For a given sample, the class corresponding to its label is called "positive," while all other classes are called "negative." When the ML model's prediction matches the true label, it is called "true"; otherwise, it is called "false." Thus, the prediction result for each sample must fall into one of four categories. We denote the total number of samples in each of the four cells as True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). We have:

$$TP + FP + TN + FN = N \tag{V.18.1}$$

Based on the confusion matrix, we first define the **Accuracy** metric, which represents the proportion of correctly predicted samples to the total number of samples:

$$\text{Accuracy} = \frac{TP + TN}{N} \tag{V.18.2}$$

Accuracy is the most intuitive metric for classification problems. However, when the data suffers from class imbalance, accuracy can be misleading. For example, in a dataset where 90% are negative samples and 10% are positive samples, even if the model predicts all samples as negative, its accuracy can still reach 90%. In such cases, accuracy does not align with subjective judgment.

To address this, we can define two more refined metrics: **Precision** and **Recall**. Precision measures the proportion of true positives among all samples predicted as positive:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{V.18.3}$$

Similarly, Recall measures the proportion of actual positive samples that are correctly predicted as positive by the model:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{V.18.4}$$

Precision emphasizes the accuracy of predictions, while Recall emphasizes the coverage of positive samples. Depending on the requirements of the task, we may prioritize one metric over the other. For example, in scenarios like spam detection where false positives are costly, Precision is more important. In contrast, for disease detection where false negatives are costly, Recall is more critical.

For scenarios where both metrics are important, we introduce the **F1 Score** to simultaneously reflect the roles of Precision and Recall. It is defined as:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{V.18.5}$$

The F1 Score is the harmonic mean of Precision and Recall and is particularly suitable for imbalanced class scenarios.

For **regression tasks**, a commonly used evaluation metric is the **Root Mean Squared Error (RMSE)**, which measures the average deviation between predicted and true values:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2} \tag{V.18.6}$$

Here, $y^{(i)}$ is the true value, and $\hat{y}^{(i)}$ is the DNN's predicted value. RMSE is more sensitive to larger errors, making it particularly suitable for scenarios requiring high prediction accuracy.

For unsupervised learning and generative tasks, there are also corresponding metrics. We will not delve into them here; interested readers can refer to other deep learning textbooks.

18.1.3 Machine Learning Loss Functions

In deep learning, the loss function serves as the bridge between the ML model and the task objective, acting as the optimization target for learnable parameters. Specifically, in supervised learning, the loss function is a function of the model output and the true data labels, measuring the difference between predicted and true values. It must be differentiable with respect to the model output. Different tasks employ different loss functions.

For classification tasks, the most commonly used loss is the **Cross-Entropy Loss**:

$$\mathcal{L}(p_{:,:}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_c^{(i)} \ln(p_c^{(i)}) \tag{V.18.7}$$

Here, N is the number of samples, C is the number of classes, $y_c^{(i)}$ indicates whether the i-th sample belongs to class c (typically using one-hot encoding), and $p_c^{(i)}$ is the neural network's output, representing the predicted probability that sample i belongs to class c. In classification tasks, the final layer of the network usually employs the softmax activation function to ensure $\sum_{c=1}^{C} p_c^{(i)} = 1$.

In traditional ML, we have the following conclusion: For a Logistic Regression model with a Gaussian prior distribution, the maximum likelihood estimation of its parameters corresponds to the cross-entropy loss[18]. For multi-layer DNNs (e.g., MLPs, discussed later), we can consider the final layer as a Softmax classifier, while the preceding $L-1$ layers act as a feature extractor. During training, both the data feature extraction method and the classification weights are learned. This embodies the idea of **feature learning**.

A drawback of cross-entropy loss is that it imposes heavy penalties on misclassified samples, potentially leading to gradient explosion. It is also sensitive to label noise, especially when labels are inaccurate, which can cause overfitting. In practice, regularization methods are often used to address such issues, as discussed in the next chapter.

The **Kullback-Leibler (KL) Divergence** is another commonly used loss for classification tasks, defined as:

$$\mathcal{L}_{KL}(P||Q) = \sum_{i} P(i) \ln \frac{P(i)}{Q(i)} \tag{V.18.8}$$

KL Divergence measures the difference between two probability distributions P and Q and is asymmetric. In classification tasks, P is typically the true label distribution, and Q is the DNN's output distribution. Similar to cross-entropy loss, the final layer of the DNN usually employs the softmax activation function when using KL Divergence.

The asymmetric nature of KL Divergence can make it unstable for certain tasks. Additionally, KL Divergence requires non-zero model outputs: when $Q(i) = 0$ and $P(i) > 0$, the KL Divergence tends to infinity, leading to numerical instability and gradient explosion. In practice, KL Divergence is often used for text generation tasks modeled as conditional classification. Large Language Models (LLMs) based on the Transformer architecture (i.e., autoregressive self-attention DNNs) typically use KL Divergence as the loss function.

For regression tasks, the **L2 Loss** is commonly used as the loss function. L2 Loss is similar to RMSE, differing only by a square root operation:

[18] Although the derivation is limited to binary classification, it also holds for multi-class Softmax regression.

$$\mathcal{L}(\hat{y}^{(:)}; y^{(:)}) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2 \tag{V.18.9}$$

Here, $y^{(i)}$ is the true label of the sample, and $\hat{y}^{(i)}$ is the model's prediction.

L2 Loss is simple and intuitive. It is more sensitive to large errors, helping the model correct deviations faster, and its derivative is easy to compute. Its drawbacks include high sensitivity to outliers, which may cause the model to overly focus on extreme samples, and poor performance when the error distribution is non-Gaussian.

In practical ML tasks, the form of the loss function is often tailored to specific requirements, but most general-purpose losses are based on the aforementioned types.

18.2 Deep Learning and DNNs

18.2.1 Basic Paradigm of DNNs

In the previous section, we introduced the fundamental problems in machine learning. We observed that these problems essentially involve discovering patterns or **mappings** f in a dataset.

A DNN is a multi-layer artificial neural network composed of artificial neurons, representing a nonlinear mapping with a large number of learnable parameters: $\text{DNN}(\cdot;\Theta) = f(\cdot)$. **The basic paradigm for using DNNs in machine learning is: Given a pre-collected dataset, construct a DNN with a specific architecture and learnable parameters; optimize these parameters using an optimization algorithm to minimize the loss function on the dataset, thereby obtaining the optimal parameters; the optimal parameters and network architecture together constitute the desired mapping f.**

Thus, we transform various machine learning problems into function approximation problems, which are then further transformed into parameter optimization problems. This idea of **mapping parameterization** is the core concept in deep learning.

From this basic paradigm, we can extract four key elements: **network architecture**, **learnable parameters**, **loss function**, and **optimization algorithm**.

As mentioned, a DNN is a multi-layer artificial neural network composed of neurons. Neurons are interconnected, with each neuron receiving certain inputs and producing an output. The output of a neuron is called its **activation value**. For a given neuron, different input activation values produce different output activation values according to a specific functional relationship, where the linear coefficients are called **(connection) weights**. Weights are the primary learnable parameters in DNNs.

The way neurons are connected is also called the **network architecture**, which is crucial. Different architectures are designed for different data and tasks. For example, the Multilayer Perceptron (MLP) discussed later in this chapter is one such architecture. In subsequent chapters, we will introduce more specialized and powerful architectures tailored to specific tasks and data.

The **loss function** is a scalar-valued function defined for the task, where higher values indicate poorer task performance on the dataset. In the previous section, we introduced basic loss functions for supervised learning tasks. In deep learning, the loss function takes the DNN's predictions as input, and these predictions depend on the network parameters. Thus, the loss function, combined with the network architecture, becomes a function of the network parameters. We design parameter update algorithms based on this relationship.

These parameter update algorithms, or **optimization algorithms**, primarily rely on the gradient (or partial derivatives) of the loss function with respect to the network parameters. Almost all DNNs use **gradient descent**-like methods to update parameters. In later sections, we will use MLPs as an example to introduce backpropagation and gradient descent algorithms for neural networks.

18.2.2 Characteristics of DNNs

Compared to traditional ML models, DNNs possess several important characteristics that distinguish them, forming the relatively independent subfield of deep learning.

Algorithm as Model In machine learning, a model generally refers to a way of modeling a distribution. In deep learning, a **model** refers to the **DNN nonlinear function mapping from inputs (and learnable parameters) to outputs**, i.e.,

$$\mathbf{y} = DNN(\mathbf{x}; \mathbf{W}) \tag{V.18.10}$$

In deep learning, this function is also called the **forward propagation algorithm**, **model architecture**, or **network architecture**[19]. DNN architectures are highly diverse and flexible. In fact, designing appropriate f (along with corresponding loss functions and optimization algorithms) for different tasks and data is the core technology of deep learning.

Moreover, the forms of DNN inputs and outputs are also highly diverse, including vectors, scalars, matrices, tensors, or combinations thereof, depending on the task. The number of learnable parameters in a DNN is predetermined, typically consisting of multiple matrices or tensors. The values of these parameters are also called **model weights**.

Design-Training-Inference The core application pipeline of DNNs consists of three stages: model design, model training, and model inference. The so-called **model design** involves defining the input and output formats for a well-defined problem and selecting the network architecture f based on design principles such as differentiability. The so-called **model training** refers to using a certain **optimization algorithm** on a specific **dataset** to minimize the value of a loss function until a set of optimal model weights is obtained. The so-called **model inference** is the process of performing forward propagation according to the predefined DNN function f, using the input and trained model weights to compute the output.

Large Models Since the structure of DNNs can involve nested nonlinear functions, their architectures can be extremely large and complex, containing **a massive number of learnable parameters**. When the parameter count reaches tens of millions or even billions, we refer to them as **large DNN models**. Our understanding of the essence of large models remains limited, but based on current research, we can posit that the parameters in large models implicitly "encode" some form of universal knowledge—about images, text, and data—far surpassing the capabilities of "small models." This exhibits **emergent properties**, making a certain level of "intelligence" possible.

Big Data The vast number of parameters in DNNs requires an equally large amount of data for training; otherwise, the model may suffer from **overfitting** (see Chapter 14 for details). In fact, one of the key reasons for the explosive growth of deep learning in the past decade is the advancement of internet and information technologies, enabling the production, collection, processing, and organization of datasets ranging from millions to billions of samples. Although some studies use enormous datasets, the deep learning community has gradually established a research paradigm of "open-source datasets + publicly benchmarked results," continuously motivating researchers to develop better methods.

Parallel Computing DNNs demand extensive data processing and parameter learning, creating a strong need for parallel processing and computing. Another major factor enabling the era of deep learning is the development of modern large-scale parallel computing hardware (e.g., GPUs), making the training of DNNs with tens of millions of parameters feasible. Currently (especially since 2023), deep learning has become the primary driving force behind advancements in high-performance computing. Leading parallel computing hardware manufacturers such as NVIDIA and AMD are striving to deliver higher-performance computing cards to meet the demands of the deep learning field.

Although the tasks and architectures of DNNs are highly diverse in practical applications, their foundational principles remain consistent. In the following sections of this chapter, we will use MLP as an example to introduce the two most fundamental algorithms in deep learning: **backpropagation** and **gradient descent**. In the next two chapters, we will cover more practical DNN training techniques and network architectures.

[19] In subsequent usage, we will not distinguish between these terms.

18.3 Multilayer Perceptron (MLP)

Among all DNN architectures, the multilayer perceptron is the simplest network structure. In this section, we will first introduce the composition of a multilayer perceptron, and then use it as an example to explain the backpropagation and gradient descent algorithms.

18.3.1 MLP Architecture

The **multilayer perceptron** (Multi-Layer Perceptron, MLP) is the simplest DNN structure. For simple tasks—where both inputs and outputs are straightforward and the mapping is not complex—MLP is also the most commonly used DNN. Among various practical DNNs, it serves as the prototype for many fundamental network modules.

An MLP consists of many **artificial neurons**. An artificial neuron is a multivariate function that mimics biological neurons. It takes a vector as input, performs an affine transformation on the vector, applies a nonlinear transformation to the result, and finally outputs a scalar. That is,

$$y(\mathbf{x}) = \phi(z) = \phi(\mathbf{w}^\top \mathbf{x} + b) \tag{V.18.11}$$

In an artificial neuron, w is called the linear weight, b is called the bias, and together they are referred to as **weights**. The result of the affine transformation, z, is called the **net input**, the nonlinear transformation ϕ is called the **activation function**, and the output of the neuron, y, is also known as the activation value.

In an MLP, multiple neurons simultaneously receive a set of vector inputs, forming a **neural network layer**. The outputs of the neurons in each layer form a new vector, called the **layer output** a. A single-layer neural network is called a **perceptron**; stacking and connecting multiple layers, where each layer takes the output of the previous layer as input, constitutes a multilayer perceptron.

For an L-layer MLP, let the input of the l-th layer be $a^{(l-1)}$ and its output be $a^{(l)}$, with the net input within the layer denoted as $z^{(l)}$. Let the linear weights of the layer be $W^{(l)}$ and the bias be $b^{(l)}$. Then,

$$\begin{aligned} z^{(l)} &= W^{(l)} a^{(l-1)} + b^{(l)} \\ a^{(l)} &= \phi(z^{(l)}) \end{aligned} \tag{V.18.12}$$

Here, assuming $a^{(l-1)} \in \mathbb{R}^{d_{l-1}}$ and $a^{(l)} \in \mathbb{R}^{d_l}$, then $z^{(l)} \in \mathbb{R}^{d_l}$, $W^{(l)} \in \mathbb{R}^{d_{l-1} \times d_l}$, and $b^{(l)} \in \mathbb{R}^{d_l}$. We define d_0 as the size of the input and d_L as the size of the output. In an MLP, the number of neurons in each layer, d_l, is also called the **width** of the layer, and the total number of layers, L, is called the **depth** of the MLP. The depth and width of an MLP are important structural hyperparameters.

For an MLP, the choice of activation function is also crucial. Typically, all layers of an MLP use the same activation function. Historically, the Sigmoid and Tanh functions were widely used, but currently, the most common activation function is the Rectified Linear Unit (ReLU) function (and its variants). The definitions of these activation functions are as follows:

Sigmoid Function

$$\phi(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{V.18.13}$$

Hyperbolic Tangent (Tanh) Function

$$\phi(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{V.18.14}$$

Rectified Linear Unit (ReLU)

$$\phi(x) = \max(0, x) \tag{V.18.15}$$

Among these three activation functions, the output range of the Sigmoid function is $(0, 1)$, and its gradient approaches 0 when x is large or small. The Tanh function has an output range of $(-1, 1)$ and offers the advantage of zero mean compared to Sigmoid, but it still suffers from the gradient approaching 0. This issue is referred to as **saturation**. The gradient of the activation function is critical and relates to the **vanishing gradient problem**, which we will discuss in detail in the next chapter.

In contrast, the forward and gradient computations of the ReLU function are very simple, and its positive half effectively mitigates the saturation issue. However, when the net input is negative, the gradient becomes 0, leading to the **dead neuron** problem. We will also explain this issue and its solutions in the next chapter.

Using ReLU as the activation function, we summarize the forward propagation algorithm for MLP as follows:

Algorithm 46: MLP Forward Propagation

Input: Network input x
Input: Layer parameters $W^{(1:L)}, b^{(1:L)}$
Parameter: Number of layers L, layer widths $d^{1:L}$
Output: Network output $\hat{y}$
$a^{(0)} \leftarrow x$
for $l \in 1, ..., L$ **do**
    $z^{(l)} \leftarrow W^{(l)} a^{(l-1)} + b^{(l)}$
    $a^{(l)} \leftarrow \max(0, z^{(l)})$
$\hat{y} \leftarrow a^{(L)}$

The corresponding Python code is shown below:

```python
class MLP:
    def __init__(self, L:int, ds:list):
        assert len(ds) == L+1
        self.params = {"W":{}, "b": {},}
        for l in range(1, L+1):
            self.params["W"][l] = np.random.randn(ds[l], ds[l-1]) * np.sqrt(2./ds[l-1])
            self.params["b"][l] = np.zeros([ds[l]])
        self.ds, self.L = ds, L
        self.a_s = []
    def forward(self, x):
        self.a_s, self.z_s = [x], [0] # self.zs[0] will never be visited
        for l in range(1, self.L+1):
            W_l, b_l = self.params["W"][l], self.params["b"][l]
            self.z_s.append(W_l @ self.a_s[l-1] + b_l)
            self.a_s.append(np.maximum(0, self.z_s[l]))
        return self.a_s[self.L]
```

18.3.2 MLP Learning Problem

The core problem in deep learning is the optimization of the large number of learnable parameters in neural networks. This process is generally referred to as "parameter learning." Similar to traditional ML models like logistic regression, RBM, and SBN, DNNs also learn parameters based on gradients. The difference is that in deep learning, a loss function is typically defined, and the goal is to minimize it rather than maximize it. Therefore, DNN learning is based on **gradient descent** rather than gradient ascent.

To perform gradient descent, the gradients must first be computed. Taking MLP as an example, we define the **gradient computation problem** and the **parameter learning problem**.

Problem	**MLP Gradient Computation**
Description	Given the MLP loss function, input, and parameters, compute the gradients for each parameter
Given	Loss function $\mathcal{L}(\hat{y}; y)$ Network input x Layer parameters $W^{(1:L)}, b^{(1:L)}$
Objective	Layer parameter gradients $\frac{\partial L}{\partial W^{(1:L)}}, \frac{\partial L}{\partial b^{(1:L)}}$

Problem	MLP Parameter Learning
Description	Given a dataset and MLP loss function, find the optimal parameters
Given	Loss function $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ Sample set $x^{(1:N)}$, label set $y^{(1:N)}$
Objective	Optimal parameters $W^{(1:L)}, b^{(1:L)}$

Note: Generally, the loss function is defined over the entire dataset. The gradient of the loss function with respect to a parameter involves all samples in the dataset. However, for most loss functions, the total loss is the average of the losses for individual samples. Therefore, computing the gradient over the entire dataset is equivalent to computing the gradients for each sample separately and then averaging them. Based on this, in the MLP gradient computation problem, we only consider the gradient computation for a single sample.

Gradient computation is the foundation of parameter learning. Next, we will first introduce the gradient computation method for MLP, namely the **backpropagation algorithm**. Then, based on this, we will introduce the simplest parameter learning method: gradient descent. In the next chapter, we will build on these methods to introduce numerous improved learning algorithms.

18.3.3 MLP Parameter Learning

First, let's consider the gradient computation problem for MLP. According to the chain rule, the (partial) derivative of the loss function $\mathcal{L}$ with respect to the parameters of the l-th layer is

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{\partial \mathcal{L}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial W^{(l)}}$$
$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \frac{\partial \mathcal{L}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial b^{(l)}}$$

Thus, the problem can be decomposed into two subproblems: how to compute the derivative of the loss function with respect to $a^{(l)}$, and how to compute the derivatives of $a^{(l)}$ with respect to $W^{(l)}$ and $b^{(l)}$.

For the former, based on Equation V.18.12 and the chain rule, we have

$$\frac{\partial \mathcal{L}}{\partial a^{(l)}} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial \hat{y}} \Pi_{m=L}^{l+1} \frac{\partial a^{(m)}}{\partial a^{(m-1)}}$$

Further, we have

$$\frac{\partial a^{(l)}}{\partial a^{(l-1)}} = \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial a^{(l-1)}}$$

For the latter, we have

$$\frac{\partial a^{(l)}}{\partial W^{(l)}} = \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial W^{(l)}}$$
$$\frac{\partial a^{(l)}}{\partial b^{(l)}} = \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial b^{(l)}}$$

Therefore, the MLP gradient computation problem can be further decomposed into five subproblems: computing the partial derivative of the loss function with respect to the network output $\frac{\partial \mathcal{L}}{\partial \hat{y}}$, computing the partial derivative of each layer's output with respect to its net input $\frac{\partial a^{(l)}}{\partial z^{(l)}}$, computing the partial derivative of the net input with respect to the layer input $\frac{\partial z^{(l)}}{\partial a^{(l-1)}}$, and computing the partial derivatives of the net input with respect to the network parameters $\frac{\partial z^{(l)}}{\partial W^{(l)}}$ and $\frac{\partial z^{(l)}}{\partial b^{(l)}}$.

For $\frac{\partial \mathcal{L}}{\partial \hat{y}}$, taking a regression problem as an example, we compute the partial derivative of the L2 loss, which has a very simple form (note: in this chapter, we uniformly use the numerator layout for derivatives):

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = (\hat{y} - y)^T \tag{V.18.16}$$

For $\frac{\partial a^{(l)}}{\partial z^{(l)}}$, this is essentially computing the derivative of the activation function. Assuming the MLP uses the ReLU activation function, we have

$$\frac{\partial a^{(l)}}{\partial z^{(l)}} = \text{diag}\left(\mathbf{1}[z_1^{(l)} > 0], \mathbf{1}[z_2^{(l)} > 0], ..., \mathbf{1}[z_{d_l}^{(l)} > 0]\right) \tag{V.18.17}$$

where $\mathbf{1}[\cdot]$ denotes the 0-1 function, which takes the value 1 when the condition in the brackets is true and 0 otherwise.

For $\frac{\partial z^{(l)}}{\partial a^{(l-1)}}$, this is simply the linear weight:

$$\frac{\partial z^{(l)}}{\partial a^{(l-1)}} = W^{(l)} \tag{V.18.18}$$

For $\frac{\partial z^{(l)}}{\partial b^{(l)}}$, this partial derivative is the identity matrix:

$$\frac{\partial z^{(l)}}{\partial b^{(l)}} = I_{d_l} \tag{V.18.19}$$

Finally, for $\frac{\partial z^{(l)}}{\partial W^{(l)}}$, we split it into derivatives with respect to different row vectors of the matrix. Assuming the matrix $W^{(l)}$ can be written in row vector form as

$$W^{(l)} = \begin{bmatrix} (w_1^{(l)})^T \\ (w_2^{(l)})^T \\ ... \\ (w_{d_l}^{(l)})^T \end{bmatrix}$$

then the partial derivative $\frac{\partial z^{(l)}}{\partial w_i^{(l)}}$ is a $d_{l-1} \times d_l$ matrix with only the i-th row being non-zero:

$$\frac{\partial z^{(l)}}{\partial w_i^{(l)}} = \begin{bmatrix} 0^T \\ ... \\ (a^{(l-1)})^T \\ ... \\ 0^T \end{bmatrix} \tag{V.18.20}$$

Based on the above derivations, using ReLU as the activation function, we summarize the **MLP backpropagation algorithm** as follows:

Algorithm	**MLP Backpropagation**
Problem type	MLP gradient computation
Given	Network input x, label y Layer parameters $W^{(1:L)}, b^{(1:L)}$ Loss function $\mathcal{L}(\hat{y}; y)$
Compute	Gradients $\frac{\partial L}{\partial W^{(1:L)}}, \frac{\partial L}{\partial b^{(1:L)}}$
Algorithm property	Analytical solution

Algorithm 47: MLP Backpropagation (MLP_BP)

Input: Network input x, labels y
Input: Layer parameters $W^{(1:L)}, b^{(1:L)}$
Input: Loss function $\mathcal{L}(\hat{y}; y)$
Output: Gradients $\frac{\partial L}{\partial W^{(1:L)}}, \frac{\partial L}{\partial b^{(1:L)}}$
$\mathcal{L}, \hat{y}, a^{(1:L)}, z^{(1:L)} \leftarrow \text{MLP}(x; W^{(1:L)}, b^{(1:L)})$
$G^T_{a^{(L)}} \leftarrow \frac{\partial \mathcal{L}}{\partial \hat{y}}$
for $l \in L, L-1, ..., 1$ **do**
  $\frac{\partial a^{(l)}}{\partial z^{(l)}} \leftarrow \text{diag}\left(\mathbf{1}[z_1^{(l)} > 0], \mathbf{1}[z_2^{(l)} > 0], ..., \mathbf{1}[z_{d_l}^{(l)} > 0]\right)$
  $\frac{\partial z^{(l)}}{\partial a^{(l-1)}} \leftarrow W^{(l)}$
  $G^T_{a^{(l-1)}} \leftarrow G^T_{a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial a^{(l-1)}}$
  $\frac{\partial z^{(l)}}{\partial b^{(l)}} \leftarrow I_{d_l}$
  $\frac{\partial \mathcal{L}}{\partial b^{(l)}} \leftarrow G^T_{a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial b^{(l)}}$
  for $i \in 1, ..., d_l$ **do**
    $\frac{\partial z^{(l)}}{\partial w_i^{(l)}} = \begin{bmatrix} 0 & ... & a^{(l)} & ... & 0 \end{bmatrix}^T$
    $\frac{\partial \mathcal{L}}{\partial w_i^{(l)}} \leftarrow G^T_{a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial w_i^{(l)}}$

The corresponding Python code is shown below:

```python
class MLP:
    def __init__(self, L:int, ds:list):
        ...
        self.zero_grad()
    def zero_grad(self):
        self.grads = {"W":{}, "b":{},}
        for l in range(1, self.L+1):
            self.grads["b"][l] = np.zeros_like(self.params["b"][l])
            self.grads["W"][l] = np.zeros_like(self.params["W"][l])
    def backward(self, dLdy):
        assert dLdy.shape == (self.ds[self.L],)
        G_as = [dLdy]
        for l in range(self.L, 0, -1):
            dadz = np.diag((self.z_s[l] > 0).astype(np.int32))
            dzda_ = self.params["W"][l]
            G_a_last = G_as[-1]
            G_as.append(G_a_last @ dadz @ dzda_)
            dzdb = np.eye(self.ds[l])
            self.grads["b"][l] += G_a_last @ dadz @ dzdb
            for i in range(self.ds[l]):
                dzdwi = np.zeros_like(self.params["W"][l])
                dzdwi[i, :] = self.a_s[l-1]
                self.grads["W"][l][i, :] += G_a_last @ dadz @ dzdwi
        pass
```

In Chapter 3, we introduced many nonlinear optimization algorithms. In fact, the parameter learning problem in deep learning is a typical nonlinear optimization problem. Since we have computed the gradients, it's natural to consider using the **gradient descent method** to iteratively optimize the network parameters of each layer in the MLP.

$$W^{(l)} \leftarrow W^{(l)} - \alpha \nabla_{W^{(l)}} \mathcal{L}$$
$$W^{(l)} \leftarrow b^{(l)} - \alpha \nabla_{b^{(l)}} \mathcal{L}$$
$$\nabla_{W^{(l)}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial W^{(l)}}\Big|_{1:N}^{T}$$
$$\nabla_{b^{(l)}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial b^{(l)}}\Big|_{1:N}^{T}$$

Here, α is the **learning rate**, an important hyperparameter in DNNs. Note that the gradient here is computed over the entire dataset, meaning we need to first compute the above gradients for each sample and then average them across the dataset. For simplicity, we use Θ to represent all parameters in the network (i.e., $W^{(1:L)}, b^{(1:L)}$), and denote:

$$G_{\Theta,i} = \frac{\partial \mathcal{L}}{\partial \Theta}\Big|_{i}^{T} \tag{V.18.21}$$

Then gradient descent can be written as:

$$\Theta \leftarrow \Theta - \alpha G_{\Theta,1:N} \tag{V.18.22}$$

The parameters Θ require initial values. We will detail the parameter initialization methods for MLPs in the next chapter. For now, we can initialize them with zero-mean random Gaussian noise using standard deviations σ_W, σ_b.

In summary, we present the **gradient descent optimization algorithm** for MLPs as follows:

Algorithm 48: GD Optimization

Input: Input samples $x^{(1:N)}$, label set $y^{(1:N)}$
Input: Loss function $\mathcal{L}(\hat{y}; y)$
Parameter: Random initialization parameters σ_W, σ_b
Parameter: Learning rate α, target loss L_{tar}
Output: Optimal parameters Θ

```
Θ ← random_init(σ_W, σ_b)
for k ∈ 1, ..., N_I do
    G_Θ, L ← 0_Θ, 0
    for i ∈ 1, ..., N do
        L_i, G_{Θ,i} ← MLP_BP(x^(i), y^(i); Θ)
        G_Θ ← G_Θ + G_{Θ,i}
        L ← L + L_i
    if L < L_tar then
        break
    Θ ← Θ − αG_Θ
```

Algorithm	**GD Optimization**
Problem Type	MLP Parameter Learning
Known	Loss function $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ Sample set $x^{(1:N)}$, Label set $y^{(1:N)}$
To Find	Optimal parameters $W^{(1:L)}, b^{(1:L)}$
Algorithm Property	Iterative Solution

The corresponding Python code is shown below:

```python
def GD_opt_MLP(L_func, dL_func, mlp:MLP, xs:array, ys:array,
↪ loss_target:float, N_I:int=100, alpha=1e-4):
    def GD_update(param, grad, alpha):
        return param - alpha * grad
    N, losses = xs.shape[0], []
    assert ys.shape[0] == N
    for k in range(N_I):
        mlp.zero_grad()
        train_loss = 0
        for i in tqdm(range(N)):
            y_est = mlp.forward(xs[i])
            train_loss += L_func(ys[i], y_est)
            dLdy = dL_func(ys[i], y_est)
            mlp.backward(dLdy)
        train_loss = train_loss / N
        if train_loss < loss_target:
            break
        for p, l in zip(mlp.params, range(1, mlp.L+1)):
            mlp.params[p][l] = GD_update(
                mlp.params[p][l], mlp.grads[p][l] / N, alpha
            )
        losses.append(train_loss)
    return losses
```

18.4 Deep Learning Theory

18.4.1 Universal Approximation Theorem

The Universal Approximation Theorem of neural networks is the cornerstone of deep learning, which describes that a DNN has the capability to approximate any function (under certain conditions), theoretically ensuring the usability of DNNs.

We assume X is a subset of Euclidean space $\mathbb{R}^n$, and the set of **continuous** functions with this set as the domain and $\mathbb{R}^m$ as the codomain is denoted as $C(X, \mathbb{R}^m)$. The activation function $\sigma \in C(\mathbb{R}, \mathbb{R})$, and for vector $x \in \mathbb{R}^n$, $(\sigma \circ x)_i = \sigma(x_i)$.

The content of the **Universal Approximation Theorem** is: if and only if σ is not a polynomial function, then $\forall n \in \mathbb{N}, m \in \mathbb{N}$, for any compact set $K \subseteq \mathbb{R}^n$, for any target function $f \in C(K, \mathbb{R}^m)$, and any $\varepsilon > 0$, there exists $k \in \mathbb{N}$, matrix $A \in \mathbb{R}^{k\times n}$, vector $b \in \mathbb{R}^k$, matrix $C \in \mathbb{R}^{m\times k}$, such that the following inequality holds:

$$\sup_{x\in K} \|f(x) - g(x)\| < \varepsilon \tag{V.18.23}$$

where,

$$g(x) = C \cdot (\sigma \circ (A \cdot x + b)) \tag{V.18.24}$$

This theorem was proved by George Cybenko and Kurt Hornik in a series of articles published between 1989-1991. Note that this theorem describes a single-layer neural network, which can approximate any continuous function in $C(X, \mathbb{R}^m)$ with arbitrary precision when the width k of the neural network can be arbitrarily large.

18.4.2 No Free Lunch

The **No Free Lunch Theorem** (NFL) is a classical theorem in the field of machine learning and serves as an important complement to the Universal Approximation Theorem introduced above. It states that although DNNs have powerful approximation capabilities, each DNN can only approximate the solution to a specific problem and cannot simultaneously approximate solutions to all problems. The essence of this theorem is:

there cannot exist a neural network that works effectively for all different problems. If two problems are fundamentally different, they cannot be approximated by the same DNN.

When defining problems, the No Free Lunch Theorem serves as an important reference principle. We need to carefully consider whether the DNN we are defining attempts to solve problems of fundamentally different natures.

18.4.3 Occam's Razor

The **Occam's Razor Principle** can be summarized in one sentence: **Entities should not be multiplied beyond necessity**.

Deep learning algorithms are black-box algorithms, and their performance is influenced by many conditions during the training process. These conditions mainly refer to various hyperparameters, including the model hyperparameters introduced above and the regularization-related hyperparameters to be discussed in the next chapter. The setting and adjustment of these hyperparameters are often empirical. We can usually only use validation set metrics as the basis for adjusting these parameters, which is essentially **result-oriented parameter tuning**. In this process, according to the Occam's Razor Principle, we should follow the following rule: if adjusting a parameter yields results that are not significantly different from not adjusting it, then maintain the original value of the hyperparameter.

Similarly, in the next chapter and subsequent chapters, we will introduce various improvement methods and techniques (tricks) for the basic algorithms. These methods and techniques are often modular, meaning they can be selectively added or not added to a specific DNN model for a particular problem. In such cases, our decision-making basis is primarily derived from experimental results (including our own experiments and those conducted by predecessors). We still make decisions according to the Occam's Razor Principle: if adding a trick yields results that are not significantly different from not adding it, then we do not adopt it.

19 Optimization and Regularization

19.1 Challenges in DNNs

In the previous chapter, we introduced the most basic MLP backpropagation/gradient descent algorithm, achieving iterative optimization of DNN parameters. However, DNNs trained in this manner face two key issues: **optimization difficulties** and **generalization difficulties**. These challenges are prevalent across various tasks using DNNs and significantly impact their performance. In this section, we will explain the principles behind these difficulties, and subsequent sections will introduce various optimization techniques.

19.1.1 Optimization Difficulties

As mentioned in the previous chapter, the parameter learning problem in deep learning is a nonlinear optimization problem. In fact, because DNNs typically have many layers, the loss function relative to the network parameters is highly non-convex and extremely high-dimensional, making optimization very challenging.

First is the **vanishing gradient problem**. Note that in the backpropagation algorithm, we have:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta^{(l)}} &= G_{a^{(L)}}^T \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial \theta^{(l)}} \\ G_{a^{(l-1)}}^T &= G_{a^{(l)}}^T \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial a^{(l-1)}} \end{aligned} \tag{V.19.1}$$

Here, $\frac{\partial a^{(l)}}{\partial z^{(l)}}$ is the partial derivative of the neuron activation value with respect to the net input, which is the derivative of the activation function itself. Note that the derivative of the Sigmoid function is:

$$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2} \tag{V.19.2}$$

And the derivative of the tanh function is:

$$\tanh'(x) = \frac{4}{(e^{x}+e^{-x})^2} \tag{V.19.3}$$

The derivatives of these activation functions quickly approach 0 as $x \to +\infty$. Therefore, when the output of a neuron in the neural network becomes too large (referred to as **output saturation**), the gradient of that layer's parameters with respect to the loss function becomes extremely small. Moreover, the gradients of layers preceding this layer will also become very small. Even if the gradient of a single neuron does not obviously approach 0, if the gradients of multiple layers are all less than 1, the gradients of earlier layers will become extremely small due to the multiplicative relationship in backpropagation, making gradient descent very difficult. This issue is called the **vanishing gradient problem**.

Conversely, the gradient of the tanh function can be greater than 1. Under multiplicative accumulation, multiple gradients greater than 1 will cause the gradients of earlier layers to become extremely large, leading to numerical instability or even overflow during optimization. This is known as the **exploding gradient problem**.

For a long time after the inception of neural networks, the vanishing and exploding gradient problems limited the scalability and depth of neural networks. The ReLU function largely resolves this issue, making it the preferred activation function for many multilayer neural networks.

Beyond vanishing and exploding gradients, DNN optimization also faces the **saddle points and flat regions** problem. Saddle points are points in high-dimensional functions where the first derivative is zero but are not local extrema. In high-dimensional spaces, saddle points are more common than local minima. Because gradients near saddle points are close to zero, optimization can stagnate when variables enter these regions. Flat regions are similar to saddle points, representing areas where gradients are close to zero. They can cause optimization to slow down significantly, making it difficult to reduce the loss.

In the following sections, we will introduce optimization algorithms to mitigate these challenges.

19.1.2 Generalization Difficulties

Apart from optimization difficulties, another major challenge for DNNs and deep learning is **generalization difficulties**. Generalization refers to the ability of a machine learning method to perform well on different data for the same task after being trained on a specific dataset.

DNNs have a large number of parameters. When training a DNN with optimization algorithms, as the number of training epochs increases, the loss function on the test set often exhibits a U-shaped curve. The descending part on the left is called **underfitting**, where the DNN has not yet fully captured the underlying patterns in the dataset. The ascending part on the right is called **overfitting**, where the DNN "memorizes" specialized features of the training data with its many parameters, losing focus on general patterns. The generalization problem primarily refers to the overfitting issue in DNNs.

Overfitting is closely related to model/dataset size. The size of the model, or the number of learnable parameters, determines its "potential representational capacity" or the "complexity" of the nonlinear functions it can express. If the task is simple but the model is complex, overfitting occurs. Similarly, if the task is complex but the dataset is small, the model will also overfit.

To better describe generalization difficulties, we can decompose the model's error on unseen data into three components: **bias, variance, and noise**:

$$\mathbb{E}_{\mathcal{D}}[(f(x;\mathcal{D}) - y)^2] = \text{Bias}^2 + \text{Variance} + \text{Noise} \tag{V.19.4}$$

Here, **bias** represents the error between the mapping learned by the DNN and the ideal nonlinear function; **variance** measures the variability in the model's output when trained on different datasets; and **noise** accounts for the inherent randomness in the additional data.

To address these generalization challenges, we aim to **reduce the model's variance** and improve its generalization ability through various design methods. This design philosophy is also known as **regularization**. In the following sections, we will also introduce model regularization techniques.

19.2 DNN Optimization Algorithms

In this section, we first introduce some algorithms to alleviate optimization difficulties. The next section will cover methods to address generalization challenges.

19.2.1 SGD and Mini-batch SGD

In the previous chapter, we introduced gradient descent (GD) for MLP parameter optimization. In GD, the gradient used for each parameter update is the average gradient over the entire dataset, which can be computationally expensive for large datasets.

In practice, we can also use the gradient from a single sample for each iteration. Although the gradient of a single sample may differ significantly from the full dataset's gradient, in expectation, this method's gradient still aligns with GD. Additionally, the randomness introduced by sampling may help the network escape saddle points or flat regions. This optimization strategy is called **Stochastic Gradient Descent** (SGD).

We summarize the **SGD optimization algorithm** for MLPs as follows:

Algorithm 49: SGD Optimization

Input: Dataset $x^{(1:N)}$, labels $y^{(1:N)}$, loss function $\mathcal{L}(\hat{y}; y)$
Parameter: Learning rate α, target loss L_{tar}
Parameter: Random initialization parameters σ_W, σ_b
Output: Optimal parameters Θ
$\Theta \leftarrow \text{random\_init}(\sigma_W, \sigma_b)$
for $k \in 1, ..., N_I$ **do**
  $i \leftarrow \text{random}(1, ..., N)$
  $\mathcal{L}_i, G_{\Theta,i} \leftarrow \text{MLP\_BP}(x^{(i)}, y^{(i)}; \Theta)$
  if $\mathcal{L}_i < L_{tar}$ **then**
    break
  $\Theta \leftarrow \Theta - \alpha G_{\Theta,i}$

The corresponding Python code is shown below:

Algorithm	SGD Optimization
Problem Type	MLP Parameter Learning
Given	Loss function $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ Dataset $x^{(1:N)}$, labels $y^{(1:N)}$
Objective	Optimal parameters $W^{(1:L)}, b^{(1:L)}$
Algorithm Property	Iterative solution

The corresponding Python code is shown below.

```python
def SGD_opt_MLP(L_func, dL_func, mlp:MLP, xs:array, ys:array, loss_target,
↪  N_I:int=100, alpha=1e-4):
    def GD_update(param, grad, alpha):
        return param - alpha * grad
    N, losses = xs.shape[0], []
    assert ys.shape[0] == N
    for k in range(N_I):
        mlp.zero_grad()
        train_loss = 0
        for i in tqdm(range(N)):
            y_est = mlp.forward(xs[i])
            train_loss += L_func(ys[i], y_est)
            dLdy = dL_func(ys[i], y_est)
            mlp.backward(dLdy)
            for p, l in zip(mlp.params, range(1, mlp.L+1)):
                mlp.params[p][l] = GD_update(
                    mlp.params[p][l], mlp.grads[p][l], alpha
                )
            mlp.zero_grad()
        train_loss = train_loss / N
        losses.append(train_loss)
        if train_loss < loss_target:
            break
    return losses
```

Compared to GD, SGD exhibits more significant oscillations in the optimization path due to larger variations in gradient update directions at each step. In practice, a compromise between the two is often used, namely the **Batch Stochastic Gradient Descent** (BSGD) method.

Specifically, the BSGD algorithm uses gradients computed from a small group of data (called a **batch**) rather than the full dataset average or a single random sample for each parameter update. Let $\mathcal{B}$ denote the index set of a batch with size b. The update rule is:

$$\Theta \leftarrow \Theta - \alpha \frac{1}{b} \sum_{i \in \mathcal{B}} G_{\Theta,i} \tag{V.19.5}$$

In fact, the constant $\frac{1}{b}$ can be absorbed into the hyperparameter α. We summarize the **BSGD optimization algorithm** for MLP as follows:

Algorithm	BSGD Optimization
Problem Type	MLP Parameter Learning
Given	Loss function $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ Sample set $x^{(1:N)}$, label set $y^{(1:N)}$
Objective	Optimal parameters $W^{(1:L)}, b^{(1:L)}$
Algorithm Property	Iterative solution

Algorithm 50: MLP-BSGD Optimization

Input: Sample set $x^{(1:N)}$, label set $y^{(1:N)}$, loss function $\mathcal{L}(\hat{y}; y)$
Parameter: Target loss L_{tar}, learning rate α, batch size b
Parameter: Random initialization parameters σ_W, σ_b
Output: Optimal parameters Θ

$\Theta \leftarrow \text{random\_init}(\sigma_W, \sigma_b)$
for $k \in 1, ..., N_I$ **do**
  $\mathcal{B} \leftarrow \text{random\_select}(1 : N, b)$
  $\mathcal{L}, G_\Theta \leftarrow 0, 0_\Theta$
  for $i \in \mathcal{B}$ **do**
    $\mathcal{L}_i, G_{\Theta,i} \leftarrow \text{MLP\_BP}(x^{(i)}, y^{(i)}; \Theta)$
    $G_\Theta \leftarrow G_\Theta + G_{\Theta,i}$
    $\mathcal{L} \leftarrow \mathcal{L} + \mathcal{L}_i$
  if $\mathcal{L}/b < L_{tar}$ **then**
    break
  $\Theta \leftarrow \Theta - \alpha G_\Theta$

The corresponding Python code is shown below:

```python
def BSGD_opt_MLP(L_func, dL_func, mlp:MLP, xs:array, ys:array, loss_target:float, batch_size:int,
↪  N_I:int=100, alpha=1e-4):
    def GD_update(param, grad, alpha):
        return param - alpha * grad
    N, N_b, losses = xs.shape[0], xs.shape[0] // batch_size, []
    assert ys.shape[0] == N
    for k in range(N_I):
        ids = np.arange(N)
        np.random.shuffle(ids)
        for b in tqdm(range(N_b)):
            mlp.zero_grad()
            b_ids = ids[b*batch_size:(b+1)*batch_size]
            b_train_loss, b_xs, b_ys  = 0, xs[b_ids], ys[b_ids]
            for i in range(batch_size):
                by_est = mlp.forward(b_xs[i])
                b_train_loss += L_func(b_ys[i], by_est)
                dLdy = dL_func(b_ys[i], by_est)
                mlp.backward(dLdy)
            b_train_loss = b_train_loss / batch_size
            if b_train_loss < loss_target:
                mlp.zero_grad()
                return
            for p, l in zip(mlp.params, range(1, mlp.L+1)):
                mlp.params[p][l] = GD_update(
                    mlp.params[p][l], mlp.grads[p][l] / batch_size, alpha
                )
            losses.append((k, b_train_loss))
    return losses
```

19.2.2 Learning Rate Decay

In the aforementioned GD and its variant algorithms, the learning rate α is a crucial hyperparameter. If the learning rate is too large, it may cause training oscillations or even divergence; if too small, it may lead to slow parameter convergence. However, a fixed learning rate may not be the optimal strategy.

Through extensive experimentation, researchers have developed **learning rate decay** methods. Learning rate decay refers to gradually reducing the learning rate as the number of dataset iterations increases—using a larger learning rate in the early stages of DNN training and a smaller one in later stages. This approach allows the DNN to rapidly decrease the loss initially and then finely search for minima within a certain range.

Specifically, the most commonly used learning rate decay strategies are **exponential decay** and **cosine decay**. For exponential decay, the learning rate α_k at the k-th iteration is:

$$\alpha_k = \alpha_0 \gamma^k \tag{V.19.6}$$

where $0 < \gamma < 1$ is a hyperparameter. For cosine decay, we have:

$$\alpha_k = \alpha_0 \cdot \frac{1}{2} \left(1 + \cos\left(\frac{k\pi}{N_I}\right)\right) \tag{V.19.7}$$

Taking exponential decay as an example, we summarize the **BSGD optimization algorithm with learning rate decay** for MLP as follows:

Algorithm	**Learning Rate Decay BSGD Optimization**
Problem Type	MLP Parameter Learning
Known	Loss function $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ Sample set $x^{(1:N)}$, label set $y^{(1:N)}$
Find	Optimal parameters $W^{(1:L)}, b^{(1:L)}$
Algorithm Property	Iterative Solution

Algorithm 51: Learning Rate Decay BSGD Optimization

Input: Sample set $x^{(1:N)}$, label set $y^{(1:N)}$, loss function $\mathcal{L}(\hat{y}; y)$
Parameter: Target loss L_{tar}, batch size b
Parameter: Initial learning rate α_0, learning rate decay rate γ
Parameter: Random initialization parameters σ_W, σ_b
Output: Optimal parameters Θ
$\Theta \leftarrow \text{random\_init}(\sigma_W, \sigma_b)$
for $k \in 1, ..., N_I$ **do**
  $\mathcal{B} \leftarrow \text{random\_select}(1 : N, b)$
  $L, G_\Theta \leftarrow 0, 0_\Theta$
  for $i \in \mathcal{B}$ **do**
    $\mathcal{L}_i, G_{\Theta,i} \leftarrow \text{MLP\_BP}(x^{(i)}, y^{(i)}; \Theta)$
    $G_\Theta \leftarrow G_\Theta + G_{\Theta,i}$
    $\mathcal{L} \leftarrow \mathcal{L} + \mathcal{L}_i$
  if $\mathcal{L}/b < L_{tar}$ **then**
    break
  $\alpha_k \leftarrow \gamma^k \alpha_0$
  $\Theta \leftarrow \Theta - \alpha_k G_\Theta$

19.2.3 RMSProp

The learning rate decay method introduced above can adjust the learning rate from large to small. However, such adjustment applies uniformly to all parameters. In reality, during each update, the gradient magnitudes vary across different parameters. Using the same learning rate may result in some parameters being updated significantly while others only slightly.

Ideally, we want the update step sizes for all parameters to be relatively similar. That is, parameters with larger gradients should have smaller learning rates, while those with smaller gradients should have larger learning rates - in other words, the learning rate should adapt automatically for different parameters. Specifically, for each $\theta \in \Theta$, we can use the following strategy:

$$\theta \leftarrow \theta - \frac{\alpha}{\sqrt{G_\theta^2 + \epsilon}} G_\Theta \tag{V.19.8}$$

However, due to the randomness in sample selection in SGD/BSGD, the gradient for the same parameter may vary significantly across iterations. We want the parameter update step sizes to be relatively smooth within a certain window. To achieve this, we replace $||G_\theta||^2$ with a historical moving average. We have:

$$\begin{aligned} v_{\theta,0} &= 0 \\ v_{\theta,k} &= \rho v_{\theta,k-1} + (1-\rho) G_\theta^2 \end{aligned} \tag{V.19.9}$$

where $\rho \in [0, 1]$ is the momentum coefficient, an important hyperparameter. ϵ is a small constant to prevent division by zero. That is:

$$\theta \leftarrow \theta - \frac{\alpha}{\sqrt{v_{\theta,k} + \epsilon}} G_\Theta \tag{V.19.10}$$

This gradient-adaptive learning rate optimization method is called **RMSProp**. We summarize the algorithm as follows:

Algorithm 52: RMSProp Optimization

Input: Sample set $x^{(1:N)}$, label set $y^{(1:N)}$, loss function $\mathcal{L}(\hat{y}; y)$
Parameter: Target loss L_{tar}, batch size b
Parameter: Initial learning rate α_0, learning rate decay rate γ, momentum coefficient ρ, small value ϵ
Parameter: Random initialization parameters σ_W, σ_b
Output: Optimal parameters Θ

$v_{\Theta,0} \leftarrow 0_\Theta$
$\Theta \leftarrow \text{random\_init}(\sigma_W, \sigma_b)$
for $k \in 1, ..., N_I$ **do**
  $\mathcal{B} \leftarrow \text{random\_select}(1:N, b)$
  $L, G_\Theta \leftarrow 0, 0_\Theta$
  for $i \in \mathcal{B}$ **do**
    $\mathcal{L}_i, G_{\Theta,i} \leftarrow \text{MLP\_BP}(x^{(i)}, y^{(i)}; \Theta)$
    $G_\Theta \leftarrow G_\Theta + G_{\Theta,i}$
    $\mathcal{L} \leftarrow \mathcal{L} + \mathcal{L}_i$
  if $\mathcal{L}/b < L_{tar}$ **then**
    break
  $\alpha_k \leftarrow \gamma^k \alpha_0$
  for $\theta \in \Theta$ **do**
    $v_{\theta,k} \leftarrow \rho v_{\theta,k-1} + (1-\rho) G_\theta^2$
    $\theta \leftarrow \theta - (\alpha_k / \sqrt{v_{\theta,k} + \epsilon}) G_\theta$

Algorithm	**RMSProp Optimization**
Problem Type	MLP Parameter Learning
Known	Loss function $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ Sample set $x^{(1:N)}$, label set $y^{(1:N)}$
Find	Optimal parameters $W^{(1:L)}, b^{(1:L)}$
Algorithm Property	Iterative Solution

The corresponding Python code is shown below

```
class MLP:
    def zero_grads(self):
        ... # same as original
```

```
        self.ms = deepcopy(self.grads)
        self.vs = deepcopy(self.grads)
    def backward_with_mv(self, dLdy, beta_1=0.9, beta_2=0.9):
        assert dLdy.shape == (self.ds[self.L],)
        G_as = [dLdy]
        for l in range(self.L, 0, -1):
            ... # same as backward()
            self.ms["b"][l] = beta_1 * self.ms["b"][l] + (1-beta_1) * self.grads["b"][l]
            self.ms["W"][l] = beta_1 * self.ms["W"][l] + (1-beta_1) * self.grads["W"][l]
            self.vs["b"][l] = beta_2 * self.vs["b"][l] + (1-beta_2) * self.grads["b"][l] ** 2
            self.vs["W"][l] = beta_2 * self.vs["W"][l] + (1-beta_2) * self.grads["W"][l] ** 2
        pass
def RMSProp_opt_MLP(L_func, dL_func, mlp:MLP, xs:array, ys:array, loss_target, batch_size,
    N_I:int=100, rho=0.9, alpha_0=1e-2, gamma=1-1e-2, epsilon=1e-6):
    def RMSProp_update(param, grad, v, alpha):
        return param - alpha * grad / np.sqrt(v + epsilon)
    ... # same as BSGD_opt_MLP()
    alpha = alpha_0
    for k in range(N_I):
        ... # same as BSGD_opt_MLP()
        for b in range(N_b):
            ... # same as BSGD_opt_MLP()
            for i in range(batch_size):
                y_est = mlp.forward(b_xs[i])
                b_train_loss += L_func(b_ys[i], y_est)
                dLdy = dL_func(b_ys[i], y_est)
                mlp.backward_with_mv(dLdy, 0, rho) # <-- different here
            ... # same as BSGD_opt_MLP()
            for p, l in zip(mlp.params, range(1, mlp.L+1)):
                mlp.param[p][l] = RMSProp_update(
                    mlp.param[p][l], mlp.grads[p][l], mlp.vs[p][l], alpha
                )
            pass
        alpha = alpha * gamma
    pass
```

19.2.4 Momentum Method

The historical weighted average technique mentioned above can also be directly applied to the gradients (of the loss with respect to parameters), and this method is called the **Momentum Method**. Specifically, we have the moving average estimate:

$$\begin{aligned} m_{\theta,0} &= 0 \\ m_{\theta,k} &= \beta m_{\theta,k-1} + (1-\beta) G_\theta \end{aligned} \tag{V.19.11}$$

And the parameter update directly uses the current moving average value:

$$\theta \leftarrow \theta - \alpha m_{\theta,k} \tag{V.19.12}$$

Thus, we summarize the **Momentum BSGD Optimization Algorithm** as follows:

Algorithm	**Momentum BSGD Optimization**
Problem Type	MLP Parameter Learning
Given	Loss function $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ Sample set $x^{(1:N)}$, label set $y^{(1:N)}$
Objective	Optimal parameters $W^{(1:L)}, b^{(1:L)}$
Algorithm Property	Iterative Solution

Algorithm 53: Momentum BSGD Optimization

Input: Sample set $x^{(1:N)}$, label set $y^{(1:N)}$, loss function $\mathcal{L}(\hat{y}; y)$
Parameter: Target loss L_{tar}, batch size b
Parameter: Initial learning rate α_0, learning rate decay rate γ, momentum coefficient β
Parameter: Random initialization parameters σ_W, σ_b
Output: Optimal parameters Θ

$m_{\Theta,0} \leftarrow 0_\Theta$
$\Theta \leftarrow \text{random\_init}(\sigma_W, \sigma_b)$
for $k \in 1, ..., N_I$ **do**
  $\mathcal{B} \leftarrow \text{random\_select}(1:N, b)$
  $L, G_\Theta \leftarrow 0, 0_\Theta$
  for $i \in \mathcal{B}$ **do**
    $\mathcal{L}_i, G_{\Theta,i} \leftarrow \text{MLP\_BP}(x^{(i)}, y^{(i)}; \Theta)$
    $G_\Theta \leftarrow G_\Theta + G_{\Theta,i}$
    $\mathcal{L} \leftarrow \mathcal{L} + \mathcal{L}_i$
  if $\mathcal{L}/b < L_{tar}$ **then**
    break
  $\alpha_k \leftarrow \gamma^k \alpha_0$
  for $\theta \in \Theta$ **do**
    $m_{\theta,k} \leftarrow \beta v_{\theta,k-1} + (1-\beta) G_\theta$
    $\theta \leftarrow \theta - \alpha_k m_{\theta,k}$

The corresponding Python code is shown below

```python
def Momentum_opt_MLP(L_func, dL_func, mlp:MLP, xs:array, ys:array, loss_target, batch_size,
    N_I:int=100, alpha_0=1e-2, gamma=1-1e-2):
    def Momentum_update(param, m, alpha):
        return param - alpha * m
    ... # same as BSGD_opt_MLP()
    alpha = alpha_0
    for k in range(N_I):
        ... # same as BSGD_opt_MLP()
        for b in range(N_b):
            ... # same as BSGD_opt_MLP()
            for i in range(batch_size):
                y_est = mlp.forward(b_xs[i])
                b_train_loss += L_func(b_ys[i], y_est)
                dLdy = dL_func(b_ys[i], y_est)
                mlp.backward_with_mv(dLdy, beta, 0) # <-- different here
            ... # same as BSGD_opt_MLP()
            for p, l in zip(mlp.params, range(1, mlp.L+1)):
                mlp.param[p][l] = Momentum_update(
                    mlp.param[p][l], mlp.ms[p][l], alpha
                )
            pass
        alpha = alpha * gamma
    pass
```

19.2.5 Adam Optimization

By combining the two historical weighted smoothing methods (Momentum and RMSProp) mentioned above, we obtain the Adam optimization algorithm. We rename ρ, β to β_2, β_1 respectively.

$$
\begin{aligned}
m_{\theta,0} &= 0 \\
v_{\theta,0} &= 0 \\
m_{\theta,k} &= \beta_1 m_{\theta,k-1} + (1-\beta_1)G_\theta \\
v_{\theta,k} &= \beta_2 v_{\theta,k-1} + (1-\beta_2)G_\theta^2
\end{aligned}
\tag{V.19.13}
$$

The parameter update is given by:

$$
\theta \leftarrow \theta - \alpha \frac{m_{\theta,k}}{v_{\theta,k}+\epsilon} \tag{V.19.14}
$$

Thus, we summarize the **Adam Optimization Algorithm** as follows:

Algorithm	**Adam Optimization**
Problem Type	MLP Parameter Learning
Given	Loss function $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ Sample set $x^{(1:N)}$, label set $y^{(1:N)}$
Find	Optimal parameters $W^{(1:L)}, b^{(1:L)}$
Algorithm Property	Iterative Solution

Algorithm 54: Adam Optimization

Input: Sample set $x^{(1:N)}$, label set $y^{(1:N)}$, loss function $\mathcal{L}(\hat{y}; y)$
Parameter: Target loss L_{tar}, batch size b
Parameter: Initial learning rate α_0, learning rate decay γ, momentum coefficient β
Parameter: Random initialization parameters σ_W, σ_b
Output: Optimal parameters Θ
$m_{\Theta,0}, v_{\Theta,0} \leftarrow 0_\Theta, 0_\Theta$
$\Theta \leftarrow$ random_init(σ_W, σ_b)
for $k \in 1, ..., N_I$ **do**
    $\mathcal{B} \leftarrow$ random_select$(1:N, b)$
    $L, G_\Theta \leftarrow 0, 0_\Theta$
    for $i \in \mathcal{B}$ **do**
        $\mathcal{L}_i, G_{\Theta,i} \leftarrow$ MLP_BP$(x^{(i)}, y^{(i)}; \Theta)$
        $G_\Theta \leftarrow G_\Theta + G_{\Theta,i}$
        $\mathcal{L} \leftarrow \mathcal{L} + \mathcal{L}_i$
    if $\mathcal{L}/b < L_{tar}$ **then**
        break
    $\alpha_k \leftarrow \gamma^k \alpha_0$
    for $\theta \in \Theta$ **do**
        $m_{\theta,k} \leftarrow \beta_1 m_{\theta,k-1} + (1-\beta_1)G_\theta$
        $v_{\theta,k} \leftarrow \beta_2 v_{\theta,k-1} + (1-\beta_2)G_\theta^2$
        $\theta \leftarrow \theta - \alpha \frac{m_{\theta,k}}{v_{\theta,k}+\epsilon}$

The corresponding Python code is shown below (here we provide the complete code)

```python
class MLP:
    def zero_grads(self):
        self.grads = {"W":{}, "b":{},}
        for l in range(1, L+1):
            self.grads["b"][l] = np.zeros_like(self.params["b"][l])
            self.grads["W"][l] = np.zeros_like(self.params["W"][l])
        self.ms = deepcopy(self.grads)
        self.vs = deepcopy(self.grads)
    def backward_with_mv(self, dLdy, beta_1=0.9, beta_2=0.9):
```

```
        assert dLdy.shape == (self.ds[self.L],)
        G_as = [dLdy]
        for l in range(self.L, 0, -1):
            dadz = np.diag((self.z_s[l] > 0).astype(np.int32))
            dzda_ = self.params["W"][l]
            G_a_last = G_as[-1]
            G_as.append(G_a_last @ dadz @ dzda_)
            dzdb = np.eye(self.ds[l])
            self.grads["b"][l] += G_a_last @ dadz @ dzdb
            for i in range(self.ds[l]):
                dzdwi = np.zeros_like(self.params["W"][l])
                dzdwi[i, :] = self.a_s[l-1]
                self.grads["W"][l][i, :] += G_a_last @ dadz @ dzdwi
            self.ms["b"][l] = beta_1 * self.ms["b"][l] + (1-beta_1) * self.grads["b"][l]
            self.ms["W"][l] = beta_1 * self.ms["W"][l] + (1-beta_1) * self.grads["W"][l]
            self.vs["b"][l] = beta_2 * self.vs["b"][l] + (1-beta_2) * self.grads["b"][l] ** 2
            self.vs["W"][l] = beta_2 * self.vs["W"][l] + (1-beta_2) * self.grads["W"][l] ** 2
        pass
def Adam_opt_MLP(L_func, dL_func, mlp:MLP, xs:array, ys:array, loss_target, batch_size,
    N_I:int=100, beta_1=0.9, beta_2=0.9, alpha_0=1e-2, gamma=1-1e-2, epsilon=1e-6):
    def Adam_update(param, m, v, alpha):
        return param - alpha * m / np.sqrt(v + epsilon) # <-- different here
    N, N_b = xs.shape[0], xs.shape[0] // batch_size
    alpha, losses = alpha_0, [] # <-- different here
    assert ys.shape[0] == N
    for k in range(N_I):
        ids = np.arange(N)
        np.random.shuffle(ids)
        for b in tqdm(range(N_b)):
            mlp.zero_grad()
            b_ids = ids[b*batch_size:(b+1)*batch_size]
            b_train_loss, b_xs, b_ys  = 0, xs[b_ids], ys[b_ids]
            for i in range(batch_size):
                y_est = mlp.forward(b_xs[i])
                b_train_loss += L_func(b_ys[i], y_est)
                dLdy = dL_func(b_ys[i], y_est)
                mlp.backward_with_mv(dLdy, beta_1, beta_2) # <-- different here
            b_train_loss = b_train_loss / batch_size
            if b_train_loss < loss_target:
                mlp.zero_grad()
                return
            for p, l in zip(mlp.params, range(1, mlp.L+1)):
                mlp.params[p][l] = Adam_update( # <-- different here
                    mlp.params[p][l], mlp.ms[p][l], mlp.vs[p][l], alpha
                )
            losses.append((k, b_train_loss))
        alpha = alpha * gamma # <-- different here
    pass
```

19.3 Regularization Techniques

In the previous section, we introduced various optimization methods for addressing optimization difficulties. For generalization challenges, we also have various regularization techniques.

19.3.1 Regularization Loss

[Main content: L1 loss, L2 loss]

[This section will be updated in subsequent versions, stay tuned]

19.3.2 Batch Normalization (BN)

Batch Normalization (BN) is a normalization method based on BSGD-type approaches, performing normalization within layers.

Specifically, for each batch $\mathcal{B}$, we calculate the mean $\mu^{(l)}$ and variance $(\sigma^2)^{(l)}$ of the layer inputs $a^{(l-1)}$ (assuming independence between neurons in the layer, ignoring covariance):

$$\begin{aligned} \mu^{(l)} &= \frac{1}{b}\sum_{i=1}^{b} a_i^{(l-1)} \\ (\sigma^2)^{(l)} &= \frac{1}{b}\sum_{i=1}^{b}(a_i^{(l-1)} - \mu^{(l)}) \odot (a_i^{(l-1)} - \mu^{(l)}) \end{aligned} \tag{V.19.15}$$

Thus the normalized input becomes:

$$z_i^{(l)} = \frac{a_i^{(l)} - \mu^{(l)}}{\sqrt{(\sigma^2)^{(l)} + \epsilon}} \tag{V.19.16}$$

At this point, since we've removed bias through normalization, the bias term $b^{(l)}$ in the network layer becomes effectively redundant. Therefore, we can omit the bias parameter b during forward propagation (using ReLU as activation function):

$$\begin{aligned} z^{(l)} &= (a^{(l-1)} - \mu^{(l)})/(\sqrt{(\sigma^2)^{(l)} + \epsilon}) \\ \hat{z}^{(l)} &= W^{(l)} z^{(l)} \\ a^{(l)} &= \max(0, \hat{z}^{(l)}) \end{aligned} \tag{V.19.17}$$

We can now summarize the **MLP forward propagation algorithm with BN** as follows:

Algorithm 55: MLP-BN Forward Propagation (MLP_BN)

Input: Batch input $x^{(1:b)}$, layer parameters $W^{(1:L)}$
Parameter: Number of layers L, layer widths $d^{(1:L)}$
Output: Batch network output $\hat{y}^{(1:b)}$
Output: Layer normalization information $\mu^{(1:L)}, \sigma^{(1:L)}$
for $i \in 1, ..., b$ **do**
  $a_i^{(0)} \leftarrow x^{(i)}$
for $l \in 1, ..., L$ **do**
  $\mu^{(l)} \leftarrow \sum_{i=1}^{b} z_i^{(l)}$
  $(\sigma^2)^{(l)} \leftarrow \sum_{i=1}^{b} (z_i^{(l)} - \mu^{(l)}) \odot (z_i^{(l)} - \mu^{(l)})$
  for $i \in 1, ..., b$ **do**
    $z_i^{(l)} \leftarrow (a_i^{(l-1)} - \mu^{(l)}) / (\sqrt{(\sigma^2)^{(l)} + \epsilon})$
    $\hat{z}_i^{(l)} \leftarrow W^{(l)} z_i^{(l)}$
    $a_i^{(l)} \leftarrow \max(0, \hat{z}_i^{(l)})$
for $i \in 1, ..., b$ **do**
  $\hat{y}^{(i)} \leftarrow a_i^{(L)}$

The corresponding Python code is shown below:

```
class MLP_BN:
    def __init__(self, L:int, ds:list, batch_size:int, momentum=0.99):
        assert len(ds) == L+1
        assert 0 < momentum and momentum < 1
        self.params = {"W":{}, "mu":{}, "sig":{}, "mu_all":{}, "sig_all":{}}
        for l in range(1, L+1):
            self.params["W"][l] = 0.01 * (np.random.rand(ds[l], ds[l-1]) -
            ↪  0.5)
            self.params["mu_all"][l] = np.zeros([ds[l]])
            self.params["sig_all"][l] = np.zeros([ds[l]])
        self.ds, self.L, self.b_size = ds, L, batch_size
        self.ba_s, self.epsilon, self.momentum = [], 1e-6, momentum
    def forward(self, bx, is_train=True):
        assert bx.shape == (self.b_size, self.ds[0])
        self.ba_s, self.bz_s = [bx], [0] # self.bz_s[0] will never be visited
        mo = self.momentum
        for l in range(1, self.L+1):
            W_l = self.params["W"][l]
            bz_l = np.einsum("ji,bi->bj", W_l, self.ba_s[l-1])
            self.bz_s.append(bz_l)
            if is_train:
                mu_l = np.mean(bz_l, axis=0)
                sig_l = np.var(bz_l, axis=0, ddof=1)
                self.params["mu"][l], self.params["sig"][l] = mu_l, sig_l
                self.params["mu_all"][l] = mo * self.params["mu_all"][l] +
                ↪  (1-mo) * mu_l
                self.params["sig_all"][l] = mo * self.params["sig_all"][l] +
                ↪  (1-mo) * sig_l
            else:
                mu_l = self.params["mu_all"][l]
                sig_l = self.params["sig_all"][l]
            bzhat_l = (bz_l - mu_l) / np.sqrt(sig_l + self.epsilon)
            self.ba_s.append(np.maximum(0, bzhat_l))
        return self.ba_s[self.L]
```

Since an additional linear transformation is introduced in neurons, the backpropagation algorithm also changes. We have:

$$\frac{d\hat{z}_i}{dz_j} = \frac{\delta_{ij} - 1/b}{\sqrt{\sigma^2 + \epsilon}} I - \frac{(z_i - \mu)(z_j - \mu)^T}{b\sqrt{\sigma^2 + \epsilon}^3} \quad \text{(V.19.18)}$$

Note: The MLP no longer requires linear bias terms, nor does it need to compute their derivatives, as the linear bias is completely replaced by μ. Its derivative is already included in $\frac{d\hat{z}}{dz}$.

Algorithm	**MLP-BN Backpropagation**
Problem Type	MLP Gradient Computation
Given	Batch input $x^{(1:b)}$, labels $y^{(1:b)}$ Layer parameters $W^{(1:L)}, b^{(1:L)}$ Loss function $\mathcal{L}(\hat{y}; y)$
Find	Gradient $\frac{\partial L}{\partial W^{(1:L)}}$
Algorithm Property	Analytical Solution

Algorithm 56: MLP-BN Backpropagation (MLP_BP_BN)

Input: Batch input $x^{(1:b)}$, labels $y^{(1:b)}$
Input: Layer parameters $W^{(1:L)}$, loss function $\mathcal{L}(\hat{y}; y)$
Output: Gradient $\frac{\partial L}{\partial W^{(1:L)}}$
$\hat{y}^{(1:b)}, a^{(:)}_{1:b}, z^{(:)}_{1:b}, \mu^{(:)}, \sigma^{(:)} \leftarrow \text{MLP\_BN}(x^{(1:b)}; W^{(:)})$
$G^T_{a^{(L)}} \leftarrow \frac{\partial \mathcal{L}}{\partial \hat{y}}(\hat{y}^{(i)}, y^{(i)})$
for $l \in L, L-1, ..., 1$ **do**
  for $i \in 1, ..., b$ **do**
    $\left.\frac{\partial a^{(l)}}{\partial \hat{z}^{(l)}}\right|_i \leftarrow \text{diag}\left(\mathbf{1}[z_i^{(l)} > 0]\right)$
    $\frac{\partial z_i^{(l)}}{\partial a^{(l-1)}{}_i} \leftarrow W^{(l)}$
    for $j \in 1, ..., b$ **do**
      $\frac{d\hat{z}_i}{dz_j} = \frac{\delta_{ij} - 1/b}{\sqrt{\sigma^2+\epsilon}} I - \frac{(z_i-\mu)(z_j-\mu)^T}{b\sqrt{\sigma^2+\epsilon}^3}$
      $G^T_{a_i^{(l-1)}} \leftarrow \sum_j G^T_{a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial \hat{z}_j^{(l)}} \frac{\partial \hat{z}_j^{(l)}}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial a_i^{(l-1)}}$
  $\frac{\partial \mathcal{L}}{\partial W^{(l)}} \leftarrow \sum_i \sum_j G^T_{a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial \hat{z}_j^{(l)}} \frac{\partial \hat{z}_j^{(l)}}{\partial z_i^{(l)}} a_i^{(l-1)}$

The corresponding Python code is shown below:

```python
class MLP_BN:
    def zero_grad(self):
        self.grads = {"W":{}}
        for l in range(1, self.L+1):
            self.grads["W"][l] = np.zeros_like(self.params["W"][l])
    def backward(self, bdLdy):
        assert bdLdy.shape == (self.b_size, self.ds[self.L],) # already mean at batch dimention
        bG_as = [bdLdy]
        for l in range(self.L, 0, -1):
            # calc bG_zh
            mu_l, sig_l, dl = self.params["mu"][l], self.params["sig"][l], self.ds[l]
            tmp = (self.ba_s[l] > 0).astype(np.float32) # (b, dl)
            bG_zh = bG_as[-1] * tmp  # ReLU gradient, (b, dl)
            # calc bG_z
            scale = 1 / np.sqrt(sig_l + self.epsilon) # (dl, )
            bG_z_1 = bG_zh * scale # (b, dl)
            bG_z_2 = - np.sum(bG_z_1 / self.b_size, axis=0)[None, :] # (1, dl)
```

```
            tmp = - np.sum((bG_zh * (self.bz_s[l] - mu_l)), axis=0) * (scale**3) # (dl,)
            bG_z_3 = (self.bz_s[l] - mu_l) / self.b_size * tmp  # (b, dl)
            bG_z = bG_z_1 + bG_z_2 + bG_z_3
            # calc bG_a, bG_W
            bG_a = bG_z @ self.params["W"][l] # (b, d^{l-1})
            bG_as.append(bG_a)
            self.grads["W"][l] += bG_z.T @ self.ba_s[l-1] # (dl, d^{l-1})
    pass
```

As can be seen, after introducing BN, both the forward propagation and backpropagation of MLP become more complex. However, this method can effectively prevent gradient explosion and vanishing gradient problems. In practice, we typically combine BN with batch-based optimization methods such as BSGD, RMSProp, Momentum, and Adam, which won't be elaborated here.

19.3.3 Dropout Method

Dropout, MLP inference with Dropout, BSGD optimization algorithm with Dropout

19.3.4 Data Regularization

Label smoothing

Data normalization

19.3.5 Other Regularization Techniques

Parameter initialization

Part VI
Reinforcement Learning Methods

20 Foundations of Reinforcement Learning

In the previous section, we introduced fundamental deep learning methods. Deep learning is a powerful set of tools that can address many intelligent problems in robotics using large-scale data. However, most problems in robotics involve control or planning, which are decision-making processes in interaction with the environment. Deep learning itself does not include interaction with the environment.

Reinforcement Learning (RL) is another branch of machine learning that specifically studies how to learn from data generated through interaction with the environment to solve certain control problems. Since 2012, various **Deep Reinforcement Learning** (DRL) methods have achieved groundbreaking progress in many fields, including robotics. In this chapter, we introduce the basic concepts of reinforcement learning; subsequent chapters will cover key algorithms in the field.

20.1 State Space

A state is a snapshot of a dynamic system. In earlier chapters, we have discussed states in optimization, system control, filtering, and other areas. In reinforcement learning, we also use states to describe systems and focus on how states transition. If the state transition follows the rule that the current state depends only on the previous state and not on earlier states, we say it has the **Markov property**. Such a stochastic process is called a Markov process.

Reward models the "purpose" of state transitions. In reinforcement learning, the specific task objective is quantified as a positive or negative number, called a reward. A Markov process with rewards is called a **Markov Reward Process** (MRP).

Before focusing on Markov Reward Processes, we need to establish the concept of a **state space**. The state space is the space composed of all possible states in a Markov process, denoted as $\mathcal{S}$. The state space can be discrete or continuous, finite (discrete case) or infinite. The larger the state space, the more complex the Markov Reward Process, and the more difficult it is to handle.

Some of the problems we discuss later are based on **discrete finite state spaces**. This is because, in such spaces, the total number of states is finite. For functions defined on the state space, we can always explicitly list the value of each state and perform subsequent analysis and decision-making accordingly. Such algorithms are called tabular methods and are discussed in Chapter 3.

However, in many real-world scenarios, we face infinite state spaces, especially **continuous state spaces**. Although they can be discretized and treated as discrete state spaces, this approach introduces errors. Moreover, discretizing high-dimensional state spaces leads to the severe **curse of dimensionality**. Suppose a continuous state has d dimensions, and each dimension is discretized into N intervals. The total number of states then becomes N^d. As N and d increase, computational resources quickly become insufficient. This is also known as the **curse of dimensionality**. In such cases, we urgently need methods to handle continuous states directly, such as the **policy gradient methods** introduced in Chapters 4-5.

20.2 MRP and Value

Before introducing these methods, let us start with a **discrete finite state space** and focus on a specific Markov Reward Process.

First, we need an initial state. Typically, we have an initial state distribution $P(S_0)$, from which we sample to obtain the initial state S_0. Then, we sample from the state transition distribution $P(S_{t+1}|S_t)$ to get the next state. Based on the reward distribution $P_R(R_{t+1}|S_t)$, we receive a reward R_{t+1}, which may be positive or negative. We then return to the second step, continuously obtaining states and rewards until a predefined stopping condition is met. The stopping condition could be a terminal state, a step limit, a maximum reward threshold, or other criteria.

Here, $P_R(R_{t+1}|S_t)$ indicates that the reward distribution depends on the state. Later, however, we more commonly use the expected reward in a given state, denoted as $\bar{R}(s)$:

$$\bar{R}(s) = \sum_{r \in \mathcal{R}} r P_R(r|s) \tag{VI.20.1}$$

Note that in the described MRP, there is only one "actor" in the process, which we might call "God" or the "**environment**." Only this "actor's" "move" determines the next state or reward. Therefore, to maximize rewards, we need to study the underlying patterns.

For now, let us set aside the complexity of infinite state spaces and focus solely on finite state spaces. Rewards are random variables, making them difficult to study. A naive idea is to study the expected reward for each state in the space.

Calculating the expected reward is a good idea. With the help of the law of large numbers, as long as enough data is collected, the average can approximate the expectation. However, there is an easily overlooked fact—sometimes a state that does not yield much reward immediately may not be a "bad" state, and vice versa.

There are many examples of this in life. Often, we do things without immediate positive feedback, but we persist because we believe that continuing will likely yield good feedback or rewards in the future. This "delayed gratification" mindset is a hallmark of higher-level human intelligence, allowing us to break free from basic desires and pursue higher values.

In MRP, our evaluation of a state is not limited to the reward it can yield immediately but also includes the rewards from states it tends to transition to—in other words, the rewards obtainable in the future of the MRP. This combined evaluation of present and future rewards is called "**value**." Value is a measure of the expected rewards from the current and future states.

To quantify value, we introduce a **discount factor**, which "discounts" future rewards to the present. The discount factor is a value γ in $[0, 1]$, representing the proportion by which future rewards are discounted for each additional step. Summing all future discounted rewards gives a global quantitative evaluation of the current state, called the **discounted return** (or cumulative return). Its formula is:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots \tag{VI.20.2}$$

The discounted return is a random variable that cannot be computed until all future steps are completed. The expectation of this random variable is the **MRP state value (function)**:

$$V(s) = E[G_t | S_t = s] \tag{VI.20.3}$$

As mentioned earlier, the state value describes not only the immediate reward from the current state but also the rewards from states it is likely to transition to. States are connected through certain probabilities. We call such state transitions "state transitions" and describe them using a transition distribution matrix P, called the **state transition matrix**. By the properties of probability, each row vector of this matrix has an L2 norm of 1.

Here, we must emphasize an important detail: **Over which random variables is the expectation in the state value function taken?** Without understanding this, it is impossible to fully grasp subsequent derivations involving the state value function. The precise answer is: **The expectation is taken over all future sequences of states and rewards.** This is because the state value function is the expectation of the discounted return G_t, which is composed of all future rewards R_{t+i}, and these rewards are determined by future states R_{t+i}. Each has its own distribution, $P(s'|s)$ and $P_R(r|s)$. Thus, this expectation is taken over all future states and rewards in the Markov chain sense.

20.3 Value Estimation in MRP

How, then, do we compute the state value in a given environment? This is the **value estimation problem in MRP**. We summarize this problem in clearer mathematical terms in the table below.

Problem	**MRP Value Estimation [Model-Dependent]**
Problem Description	Given the MRP environment model, compute the state value for each state
Given	State transition matrix $P(s'|s)$ and reward function $\bar{R}(s)$
Objective	State value $V(s), s \in \mathcal{S}$

Here, "**model**" refers to the MRP's state transition matrix P and reward function $\bar{R}(s)$. In later problems, we will see that whether the model is known is a critical factor. Most mainstream reinforcement learning algorithms today do not require or rely on a model.

Based on the definition of MRP state value in Equation VI.20.3, we can expand the state value in terms of one-step state transitions:

$$\begin{aligned}
V(s) &= E[G_t|S_t = s] \\
&= E[R_t + \gamma R_(t+1) + \gamma^2 R_(t+2) + \ldots |S_t = s] \\
&= E[R_t|S_t = s] + \gamma E[R_(t+1) + \gamma R_(t+2) + \gamma^2 R_(t+3) + \ldots |S_t = s] \\
&= \bar{R}(s) + \gamma E[G_{t+1}|S_t = s] \\
&= \bar{R}(s) + \gamma \mathbb{E}_{p(s'|s)}[E[G_{t+1}|S_{t+1} = s']] \\
&= \bar{R}(s) + \gamma \mathbb{E}_{p(s'|s)}[V(s')] \\
&= \bar{R}(s) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s)V(s')
\end{aligned}$$

Thus, we have:

$$V(s) = \bar{R}(s) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s)V(s') \tag{VI.20.4}$$

This equation is also known as the **Bellman equation for MRP**.

When the model (state transition matrix/reward function) is known, solving the state value function using the Bellman equation for MRP is straightforward. For convenience, we first represent the state value as a vector $\mathbf{V}$, with dimensionality equal to the number of states in the state space. Similarly, we represent the expected reward function as a vector $\mathbf{R}$. Combining these with the state transition matrix $\mathbf{P}$, we can write:

$$\mathbf{V} = \mathbf{R} + \gamma \mathbf{P}\mathbf{V} \tag{VI.20.5}$$

Therefore, given the model, we can derive the analytical solution of $\mathbf{V}$ through simple calculations:

$$\mathbf{V} = (I - \gamma \mathbf{P})^{-1}\mathbf{R} \tag{VI.20.6}$$

The analytical solution algorithm is summarized as follows.

Algorithm	**MRP-Value Analytical Solution**
Problem Type	MRP Value Estimation [Model-Dependent]
Given	State transition matrix $P(s'\|s)$ and reward function $\bar{R}(s)$
Objective	State value $V(s), s \in \mathcal{S}$
Algorithm Property	Model-based, Tabular

Algorithm 57: MRP-Value Analytical Solution

Input: State transition matrix $P(s'|s)$ and reward function $\bar{R}(s)$

Output: State value $V(s), s \in \mathcal{S}$

$\mathbf{V} \leftarrow (I - \gamma \mathbf{P})^{-1}\mathbf{R}$

While the analytical solution is very convenient, it is often difficult to obtain in many cases. In the previous section, we discussed the issue of dimensionality explosion—the larger the state space dimension, the larger the size of the $\mathbf{P}$ matrix, and the more complex it becomes to compute the inverse matrix for the analytical solution. In such cases, we may opt not to solve for all state values at once but instead use the Bellman equation for iterative solutions.

Dynamic Programming (DP) is one such method. As a broad class of algorithmic construction principles, dynamic programming refers to recursively decomposing a large problem into smaller subproblems based on the problem's inherent characteristics, then solving these simpler subproblems step by step to ultimately

solve the original problem. The fundamental principle of all dynamic programming methods is the Bellman optimality principle.

In Part III of this book, we introduced the dynamic programming approach for optimal control problems. Through the Bellman optimality principle, we decompose the optimal control problem into smaller subproblems, enabling iterative solutions for the optimal control policy.

The core of dynamic programming lies in subproblem decomposition. For MDP value estimation, the Bellman equation (Equation VI.20.4) provides a way to decompose the problem. However, this decomposition differs from other dynamic programming solutions in that the subproblems do not reduce in size, and the decomposed subproblems remain interdependent through the P matrix.

Thus, in MDP value estimation, we cannot solve the problem through finite decomposition. Instead, our approach is iterative estimation—iteratively updating the value function table. Starting with random initialization, after sufficiently many iterations, the value function estimate converges to the true value. This convergence is guaranteed by theoretical proofs.

For broader applications of dynamic programming, readers may refer to the extended content in Chapter I.7 of this book.

Algorithm	**MRP-DP Value Estimation**
Problem Type	MRP Value Estimation [Model-Dependent]
Given	State transition matrix $P(s'\|s)$ and reward function $\bar{R}(s)$
Objective	State value $V(s), s \in \mathcal{S}$
Algorithm Property	Model-based, Tabular, Bootstrapping

Algorithm 58: MRP-DP Value Estimation

Input: State transition matrix $P(s'|s)$ and reward function $\bar{R}(s)$

Output: State value $V(s), s \in \mathcal{S}$

$\hat{V}(s) \leftarrow random([V_{min}, V_{max}]), \forall s \in \mathcal{S}$

for $t \in 1, \ldots, T$ **do**

  for $s \in \mathcal{S}$ **do**

    $\hat{V}(s) \leftarrow \bar{R}(s) + \sum_{s' \in \mathcal{S}} p(s'|s)\hat{V}(s')$

In practical applications of dynamic programming for MRP value estimation, the stopping condition for the outer loop is often not a fixed number of iterations T but rather the difference between value functions in consecutive iterations $||\mathbf{V}_{t+1} - \mathbf{V}_t||_2$. When this difference falls below a given error threshold ϵ, the iteration is considered converged.

The dynamic programming method avoids the drawback of large matrix inversion in the analytical solution, achieving value estimation through iteration. However, this method still relies on the model, i.e., the state transition matrix and reward function. In many cases, we have little information about the environment. Under such circumstances, can we infer the value function table from multiple experiments?

Following this idea, we arrive at the Monte Carlo value estimation method. Before proceeding, however, we need to clarify the problem we are currently addressing: model-free MRP value estimation.

Problem	**MRP Value Estimation [Model-Free]**
Problem Description	Estimate state values for an MRP with unknown environment model
Given	MRP environment $\mathcal{E}_R$
Objective	State value $V(s), s \in \mathcal{S}$

Here, we introduce the concept of an MRP environment $\mathcal{E}_R$. This term refers to a specific Markov reward process environment that can sequentially generate state and reward sequences $s_1, r_1, s_2, r_2, \ldots$, which are sequences of random variables. In this book, when such an environment is provided, it means **we can obtain such random variable sequences from the environment but cannot access the environment's model**.

Monte Carlo methods (MC) stem from a simple idea: conduct multiple experiments in an environment, and according to the law of large numbers, frequencies will gradually converge to probabilities. For MRP value estimation, we aim to estimate the value function, which is an expectation defined over the state space. By conducting sufficiently many experiments and computing the discounted returns and their statistical expectations, we can approximate the true value function.

Specifically, suppose in each episode of interaction with the environment, we obtain a finite state-reward sequence $SRS(t)$:

$$SRS(n) := s_1, r_1, s_2, r_2, \ldots, s_{L_e}, r_{L_e}$$

where L_e denotes the length of an episode. For MRPs with terminal states, L_e is the number of steps until termination. For MRPs without terminal states or where termination is difficult to achieve, L_e can be artificially set to truncate the interaction with the environment.

For the $SRS(t)$ sampled from the environment in the n-th episode, we can compute the discounted return for the t-th step through backward calculation:

$$G_t(n) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{L_e - t} r_{L_e}, t = 1, 2, ..., L_e$$

$G_t(n)$ corresponds to s_t and is a sample of $V(s_t)$. When the number of experiments is sufficiently large, and the same state is sampled enough times, we can average all $G_t(n)$ for that state to estimate $V(s_t)$.

Based on this idea, we summarize the MC value estimation algorithm for MRPs as follows.

Algorithm	**MRP-MC Value Estimation**
Problem Type	MRP Value Estimation [Model-Free]
Given	MRP environment $\mathcal{E}_R$
Objective	State value $V(s), s \in \mathcal{S}$
Algorithm Property	Model-free, Tabular, MC

Algorithm 59: MRP-MC Value Estimation

Input: MRP environment $\mathcal{E}_R$
Output: State value $V(s), s \in \mathcal{S}$
$\hat{V}(s) \leftarrow 0, \forall s \in \mathcal{S}$
$N(s) \leftarrow 0, \forall s \in \mathcal{S}$
for $n \in 1, \ldots, N$ **do**
  $\{s_1, r_1, s_2, r_2, \ldots, s_{L_e}, r_{L_e}\} \leftarrow \mathcal{E}_R$
  $G_{L_e+1} \leftarrow 0$
  for $t \in \mathcal{L}_e, L_e - 1, ..., 1$ **do**
    $G_t \leftarrow r_t + \gamma G_t$
  for $t \in 1, 2, \ldots, L_e$ **do**
    $N(s_t) \leftarrow N(s_t) + 1$
    $\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \frac{1}{N(s_t)}(G_t - \hat{V}(s_t))$

In the above algorithm, we utilize a conclusion called the **Sequential Update Formula**. Suppose we have a sequence of numbers $x_i, i = 1, 2, 3, \ldots$ obtained consecutively, and we want to update the average of these numbers immediately upon receiving the next value. If we follow the definition of average, we would need to retain all previous x values up to the current time t:

$$\bar{x}_t = \frac{x_1 + x_2 + x_3 + \cdots + x_t}{t}$$

However, if we subtract $\bar{x}_t$ from $\bar{x}_t$ and perform some transformations, we can derive the following formula:

$$\bar{x}_t = \bar{x}_{t-1} + \frac{1}{t}(x_t - \bar{x}_{t-1})$$

Adopting the sequential update approach can effectively save storage space. Meanwhile, it reveals that the "correction" effect of new values on the average gradually diminishes as data accumulates—consistent

with our intuition from the law of large numbers that "the mean converges to the expectation." Readers may verify for themselves whether the sequential update we provided is equivalent to recording all G_t values corresponding to states and then computing their average.

20.4 MDP and Action Value

In the previous section, we discussed value and value estimation problems related to MRP. Let us revisit a description introduced when discussing MRP: in an MRP, there is only one "role" taking actions—either referred to as "God" or the "**environment**"; only this "role's" "move" determines the next state or reward.

In reality, we are more concerned with how to use a "controllable" role to actively make decisions and obtain greater rewards, rather than passively waiting for the environment to provide state transitions and values. In other words, we need a "role" of our own that can effectively "interact" with the environment. Compared to "God" or the "environment," our role is the "player."

This leads us to the definition of a **Markov Decision Process** (MDP): the agent and the environment interact sequentially; the environment provides the agent with the current state and reward, and the agent submits an action to the environment. The current state and reward provided by the environment depend only on the previous state (Markov property).

Here, we introduce the concept of an **action**. The choices made by the agent in the environment are called actions. All possible actions form the **action space**, denoted as $\mathcal{A}$. Similar to the state space, the action space can also be finite or infinite, discrete or continuous. In Chapter 3 and earlier, we mainly discuss finite discrete action spaces.

In addition, we introduce the concept of an agent (sometimes translated as "proxy" or "intelligent agent"). This concept refers to a "role" that can interact with the environment and actively decide the next action. In reinforcement learning, we continuously collect data and learn knowledge about the environment from the data, enabling the agent to make increasingly better decisions. This is the key idea behind the "reinforcement" in "reinforcement learning."

Similar to MRP, an MDP environment also has its mathematical formulation. In an MRP, the environment is described by the state transition matrix $P(s'|s)$ and the reward function $\bar{R}(s)$. In an MDP, the environment determines the next state not only based on the previous state but also based on the action taken by the agent. Therefore, both functions include the action a as a condition. That is, the mathematical description (model) of the MDP environment consists of the **action-state transition matrix** $P(s'|s,a)$ and the reward function $\bar{R}(s,a)$. Similarly, if the model is unknown but the environment can interact with the agent, we refer to it as an MDP environment $\mathcal{E}_D$ in the following discussion.

Let us focus on a complete process of an MDP. First, the environment provides an initial state s_0 according to an initial normal distribution (sometimes accompanied by an initial reward r_0). Then, the agent decides on an action a_{t+1} based on the previous state s_t. The environment then determines the next state S_{t+1} and corresponding reward r_{t+1} based on the action a_{t+1} and s_t. This forms a **state-reward-action sequence** $SRAS(t)$:

$$SRAS(t) := s_0, r_0, a_1, s_1, r_1, a_2, s_2, r_2, \dots, a_{L_e}, s_{L_e}, r_{L_e}$$

Similar to the state-reward sequence in MRP, a state-reward-action sequence is also a sample from an MDP. The agent obtains a reward sequence $r_1, r_2, \dots, r_{L_e}$ during interaction with the environment. Our goal is naturally to maximize the expected value of current and future rewards. We continue to use the discounted return formula from MRP (Equation VI.20.4) to "discount" future rewards to the present.

The discounted return is a random variable, and in MDP we use "value" to represent its expectation. Similarly, we define the **state value function** in MDP as $V_\pi(s)$:

$$V_\pi(s) = \mathbb{E}_{s' \sim P(s'|s'',a), r \sim p_r(r|s',a), a \sim \pi(a|s')}[G_t|S_t = s] \tag{VI.20.7}$$

Here, we explicitly (verbosely but not rigorously) write out the "expectation" in the value function to show over which variables the expectation is taken. As seen, there are actually three groups of random variables being averaged.[20] On one hand, the future state s' depends on the previous state s'' and action a, and the future reward depends on the reward distribution $p_r(r|s',a)$; on the other hand, the future action a depends on the **agent-related action distribution** $\pi(a|s')$.[21]

[20] To distinguish the argument of the state value function from future states, we use s' and s'' to denote future states

[21] s' and a refer not to a specific future time but to all future states and actions collectively, in an imprecise expression

This leads us to introduce the concept of a **policy**. A policy is the method by which the agent decides the next action based on the previous state. In a chess game, this is the strategy for making moves; in a video game, it is the logic for the next operation. If the agent may choose different actions in the same state, we write the policy as a conditional probability distribution over actions: $\pi(a|s)$, which is called a **stochastic policy**. Conversely, if a unique action is chosen in a given state, it is called a **deterministic policy**.

We can now see that the key difference between the state value function in MDP and that in MRP is: **the MDP state value function depends on a specific policy** π. This is because MDP involves two "roles": the environment and the agent. The future is jointly determined by both. If the environment is fixed, the distributions of future states and rewards remain unchanged; but if the agent changes, the distribution of future actions changes. Therefore, **in MDP, it is meaningless to discuss state value without reference to a policy**.

In MRP, we define the state value function to understand which states are "favorable" to us. In MDP, due to the addition of actions/policies, we should not only evaluate states independently but also evaluate the **favorability of state-action pairs**. Thus, we introduce the MDP **(state-)action value function** $Q_\pi(s, a)$:

$$Q_\pi(s, a) = \mathbb{E}_{a' \sim \pi(a'|s')}[G_t|S_t = s, A_{t+1} = a] \tag{VI.20.8}$$

In subsequent discussions, we will refer to it simply as the "action value function" or "Q-function." If the state/action space is discrete and finite, we call it a "Q-table." Correspondingly, the state value function is referred to as the "V-function."

It is important to note that, similar to the state value function in MDP, **the action value function also depends on the specific policy** π. In the above expression, we omit the distributions on which future states and rewards depend for simplicity, but readers should be aware that they are also part of the expectation. The explicit term $a' \sim \pi(a'|s')$ is used to emphasize that the Q-function necessarily depends on the policy π.

Above, we defined both the state value function and the action value function. Both are expectations over future random variables. So, can these two types of value be converted into each other?

According to the definition of Q-value, we can easily convert Q-values into V-values:

$$\begin{aligned}
V_\pi(s) &= \mathbb{E}_{a \sim \pi(a|s'), o \sim p_o(o)}[G_t|S_t = s] \\
&= \sum_{a \in \mathcal{A}} \pi(a|s) \mathbb{E}_{o \sim p_o(o)}[G_t|S_t = s, A_t = a] \\
&= \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a)
\end{aligned}$$

That is,

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a) \tag{VI.20.9}$$

Note that in the above derivation, we use o to represent all other random variables besides the next action a when computing the expectation in the state value function. This form emphasizes that both V and Q definitions involve expectations over many future random variables.

Similarly, we can give the formula for computing action value Q from state value V:

$$Q_\pi(s, a) = \bar{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V_\pi(s') \tag{VI.20.10}$$

Readers should note: in Equation VI.20.9, only the current policy π is needed to compute V from Q. But in Equation VI.20.10, we need to know the expected reward $\bar{R}(s, a)$ and the transition matrix $P(s'|s, a)$, i.e., the model must be known. This distinction is crucial.

If we substitute the V-to-Q formula into the Q-to-V formula, we obtain the relationship between state value and the next round's state value:

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) (\bar{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V_\pi(s')) \tag{VI.20.11}$$

We can also substitute the Q-to-V formula into the V-to-Q formula to obtain the relationship between action value and the next round's action value:

$$Q_\pi(s,a) = \bar{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)\left(\sum_{a' \in \mathcal{A}} \pi(a'|s') Q_\pi(s',a')\right) \tag{VI.20.12}$$

Equations VI.20.12 and VI.20.11 are also referred to as the Bellman expectation equations of MDP. They reveal the recursive nature of the action value function and the state value function. Note that using these two formulas requires not only the policy but also the known model.

20.5 The MDP Prediction Problem

In MDP, the state value function (and the action value function) depends on the policy. Therefore, a natural question is: how can we compute the state value function given a known policy?

This question is very similar to the previously discussed problem of "how to compute the state value function in MRP." Indeed, the two are closely related. However, readers should note a key difference: in MRP, there is no "player," and one environment corresponds to one value function; in MDP, there is a "player" and a policy π, so the value function is bound to the policy. If the policy changes while the model remains the same, the value function also changes.

Therefore, in MDP, the problem of computing the value function is called **policy evaluation**, which assesses whether a policy is good or bad in the same environment. The notion of "good/bad" will be formalized using a partial order later. Sometimes, we also refer to this as the **MDP prediction problem**, to distinguish it from the control problem discussed later.

Problem	**MDP Prediction Problem [Model-Dependent]**
Description	Given the environment model and policy of an MDP, compute the state value for each state
Given	State transition matrix $P(s'|s,a)$, reward function $\bar{R}(s,a)$, policy $\pi(s|a)$
Objective	State value $V_\pi(s), s \in \mathcal{S}$

Since this problem is similar to MRP value estimation, we can adapt the solution approach from MRP to the MDP prediction problem.

Consider applying the dynamic programming algorithm (Algorithm 58) to the prediction problem. Algorithm 58 iterates based on Equation VI.20.4, which should be modified to use Equation VI.20.11 for MDPs.

Algorithm	**MDP-DP Policy Evaluation**
Problem Type	MDP Prediction Problem [Model-Dependent]
Given	State transition matrix $P(s'|s,a)$, reward function $\bar{R}(s,a)$, policy $\pi(s|a)$
Objective	State value $V_\pi(s), s \in \mathcal{S}$
Algorithm Properties	model-based, tabular, bootstrapping

Algorithm 60: MDP-DP Policy Evaluation

Input: State transition matrix $P(s'|s,a)$, reward function $\bar{R}(s,a)$, policy $\pi(s|a)$
Parameter: Discount factor γ
Output: State value $V_\pi(s), s \in \mathcal{S}$
$\hat{V}(s) \leftarrow random([V_{min}, V_{max}]), \forall s \in \mathcal{S}$
for $i \in 1, \ldots, N_I$ **do**
  for $s \in \mathcal{S}$ **do**
    $V_\pi(s) \leftarrow 0$
    for $a \in \mathcal{A}$ **do**
      $\hat{V}_{next}(s,a) \leftarrow \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s')$
      $\hat{V}_\pi(s) \leftarrow \hat{V}_\pi(s) + \pi(a|s)(\bar{R}(s,a) + \hat{V}_{next}(s,a))$

The corresponding Python code is shown below:

```python
class MDPModel:
    def __init__(self, Ns:int, Na:int):
        self.Ns, self.Na = Ns, Na
        self.p_sas = np.zeros([self.Ns, self.Na, self.Ns])
        self.p_s0 = np.zeros([self.Ns])
        ......# p_sas and p_s0 init
        pass
    def state_transfer_distb(self, s:int, a:int):
        assert 0 <= s and s < self.Ns and 0 <= a and a < self.Na
        return self.p_sas[s, a]
    def init_state_distb(self):
        return self.p_s0
    def reward(self, s:int, a:int):
        r = 0
        ...... # reward calculation
        return r
def MDP_DP_policy_est(mdp:MDPModel, pi_func, gamma=0.9, vmin=0, vmax=99):
    V_pi = np.random.uniform(low=vmin, high=vmax, size=(mdp.Ns,))
    for i in range(N_I):
        for s in range(mdp.Ns):
            v_s = 0
            pi_s = pi_func(s)
            for a in range(mdp.Na):
                r_sa = mdp.reward(s, a)
                psa = mdp.state_transfer_distb(s, a)
                v_next = gamma * psa[None, :] @ V[:, None]
                v_s += pi_s[a] * (r_sa + v_next)
            V_pi[s] = v_s
    return V_pi
```

As we can see, the DP method is entirely model-based and cannot be used without a model. Additionally, during each update step, it needs to traverse all combinations of (a, s'), i.e., the $A \times S$ space. When the state space and action space are large, such operations are undoubtedly inefficient.

For model-free scenarios, we can also formulate the MDP prediction problem.

Problem	**MDP Prediction Problem [Model-Free]**
Description	Given an MDP with unknown environment model, compute the state value under a given policy
Given	MDP environment $\mathcal{E}_D$, policy $\pi(s\|a)$
Objective	State value $V_\pi(s), s \in \mathcal{S}$

Similarly, we can adapt the Monte Carlo approach from model-free MRP value estimation to solve this problem.

Algorithm	**MDP-MC Policy Evaluation**
Problem Type	MDP Prediction Problem [Model-Free]
Given	MDP environment $\mathcal{E}_D$, policy $\pi(s\|a)$
Objective	State value $V_\pi(s), s \in \mathcal{S}$
Algorithm Properties	model-free, tabular, MC

We still employ the incremental learning approach. The difference from MRP is that the environment doesn't directly provide complete episode data; instead, we must interact with the environment according to policy π. After interaction, we update the value function table using the incremental learning formula.

Incremental learning has another advantage. While $\hat{V}_\pi(s_t) \leftarrow \hat{V}_\pi(s_t) + \frac{1}{N(s_t)}(G_t - \hat{V}_\pi(s_t))$ satisfies the update rule for the mean, the coefficient before the increment doesn't necessarily have to be the reciprocal

of the sample count. As long as it's a parameter that gradually decreases under certain conditions, the value function will converge. Therefore, we sometimes express incremental learning in this form:

$$\hat{V}_\pi(s_t) \leftarrow \hat{V}_\pi(s_t) + \alpha(G_t - \hat{V}_\pi(s_t)) \tag{VI.20.13}$$

In the algorithm box above, we actually show the case where $\alpha = \frac{1}{N(s_t)}$. However, in practice, setting it as a hyperparameter α allows us to tune the learning speed based on actual learning performance, similar to learning rate tuning in deep learning. In subsequent incremental learning methods, we will consistently use α to denote the learning rate parameter.

The Monte Carlo method compensates for the lack of a known model through sampling. However, in the MC policy evaluation algorithm, we must sample an entire episode before updating the value table. Yet, Equations VI.20.9 and VI.20.10 reveal that value functions have interdependent relationships that enable iterative updates. Previously, with a known model, we leveraged this dependency to design dynamic programming methods for bootstrapping value table estimation. So, in model-free scenarios, can we adopt a similar approach to iteratively update the value table after each sampling step?

Algorithm 61: MDP-MC Policy Evaluation

Input: MDP environment $\mathcal{E}_D$, policy $\pi(s|a)$
Parameter: Discount factor γ
Output: State value $V_\pi(s), s \in \mathcal{S}$
$\hat{V}_\pi(s), N(s) \leftarrow 0, 0, \forall s \in \mathcal{S}$
for $i \in 1, \ldots, N_I$ **do**
  $s_0, r_0 \sim p(s_0), p(r_0)$
  for $t \in 1, 2, \ldots, L_e$ **do**
    $a_t \sim \pi(a|s = s_{t-1})$
    $s_t, r_t \leftarrow \mathcal{E}_D(s_{t-1}, a_t)$
  $G_{L_e+1} \leftarrow 0$
  for $t \in \mathcal{L}_e, L_e - 1, ..., 1$ **do**
    $G_t \leftarrow r_t + \gamma G_t$
  for $t \in 1, 2, \ldots, L_e$ **do**
    $N(s_t) \leftarrow N(s_t) + 1$
    $\hat{V}_\pi(s_t) \leftarrow \hat{V}_\pi(s_t) + (G_t - \hat{V}_\pi(s_t))/N(s_t)$

The corresponding Python code is shown below:

```
class MDP:
    def __init__(self, Ns:int, Na:int):
        self.Ns, self.Na = Ns, Na
        self.p_sas = np.zeros([self.Ns, self.Na, self.Ns])
        self.p_s0 = np.zeros([self.Ns])
        ......# p_sas and p_s0 init
        pass
    def state_transfer(self, s:int, a:int):
        assert 0 <= s and s < self.Ns and 0 <= a and a < self.Na
        s_out = np.random.choice(self.Ns, p=self.p_sas[s, a])
        return s_out
    def init_state(self):
        return s_out = np.random.choice(self.Ns, p=self.p_s0)
    def reward(self, s:int, a:int):
        r = 0
        ...... # reward calculation
        return r
def sample_trail_policy(env:MDP, pi_func=None):
    states, rs, a_s = [env.init_state()], [env.reward(s)], [None]
    for t in range(1, L_e+1):
        a_s.append(np.random.choice(env.Na, p=pi_func(s)))
```

```
        states.append(env.state_transfer(states[t-1], a_s[t]))
        rs.append(env.reward(states[t]))
    return rs, states, a_s
def MDP_MC_policy_est(env:MDP, pi_func, gamma=0.9, L_e=100, vmin=0, vmax=99):
    V_pi = np.random.uniform(low=vmin, high=vmax, size=(env.Ns,))
    num_s = np.zeros([env.Ns])
    for _ in range(N_I):
        rs, states, a_s = sample_trail_policy(env, pi_func) # related to Q !!
        Gs = np.zeros([L_e + 2])
        for t in range(L_e, 0, -1):
            Gs[t] = rs[t] + gamma * Gs[t+1]
        for t in range(1, L_e + 1):
            num_s[states[t]] += 1
            V_pi[states[t]] += (Gs[t] - V_pi[states[t]]) / num_s[states[t]]
    return V_pi
```

In fact, this class of methods is called **Temporal Difference** (TD) methods. Specifically, the reason Monte Carlo methods need to wait for an episode to complete is that value learning depends on the cumulative return G_t, and the calculation of G_t requires working backward in time from the end. So, can we find a better way to estimate G_t?

Let's start from the definition of G_t (Equation VI.20.2). Note that G_t can be expanded into a recursive form in one step:

$$G_t = r_t + \gamma G_{t+1}$$

Although we cannot know either G_t or G_{t+1} without obtaining the entire episode, we know from the value function definition (Equation VI.20.7) that $V(S_{t+1})$ is the expectation of G_{t+1}. In other words, if we know r_t and $V(S_{t+1})$, we can obtain an estimate of G_t.

$$G_t \approx r_t + \gamma V(s_{t+1})$$

Here, we refer to this estimate as the **temporal difference target**, denoted as T_t, which is the "learning" target of the policy evaluation algorithm. For the simplest case, expanding G_t by one time step means this target only requires knowledge of the next state to compute, which is much easier than calculating G_t that requires the entire episode.

Thus, we have the general form of incremental learning:

$$\hat{V}_\pi(s_t) \leftarrow \hat{V}_\pi(s_t) + \alpha(T_t - \hat{V}_\pi(s_t)) \tag{VI.20.14}$$

And the simplest one-step estimation method:

$$T_t = r_t + \gamma V(s_{t+1}) \tag{VI.20.15}$$

Putting it all together, we obtain the **0-step Temporal Difference** (TD(0)) policy evaluation algorithm, where TD stands for Time-Difference.

Algorithm	**MDP-TD(0) Policy Evaluation**
Problem Type	MDP Prediction Problem [Model-Free]
Given	MDP environment $\mathcal{E}_D$, policy $\pi(s\|a)$
Objective	State value $V_\pi(s), s \in \mathcal{S}$
Algorithm Properties	Model-free, Tabular, TD

Algorithm 62: MDP-TD(0) Policy Evaluation

Input: MDP environment $\mathcal{E}_D$, policy $\pi(s|a)$
Parameter: Discount factor γ, update parameter α
Output: State value $V_\pi(s), s \in \mathcal{S}$
$\hat{V}_\pi(s) \leftarrow 0, \forall s \in \mathcal{S}$
$N(s) \leftarrow 0, \forall s \in \mathcal{S}$
for $n \in 1, \ldots, N$ **do**
  $s_0, r_0 \sim p(s_0), p(r_0)$
  for $t \in 1, 2, \ldots, L_e$ **do**
    $a_t \sim \pi(a|s = s_{t-1})$
    $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
    $T_{t-1} = r_{t-1} + \gamma \hat{V}_\pi(s_t)$
    $\hat{V}_\pi(s_{t-1}) \leftarrow \hat{V}_\pi(s_{t-1}) + \alpha(T_{t-1} - \hat{V}_\pi(s_{t-1}))$

The corresponding Python code is shown below:

```
def MDP_td0_policy_est(env:MDP, pi_func, gamma=0.9, alpha = 0.5, L_e=100, vmin=0, vmax=99):
    V_pi = np.random.uniform(low=vmin, high=vmax, size=(env.Ns,))
    num_s = np.zeros([env.Ns])
    for _ in range(N_I):
        states = [env.init_state()]
        rs = [env.reward(s)]
        for t in range(1, L_e+1):
            a_t = np.random.choice(env.Na, p=pi_func(s))
            states.append(env.state_transfer(states[t-1], a_t))
            rs.append(env.reward(states[t]))
            target_last = rs[t-1] + gamma * V_pi[states[t]]
            V_pi[states[t-1]] += alpha * (target_last - V_pi[states[t-1]])
    return V_pi
```

In TD(0), the temporal difference target only includes one actual sampled reward value. In fact, we can further integrate the MC method with TD(0) by allowing the temporal difference target to include multiple subsequent sampled values. This is the TD(n) method.

Compared to TD(0), TD(n) only requires modifying the temporal difference target:

$$T_t = r_t + \gamma r_{t+1} + \ldots + \gamma^n r_{t+n} + V(s_{t+n+1}) \tag{VI.20.16}$$

We can summarize the $TD(n)$ policy evaluation algorithm as follows:

Algorithm	**MDP-TD(n) Policy Evaluation**
Problem Type	MDP Prediction Problem [Model-Free]
Given	MDP Environment $\mathcal{E}_D$, Policy $\pi(s\|a)$
Objective	State Value $V_\pi(s), s \in \mathcal{S}$
Algorithm Properties	Model-Free, Tabular, TD

Algorithm 63: MDP-TD(n) Policy Evaluation

Input: MDP Environment $\mathcal{E}_D$, Policy $\pi(s|a)$
Parameter: Discount Factor γ, Iteration Count n, Update Parameter α
Output: State Value $V_\pi(s), s \in \mathcal{S}$
$\hat{V}_\pi(s) \leftarrow 0, \forall s \in \mathcal{S}$
$N(s) \leftarrow 0, \forall s \in \mathcal{S}$
for $n \in 1, \ldots, N$ **do**
  $s_0, r_0 \sim p(s_0), p(r_0)$
  for $t \in 1, 2, \ldots, L_e$ **do**
    $a_t \sim \pi(a|s = s_{t-1})$
    $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
    if $t > n + 1$ **then**
      $T_{t-n-1} = r_{t-n-1} + \gamma r_{t-n} + ... + \gamma^n r_{t-1} + \gamma^{n+1} V(s_t)$
      $\hat{V}_\pi(s_{t-n-1}) \leftarrow \hat{V}_\pi(s_{t-n-1}) + \alpha(T_{t-n-1} - \hat{V}_\pi(s_{t-n-1}))$

The corresponding Python code is shown below:

```python
def MDP_tdn_policy_est(env:MDP, pi_func, gamma=0.9, n_td=4, alpha = 0.5, L_e=100, vmin=0,
↪  vmax=99):
    V_pi = np.random.uniform(low=vmin, high=vmax, size=(env.Ns,))
    num_s = np.zeros([env.Ns])
    for _ in range(N_I):
        states = [env.init_state()]
        rs = [env.reward(s)]
        for t in range(1, L_e+1):
            a_t = np.random.choice(env.Na, p=pi_func(s))
            states.append(env.state_transfer(states[t-1], a_t))
            rs.append(env.reward(states[t]))
            if t > n_td + 1:
                target_old = gamma**(n_td + 1) * V_pi[states[t]]
                for i in range(n):
                    target_old += (gamma ** i) * rs[t-n_td-1+i]
                V_pi[states[t-n-1]] += alpha * (target_old - V_pi[states[t-n-1]])
    return V_pi
```

20.6 MDP Control Problem

So far, we have introduced many methods related to value estimation. In MRP, while value estimation under a given policy is important, what we are more concerned with is how to find better policies.

This leads to questions we haven't explored deeply before: How do we define that one policy is "better" than another? What constitutes a good policy? With the state value function, we can directly provide a definition: A policy is better if its state value function is superior for every state. We denote this as $\pi \geq \pi'$

$$\pi \geq \pi' := V_\pi(s) \geq V_{\pi'}(s) \tag{VI.20.17}$$

Since we have the concept of "better," we naturally consider "best." Under this definition, we can define the **optimal policy**:

$$\pi^* := \arg\max_\pi V_\pi(s), \forall s \in \mathcal{S} \tag{VI.20.18}$$

or

$$V_{\pi^*}(s) \geq V_\pi(s), \forall s \in \mathcal{S}, \forall \pi$$

The corresponding value function is also called the **optimal value function** $V^*(s)$:

$$V^*(s) := \max_\pi V_\pi(s) = V_{\pi^*}(s) \tag{VI.20.19}$$

There are several points to note about the optimal policy. First, the optimal policy is specific to a particular environment. If the environment changes (e.g., the state transition matrix or expected value function changes), the optimal policy will also change. Second, the optimal policy is a theoretical construct. For environments with unknown models, we can only approximate it using various methods; for environments with known models, calculating its true value is often extremely difficult. Third, the optimal policy is not necessarily unique. If the environment exhibits certain symmetries, there may be multiple or even infinitely many optimal policies.

Additionally, Equation VI.20.18 hides an easily overlooked question: **Assuming we know the optimal value function, how do we derive the optimal policy?**

The optimal policy is a probability distribution $\pi(a|s)$, but it can also represent a deterministic policy (i.e., assigning all probability to a specific action a). To obtain the optimal policy, we only need to know the optimal action a^* (or equivalent optimal actions) for a given state and assign all probability to it. The problem is: the state value function $V^*(s)$ does not involve actions at all.

The action-state value function, however, meets our needs. If we have the optimal $Q^*(s, a)$ corresponding to the optimal $V^*(s)$, we can compare the Q-values of different actions under the same state s and select the highest one (or ones) as the policy. The question is: **How do we derive the Q-function from the V-function?**

Recall the formulas for the relationship between the V-function and Q-function in Section 20.4 (Equations VI.20.9 and VI.20.10). Notably, these formulas can also be written in terms of the optimal policy:

$$V^*(s) = \sum_{a \in \mathcal{A}} \pi^*(a|s) Q^*(s, a) \tag{VI.20.20}$$

$$Q^*(s, a) = \bar{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V^*(s') \tag{VI.20.21}$$

$$V^*(s) = \sum_{a \in \mathcal{A}} \pi^*(a|s)(\bar{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V^*(s')) \tag{VI.20.22}$$

$$Q^*(s, a) = \bar{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a)(\sum_{a' \in \mathcal{A}} \pi^*(a'|s') Q^*(s', a')) \tag{VI.20.23}$$

Among these formulas, VI.20.22 and VI.20.23 are also known as the **Bellman optimality equations**.

However, what we need is the formula to derive the Q-function from the V-function, i.e., Equation VI.20.21, which clearly depends on the environment model (state transition matrix). In other words, **without a model, we cannot derive the optimal policy $\pi^*(a|s)$ from the optimal state value $V^*(s)$.**

This critical conclusion reveals the **fundamental difference between the Q-function and V-function**: although both are value functions, the Q-function provides us with the key information for action selection, i.e., policy design, while the V-function only tells us how to evaluate policies and states. That is, despite the conversion relationship between the two functions, in practical terms, **if we aim for the optimal policy, obtaining the Q-function is more crucial than obtaining the V-function**.

Thus, we can formally introduce the **MDP control problem**: Given an MDP environment, find the optimal policy $\pi^*(a|s)$ under known or unknown model conditions.

Before addressing the unknown model case, let's start with the simpler scenario where the model is known.

Problem	**MDP Control Problem [Model-Dependent]**
Problem Description	Given the MDP environment model, find the optimal policy
Given	State transition matrix $P(s'|s, a)$, reward function $\bar{R}(s, a)$
Objective	Optimal policy $\pi^*(a|s)$

A naive idea is to use the policy evaluation methods we discussed earlier. Policy evaluation can tell us the value function corresponding to a policy. Based on this value function, if we have a way to improve the policy, we can derive a new policy. We can then use policy evaluation again to obtain an even better value function. By repeating this process, the policy will eventually converge to the optimal policy.

This approach has been proven feasible and is called **policy iteration**. It consists of two iterative steps: **policy evaluation** and **policy improvement**. In the previous section, we introduced many policy evaluation methods. So, how do we perform policy improvement?

Clearly, we need to use the formula defining a "better policy" (Equation VI.20.17). According to this formula, during policy improvement, we only need to ensure that the policy π_k from the k-th iteration has a value function $V_k(s)$ that is better than the value function $V_{k-1}(s)$ of the policy π_{k-1} from the $(k-1)$-th iteration (for every state in the state space). How can we achieve this?

In fact, we can simply apply a **greedy** approach:

$$\pi_{k+1}(a|s) = \begin{cases} 1, a = \arg\max_a Q_k(s,a) \\ 0, \text{otherwise} \end{cases} \tag{VI.20.24}$$

"Greedy" here means selecting the current best (locally optimal) action. Although the value function $V_k(s)$ is not the optimal value function $V^*(s)$ before convergence, we can design the policy $\pi_k(a|s)$ at each iteration as the optimal policy corresponding to the local $V_k(s)$. That is, we identify the action a that maximizes $Q_k(s,a)$ and fix it as the current policy $\pi_k(a|s)$.

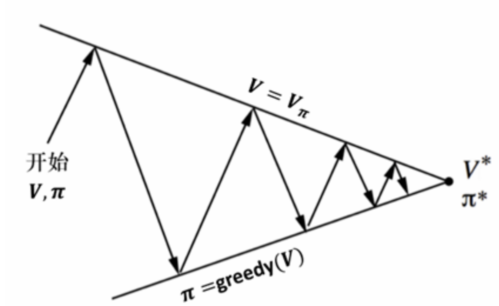

Figure VI.20.1: Policy iteration process

Note that this greedy process relies on the Q-values. As discussed earlier, deriving Q from V requires a known model. Thus, this method solves the model-dependent MDP control problem. Building on the DP policy evaluation methods introduced earlier, we now present the **DP policy iteration** method for solving the MDP control problem.

Algorithm	**MDP-DP Policy Iteration (Greedy)**
Problem Type	MDP Control Problem [Model-Dependent]
Given	State transition matrix $P(s'|s,a)$, reward function $\bar{R}(s,a)$
Objective	Optimal policy $\pi^*(a|s)$
Algorithm Properties	model-based, tabular, bootstrapping

Algorithm 64: MDP-DP Policy Iteration (Greedy)

Input: State transition matrix $P(s'|s,a)$, reward function $\bar{R}(s,a)$
Output: Optimal policy $\pi^*(a|s)$
$\pi_0(a|s) \leftarrow random(A \times S)$
for $k \in 1, \dots, N_K$ **do**
  $V_k(s) \leftarrow$ MDP_DP_policy_est(π_k)
  for $s \in \mathcal{S}$ **do**
    for $a \in \mathcal{A}$ **do**
      $Q_k(s,a) \leftarrow \bar{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s')$
    $$\pi_{k+1}(a|s) \leftarrow \begin{cases} 1, a = \arg\max_a Q_k(s,a) \\ 0, others \end{cases}$$

The corresponding Python code is shown below:

```python
def MDP_dp_iter(mdp:MDPModel, N_k, gamma=0.9):
    Q_mat = np.random.rand([mdp.Ns, mdp.Na])
    def pi_func_greedy(s:int):
        assert 0 <= s and s < Q_mat.shape[0]
        pi = np.zeros_like(Q_mat[s])
        pi[np.argmax(Q_mat[s])] = 1
        return pi
    for k in range(N_k):
        V_k = MDP_DP_policy_est(mdp, pi_func_greedy, gamma=gamma)
        for s in range(mdp.Ns):
            for a in range(mdp.Na):
                r_sa = mdp.reward(s, a)
                v_next = gamma * psa[None, :] @ V[:, None]
                Q_mat[s, a] = r_sa + v_next
        pass
    return pi_func_greedy
```

In DP-based policy iteration, for each major loop, we alternately perform policy evaluation and policy improvement. Both policy evaluation and policy improvement require traversing every possible state-action pair. Clearly, this algorithm is rather cumbersome—can we make it slightly more concise?

We observe that the major loop essentially follows the cycle of $V_k \to Q_k \to \pi_{k+1} \to V_{k+1}$, where the Bellman equations (Eq. VI.20.12 and Eq. VI.20.11) are used in policy evaluation and policy improvement, respectively. In the major loop, the optimal policy is merely an intermediate variable. However, according to Eq. VI.20.9, we can directly derive the value function from the Q-function. In fact, we can accomplish this process with a very simple greedy strategy:

$$V_{k+1}(s) = \arg\max_a Q_\pi(s,a) \tag{VI.20.25}$$

This way, we can directly perform the iteration of $V_k \to Q_k \to \pi_{k+1} \to V_{k+1}$ without using the Bellman equations. As long as $V_k(s)$ converges to $V^*(s)$, it does not affect the final policy being the optimal policy $\pi^*(s)$. This algorithm, which skips intermediate optimal policies and directly iterates between Q and V tables to derive the optimal value, is called **value iteration**. Next, we summarize the deterministic value iteration algorithm.

Algorithm	**MDP-Deterministic Value Iteration**
Problem Type	MDP Control Problem [Model-Dependent]
Known	State transition matrix $P(s'\|s,a)$, reward function $\bar{R}(s,a)$
Objective	Optimal policy $\pi^*(a\|s)$
Algorithm Properties	Model-based, Tabular, Bootstrapping

Algorithm 65: MDP-Deterministic Value Iteration

Input: State transition matrix $P(s'|s,a)$, reward function $\bar{R}(s,a)$

Output: Optimal policy $\pi^*(a|s)$

$V_0(s) \leftarrow random([V_{min}, V_{max}]), \forall s \in \mathcal{S}$

for $k \in 1, \ldots, N_K$ **do**

  for $s \in \mathcal{S}$ **do**

    for $a \in \mathcal{A}$ **do**

      $Q_k(s,a) = \bar{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s')$

    $V_{k+1}(s) = \arg\max_a Q_\pi(s,a)$

$$\pi^*(a|s) \leftarrow \begin{cases} 1, a = \arg\max\limits_a Q_K(s,a) \\ 0, others \end{cases}$$

The corresponding Python code is shown below:

```
def MDP_value_iter(mdp:MDPModel, N_k, gamma=0.9):
    Q_mat = np.random.rand([mdp.Ns, mdp.Na])
    def pi_func_greedy(s:int):
        assert 0 <= s and s < Q_mat.shape[0]
        pi = np.zeros_like(Q_mat[s])
        pi[np.argmax(Q_mat[s])] = 1
        return pi
    V = np.random.uniform(low=vmin, high=vmax, size=(mdp.Ns,))
    for k in range(N_K):
        for s in range(mdp.Ns):
            for a in range(mdp.Na):
                r_sa = mdp.reward(s, a)
                psa = mdp.state_transfer_distb(s, a)
                v_next = gamma * psa[None, :] @ V[:, None]
                Q_mat[s, a] = r_sa + v_next
            v_s += pi_s[a] * (r_sa + v_next)
        V = np.max(Q_mat, axis=1)
    return pi_func_greedy
```

Policy iteration and value iteration are the two main approaches for solving MDP control problems when the model is known. The core distinction between these two approaches lies in whether they generate complete intermediate policies π_k. Policy iteration produces a policy in each outer loop and continuously improves it within the outer loop. Value iteration focuses solely on deriving the best value function and only outputs the policy at the very end.

20.7 Reinforcement Learning Problem

Reflecting on our discussion so far, we began with Markov processes involving rewards, exploring what MRPs are and how to estimate their values. We introduced the concept of a model, emphasizing that problem-solving approaches differ fundamentally depending on whether a model is available. Next, we discussed MDPs with active interaction processes (policies), explaining what actions are and distinguishing between state values and action values, while highlighting that values in MDPs depend on policies. We examined prediction problems and control problems, detailing how to evaluate policies in MDPs and how to compute optimal policies when a model is available.

Undoubtedly, our journey thus far has been substantial. However, we have yet to address a critical issue faced by real-world MDPs: **How should we solve MDP control problems when no model is available?** It could be said that our lengthy chapter-long journey has essentially been laying the groundwork for this very question.

Problem	**MDP Control Problem [Model-Free]**
Brief Description	Finding optimal policy in unknown MDP environment model
Given	MDP environment $\mathcal{E}_D$
Objective	Optimal policy $\pi^*(a\|s)$

Without a model, the value function must be approximated through experiments. Without a model, the state value V cannot be used to compute the action value Q. The challenge we face becomes greater, but there's no need to worry - in some algorithms presented in this chapter, solutions are already beginning to emerge.

Reinforcement learning refers to continuously strengthening an agent's policy through interaction with the environment, learning from existing data to improve the agent's problem-solving capabilities. The model-free MDP control problem is a classic type of MDP problem.

However, reinforcement learning goes beyond this. In MDP control problems, we assume the state s provided by the environment contains sufficient information for the agent to make the next decision. But in practical problems, the environmental information we can observe may be incomplete. This is called **partial**

observability. MDPs under partial observability conditions are called **Partially Observable Markov Decision Processes** (POMDP).

If familiar with state-space methods in control theory, we understand how significant observability is for controller design.

Solving model-free MDP control problems under partial observability conditions represents the formally most challenging reinforcement learning problem within the Markov framework.

Problem	**POMDP Control Problem [Model-Free]**
Brief Description	POMDP with partial observability and unknown environment model, seeking optimal policy
Given	POMDP environment $\mathcal{E}_{PO}$
Objective	Optimal policy $\pi^*(a\|s)$

The deep reinforcement learning methods introduced in subsequent chapters of this book will provide important solutions to POMDP problems.

20.8 Reinforcement Learning Perspectives

Value-based/policy-based/actor-critic

World models, Model-based/model-free

Multi-armed bandit problem, Exploration-exploitation dilemma, Sampling, on-policy/off-policy

21 Tabular Methods

At the end of the previous chapter, we introduced two important categories of reinforcement learning problems: model-free MDP control problems and POMDP control problems. Prior to that, we also discussed some methods for simplified versions of these problems. The core of these methods revolves around estimating the value functions V or Q. Since we confine our discussion to finite discrete state/action spaces, the V and Q functions can actually be enumerated in a tabular manner. Methods with this characteristic are referred to as **Tabular Methods**.

21.1 Monte Carlo Policy Iteration

For model-free MDP control problems, we still adopt the tabular approach. Compared to the model-based control problem introduced at the end of Chapter 1, the distinction of model-free methods lies in the inability to access $P(s'|s,a)$ and $\bar{R}(s,a)$.

Earlier, we addressed similar problems using Monte Carlo sampling (Algorithm 61). We can integrate this idea into the policy iteration method (Algorithm 64), replacing the model-calculated values with estimates derived from Monte Carlo sampling.

Algorithm	**MDP-MC Policy Iteration (ϵ-greedy)**
Problem Type	MDP Control Problem [Model-Free]
Known	MDP Environment $\mathcal{E}_D$
Objective	Optimal Policy $\pi^*(a|s)$
Algorithm Properties	value-based, on-policy?, tabular

However, this introduces the **exploration-exploitation dilemma** discussed in the previous chapter. Monte Carlo is a sampling method that relies on a certain policy for sampling. The sampling results are used to update the Q-table, which in turn defines the new policy. Without sufficient exploration of the environment early on, the algorithm may converge to a local optimum.

To address this, we employ two techniques. The first is called **exploratory starts**, and the second is **ϵ-greedy**.

Exploratory starts refers to the initial phase of policy iteration, where we avoid using the policy derived from the Q-table and instead adopt a random policy to sample different (a, s) pairs as much as possible.

ϵ-greedy means that, to achieve the effect of exploratory starts, we use a parameter ϵ to control the proportion of random actions during learning. Initially, ϵ is close to 1, and the proportion of random actions decreases over time. In other words, we follow the principle of "explore first, exploit later."

To achieve this, we can design an ϵ update strategy dependent on the episode count k, such as:

$$\epsilon(k) = \frac{1}{k} \tag{VI.21.1}$$

Thus, the ϵ-greedy policy can be expressed as:

$$\pi_\epsilon(a; s, Q_k) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{||\mathcal{S}||}, & a = \arg\max_a Q_k(s,a) \\ \frac{\epsilon}{||\mathcal{S}||}, & \text{otherwise} \end{cases} \tag{VI.21.2}$$

We summarize the MC policy iteration (ϵ-greedy) algorithm as follows.

Algorithm 66: MDP-MC Policy Iteration (ϵ-greedy)

Input: MDP Environment $\mathcal{E}_D$
Output: Optimal Policy $\pi^*(a|s)$
$Q_0(s,a) \leftarrow \text{random}(A \times S)$
$N(s,a) \leftarrow 0, \forall (a,s) \in \mathcal{S} \times \mathcal{A}$
for $k \in 1, \ldots, K$ **do**
  $\epsilon \leftarrow \epsilon(k)$
  $s_0, r_0 \sim p(s_0), p(r_0)$
  for $t \in 1, 2, \ldots, L_e$ **do**
    $a_t \sim \pi_\epsilon(a; s, Q_k)$
    $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
  $G_{L_e+1} \leftarrow 0$
  for $t \in \mathcal{L}_e, L_e - 1, ..., 1$ **do**
    $G_t \leftarrow r_t + \gamma G_t$
  for $t \in 1, 2, \ldots, L_e$ **do**
    $N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$
    $Q_k(s_t, a_t) \leftarrow Q_k(s_t, a_t) + \frac{1}{N(s_t, a_t)}(G_t - Q_k(s_t, a_t))$
$\pi^*(a|s) \leftarrow \pi_\epsilon(s, a; Q_K)$

The corresponding Python code is shown below:

```python
def MDP_MC_policy_iter(env:MDP, epsilon, N_K, gamma=0.9, L_e=100, vmin=0, vmax=99):
    Q_mat = np.random.uniform(low=vmin, high=vmax, size=[env.Ns, env.Na])
    def pi_func_greedy_eps(s:int, epsilon:float):
        assert 0 <= s and s < Q_mat.shape[0]
        pi = np.zeros_like(Q_mat[s])
        pi[np.argmax(Q_mat[s])] = 1 - epsilon
        pi += epsilon / Q_mat.shape[0]
        return pi
    num_s = np.zeros([env.Ns, env.Na])
    for k in range(N_K):
        rs, states, a_s = sample_trail_policy(env, pi_func_greedy_eps) # related to Q !!
        Gs = np.zeros([L_e + 2])
        for t in range(L_e, 0, -1):
            Gs[t] = rs[t] + gamma * Gs[t+1]
        for t in range(1, L_e + 1):
            num_s[states[t], a_s[t]] += 1
            Q_mat[states[t], a_s[t]] += (Gs[t] - Q_mat[states[t], a_s[t]]) / num_s[states[t],
            ↪  a_s[t]]
    return pi_func_greedy_eps
```

Compared to Monte Carlo policy evaluation (Algorithm 61), in Monte Carlo value iteration, we learn the Q-table instead of the V-table. This is because the Q-table enables us to select actions and directly derive the policy.

21.2 SARSA Method

In the previous chapter, we introduced three broad approaches to value estimation: Dynamic Programming (DP), Monte Carlo (MC), and Temporal Difference (TD). Compared to DP's reliance on models and MC's slow update cycles, TD effectively combines the advantages of both, balancing sampling and learning. In this section, we attempt to apply TD methods to control problems.

We first explore a TD(0)-based policy iteration. The main idea is to integrate Algorithm 62 into the framework of Algorithm 66.

Algorithm	MDP-SARSA Policy Iteration
Problem Type	MDP Control Problem [Model-Free]
Known	MDP Environment $\mathcal{E}_D$
Objective	Optimal Policy $\pi^*(a\|s)$
Algorithm Properties	value-based, on-policy, tabular

Here, it is still important to note that, depending on the objective, the learning target shifts from the V-table to the Q-table. Consequently, the temporal difference target also changes from future V-values to future Q-values. Thus, in addition to knowing the current step's (state s, action a, reward r), we also need to know the next step's (state s, action a). Combining these five letters gives us **SARSA**, which is another name for this algorithm.

Algorithm 67: MDP-SARSA Policy Iteration

Input: MDP environment $\mathcal{E}_D$
Output: Optimal policy $\pi^*(a|s)$
$Q_0(s,a) \leftarrow random(A \times S)$
$N(s,a) \leftarrow 0, \forall (a,s) \in \mathcal{S} \times \mathcal{A}$
for $k \in 1, \ldots, K$ **do**
  $\epsilon \leftarrow \epsilon(k)$
  $s_0, r_0 \sim p(s_0), p(r_0)$
  for $t \in 1, 2, \ldots, L_e$ **do**
    $a_t \sim \pi_\epsilon(a; s, Q_k)$
    $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
    $T_{t-1} = r_{t-1} + \gamma Q_k(s_t, a_t)$
    $Q_k(s_{t-1}, a_{t-1}) \leftarrow Q_k(s_{t-1}, a_{t-1}) + \alpha(T_{t-1} - Q_k(s_{t-1}, a_{t-1}))$
$\pi^*(a|s) \leftarrow \pi_\epsilon(s, a; Q_K)$

The corresponding Python code is shown below:

```python
def MDP_SARSA(env:MDP, epsilon, N_K, alpha, gamma=0.9, L_e=100, vmin=0, vmax=99):
    Q_mat = np.random.uniform(low=vmin, high=vmax, size=[env.Ns, env.Na])
    def pi_func_greedy_eps(s:int, epsilon:float):
        assert 0 <= s and s < Q_mat.shape[0]
        pi = np.zeros_like(Q_mat[s])
        pi[np.argmax(Q_mat[s])] = 1 - epsilon
        pi += epsilon / Q_mat.shape[0]
        return pi
    num_s = np.zeros([env.Ns, env.Na])
    for k in range(N_K):
        rs, states, a_s = sample_trail_policy(env, pi_func_greedy_eps) # related to Q !!
        for t in range(L_e):
            target_last = rs[t] + gamma * Q_mat[states[t], a_s[t]]
            Q_mat[states[t], a_s[t]] += alpha * (target_last - Q_mat[states[t], a_s[t]])
    return pi_func_greedy_eps
```

Similarly, we can derive the n-step SARSA method, or **n-step SARSA** method.

Algorithm	MDP n-step SARSA Policy Iteration
Problem Type	MDP Control Problem [Model-Free]
Known	MDP environment $\mathcal{E}_D$
Objective	Optimal policy $\pi^*(a\|s)$
Algorithm Properties	value-based, on-policy, tabular

Algorithm 68: MDP n-step SARSA Policy Iteration

Input: MDP environment $\mathcal{E}_D$
Output: Optimal policy $\pi^*(a|s)$
$Q_0(s,a) \leftarrow random(A \times S)$
$N(s,a) \leftarrow 0, \forall (a,s) \in \mathcal{S} \times \mathcal{A}$
for $k \in 1, \ldots, K$ **do**
  $\epsilon \leftarrow \epsilon(k)$
  $s_0, r_0 \sim p(s_0), p(r_0)$
  for $t \in 1, 2, \ldots, L_e$ **do**
    $a_t \sim \pi_\epsilon(a; s, Q_k)$
    $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
    $T_{t-n} = r_{t-1} + \sum_{i=1}^{n} \gamma^i Q_k(s_{t-n+i}, a_{t-n+i})$
    $Q_k(s_{t-n}, a_{t-n}) \leftarrow Q_k(s_{t-n}, a_{t-n}) + \alpha(T_{t-1} - Q_k(s_{t-n}, a_{t-n}))$
$\pi^*(a|s) \leftarrow \pi_\epsilon(s, a; Q_K)$

The corresponding Python code is shown below:

```python
def MDP_SARSA_n(env:MDP, N_K, n_sarsa, epsilon, alpha, gamma=0.9, L_e=100, vmin=0, vmax=99):
    Q_mat = np.random.uniform(low=vmin, high=vmax, size=[env.Ns, env.Na])
    def pi_func_greedy_eps(s:int, epsilon:float):
        assert 0 <= s and s < Q_mat.shape[0]
        pi = np.zeros_like(Q_mat[s])
        pi[np.argmax(Q_mat[s])] = 1 - epsilon
        pi += epsilon / Q_mat.shape[0]
        return pi
    num_s = np.zeros([env.Ns, env.Na])
    for k in range(N_K):
        rs, states, a_s = sample_trail_policy(env, pi_func_greedy_eps) # related to Q !!
        for t in range(n_sarsa, L_e):
            target_old = rs[t-n_sarsa]
            for i in range(n_sarsa):
                tmp = t - n_sarsa + i
                target_old += gamma ** i * Q_mat[states[tmp], a_s[tmp]]
            Q_mat[states[t], a_s[t]] += alpha * (target_old - Q_mat[states[t], a_s[t]])
    return pi_func_greedy_eps
```

The n-step SARSA method requires selecting an appropriate step size n for the temporal difference target T_{t-n}. If n is too large or too small, it may lead to insufficient sampling, making the algorithm unstable. Then, can we simultaneously use Q-value estimates corresponding to different n values as temporal difference targets?

SARSA(λ) is such a method. It computes multiple Q-value estimates corresponding to different n values and uses a parameter $\lambda \in [0, 1]$ to control the weighting coefficients of these estimates. Specifically, the temporal difference target for SARSA(λ) is:

$$T_{t+N}^{\lambda} = (1-\lambda) \sum_{n=1}^{N} \lambda^{n-1} Q(s_{t+n}, a_{t+n}) \tag{VI.21.3}$$

Based on this, we can summarize the SARSA(λ) algorithm as follows (TODO).

21.3 Q-Learning

A major issue with SARSA is its conservative nature. Even with ϵ-greedy policy, the agent tends to rely on the learned Q-table for decision-making in later stages, making it prone to local optima.

The root cause of this problem lies in SARSA's learning strategy. In SARSA, the Q-value in the temporal difference target corresponds to the sampled future (s, a) pair. However, the future action a may not

necessarily be the optimal action in that situation. This discrepancy may cause SARSA to miss potential "high-value paths," resulting in suboptimal learning.

Q-learning addresses this issue. It still requires sampling the next state, but the learning target is the maximum Q-value among all possible actions in that state. The key difference between Q-learning and SARSA is that we modify the temporal difference target to:

$$T_{t-1} = r_{t-1} + \gamma \max_a Q_k(s_t, a) \tag{VI.21.4}$$

Therefore, we summarize the Q-learning method as follows:

Algorithm	**MDP-Q Learning Method**
Problem Type	MDP Control Problem [Model-Free]
Known	MDP Environment $\mathcal{E}_D$
Objective	Optimal Policy $\pi^*(a\|s)$
Algorithm Properties	value-based, off-policy, tabular

Algorithm 69: MDP-Q Learning Method

Input: MDP Environment $\mathcal{E}_D$
Output: Optimal Policy $\pi^*(a|s)$
$Q_0(s,a) \leftarrow random(A \times S)$
$\pi_0(a|s) \leftarrow random(A \times S)$
$N(s,a) \leftarrow 0, \forall (a,s) \in \mathcal{S} \times \mathcal{A}$
for $k \in 1, \ldots, K$ **do**
  $\epsilon \leftarrow \epsilon(k)$
  $s_0, r_0 \sim p(s_0), p(r_0)$
  for $t \in 1, 2, \ldots, L_e$ **do**
    $a_t \sim \pi_\epsilon(a; s, Q_k)$
    $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
    $T_{t-1} \leftarrow r_{t-1} + \gamma \max_a Q_k(s_t, a)$
    $Q_k(s_{t-1}, a_{t-1}) \leftarrow Q_k(s_{t-1}, a_{t-1}) + \alpha(T_{t-1} - Q_k(s_{t-1}, a_{t-1}))$
$\pi^*(a|s) \leftarrow \pi_\epsilon(s, a; Q_K)$

The corresponding Python code is shown below:

```python
def MDP_Q_learning(env:MDP, N_K, epsilon, alpha, gamma=0.9, L_e=100, vmin=0, vmax=99):
    Q_mat = np.random.uniform(low=vmin, high=vmax, size=[env.Ns, env.Na])
    def pi_func_greedy_eps(s:int, epsilon:float):
        assert 0 <= s and s < Q_mat.shape[0]
        pi = np.zeros_like(Q_mat[s])
        pi[np.argmax(Q_mat[s])] = 1 - epsilon
        pi += epsilon / Q_mat.shape[0]
        return pi
    num_s = np.zeros([env.Ns, env.Na])
    for k in range(N_K):
        rs, states, a_s = sample_trail_policy(env, pi_func_greedy_eps) # related to Q !!
        for t in range(1, L_e):
            target_last = rs[t-1] + gamma * np.max(Q_mat[states[t]])
            Q_mat[states[t-1], a_s[t-1]] += alpha * (target_last - Q_mat[states[t-1], a_s[t-1]])
    return pi_func_greedy_eps
```

It is worth noting that Q-learning is classified as an off-policy algorithm. This is to emphasize that the policy generating the sampled data differs from the learning target. In temporal difference algorithms, the learning target is precisely the temporal difference target defined earlier, which serves as the direction for algorithm updates.

The various tabular methods introduced in this chapter are intuitive and easy to understand, but they face many limitations in practical applications. Particularly in continuous spaces, they can lead to the curse

of dimensionality. Although there are deep learning-integrated tabular methods like DQN, policy gradient methods have broader applicability for robot learning. In the following chapters, we will introduce policy gradient methods.

21.4 Inverted Pendulum Solution

In the previous sections, we introduced different tabular reinforcement learning methods. We will now verify the effectiveness of these methods using the classic first-order/second-order planar inverted pendulum problem.

[]

[This section will be updated in subsequent versions. Stay tuned.]

22 Policy Gradient Methods

22.1 Policy Parameterization

In the previous chapter, we introduced several value-based deep learning methods. These methods all use approximations of value definitions or Bellman equations to estimate values, ultimately producing some form of greedy policy based on values (V or Q).

In fact, recalling the MDP control problem and the POMDP control problem, what we aim to solve is the optimal policy, which does not necessarily rely on value estimation. **Policy-based reinforcement learning methods** (policy-based RL methods) are precisely this category of reinforcement learning methods—we attempt to directly fit a policy function $\pi(a|s)$.

As shown by many algorithms in the previous chapter, $\pi(a|s)$ is a function that takes the state space as input and outputs a distribution over the action space, which can be considered as $\mathcal{S} \to \mathcal{P}(\mathcal{A})$. On one hand, it requires the output probability distribution to be normalized; on the other hand, its input and output dimensions are generally large, and $\mathcal{S}, \mathcal{A}$ may be continuous spaces. Additionally, MDP problems themselves are typically complex, making $\pi(a|s)$ a highly intricate function.

In Part V, we introduced deep learning as a powerful tool that enables us to fit high-dimensional, complex, nonlinear functions using DNNs, along with training methods based on gradient descent and backpropagation. Thus, we can use DNNs to represent the policy function $\pi(a|s)$. A DNN has a set of parameters θ, and when we use a DNN with θ to fit the policy function $\pi(a|s)$, we effectively transform the problem of solving in function space into an optimization problem over parameters. This idea is referred to as **policy parameterization**. We generally denote the parameterized policy as $\pi_\theta(a|s)$ or $\pi(a|s;\theta)$.

Using DNNs to model policies offers an additional advantage: for the challenge of continuous action/state spaces, DNNs can directly input and output continuous quantities. If we want to fit a **deterministic policy** (where a state corresponds to a unique action with probability 1), we can have the DNN output directly in the continuous action space. If we need a **stochastic policy**, we can fit a parametric distribution and then use the reparameterization trick for sampling (see Section 22.5.2 for details). This way, policy parameterization avoids the need to discretize continuous spaces, as required by most tabular methods, thereby **circumventing the curse of dimensionality**.

Under the assumption of policy parameterization, the problem of finding the optimal policy π^* becomes the problem of finding the optimal DNN parameters θ^*. Thus, we can reformulate the MDP control problem.

Problem	**MDP Control Problem [Model-Free Parameterization]**
Problem Description	MDP with unknown environment model, seeking the optimal parameterized policy
Given	MDP environment $\mathcal{E}_D$
Objective	Optimal parameters θ^* for the policy $\pi_\theta(a|s)$

In deep learning, in addition to designing the neural network architecture, we also need to design the objective function (referred to as the loss function in DL) for optimization. In reinforcement learning, this objective is related to the value function.

In Chapter 20, we provided the definition of an optimal policy (Equation VI.20.18), which states that **the optimal policy is the one whose corresponding value function is maximized in every state**. Therefore, our optimization objective should be to maximize the value function. However, the state space contains many states, and we cannot simultaneously optimize the value for every state. Thus, we consider introducing a **weight** to weight the value of each policy:

$$J(\theta) = \sum_{s \in \mathcal{S}} \eta(s) V_{\pi(\theta)}(s) \tag{VI.22.1}$$

Here, the weight $\eta(s)$ represents our emphasis on state s—the larger its value, the more we prioritize this state. Which states deserve our attention? Clearly, those that are more likely to appear in the overall space. Therefore, $\eta(s)$ can be defined as: **the probability/probability density of state s appearing at any step**. This weight $\eta(s)$ must satisfy the normalization condition, meaning the sum (or integral for continuous spaces) over the state space should equal 1:

$$\sum_{s \in \mathcal{S}} \eta(s) = 1 \tag{VI.22.2}$$

Equation VI.22.1 addresses the definition of the optimization objective for the policy DNN, but this definition has a significant flaw: **the state distribution** $\eta(s)$ **depends on the policy** π, i.e., $\eta_\theta(s)$. Different policies yield different value functions, and under different value functions, the probability of selecting a state varies, as does its overall frequency of occurrence.

To avoid this issue, we must specify a state distribution that does not change with the policy[22]. Here, we use the **initial state distribution** $p(s_0)$, which is determined by the environment $\mathcal{E}_D$. Thus, our actual optimization objective is:

$$J(\theta) = \mathbb{E}_{s_0 \sim p(s_0)}[V_{\pi(\theta)}(s_0)] \tag{VI.22.3}$$

Therefore, solving the model-free parameterized MDP control problem essentially involves finding the optimal parameters:

$$\theta^* = \arg\max_\theta \mathbb{E}_{s_0 \sim p(s_0)}[V_{\pi(\theta)}(s_0)] \tag{VI.22.4}$$

Note that unlike deep learning, where the loss function is minimized, the policy parameterization optimization problem is a **maximization problem**. Of course, we can easily convert it to a minimization problem by adding a negative sign.

22.2 Policy Gradient Theorem

In deep learning, the optimization of DNN parameters relies on computing the gradient of the loss function with respect to the parameters. To compute this gradient, we only need to know the gradient of the loss function with respect to the DNN output $\frac{\partial L}{\partial \hat{y}}$ and use backpropagation to compute $\frac{\partial \hat{y}}{\partial \Theta}$. Here, the relationship between the loss function and DNN parameters is very straightforward.

In reinforcement learning, the DNN parameters θ are used to generate a conditional probability distribution $\pi_\theta(a|s)$ over the action space, but the optimization objective is the expected value function over the initial state distribution $V_{\pi(\theta)}(s_0)$. In fact, the policy $\pi_\theta(a|s)$ influences every timestep, and the rewards at each timestep must be discounted to contribute to the state value. In other words, in RL, the relationship between the objective function and DNN parameters is more complex. We need to express this complex dependency as a solvable mathematical formula to derive $\nabla_\theta J(\theta)$, the **policy gradient**.

To connect the initial state's value function with the policy $\pi_\theta(a|s)$, we can leverage the dependencies between value functions. According to Equation VI.20.9, for a given s_0, we have:

$$\nabla_\theta V_{\pi(\theta)}(s_0) = \nabla_\theta \sum_{a_1 \in \mathcal{A}} \pi_\theta(a_1|s_0) Q_{\pi(\theta)}(s_0, a_1)$$

The above equation already includes the policy $\pi_\theta(a_1|s_0)$, which is computable. However, the subsequent $Q_{\pi(\theta)}(s_0, a_1)$ still depends on θ. Using the properties of gradients, we have:

$$\begin{aligned}
\nabla_\theta V_{\pi(\theta)}(s_0) &= \nabla_\theta \sum_{a_1 \in \mathcal{A}} \pi_\theta(a_1|s_0) Q_\theta(s_0, a_1) \\
&= \sum_{a_1 \in \mathcal{A}} (\nabla_\theta \pi_\theta(a_1|s_0)) Q_\theta(s_0, a_1) + \pi_\theta(a_1|s_0)(\nabla_\theta Q_\theta(s_0, a_1))
\end{aligned}$$

Here, for convenience, we abbreviate $V_{\pi(\theta)}$ as V_θ and $Q_{\pi(\theta)}$ as Q_θ. Note that in MDPs, both types of values depend on the specific policy, i.e., they depend on θ.

In the above equation, we transform the problem of computing the state value gradient into computing the action value gradient. According to Equation VI.20.10, we have:

[22] This is essentially a compromise

$$\nabla_\theta V_{\pi(\theta)}(s_0) = \sum_{a_1 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0, a_1) + \pi_\theta(a_1|s_0) \nabla_\theta Q_\theta(s_0, a_1)$$

$$= \sum_{a_1 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0, a_1) + \sum_{a_1 \in \mathcal{A}} \pi_\theta(a_1|s_0) \nabla_\theta (R_1 + \sum_{s' \in \mathcal{S}} p(s_1|s_0, a_1) \gamma V_\theta(s_1))$$

Note that in policy gradient methods, we assume the discount factor $\gamma = 1$. In the above equation, R and $p(s'|s, a)$ are determined by the environment and are independent of θ, so...

$$\nabla_\theta V_{\pi(\theta)}(s_0) = \sum_{a_1 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0, a_1) + \sum_{a_1 \in \mathcal{A}} \pi_\theta(a_1|s_0) \sum_{s_1 \in \mathcal{S}} p(s_1|s_0, a_1) \nabla_\theta V_\theta(s_1) \tag{VI.22.5}$$

Note that the term $\sum_{a_1 \in \mathcal{A}} \pi_\theta(a_1|s_0) p(s_1|s_0, a_1)$ represents the probability of transitioning from state s_0 to state s_1 under policy θ. Therefore, we can introduce a **state transition notation**:

$$\Pr_\theta(S_{t'} = s_{t'} | S_t = s_t)$$

This notation denotes: under the current policy, the probability that the state at time t' takes the value $s_{t'}$ given that the state at time t is s_t[23]. Thus, we have:

$$\Pr_\theta(S_1 = s_1 | S_0 = s_0) = \sum_{a_1 \in \mathcal{A}} \pi_\theta(a_1|s_0) p(s_1|s_0, a_1) \tag{VI.22.6}$$

This notation can be cascaded. For example, for $t = 0, t' = 2$, we have:

$$\Pr_\theta(S_2 = s_2 | S_0 = s_0) = \sum_{s_1 \in \mathcal{S}} \Pr_\theta(S_1 = s_1 | S_0 = s_0) \Pr_\theta(S_2 = s_2 | S_1 = s_1) \tag{VI.22.7}$$

Using this notation, we can simplify Equation VI.22.5 as:

$$\nabla_\theta V_\theta(s_0) = \sum_{a_1 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0, a_1) + \sum_{s_1 \in \mathcal{S}} \Pr_\theta(S_1 = s_1 | S_0 = s_0) \nabla_\theta V_\theta(s_1) \tag{VI.22.8}$$

It can be observed that we have essentially derived a recursive relationship for $\nabla_\theta V_\theta(s_t)$. By simply incrementing the subscripts, we can write the next step of the recursion:

$$\nabla_\theta V_\theta(s_1) = \sum_{a_2 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_2|s_1) Q_\theta(s_1, a_2) + \sum_{s_2 \in \mathcal{S}} \Pr_\theta(S_1 = s_1, S_2 = s_2) \nabla_\theta V_\theta(s_2)$$

We can summarize this general form as:

$$\nabla_\theta V_\theta(s_t) = \sum_{a_{t+1} \in \mathcal{A}} \nabla_\theta \pi_\theta(a_{t+1}|s_t) Q_\theta(s_t, a_{t+1}) + \sum_{s_t \in \mathcal{S}} \Pr_\theta(S_{t+1} = s_{t+1} | S_t = s_t) \nabla_\theta V_\theta(s_{t+1})$$

Substituting the recursive formula into Equation VI.22.8, note that the cascading property of the state transition notation can be used for simplification:

[23] Here, uppercase and lowercase letters are used to distinguish random variables from their realizations

$$
\begin{aligned}
\nabla_\theta V_\theta(s_0) &= \sum_{a_1 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0, a_1) + \sum_{s_1 \in \mathcal{S}} \Pr_\theta(S_1 = s_1 | S_0 = s_0) \nabla_\theta V_\theta(s_1) \\
&= \sum_{a_1 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0, a_1) \\
&+ \sum_{s_1 \in \mathcal{S}} \Pr_\theta(S_1 = s_1 | S_0 = s_0) \sum_{a_2 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_2|s_1) Q_\theta(s_1, a_2) \\
&+ \sum_{s_1 \in \mathcal{S}} \Pr_\theta(S_1 = s_1 | S_0 = s_0) \sum_{s_2 \in \mathcal{S}} \Pr_\theta(S_1 = s_1, S_2 = s_2) \nabla_\theta V_\theta(s_2) \\
&= \sum_{a_1 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0, a_1) \\
&+ \sum_{s_1 \in \mathcal{S}} \Pr_\theta(S_1 = s_1 | S_0 = s_0) \sum_{a_2 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_2|s_1) Q_\theta(s_1, a_2) \\
&+ \sum_{s_2 \in \mathcal{S}} \Pr_\theta(S_2 = s_2 | S_0 = s_0) \nabla_\theta V_\theta(s_2)
\end{aligned}
$$

Expanding one more step, we observe that the three terms in the expansion correspond to the state transition term, the gradient of the policy DNN, and the action-value function:

$$
\begin{aligned}
\nabla_\theta V_\theta(s_0) = &\sum_{a_1 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0, a_1) \\
&+ \sum_{s_1 \in \mathcal{S}} \Pr_\theta(S_1 = s_1 | S_0 = s_0) \sum_{a_2 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_2|s_1) Q_\theta(s_1, a_2) \\
&+ \sum_{s_2 \in \mathcal{S}} \Pr_\theta(S_2 = s_2 | S_0 = s_0) \sum_{a_3 \in \mathcal{A}} \nabla_\theta \pi_\theta(a_3|s_2) Q_\theta(s_2, a_3) \\
&+ \sum_{s_3 \in \mathcal{S}} \Pr_\theta(S_3 = s_3 | S_0 = s_0) \nabla_\theta V_\theta(s_3)
\end{aligned}
$$

Extending this time-step expansion to infinity, we obtain:

$$
\nabla_\theta V_\theta(s_0) = \sum_{k=0}^{\infty} \sum_{s_k \in \mathcal{S}} \Pr_\theta(S_k = s | S_0 = s_k) \sum_{a_{k+1} \in \mathcal{A}} \nabla_\theta \pi_\theta(a_{k+1}|s_k) Q_\theta(s_k, a_{k+1})
$$

Note that through the state transition notation, the gradient term here is no longer dependent on the time step k. Therefore, we can simplify the notation without emphasizing the specific timing of state and action random variables:

$$
\nabla_\theta V_\theta(s_0) = \sum_{k=0}^{\infty} \sum_{s \in \mathcal{S}} \Pr_\theta(S_k = s | S_0 = s_0) \sum_{a \in \mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s, a)
$$

Note the term $\sum_{k=0}^{\infty} \Pr_\theta(S_k = s | S_0 = s_0)$ in the formula. Its meaning is: given the initial state s_0, the sum of probabilities that state s appears at any time step. In fact, this is precisely the **(conditional) expected number of visits** to state s. We denote this as $\mu_\theta(s|s_0)$, where:

$$
\mu_\theta(s|s_0) = \sum_{k=0}^{\infty} \Pr_\theta(S_k = s | S_0 = s_0) \tag{VI.22.9}
$$

Thus, we have:

$$
\nabla_\theta V_\theta(s_0) = \sum_{s \in \mathcal{S}} \mu_\theta(s|s_0) \sum_{a \in \mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s, a)
$$

In the state space, the sum of the expected occurrence counts of all states equals the average episode length. For a specific policy θ and initial state s_0, this length is a constant L_m:

$$L_m = \sum_{s \in \mathcal{S}} \mu_\theta(s|s_0) \tag{VI.22.10}$$

Note that the ratio of $\mu_\theta(s|s_0)$ to L_m is precisely the "state occurrence probability" $\eta(s)$ defined in the previous section, but conditioned on the initial state:

$$\eta(s|s_0) = \frac{\mu_\theta(s|s_0)}{L_m} \tag{VI.22.11}$$

Therefore, given s_0, the gradient of the initial state's value can be further expressed as:

$$\nabla_\theta V_\theta(s_0) = L_m \sum_{s \in \mathcal{S}} \eta(s|s_0) \sum_{a \in \mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)$$

According to Equation VI.22.1, taking the expectation of the above expression yields the policy gradient formula:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{s_0 \sim p(s_0)}[L_m \sum_{s \in \mathcal{S}} \eta(s|s_0) \sum_{a \in \mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)] \\
&= L_m \mathbb{E}_{s_0 \sim p(s_0)}[\sum_{s \in \mathcal{S}} \eta(s|s_0) \sum_{a \in \mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)] \\
&= L_m \mathbb{E}_{s_0,s}\left[\sum_{a \in \mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)\right]
\end{aligned}$$

In this step, we incorporate the probability-weighted summation over states into the expectation. Note that:

$$\begin{aligned}
\nabla_\theta J(\theta) &= L_m \mathbb{E}_{s_0,s}\left[\sum_{a \in \mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)\right] \\
&= L_m \mathbb{E}_{s_0,s}\left[\sum_{a \in \mathcal{A}} \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)}{\pi_\theta(a|s)}\right] \\
&= L_m \mathbb{E}_{s_0,s,a}\left[\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} Q_\theta(s,a)\right] \\
&= L_m \mathbb{E}_{s_0,s,a}\left[\nabla_\theta \ln \pi_\theta(a|s) Q_\theta(s,a)\right]
\end{aligned}$$

Here, we employ the trick of multiplying and dividing by the same term to construct a logarithmic derivative of the policy, thereby incorporating the action a into the expectation. Simplifying, we obtain:

$$\nabla_\theta J(\theta) = E\left[\nabla_\theta \ln \pi_\theta(a|s) Q_\theta(s,a)\right] \tag{VI.22.12}$$

Equation VI.22.12 is also known as the **Policy Gradient Theorem**, which serves as the foundation for all policy gradient methods introduced in this and the next chapter. Note: In this formula, we omit the constant L_m because for each policy θ, the policy gradient is used to update the neural network. According to the gradient descent algorithm for neural networks introduced in Part V, L_m can be absorbed by the adjustable learning rate parameter α.

22.3 Original REINFORCE Algorithm

So far, we have derived an expression for the policy gradient (Equation VI.22.12). However, this expression still falls short of our goal because it involves the action-value function (i.e., the Q-function), which we cannot directly obtain. Instead, we can only receive rewards from the environment. So, how can we estimate the Q-value using rewards?

Recalling the tabular methods introduced in the previous chapter, in the Monte Carlo policy iteration algorithm for solving model-free MDP control problems (Algorithm 21.1), we adopted a Monte Carlo sampling approach. Specifically, to estimate Q for each (s_t, a_t) pair, we referred to the definition of the Q-function,

using the cumulative return G_t as a substitute for the action-value and approximating the expectation with the average of sampled trajectories.

Here, we can still apply a similar idea, namely:

$$\nabla_\theta J(\theta) = \mathbb{E}_{a \sim \pi_\theta} \left[G_t \nabla_\theta \ln \pi_\theta(a_t|s_{t-1}) \right] \tag{VI.22.13}$$

Thus, we have fully resolved the problem of gradient computation between the objective function and the parameters of the policy DNN in policy parameterization methods. Comparing this to backpropagation in deep learning, Equation VI.22.13 is analogous to $\frac{\partial L}{\partial \hat{y}}$. In deep learning, $\frac{\partial L}{\partial \hat{y}}$ bridges the network gradient $\frac{\partial \hat{y}}{\partial \theta} = \nabla_\theta NN(\cdot;\theta)$ and the loss function gradient $\frac{\partial L}{\partial \theta}$. In reinforcement learning, Equation VI.22.13 connects the policy gradient $\nabla_\theta J(\theta)$ with the network's logarithmic gradient $\nabla_\theta \ln \pi_\theta(a_t|s_{t-1})$. Although the derivation of this equation is far more complex than the chain rule in DL, the result is remarkably concise.

In Part V, when introducing the backpropagation (BP) algorithm for gradient computation in DL, we mentioned that in practice, manual implementation of BP is often unnecessary. Instead, frameworks like PyTorch can automatically compute gradients through automatic differentiation, given the forward propagation steps. In other words, as long as we implement the entire process from DNN input to output and then to the loss function, manual gradient calculation is not required.

In RL, we can similarly leverage such DL frameworks. Indeed, based on the policy gradient theorem, we can define a **pseudo** value function $\bar{J}(\theta)$, enabling us to utilize the automatic differentiation capabilities of DL frameworks:

$$\bar{J}(\theta) = \sum_{t=0}^{L_e} G_t \ln \pi_\theta(a_t|s_{t-1}) \tag{VI.22.14}$$

The term "pseudo" is used because its value does not match the definition of the value function (Equation VI.22.3), but its gradient approximates that of the true value function (differing only by a constant). The purpose of policy parameterization is to transform the MDP control problem in RL into a DL optimization problem, leveraging DL's gradient descent and other optimization methods. Optimizing the pseudo value function via gradient descent is equivalent to optimizing the true value function.

Thus, we can summarize the algorithm for solving MDP control problems based on DNN gradient descent, the policy gradient theorem, and Monte Carlo approximation. We refer to this as the **Original REINFORCE Algorithm**:

Algorithm	**Original REINFORCE Algorithm**
Problem Type	MDP Control Problem [Model-Free]
Known	MDP Environment $\mathcal{E}_D$
Objective	Optimal Parameterized Policy $\pi^*_\theta(a|s)$
Algorithm Properties	Policy-Based, On-Policy, Policy Gradient

Algorithm 70: Original REINFORCE Algorithm

```
Input: MDP environment E_D
Output: Optimal parameterized policy π*_θ(a|s)
θ ← random(Θ)
for m ∈ 1, ..., M do
    s_0, r_0 ~ p(s_0), p(r_0)
    for t ∈ 1, 2, ..., L_e do
        a_t ~ π_θ(a|s_{t-1})
        s_t, r_t ← E_D(a_t)
    for t ∈ ℒ_e, L_e − 1, ..., 1 do
        G_t ← Σ_{τ=t}^{L_e} r_τ
    J̄(θ) = (1/L_e) Σ_{t=0}^{L_e} G_t ln π_θ(a_t|s_{t-1})
    θ ← θ + α∇_θ J̄(θ)
π*(a|s) ← π_θ(a|s)
```

Below, we present the Python implementation of the aforementioned original REINFORCE algorithm. We maintain the assumptions from the previous chapter, where $\mathcal{S}, \mathcal{A}$ are discrete state spaces, with states and actions represented by integers. We use the MLP introduced in Section 18 as the policy DNN $\pi_\theta(a|s)$, whose input is a $|\mathcal{S}|$-dimensional one-hot vector.

Combining the MLP's BSGD optimization algorithm (Algorithm 50), the Python code for the above algorithm is as follows:

```
def sample_trail_policy_NN(env:MDP, piMLP:MLP):
    states, rs, a_s = [env.init_state()], [env.reward(states[0])], [None]
    for t in range(1, L_e+1):
        piMLP.zero_grads()
        s_in = np.eye(env.Ns)[states[t-1]] # one-hot
        pi_a = piMLP.forward(s_in)
        a_s.append(np.random.choice(env.Na, p=pi_a))
        states.append(env.state_transfer(states[t-1], a_s[t]))
        rs.append(env.reward(states[t]))
    return rs, states, a_s
def REINFORCE_og(env:MDP, batch_size:int, N_S:int=10, N_M:int=100, alpha=1e-4, L_e=100, W_NN=100):
    piMLP = MLP(4, [env.Ns, W_NN, W_NN, env.Na])
    def pi_func_mlp(s:int):
        s_in = np.eye(env.Ns)[s]
        return piMLP.forward(s_in)
    for k in range(N_M):
        xs, allGs, alla_s = [], [], []
        for _ in N_S:
            rs, states, a_s = sample_trail_policy_NN(env, piMLP)
            Gs = [0]
            for t in range(1, L_e):
                Gs.append(Gs[t-1] + rs[t])
                xs.append(states[t-1]), allGs.append(Gs[t]), alla_s.append(a_s[t])
        N, N_b = len(xs), len(xs) // batch_size
        ids = np.random.shuffle(np.arange(N))
        for b in range(N_b):
            mlp.zero_grad()
            b_ids = ids[b*batch_size:(b+1)*batch_size]
            b_J, b_xs, b_Gs, b_as = 0, xs[b_ids], allGs[b_ids], alla_s[b_ids]
            for i in range(batch_size):
                pi_est = piMLP.forward(np.eye(env.Ns)[b_xs[i]])
                b_J += b_Gs[i] * np.log(pi_est[b_as[i]])
                dJdy = - b_Gs[i] / pi_est[b_as[i]] # add "-" to gradient acsend
                piMLP.backward(dJdy)
            if b_J > J_target:
                piMLP.zero_grad()
                return pi_func_mlp
            for p, l in zip(mlp.params, range(1, piMLP.L+1)):
                piMLP.param[p][l] = GD_update(piMLP.param[p][l], piMLP.grads[p][l], alpha)
    return pi_func_mlp
```

In MC policy iteration (Algorithm 21.1), we also employed the ϵ-greedy strategy to balance exploration and exploitation. This is because the greedy strategy in tabular methods is deterministic and not conducive to exploration. However, when using policy gradient methods, the output of the policy network is the action distribution, which inherently ensures randomness during sampling, eliminating the need for additional randomization.

22.4 REINFORCE Algorithm

The original REINFORCE algorithm can already be used for optimizing parameterized policies in MDPs. However, this algorithm contains a paradox.

Consider the following scenario: Suppose an environment has three possible actions A, B, C, and all states and actions yield positive rewards (e.g., 1-100). In one sampling round, only actions A and B are sampled, each accounting for half. The cumulative rewards for action A are relatively low (e.g., no more than 5), while those for action B are relatively high (e.g., no less than 80). After one update, how will the sampling probabilities of the three actions change?

According to the original REINFORCE algorithm, the selection probabilities of both actions A and B will increase because their corresponding G_t values are positive. Since B's rewards are higher than A's and their proportions are equal, the probability increase for B will be greater than for A. As the total probability must sum to 1, the selection probability of action C will decrease. This implies that the model believes the utility of the three actions should be $B > A > C$.

However, is action C truly worse than action A? In reality, the cumulative rewards for C are likely between those of A and B (i.e., 5-80), meaning $B > C > A$. The erroneous update favoring A over C arises solely due to biased sampling.

The paradox hinges on two key factors: **insufficient sampling** and **consistently positive rewards**. If more samples were collected and action C had sufficient representation, the network would naturally increase C's selection probability. However, insufficient sampling is unavoidable in the early stages of the algorithm. In contrast, we can address the issue of consistently positive rewards by adjusting the average reward of all actions to around zero, which is the essence of the **baseline trick**. We define

$$A(a_t, s_t) = G_t - b \tag{VI.22.15}$$

where b is the baseline, representing the average of all current G_t values. We can perform incremental updates:

$$\begin{aligned} b_k &= \frac{1}{L_e} \sum_{t=0}^{L_e} G_t \\ b &\leftarrow b + \frac{1}{m}(b_k - b) \end{aligned} \tag{VI.22.16}$$

Now, the policy gradient and pseudo-value function become:

$$\nabla_\theta J(\theta) = \mathbb{E}_{a \sim \pi_\theta} [A(a_t, s_t) \nabla_\theta \ln \pi_\theta(a_t | s_{t-1})] \tag{VI.22.17}$$

$$J(\theta) = \sum_{t=0}^{L_e} A(a_t, s_t) \pi_\theta(a_t | s_t) \tag{VI.22.18}$$

Here, the introduced $A(a_t, s_t)$ is called the **advantage function**. It represents the advantage of taking a specific action a_t in state s_t **relative to other actions** under the given policy. With the baseline trick, the advantage function can reflect the relative advantages between actions regardless of whether the cumulative rewards G_t are consistently positive or negative, thereby avoiding the paradox of the original REINFORCE algorithm.

Note that the advantage function A, like the state-value function V and action-value function Q, depends on a specific policy. There is no universal advantage function applicable to all policies. Additionally, $A(a_t, s_t)$ is independent of the parameters θ, and we do not compute gradients for it.

Thus, we can summarize the **REINFORCE algorithm**.

Algorithm	**REINFORCE Algorithm**
Problem Type	MDP Control Problem [Model-Free]
Given	MDP Environment $\mathcal{E}_D$
Objective	Optimal Parameterized Policy $\pi_\theta^*(a\|s)$
Algorithm Properties	Policy-based, On-policy, Policy Gradient

Algorithm 71: REINFORCE Algorithm

Input: MDP Environment $\mathcal{E}_D$
Output: Optimal Parameterized Policy $\pi^*_\theta(a|s)$
$\theta \leftarrow random(\Theta)$
$b \leftarrow 0$
for $m \in 1, \ldots, M$ **do**
  for $k \in 1, \ldots, K$ **do**
    $s_0, r_0 \sim p(s_0), p(r_0)$
    for $t \in 1, 2, \ldots, L_e$ **do**
      $a_t \sim \pi_\theta(a|s_{t-1})$
      $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
    for $t \in \mathcal{L}_e, L_e - 1, ..., 1$ **do**
      $G_t \leftarrow \sum_{\tau=t}^{L_e} r_\tau$
      $A(a_t, s_t) \leftarrow G_t - b$
    $b_k \leftarrow \frac{1}{L_e} \sum_{t=0}^{L_e} G_t$
    $b \leftarrow b + \frac{1}{m}(b_k - b)$
  $\bar{J}(\theta) = \frac{1}{KL_e} \sum_{k=0}^{K} \sum_{t=0}^{L_e} A(a_t, s_t) \ln \pi_\theta(a_t|s_t)$
  $\theta \leftarrow \theta + \alpha \nabla_\theta \bar{J}(\theta)$
$\pi^*(a|s) \leftarrow \pi_\theta(a|s)$

The corresponding Python code is shown below (partial content omitted same as original REINFORCE algorithm)

```
def REINFORCE(env:MDP, batch_size:int, N_S:int=10, N_M:int=100, alpha=1e-4, L_e=100, W_NN=100):
    ......
    for m in range(N_M):
        xs, allGs, alla_s = [], [], []
        for _ in N_S:
            ......
        baseline = np.mean(allGs) # add baseline calculation
        ......
        for b in range(N_b):
            ......
            for i in range(batch_size):
                pi_est = piMLP.forward(np.eye(env.Ns)[b_xs[i]])
                b_J += (b_Gs[i] - baseline) * np.log(pi_est[b_as[i]]) # <-- here is different
                dJdy = - (b_Gs[i] - baseline) / pi_est[b_as[i]] # <-- here is different
                piMLP.backward(dJdy)
            ......
    return pi_func_mlp
```

22.5 Inverted Pendulum Solution

Similar to tabular methods, policy gradient methods can also be applied to solve the inverted pendulum problem (see Section 10.6 for details).

22.5.1 Discrete Action Solution

[This section will be updated in a future version. Stay tuned.]

22.5.2 Continuous Action Solution

Unlike tabular methods, policy gradient methods can not only fit distributions for discrete actions but also directly output probability distributions for continuous actions.

Specifically, in the inverted pendulum problem, we want the policy network $\pi_\theta(a|s)$ to fit a continuous conditional probability distribution $p(a|s)$. Here, the state represents the continuous state of the inverted pendulum mentioned above—X-dimensional for a first-order inverted pendulum and Y-dimensional for a second-order inverted pendulum—while the action is a continuous one-dimensional quantity, namely the lateral force F. In practice, we only need to model the probability distribution as a Gaussian distribution:

$$\pi_\theta(a|s) = \mathcal{N}(\mu_{a;\theta}, \sigma^2_{a;\theta}) \tag{VI.22.19}$$

where $\mu_{a;\theta,s}$ and $\sigma^2_{a;\theta,s}$ represent the mean and variance of the continuous action a as a random variable, respectively. Both quantities depend on the network parameters and the state. In other words, we only need to construct a policy neural network whose input is the system state and whose output dimension is twice the action dimension, representing the mean and variance of the action, to complete the policy parameterization.

Under this setup, the network's loss function also undergoes a minor modification. We need to express $\ln \pi_\theta(a|s)$ in the loss function as an explicit function of the network's direct outputs μ_a and σ^2_a.

[This section will be updated in a future version. Stay tuned.]

23 PPO Algorithm

For the discussion in this chapter, unless otherwise specified, the MDP is assumed to have a discrete action space by default.

23.1 Importance Sampling

The REINFORCE algorithm mentioned earlier uses the policy gradient theorem (along with some other techniques) to continuously sample from the environment and update the policy network, making it a decent deep reinforcement learning algorithm. However, this algorithm requires that after each batch of trajectories is sampled, the network can only be updated once. Even in simulated environments, the sampling rate is generally slower than network updates. As a result, the overall efficiency of the algorithm is reduced.

Why can't REINFORCE reuse the same batch of data for multiple updates? This is because the updated network (denoted as π_θ) is no longer the network that generated the sampled data (denoted as $\pi_{\theta'}$), but a similar yet distinct network. Forcing an update with this data would violate the assumptions of the policy gradient theorem, preventing the network from converging.

23.1.1 Importance Weight

Specifically, let us consider a more general scenario. Suppose we have data drawn from a distribution $q(x)$, which can be thought of as the network $\pi_{\theta'}$ from the previous batch sampling. There is another distribution $p(x)$, similar to but different from $q(x)$, representing the updated network π_θ. There is also a sample-dependent function $f(x)$, which corresponds to the policy gradient $\nabla_\theta \bar{J}(\theta)$. The problem now is: we want to compute $\mathbb{E}_{x\sim p(x)}[f(x)]$, but we can only sample from $q(x)$.

In fact, we can introduce an importance weight to adjust the "importance" of each sampled sample. For a given sample, if $p(x)$ is large while $q(x)$ is small, its weight should be increased. Conversely, if $p(x)$ is small while $q(x)$ is large, its weight should be reduced. This is the essence of **importance sampling**:

$$\mathbb{E}_{x\sim p}[f(x)] = \mathbb{E}_{x\sim q}\left[\frac{p(x)}{q(x)}f(x)\right] \tag{VI.23.1}$$

It is important to note that while the above relationship holds for any two distributions $p(x)$ and $q(x)$, we should not use two vastly different distributions to approximate each other. This is because, when we compute the variance, we find:

$$\begin{aligned}
\mathrm{Var}_{x\sim p}[f(x)] &= \mathbb{E}_{x\sim p}\left[f(x)^2\right] - \left(\mathbb{E}_{x\sim p}[f(x)]\right)^2 \\
\mathrm{Var}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right] &= \mathbb{E}_{x\sim q}\left[\left(f(x)\frac{p(x)}{q(x)}\right)^2\right] - \left(\mathbb{E}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right]\right)^2 \\
&= \mathbb{E}_{x\sim p}\left[f(x)^2\frac{p(x)}{q(x)}\right] - \left(\mathbb{E}_{x\sim p}[f(x)]\right)^2
\end{aligned} \tag{VI.23.2}$$

In other words, there is a discrepancy between $\mathrm{Var}_{x\sim p}[f(x)]$ and $\mathrm{Var}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$. The greater the difference between $p(x)$ and $q(x)$, the larger the variance gap. When sampling is insufficient, the error in approximating the expectation also increases.

23.1.2 Value of Importance Sampling

Returning to the REINFORCE network update problem, as mentioned earlier, $q(x)$ corresponds to $\pi_{\theta'}$, $p(x)$ corresponds to the updated network π_θ, and $f(x)$ corresponds to the policy gradient $\nabla_\theta \bar{J}(\theta)$. Therefore, the policy gradient under importance sampling can be written as:

$$\nabla_\theta J(\theta;\theta') = \mathbb{E}_{a\sim\pi_{\theta'}}\left[\frac{\Pr_\theta(a_t, s_{t-1})}{\Pr_{\theta'}(a_t, s_{t-1})}A(a_t, s_{t-1})\nabla_\theta \ln \pi_\theta(a_t|s_{t-1})\right]$$

Further, we have:

$$
\begin{aligned}
\frac{\Pr_\theta(a_t, s_{t-1})}{\Pr_{\theta'}(a_t, s_{t-1})} &= \frac{\pi_\theta(a_t|s_{t-1})\Pr_\theta(s_{t-1})}{\pi_{\theta'}(a_t|s_{t-1})\Pr_{\theta'}(s_{t-1})} \\
&\approx \frac{\pi_\theta(a_t|s_{t-1})}{\pi_{\theta'}(a_t|s_{t-1})}
\end{aligned}
$$

In this step, we ignore the difference between $Pr_\theta(s_{t-1})$ and $Pr_{\theta'}(s_{t-1})$. We approximate that the slight difference in state distribution does not significantly affect the policy gradient. Thus, we obtain:

$$
\nabla_\theta J(\theta;\theta') \approx \mathbb{E}_{a\sim\pi_{\theta'}}\left[\frac{\pi_\theta(a_t|s_{t-1})}{\pi_{\theta'}(a_t|s_{t-1})}A(a_t, s_{t-1})\nabla_\theta \ln \pi_\theta(a_t|s_{t-1})\right]
$$

Using the log-derivative trick again, we can express the pseudo-value function under importance sampling as:

$$
\bar{J}(\theta;\theta') = \mathbb{E}_{a\sim\pi_{\theta'}}\left[\frac{\pi_\theta(a_t|s_{t-1})}{\pi_{\theta'}(a_t|s_{t-1})}A(a_t, s_{t-1})\right] \tag{VI.23.3}
$$

In the previous section, we mentioned that importance sampling requires the two distributions to be as similar as possible. Here, we add a penalty term to the value function to constrain the old distribution $\pi_{\theta'}(a_t|s_{t-1})$ and the new distribution $\pi_\theta(a_t|s_{t-1})$ to remain similar. We choose the KL divergence as this penalty term. Thus, we have the value function under importance sampling:

$$
J_{IS}(\theta;\theta') = \mathbb{E}_{a\sim\pi_{\theta'}}\left[\frac{\pi_\theta(a_t|s_{t-1})}{\pi_{\theta'}(a_t|s_{t-1})}A(a_t, s_{t-1})\right] - \mathrm{KL}(\pi_\theta||\pi_{\theta'}) \tag{VI.23.4}
$$

The larger the KL divergence, the greater the difference between the two distributions, which reduces the total value function and inhibits parameter updates in that direction.

23.1.3 Approximate Calculation of KL Divergence

For two discrete distributions defined on the same space, the KL divergence is defined as:

$$
\mathrm{KL}(q||p) = \sum_x q(x)\ln\frac{q(x)}{p(x)} \tag{VI.23.5}
$$

It can be seen that calculating the KL divergence requires sampling all x in the space and computing the probability values for both distributions.

However, in practice, our sampling is limited and cannot cover the entire probability space. In such cases, the following three methods are generally used to approximate the KL divergence:

$$
\mathrm{KL}(q||p) = \mathbb{E}[k1] = \mathbb{E}_{x\sim q}\left[\ln\frac{q(x)}{p(x)}\right]
$$

The $k1$ estimator is unbiased but has high variance.

$$
\mathrm{KL}(q||p) \approx \mathbb{E}[k2] = \mathbb{E}_{x\sim q}\left[\frac{1}{2}\left(\ln\frac{q(x)}{p(x)}\right)^2\right]
$$

The $k2$ estimator has lower variance but is biased.

$$
\mathrm{KL}(q||p) = \mathbb{E}[k3] = \mathbb{E}_{x\sim q}\left[\frac{p(x)}{q(x)} - 1 + \ln\frac{q(x)}{p(x)}\right] \tag{VI.23.6}
$$

The $k3$ estimator has low variance and is unbiased. Note that the denominator in the first term is $p(x)$, while elsewhere it is $q(x)$.

In PPO, we generally use the $k3$ estimator to approximate the KL divergence.

23.2 Critic Network

23.2.1 Advantage Function

As discussed in the previous chapter, the advantage function represents "the advantage of taking a specific action over other actions in a given state under a certain policy." In the previous chapter, we constructed the advantage function by subtracting a baseline from the reward. Is there a better way to construct the advantage function?

Recalling the definitions of the state value function $V_\pi(s)$ and the action value function $Q_\pi(s,a)$, we observe that the action value represents the advantage of choosing a specific action in a given state, while the state value function represents the advantage of choosing that state over others. The state value function is the average of the action value function (see Equation VI.20.9). Therefore, $Q_\pi(s,a) - V_\pi(s)$ is a good estimate of "the advantage of a specific action over others."

$$A_\pi(s,a) := Q_\pi(s,a) - V_\pi(s) \tag{VI.23.7}$$

In fact, this is the definition of the advantage function, and the earlier introduction was an estimate of this function.

Recalling the previous section, the G_t in the original REINFORCE algorithm can be replaced with the advantage function. If we can obtain the values of $V_\pi(s)$ and $Q_\pi(s,a)$, we can derive a better version of the REINFORCE algorithm.

How can we estimate $Q_\pi(s,a) - V_\pi(s)$? Suppose we collect many state-reward-action sequences $SARS(t)$ under policy π. From Equation VI.20.9, we can transform the Q-function into the V-function of the next step.

$$Q_\pi(s,a) = \mathbb{E}_{\pi,s,a}[r_{t+1} + \gamma V_\pi(s_{t+1})]$$

The subscript $\mathbb{E}_{\pi,s,a}$ indicates that each sequence $SARS(t)$ is collected under policy π, with $s_t = s$ and $a_{t+1} = a$.

Thus, the estimation of the advantage function becomes:

$$A_\pi(s,a) = \mathbb{E}_{\pi,s,a}[r_{t+1} + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)]$$

In practical computation, for each pair (s_t, a_{t+1}), we can directly use $V_\pi(s_t)$ and $V_\pi(s_{t+1})$ to estimate $A_\pi(s,a)$. This simplifies the calculation for each data point. When a batch of data is used to update the policy network collectively, it achieves the effect of expectation estimation.

23.2.2 Value Network

Given the advantage function, how do we determine $V_\pi(s)$? Similar to the parameterization of the policy, we also parameterize the state-value function using a neural network to approximate $V_\pi(s)$, referred to as the **value network**.

Assume the neural network parameters are ω, denoted as $V_\pi(s;\omega)$. If the value function corresponds to a policy network π_θ, we can also denote the value network as $V_\theta(s;\omega)$. Note: The subscript in this notation does not represent the parameters of the value network itself but rather the parameters of the corresponding policy network.

How do we train the value network? Note that this is a prediction problem for MDP (Problem 20.5), and we can borrow methods from previous solutions to prediction problems. For example, we can use an approach similar to TD(0) policy evaluation, constructing temporal difference targets for bootstrapped iteration. Specifically, we can define the following loss function using the L2 norm:

$$\mathcal{L}(\omega) = \frac{1}{2}(r + \gamma V_\pi(s_{t+1};\omega) - V_\pi(s_t;\omega))^2 \tag{VI.23.8}$$

Note that both terms in this loss function are outputs of the policy network. During actual training, we do not compute gradients for $V_\pi(s_{t+1};\omega)$ but only for $V_\pi(s_t;\omega)$, i.e.,

$$\nabla_\omega \mathcal{L}(\omega) = (r_t + \gamma V_\pi(s_{t+1};\omega) - V_\pi(s_t;\omega))\nabla_\omega V_\pi(s_t;\omega)$$

Observe that the error term of the value network here is identical in form to the estimation of the advantage function. We can denote it as $\delta_\omega(t)$:

$$\delta_\omega(t) := r_t + \gamma V_\pi(s_{t+1}; \omega) - V_\pi(s_t; \omega) \tag{VI.23.9}$$

Thus, we have:

$$\nabla_\omega \mathcal{L}(\omega) = \delta_\omega(t) \nabla_\omega V_\pi(s_t; \omega) \tag{VI.23.10}$$

and

$$\hat{A}_\pi(s_t, a_{t+1}) = \delta_\omega(t)$$

Additionally, note that we define a loss function for the policy network, not a value function. The loss function is minimized, so gradient descent rather than gradient ascent should be used during training.

Although the policy network and value network serve different purposes, their input is the same state s. From a network architecture perspective, their difference lies in the output: the policy network outputs a distribution of actions, while the value network outputs a scalar value. Therefore, in practical engineering, the first few layers of the policy network and value network are often shared, and the parameters of these layers are updated during the training of both networks. This is equivalent to adding a policy head and a value head to the same state feature extraction network, training them alternately with different loss functions. However, in the following discussion, we will still describe the policy network parameters θ and value network parameters ω separately.

23.2.3 Actor-Critic Algorithm

To summarize, to parameterize the policy, we define the policy network $\pi_\theta(a|s)$. Next, to provide the advantage function for the policy network, we define the value network $V_\theta(s; \omega)$. To obtain the policy network, the value network must be trained while the policy network is frozen. This results in a reinforcement learning algorithm where two neural networks are trained alternately, known as the **Actor-Critic algorithm**.

In this name, the Actor corresponds to the policy network. The environment is like a stage, and the policy network is the actor practicing on it. The Critic corresponds to the value network, evaluating the value function under the current policy, akin to a critic reviewing the actor's performance. The actor improves based on the critic's feedback, and the critic's evaluations evolve as the actor changes. This is essentially the **policy iteration** approach introduced in Chapter 3.

Next, we summarize the Actor-Critic algorithm in detail.

Algorithm	**Actor-Critic Algorithm**
Problem Type	MDP Control Problem [Model-Free]
Known	MDP Environment $\mathcal{E}_D$
Objective	Optimal Parameterized Policy $\pi^*_\theta(a|s)$
Algorithm Properties	Policy-Based, On-Policy, Policy Gradient

Algorithm 72: Actor-Critic Algorithm

Input: MDP Environment $\mathcal{E}_D$
Output: Optimal Parameterized Policy $\pi_\theta^*(a|s)$
$\theta \leftarrow random(\Theta)$
$\omega \leftarrow random(\Omega)$
for $m \in 1, \ldots, M$ **do**
  for $k \in 1, \ldots, K$ **do**
    $s_0, r_0 \sim p(s_0), p(r_0)$
    for $t \in 1, 2, \ldots, L_e$ **do**
      $a_t \sim \pi_\theta(a|s_{t-1})$
      $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
    $V_\pi(s_{L_e+1}; \omega) \leftarrow 0$
    for $t \in \mathcal{L}_e, L_e - 1, ..., 1$ **do**
      $\delta_\omega(t) \leftarrow r_t + \gamma V_\pi(s_{t+1}; \omega) - V_\pi(s_t; \omega)$
      $\hat{A}_\theta(a_t, s_t) \leftarrow \delta_\omega(t)$
      $\nabla_\omega \mathcal{L}(a_t, s_t) \leftarrow \delta_\omega(t) \nabla_\omega V_\pi(s_t; \omega)$
  $\nabla_\omega \mathcal{L}(\omega) \leftarrow \frac{1}{KL_e} \sum_{k=0}^{K} \sum_{t=0}^{L_e} \nabla_\omega \mathcal{L}(a_t, s_t)$
  $\omega \leftarrow \omega - \alpha_\omega \nabla_\omega \mathcal{L}(\omega)$
  $\bar{J}(\theta) = \frac{1}{KL_e} \sum_{k=0}^{K} \sum_{t=0}^{L_e} \hat{A}_\theta(a_t, s_t) \ln \pi_\theta(a_t|s_t)$
  $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \bar{J}(\theta)$
$\pi^*(a|s) \leftarrow \pi_\theta(a|s)$

23.2.4 Generalized Advantage Estimation (GAE)

In the above Actor-Critic algorithm, we simply used $\delta_\omega(t)$ to estimate the advantage function $A_\pi(s_t, a_{t+1})$. This is a single-step expansion and a rough estimate; with insufficient sampling, the bias can be significant.

To better estimate the advantage function, we can borrow ideas from the $TD(0) \rightarrow TD(n)$ process and perform multi-step expansion, i.e.,

$$\hat{A}_\pi^{(n)}(s_t, a_{t+1}) = -V_\pi(s_t; \omega) + \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n V_\pi(s_{t+1}; \omega)$$

The Generalized Advantage Estimation (GAE) algorithm posits that higher expansion orders lead to larger variance but smaller bias in estimation. Therefore, it is best to use a weighted average of different orders to reduce variance while ensuring bias remains small.

$$\hat{A}_\pi^{GAE}(s_t, a_{t+1}) = (1 - \lambda) \sum_{j=1}^{n} \hat{A}_\pi^{(j)}(s_t, a_{t+1})$$

In fact, if expressed using $\delta_\omega(t)$, it has a more concise form (when n is large, the smaller last term $\gamma^n V_\pi(s_{t+1}; \omega)$ can be omitted):

$$\hat{A}_\pi^{GAE}(s_t, a_{t+1}) = \sum_{t=0}^{n} (\gamma\lambda)^t \delta_\omega(t)$$

Thus, we can write the GAE version of the Actor-Critic algorithm:

Algorithm	**Actor-Critic Algorithm-GAE**
Problem Type	MDP Control Problem [Model-Free]
Known	MDP Environment $\mathcal{E}_D$
Objective	Optimal Parameterized Policy $\pi_\theta^*(a\|s)$
Algorithm Properties	policy-based, on-policy, policy gradient

Algorithm 73: Actor-Critic Algorithm-GAE

Input: MDP Environment $\mathcal{E}_D$
Output: Optimal Parameterized Policy $\pi_\theta^*(a|s)$
$\theta \leftarrow random(\Theta)$
$\omega \leftarrow random(\Omega)$
for $m \in 1, \ldots, M$ **do**
  for $k \in 1, \ldots, K$ **do**
    $s_0, r_0 \sim p(s_0), p(r_0)$
    for $t \in 1, 2, \ldots, L_e$ **do**
      $a_t \sim \pi_\theta(a|s_{t-1})$
      $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
    $\hat{A}_{\theta'}^{GAE}(s_{L_e}, a_{L_e+1}) \leftarrow 0$
    $V_\pi(s_{L_e+1}; \omega) \leftarrow 0$
    for $t \in \mathcal{L}_e - 1, ..., 0$ **do**
      $\delta_\omega(t) \leftarrow r_t + \gamma V_\pi(s_{t+1}; \omega) - V_\pi(s_t; \omega)$
      $\hat{A}_{\theta'}^{GAE}(s_t, a_{t+1}) \leftarrow \delta_\omega(t) + \gamma\lambda \hat{A}_{\theta'}^{GAE}(s_{t+1}, a_{t+2})$
      $\nabla_\omega \mathcal{L}(a_t, s_t) \leftarrow \delta_\omega(t) \nabla_\omega V_\pi(s_t; \omega)$
  $\nabla_\omega \mathcal{L}(\omega) \leftarrow \frac{1}{KL_e} \sum_{k=0}^{K} \sum_{t=0}^{L_e} \nabla_\omega \mathcal{L}(a_{t+1}, s_t)$
  $\omega \leftarrow \omega - \alpha_\omega \nabla_\omega \mathcal{L}(\omega)$
  $\bar{J}(\theta) = \frac{1}{KL_e} \sum_{k=0}^{K} \sum_{t=0}^{L_e} \hat{A}_{\theta'}^{GAE}(s_t, a_{t+1}) \ln \pi_\theta(a_{t+1}|s_t)$
  $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \bar{J}(\theta)$
$\pi^*(a|s) \leftarrow \pi_\theta(a|s)$

23.3 PPO Algorithm

23.3.1 Vanilla PPO Algorithm

In the previous two sections, we discussed the REINFORCE algorithm with importance sampling through multi-round update sampling, introduced the Critic network, and presented the Actor-Critic algorithm under generalized advantage estimation.

By integrating these methods to achieve multi-round gradient ascent/descent updates for the policy neural network, we obtain the vanilla PPO (Proximal Policy Optimization) algorithm.

Algorithm	**Vanilla PPO Algorithm**
Problem Type	MDP Control Problem [Model-Free]
Known	MDP Environment $\mathcal{E}_D$
Objective	Optimal Parameterized Policy $\pi_\theta^*(a\|s)$
Algorithm Properties	Policy-Based, On-Policy, Policy Gradient

Algorithm 74: Vanilla PPO Algorithm

Input: MDP Environment $\mathcal{E}_D$
Output: Optimal Parameterized Policy $\pi_\theta^*(a|s)$
$\theta \leftarrow random(\Theta)$
$\omega \leftarrow random(\Omega)$
for $m \in 1, \ldots, M$ **do**
  for $k \in 1, \ldots, K$ **do**
    $s_0, r_0 \sim p(s_0), p(r_0)$
    for $t \in 1, 2, \ldots, L_e$ **do**
      $a_t \sim \pi_{\theta'}(a|s_{t-1})$
      $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
    $\hat{A}_{\theta'}^{GAE}(s_{L_e}, a_{L_e+1}) \leftarrow 0$
    $V_\pi(s_{L_e+1}; \omega) \leftarrow 0$
    for $t \in \mathcal{L}_e - 1, ..., 0$ **do**
      $\delta_\omega(t) \leftarrow r_t + \gamma V_{\theta'}(s_{t+1}; \omega) - V_{\theta'}(s_t; \omega)$
      $\hat{A}_{\theta'}^{GAE}(s_t, a_{t+1}) \leftarrow \delta_\omega(t) + \gamma\lambda \hat{A}_{\theta'}^{GAE}(s_{t+1}, a_{t+2})$
      $\nabla_\omega \mathcal{L}(a_t, s_t) \leftarrow \delta_\omega(t) \nabla_\omega V_{\theta'}(s_t; \omega)$
      $r_{IS}(a_{t+1}, s_t) \leftarrow \frac{\pi_\theta(a_{t+1}|s_t)}{\pi_{\theta'}(a_{t+1}|s_t)}$
  for $b \in 1, 2, \ldots, B_\theta$ **do**
    Randomly sample N_1 Trails from K Trails.
    $\bar{J}(\theta; \theta') = \frac{1}{N_1 L_e} \sum_{i=0}^{N_1} \sum_{t=0}^{L_e} r_{IS}(a_{t+1}, s_t) \hat{A}_{\theta'}^{GAE}(s_t, a_{t+1})$
    $\hat{\mathrm{KL}}(\pi_\theta || \pi_{\theta'}) \leftarrow \frac{1}{N_1 L_e} \sum_{i=0}^{N_1} \sum_{t=0}^{L_e} [r_{IS}(a_{t+1}, s_t) - 1 - \ln r_{IS}(a_{t+1}, s_t)]$
    $J_{PPO}(\theta; \theta') \leftarrow \bar{J}(\theta; \theta') - \beta \hat{\mathrm{KL}}(\pi_\theta || \pi_{\theta'})$
    $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J_{PPO}(\theta; \theta')$
  for $b \in 1, 2, \ldots, B_\omega$ **do**
    Randomly sample N_2 Trails from K Trails.
    $\nabla_\omega \mathcal{L}(\omega) \leftarrow \frac{1}{N_2 L_e} \sum_{i=0}^{N_2} \sum_{t=0}^{L_e} \nabla_\omega \mathcal{L}(a_{t+1}, s_t)$
    $\omega \leftarrow \omega - \alpha_\omega \nabla_\omega \mathcal{L}(\omega)$
  $\theta' \leftarrow \theta$
$\pi^*(a|s) \leftarrow \pi_\theta(a|s)$

Here, we introduce the hyperparameter β to balance the pseudo-value function and KL divergence in the policy network's objective. Currently, this parameter can be manually specified, just like other hyperparameters such as $\gamma, \alpha_\omega, \alpha_\theta$. In the next section, we will introduce a better update strategy.

Note that although the sampling network and the learning network are two distinct networks in PPO, the KL divergence constraint ensures their outputs do not diverge significantly. Therefore, we still consider it an on-policy method.

Similarly, although the value network $V_{\theta'}(s; \omega)$ corresponds to different policy networks $\pi_{\theta'}$ in each major iteration, the limited update size between consecutive policy networks ensures their value functions remain approximately equivalent. Thus, for each major iteration, the value network can continue training from the previous round's parameters rather than being reinitialized.

23.3.2 PPO-Penalty Algorithm

As mentioned earlier, the KL divergence penalty constrains the policy network from making excessively large updates, with the hyperparameter β balancing this penalty against the pseudo-value function objective. However, determining this parameter is non-trivial. If β is too large, the network becomes overly conservative and reluctant to update; if β is too small, the network behaves too aggressively, causing sampling issues.

To address this, we can adopt an adaptive β adjustment method. We specify a KL divergence range $[\mathrm{KL}_{\min}, \mathrm{KL}_{\max}]$. If the estimated KL divergence exceeds the upper bound, we increase β; if it falls below the lower bound, we decrease β. This approach essentially implements a hysteresis loop, similar to hysteresis comparator circuits or hysteresis control.

Thus, we introduce the **PPO-Penalty Algorithm**.

Algorithm 75: PPO-Penalty Algorithm

Input: MDP environment $\mathcal{E}_D$
Output: Optimal parameterized policy $\pi_\theta^*(a|s)$
$\theta \leftarrow random(\Theta)$
$\omega \leftarrow random(\Omega)$
$\beta \leftarrow \beta_0$
for $m \in 1, \ldots, M$ **do**
  for $k \in 1, \ldots, K$ **do**
    $s_0, r_0 \sim p(s_0), p(r_0)$
    for $t \in 1, 2, \ldots, L_e$ **do**
      $a_t \sim \pi_{\theta'}(a|s_{t-1})$
      $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
    $\hat{A}_{\theta'}^{GAE}(s_{L_e}, a_{L_e+1}) \leftarrow 0$
    $V_\pi(s_{L_e+1}; \omega) \leftarrow 0$
    for $t \in \mathcal{L}_e - 1, ..., 0$ **do**
      $\delta_\omega(t) \leftarrow r_t + \gamma V_{\theta'}(s_{t+1}; \omega) - V_{\theta'}(s_t; \omega)$
      $\hat{A}_{\theta'}^{GAE}(s_t, a_{t+1}) \leftarrow \delta_\omega(t) + \gamma\lambda \hat{A}_{\theta'}^{GAE}(s_{t+1}, a_{t+2})$
      $\nabla_\omega \mathcal{L}(a_t, s_t) \leftarrow \delta_\omega(t) \nabla_\omega V_{\theta'}(s_t; \omega)$
      $r_{IS}(a_{t+1}, s_t) \leftarrow \frac{\pi_\theta(a_{t+1}|s_t)}{\pi_{\theta'}(a_{t+1}|s_t)}$
  for $b \in 1, 2, \ldots, B_\theta$ **do**
    Randomly sample N_1 Trails from K Trails.
    $\bar{J}(\theta; \theta') \leftarrow \frac{1}{N_1 L_e} \sum_{i=0}^{N_1} \sum_{t=0}^{L_e} r_{IS}(a_{t+1}, s_t) \hat{A}_{\theta'}^{GAE}(s_t, a_{t+1})$
    $\hat{\mathrm{KL}}(\pi_\theta || \pi_{\theta'}) \leftarrow \frac{1}{N_1 L_e} \sum_{i=0}^{N_1} \sum_{t=0}^{L_e} [r_{IS}(a_{t+1}, s_t) - 1 - \ln r_{IS}(a_{t+1}, s_t)]$
    $J_{KL} \leftarrow \hat{\mathrm{KL}}(\pi_\theta || \pi_{\theta'})$
    $J_{PPO}(\theta; \theta') \leftarrow \bar{J}(\theta; \theta') - \beta J_{KL}$
    $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J_{PPO}(\theta; \theta')$
    if $J_{KL} > \mathrm{KL}_{\max}$ **then**
      $\beta \leftarrow (1 + \delta)\beta$
    else if $J_{KL} < \mathrm{KL}_{\min}$ **then**
      $\beta \leftarrow (1 - \delta)\beta$
  for $b \in 1, 2, \ldots, B_\omega$ **do**
    Randomly sample N_2 Trails from K Trails.
    $\nabla_\omega \mathcal{L}(\omega) \leftarrow \frac{1}{N_2 L_e} \sum_{i=0}^{N_2} \sum_{t=0}^{L_e} \nabla_\omega \mathcal{L}(a_{t+1}, s_t)$
    $\omega \leftarrow \omega - \alpha_\omega \nabla_\omega \mathcal{L}(\omega)$
$\theta' \leftarrow \theta$
$\pi^*(a|s) \leftarrow \pi_\theta(a|s)$

Algorithm	**PPO-Penalty Algorithm**
Problem Type	MDP control problem [model-free]
Known	MDP environment $\mathcal{E}_D$
Objective	Optimal parameterized policy $\pi_\theta^*(a|s)$
Algorithm Properties	policy-based, on-policy, policy gradient

23.3.3 PPO-Clip Algorithm

The aforementioned method can indeed implement importance sampling with multiple policy network updates; however, the cost is that KL divergence computation is relatively resource-intensive. For each sampled step, we need to calculate its KL divergence and update the network during backpropagation.

The ultimate purpose of using KL divergence is to prevent the policy network from making large updates at once. To this end, we can adopt an alternative approach by directly limiting the magnitude of importance weights. This method is called **importance clipping**.

[This section will be updated in subsequent versions. Stay tuned.]
Thus, we obtain the **PPO-Clip Algorithm**

Algorithm 76: PPO-Clip Algorithm (Part 1)

Input: MDP environment $\mathcal{E}_D$
Output: Optimal parameterized policy $\pi^*_\theta(a|s)$
$\theta \leftarrow random(\Theta)$
$\omega \leftarrow random(\Omega)$
$\beta \leftarrow \beta_0$
for $m \in 1, \ldots, M$ **do**
  for $k \in 1, \ldots, K$ **do**
    $s_0, r_0 \sim p(s_0), p(r_0)$
    for $t \in 1, 2, \ldots, L_e$ **do**
      $a_t \sim \pi_{\theta'}(a|s_{t-1})$
      $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$
    $\hat{A}^{GAE}_{\theta'}(s_{L_e}, a_{L_e+1}) \leftarrow 0$
    $V_\pi(s_{L_e+1}; \omega) \leftarrow 0$
    for $t \in \mathcal{L}_e - 1, ..., 0$ **do**
      $\delta_\omega(t) \leftarrow r_t + \gamma V_{\theta'}(s_{t+1}; \omega) - V_{\theta'}(s_t; \omega)$
      $\hat{A}^{GAE}_{\theta'}(s_t, a_{t+1}) \leftarrow \delta_\omega(t) + \gamma\lambda \hat{A}^{GAE}_{\theta'}(s_{t+1}, a_{t+2})$
      $\nabla_\omega \mathcal{L}(a_t, s_t) \leftarrow \delta_\omega(t) \nabla_\omega V_{\theta'}(s_t; \omega)$
      $r_{IS}(a_{t+1}, s_t) \leftarrow \frac{\pi_\theta(a_{t+1}|s_t)}{\pi_{\theta'}(a_{t+1}|s_t)}$
      if $\hat{A}^{GAE}_{\theta'}(s_t, a_{t+1}) > 0$ **then**
        $\bar{r}_{IS}(a_{t+1}, s_t) \leftarrow \min(r_{IS}(a_{t+1}, s_t), 1 + \epsilon)$
      else
        $\bar{r}_{IS}(a_{t+1}, s_t) \leftarrow \min(r_{IS}(a_{t+1}, s_t), 1 - \epsilon)$
    ...(Continued on next page)

Algorithm 77: PPO-Clip Algorithm (Part 2)

for $m \in 1, \ldots, M$ **do**
  ...(Continued from previous page)
  for $b \in 1, 2, \ldots, B_\theta$ **do**
    Randomly sample N_1 Trails from K Trails.
    $J_{PPO2}(\theta; \theta') \leftarrow \frac{1}{N_1 L_e} \sum_{i=0}^{N_1} \sum_{t=0}^{L_e} \bar{r}_{IS}(a_{t+1}, s_t) \hat{A}^{GAE}_{\theta'}(s_t, a_{t+1})$
    $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J_{PPO2}(\theta; \theta')$
  for $b \in 1, 2, \ldots, B_\omega$ **do**
    Randomly sample N_2 Trails from K Trails.
    $\nabla_\omega \mathcal{L}(\omega) \leftarrow \frac{1}{N_2 L_e} \sum_{i=0}^{N_2} \sum_{t=0}^{L_e} \nabla_\omega \mathcal{L}(a_{t+1}, s_t)$
    $\omega \leftarrow \omega - \alpha_\omega \nabla_\omega \mathcal{L}(\omega)$
$\theta' \leftarrow \theta$
$\pi^*(a|s) \leftarrow \pi_\theta(a|s)$

Algorithm	**PPO-Clip Algorithm**
Problem Type	MDP control problem[model-free]
Known	MDP environment $\mathcal{E}_D$
Objective	Optimal parameterized policy $\pi^*_\theta(a|s)$
Algorithm Properties	policy-based, on-policy, policy gradient

(Reference: Mushroom Book)

23.3.4 Continuous Action Space

As introduced in Section 22.5.2, the PPO algorithm can also be adapted to continuous action spaces through Gaussian distribution modeling.

[This section will be updated in future versions, stay tuned]

23.4 Inverted Pendulum Solution

We still use the planar inverted pendulum problem (details in Section 10.6) as an example to test the continuous action PPO algorithm introduced in the previous section.

[This section will be updated in future versions, stay tuned]

Part VII
Visual Navigation Methods

24 Fundamentals of Visual Odometry

Vision is the most important sensory organ for humans. Approximately 80% of the information we receive daily is acquired through vision. For autonomous robots, it is crucial to fully utilize the diverse information provided by visual sensors. In this section, we primarily focus on visual navigation methods.

In this chapter, we will introduce the sequential visual measurement problem known as visual odometry, along with key foundational concepts and techniques. In the subsequent chapters, we will delve into critical technologies such as image feature extraction and point-based pose estimation, followed by detailed discussions of various visual odometry methods.

24.1 Basic Concepts

24.1.1 Visual Odometry

For modern robots, whether fixed-base robotic arms or mobile robots, estimating the pose state of their end-effectors or themselves is a fundamental requirement.

Visual motion estimation provides an important solution to this problem. Visual motion estimation refers to the process of estimating the position, orientation, velocity, and other motion-related information of a camera (and its attached carrier) using data from visual sensors (i.e., cameras). Such tasks and methods are collectively termed **Visual Odometry** (VO). In fact, for the problem of estimating the end-effector state of fixed-base robots, information from multiple sensors can be fused using methods like Kalman filtering to achieve more accurate state estimation, with visual sensors serving as a key end-effector sensor.

Specifically, let us define the visual odometry problem.

Problem	**Monocular Visual Odometry**
Description	Given an image sequence, estimate the robot's pose sequence
Known	Images $I_0, I_1, ..., I_n$ Intrinsic matrix K, distortion parameters
Estimate	Pose sequence $T_1, ..., T_n$

In the above definition, the output of visual odometry is a pose sequence. A pose consists of position and orientation, represented as the rotation R and translation t between two coordinate frames. Typically, this is expressed as a homogeneous matrix T. In this section, "pose" and "homogeneous matrix" are essentially equivalent, and we will not distinguish between them hereafter. The intrinsic matrix and distortion parameters are camera-specific imaging parameters, usually obtained through calibration. Some visual odometry methods can also estimate poses without prior knowledge of these parameters.

It is important to note that in visual odometry, the poses are often not absolute with respect to a fixed coordinate frame (e.g., East-North-Up) but rather relative poses between different states. Generally, we define the camera coordinate frame of the first image I_0 (introduced in the next section) as the **world coordinate frame**, denoted as w. The remaining pose sequences represent the poses of the camera coordinate frames relative to the world coordinate frame at each time step.

Visual odometry can employ different types of visual sensors. The configuration described above uses the most basic setup—a monocular camera—and is thus termed **monocular visual odometry**. However, monocular visual odometry suffers from a fundamental theoretical limitation—scale ambiguity[24]—which we will discuss in detail in the epipolar geometry section. Additionally, its cumulative error issue is significant. To address these challenges, various visual odometry methods have been developed, incorporating different sensors and algorithms.

In the following subsections, we will introduce key concepts related to visual odometry from three perspectives: sensors, image utilization methods, and pose optimization techniques.

[24] In practice, if the camera captures common objects in typical scenes, absolute scale can sometimes be estimated using visual cues. End-to-end visual odometry leverages this principle to recover scale.

24.1.2 Sensors

In the previous subsection, we mentioned an important concept in visual navigation: monocular scale ambiguity. The camera imaging process projects 3D space onto a 2D plane (see the "Camera Model" section below). During this process, true scale and depth information are lost. Although relative depth can be recovered using images from multiple viewpoints, the absolute scale remains unrecoverable. Imagine a simulated 3D environment where all objects can be uniformly scaled without affecting the imaging.

To address this issue, we can introduce known length information. Humans and many animals have two eyes with overlapping fields of view. When observing an object with both eyes, the slight disparity between the images received by each eye (due to the **baseline** distance between the eyes) allows our brains to intuitively judge the object's actual distance. Mimicking this biological mechanism, we can use two cameras with a fixed baseline and overlapping fields of view to simultaneously capture images, enabling pose estimation with absolute scale. This is the principle behind **stereo visual odometry**.

Problem	**Stereo Visual Odometry**
Description	Given the stereo baseline and stereo image sequence, estimate the robot's pose sequence
Known	Left images $I_{0,l}, I_{1,l}, ..., I_{n,l}$ Right images $I_{0,r}, I_{1,r}, ..., I_{n,r}$ Stereo baseline T_l^r Stereo intrinsic matrices K_1, K_2, stereo distortion parameters
Estimate	Pose sequence $T_1, ..., T_n$

Stereo cameras image the same object from overlapping viewpoints, enabling depth recovery through triangulation (see Section 26.2). However, this requires stereo image matching and imposes certain requirements on disparity. To directly obtain depth from a single image, various **depth cameras** have been developed. Depth cameras primarily operate on two principles: structured light and time-of-flight. Regardless of the principle, depth cameras output both a photometric image I and a depth map D for each frame. Visual odometry based on depth cameras is termed **RGBD visual odometry**.

Problem	**RGBD Visual Odometry**
Description	Given photometric and depth image sequences, estimate the robot's pose sequence
Known	Photometric images $I_0, I_1, ..., I_n$ Depth images $D_0, D_1, ..., D_n$ Intrinsic matrix K, distortion parameters
Estimate	Pose sequence $T_1, ..., T_n$

In addition to cameras, another sensor commonly used for motion estimation is the **Inertial Measurement Unit** (IMU). An IMU consists of accelerometers and gyroscopes, typically implemented as MEMS (Micro-Electro-Mechanical Systems). IMUs are compact, do not rely on external information, and do not actively emit signals, making them convenient and low-cost sensors.

IMU measurements consist of two components: **specific force** and **angular velocity**. Specific force is defined as the total acceleration of the carrier due to external forces excluding gravity, denoted as f:

$$f = \frac{F - mg}{m} = a - g \tag{VII.24.1}$$

IMUs typically have three sensitive axes (x, y, z) arranged in a right-handed coordinate system. Generally, we consider **the coordinate frame defined by the IMU on the carrier as the body frame**. **The IMU directly measures specific force and angular velocity in the body frame**, denoted as f^b and ω_{nb}^b, respectively. Both specific force and angular velocity are 3D vectors.

IMUs typically update at 100–200 Hz, far exceeding the frame rate of ordinary cameras (20–50 Hz). Thus, the IMU measurements between two image frames form a vector sequence, denoted as $\mathbf{f}^b$ and $\mathbf{w}_{nb}^b$, representing all specific forces and angular velocities between the frames.

IMUs provide visual odometry with gravity direction information, reducing the unobservability of visual odometry at the methodological level. Visual odometry that incorporates IMU data is termed **Visual-Inertial Odometry** (VIO).

Problem	**(Monocular) Visual-Inertial Odometry**
Description	Given image sequences and IMU measurement sequences, estimate the robot's pose sequence
Known	Images $I_0, I_1, ..., I_n$ Specific forces $\mathbf{f}_1^b, ..., \mathbf{f}_n^b$, angular velocities $\mathbf{w}_1^b, ..., \mathbf{w}_n^b$ Intrinsic matrix K, extrinsic matrix T_b^c, distortion parameters
Estimate	Pose sequence $T_1, ..., T_n$

The methods introduced in subsequent chapters of this section primarily cover monocular/stereo visual odometry and visual-inertial odometry, collectively referred to as VO.

24.1.3 Image Utilization

A camera's photometric image I is typically a 2D or 3D matrix, where the dimensions along the horizontal and vertical axes are called resolution, and each element is called a pixel. Most cameras capture color images, where each pixel stores the intensities of red, green, and blue channels as a vector, resulting in a matrix $I \in \mathbf{R}^{H \times W \times 3}$. However, some non-dense mapping methods in VO do not rely on color information and instead convert the three color channels into a single grayscale channel, resulting in a 2D image $I \in \mathbf{R}^{H \times W}$.

Regardless of the specific VO method, the photometric image I from the camera is always a required input. Camera resolutions H and W typically range from hundreds to thousands, making I a very large matrix with hundreds of thousands to millions of elements. This matrix contains both critical information for pose estimation and substantial redundancy. Efficiently utilizing visual image information for pose estimation is the core challenge of visual odometry. Currently, based on how image information is utilized, mainstream VO methods can be categorized into four types: **indirect VO**, **direct VO**, **end-to-end VO**, and **rendering-based VO**.

Indirect VO converts image information into an intermediate representation and then estimates motion from this representation. Such intermediate information generally facilitates rapid matching of the same object across different images, with the most common approaches being **feature-based methods** and **optical flow methods**. Using these, indirect VO solves for the optimal motion estimate by optimizing **feature reprojection error**. For detailed explanations of indirect VO, refer to Chapter 27.

Feature points are pixels in an image that can be easily distinguished and matched against others, typically corresponding to corners or edges of objects in space. Feature-based methods first detect corners or edges in images to obtain their pixel coordinates (2D), then recover their spatial coordinates (3D) by matching points across images with different poses. These 3D coordinates serve as critical references for camera pose estimation. For details on feature points, see the next chapter.

Optical flow is the projection of object motion relative to the camera onto the imaging plane, forming a planar field. Many mature methods exist to compute optical flow between consecutive frames, enabling relative motion estimation. Compared to feature-based methods, optical flow reduces some computational overhead in matching. For details on optical flow, see the next chapter.

Unlike indirect methods, **direct VO** directly uses photometric information and gradients from images for camera pose estimation. Direct VO typically solves optimization problems with objectives like **photometric error** and depth error. Some hybrid approaches combine aspects of both direct and indirect methods, termed **semi-direct VO**. For details on direct and semi-direct methods, refer to Chapter 28.

End-to-end VO refers to deep neural network-based visual odometry. It frames VO as a regression problem, where networks take inputs like photometric images or IMU data and output poses T, trained on large datasets. Another category retains indirect/direct frameworks but uses neural networks to optimize intermediate representations (e.g., feature points, optical flow, depth maps).

Rendering-based VO is an emerging class of VO methods. These approaches, leveraging scene representations optimized for novel-view rendering and efficient modeling of complex scenes, have gained attention—particularly in AR/VR/CG. Their theoretical foundation is **differentiable rendering**, with two main

branches: **NeRF (Neural Radiance Fields)** and **3DGS (3D Gaussian Splatting)**. Currently, 3DGS-based SLAM methods are notable for their explicit representation and low rendering overhead. For details, see Chapter 29.

Compared to end-to-end and rendering-based VO, direct and indirect VO are also termed **traditional methods**. Currently, indirect VO is the most mature and widely used in mobile robotics. This and subsequent chapters focus on indirect VO, direct VO, and rendering-based VO.

24.1.4 Sliding Window Pose Optimization

For VO, regardless of the method used to extract pose information from images, initial pose estimates inevitably contain **errors**. These may arise from inaccurate camera parameters, incorrect feature matching, local minima in non-convex optimization, or sensor noise (e.g., IMU).

Since visual odometry estimates poses sequentially, errors propagate along the trajectory—a phenomenon called **error accumulation**. Additionally, for non-RGBD VO, 3D (depth) information is recovered from 2D data, and depth errors further corrupt pose estimates. Unchecked error accumulation quickly leads to divergence.

To address this, we can: (1) mitigate error sources via precise sensor calibration or outlier rejection (detailed later), or (2) formulate a **joint optimization problem** based on 3D vision principles. Here, multiple historical poses and their co-observed 3D features become optimization variables, with objectives derived from camera models. Solving this yields refined historical poses.

This approach introduces a challenge: real-time algorithms must retain historical poses and 3D features, causing optimization variables to grow excessively as trajectories lengthen.

The solution is **sliding window** optimization—a queue where new poses/features enter the window, and older ones are pruned when capacity is reached, maintaining a bounded problem size. To avoid redundancy from static scenes, windows only retain distinct frames called **keyframes**.

Sliding window optimization is widely adopted, but it creates a **precision-speed trade-off**. Larger windows improve accuracy but increase computational cost, while VO must process each new frame (called **tracking**) at camera frame rates.

To balance this, mainstream VO decouples **new pose tracking** (frontend) and **historical pose optimization** (backend), often running on separate CPU threads for parallelism.

However, the "frontend-backend" boundary lacks universal standards. Some indirect methods limit the frontend to image processing (e.g., feature matching), while others include pose tracking. We follow each algorithm's convention in later chapters.

As mentioned, VO frontends vary (direct/indirect methods). Backends also differ, reflecting two interpretations of VO:

1. **Optimization-based VO**: Treats VO as an optimization problem where historical poses are variables. The backend constructs joint objectives from observations and solves them numerically.

2. **Filter-based VO**: Frames VO as a sequential state estimation problem with sensor fusion. It employs filters (e.g., Kalman filters from Chapter 11) to correct state estimates using observations.

Both approaches perform optimal state estimation. Subsequent chapters cover both filter- and optimization-based methods.

24.1.5 Simultaneous Localization and Mapping

Earlier, we noted that VO's joint optimization of historical poses and 3D observations implicitly builds a **map**—a 3D representation of the environment.

From VO's perspective, maps are byproducts. However, maps have intrinsic value: (1) **3D reconstruction** converts 2D images into 3D models; (2) Tracking relies on matching current images to historical data (windowed). If matching fails (e.g., due to occlusion or rapid motion), **tracking loss** occurs.

In VO, tracking loss may require reinitialization if resumed frames differ too much from the window. However, storing environment data beyond the window enables **relocalization**—matching against the full map to recover tracking without interruption.

When mapping becomes as critical as pose estimation, the process is termed **Simultaneous Localization and Mapping (SLAM)**. Vision-based SLAM is called **visual SLAM (vSLAM)**.

Conventionally, vSLAM systems divide into frontend (real-time pose tracking) and backend (three components: **global optimization**, **relocalization**, and **loop closure detection**). The backend handles computationally intensive tasks with relaxed latency.

Loop closure detection identifies when a trajectory revisits previously observed areas, enabling pose corrections across the loop to suppress accumulated errors. For vSLAM with loop closure, incorporating loops (manually or autonomously) is crucial.

VO and vSLAM lack strict boundaries and are sometimes used interchangeably. In this book, algorithms with relocalization/loop closure or global mapping are termed vSLAM; others are VO.

vSLAM maps generally fall into: sparse, semi-dense/dense, and implicit maps. **Sparse maps** (point clouds) consist of 3D feature points with large inter-point spacing. **Dense maps** are near-continuous 3D points with color data, usable for rendering or conversion to mesh/volume formats. **Semi-dense maps** are edge-dense but sparse in textureless regions.

Implicit maps are 3D representations in rendering-based VO, modeling space as opacity/color blocks for **differentiable rendering** and simplified novel-view synthesis. Currently, **3DGS maps** dominate; see Chapter 29.

VO/vSLAM encompasses diverse methods with complex interrelations. This book selects representative algorithms for detailed coverage in later chapters.

To conclude, we summarize key VO/vSLAM methods by sensor type, frontend, backend, and map type (Table 4).

Table 4: Key VO/vSLAM Methods and Their Properties

Method	**Sensor**	**Frontend**	**Backend**	**Map Type**
MSCKF	RGB+IMU	Feature points	Filter	Sparse map
VINS-Fusion	RGB+IMU	Feature points	Optimization	Sparse map
DSO	RGB	Direct	Optimization	Sparse map
MonoGS	RGB	Rendering-based	3DGS	3DGS map

The following sections introduce camera models (VO's foundation) and feature/optical flow processing in indirect methods.

24.2 Camera Models

—

Note: All LaTeX commands, math formulas (e.g., T), cross-references (e.g., 27), and table structures remain unchanged as requested. Only natural language text has been translated.

The camera is the primary sensor used in vSLAM/VO. The essence of a camera is an array of photoelectric sensors, known as the photosensitive chip. The photosensitive chip can measure the intensity of light in units of **pixels** and convert it into numerical values. The basic parameters of a camera are the number of pixels horizontally and vertically on the photosensitive chip, referred to as Width (W) and Height (H), respectively.

The data obtained by the camera is called an **image**. The horizontal and vertical pixel indices corresponding to a point in the image are referred to as the **pixel coordinates**. Typically, we establish a **pixel coordinate system** with the top-left corner of the image as the origin, the right and downward directions as the positive axes. The unit of the pixel coordinate system is 1 pixel, i.e., 1 pix. Pixel coordinates are generally represented as $[u, v]^T$ or sometimes in augmented form as $[u, v, 1]^T$. In the augmented representation, we denote:

$$\upsilon = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{VII.24.2}$$

In addition to the pixel coordinate system, we often establish a three-dimensional coordinate system fixed to the camera: the **camera coordinate system**. The point where all light rays in the camera's optical path converge is called the **optical center**. The camera coordinate system is typically defined with the optical center as the origin, the forward direction of the camera as the z-axis, the rightward direction of the photosensitive chip as the x-axis, and follows the right-hand rule. The camera coordinate system is generally denoted as the c frame.

The imaging process of a camera is a complex optical procedure, where designed lens assemblies converge light rays emitted from an external point onto the focal plane (i.e., the plane where the photosensitive chip is located). Depending on the transformation relationships, camera lenses are categorized into wide-angle lenses, fisheye lenses, telephoto lenses, etc. In visual SLAM, fisheye and wide-angle lenses are commonly used. For simplicity, we will focus on wide-angle lenses in the following discussion.

The mathematical model describing the effect of the lens on light rays is called the **camera model**. For wide-angle lenses, the imaging process can be approximated by the **pinhole imaging model**, where light rays from the image plane pass through a small aperture at the camera's optical center and form an inverted real image on the focal plane. For simplicity, we equate this inverted real image behind the optical center to an upright virtual image at the same distance in front of the optical center. This ensures that the x and y axes of the camera coordinate system align with the u and v axes of the pixel coordinate system.

Under this modeling, the pinhole camera model can be expressed as:

$$\upsilon = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K\nu^c = \frac{1}{z^c} K p^c = \frac{1}{z^c} K \begin{bmatrix} x^c \\ y^c \\ z^c \end{bmatrix} \tag{VII.24.3}$$

Here, we denote the normalized coordinates as ν, i.e.,

$$\nu^c = \frac{1}{z^c} p^c = \frac{1}{z^c} \begin{bmatrix} x^c \\ y^c \\ z^c \end{bmatrix} \tag{VII.24.4}$$

where p^c represents the coordinates of a point on an object in the camera coordinate system. $[u, v, 1]^T$ represents the corresponding (augmented) pixel coordinates of the imaged point. K is called the **camera intrinsic matrix**, with the following composition:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{VII.24.5}$$

Here, c_x and c_y are the pixel distances of the optical center's projection from the top-left corner of the photosensitive chip along the horizontal and vertical directions, respectively. For an ideal camera, c_x and c_y are half of W and H, though in practice, they are close to these values. f_x and f_y are the ratios of the unit length in the camera coordinate system (1m) to the unit length in the pixel coordinate system (1pix) along the horizontal and vertical directions.

In the actual imaging process of wide-angle lenses, **distortion** also occurs. Distortion refers to the offset between the actual position of an imaged point on the focal plane and its ideal position, causing the image to lose properties such as angle and line preservation. Distortion can also be described by a set of parameters, which will not be elaborated here.

The camera's intrinsic and distortion parameters are determined solely by the camera and lens optics. Methods exist to measure these parameters, known as **camera intrinsic calibration**. During calibration, a precisely dimensioned calibration board is used in conjunction with corner detection and intrinsic calibration algorithms, typically solving an optimization problem to obtain the intrinsic parameters.

The camera is usually fixed to a robotic platform. Conventionally, the right/forward/upward directions of the robot's motion are defined as the $x/y/z$ axes of the body frame (i.e., the b frame), or the three sensitive axes of the IMU on the platform are used as the $x/y/z$ axes of the b frame. The camera's z-axis generally points forward. Thus, a homogeneous matrix T_b^c is needed to represent the relative pose between the camera c frame and the body b frame. The matrix T_b^c is also called the **camera extrinsic parameters**.

If the VO system includes IMU input, the homogeneous matrix T_b^c can be determined through certain methods, a process known as **camera extrinsic calibration**. Extrinsic calibration generally relies on the results of intrinsic calibration and essentially involves solving an optimization problem with known environmental feature dimensions.

In VO and vSLAM, camera intrinsics, distortion parameters, and extrinsics are typically assumed to be known, and images are assumed to have undergone distortion compensation. However, in later chapters, we will introduce methods that consider inaccurate intrinsics as affecting VO/vSLAM accuracy; thus, they also treat intrinsics as one of the optimization variables to be corrected during the algorithm.

25 Feature Points and Optical Flow

In this chapter, we will discuss the fundamental principles and methods of indirect image utilization, namely feature point methods and optical flow methods. First, we will introduce some basic concepts, notations, and conventions related to digital images. Subsequently, we will separately present several important feature point and optical flow methods.

25.1 Basics of Digital Images

The term **digital image** generally refers to a matrix stored in a computer in the form of floating-point numbers/integers, representing the image's luminance. The most common type of digital image is the **RGB image**, which mimics the three types of cone cells in the human eye and consists of three color channels (red, green, and blue). The shape of its matrix is $H \times W \times 3$. In VO and vSLAM, color information is often unnecessary, so we typically use **grayscale images**, whose matrix shape is $H \times W$.

In the above expressions, H and W represent the height and width of the image, consistent with the definitions in the camera model section. The pixel coordinates of the image are denoted as u for the width direction and v for the height direction. (u, v) can take integer or non-integer values. Generally, we assume $(u, v) \in [0, W] \times [0, H]$.

For each grayscale image I, the grayscale value at (u, v) is denoted as $I[u, v]$. By default, u and v are assumed to be integers when retrieving grayscale values; for non-integer cases, bilinear interpolation can be applied. **A grayscale image I is a 2D discrete signal, which can be considered as a sampling of a 2D continuous signal (continuous function)**. That is, for each grayscale image I, an image function $\mathcal{I}(x, y)$ can be defined such that

$$I[u, v] = \mathcal{I}(u, v)$$

For the continuous function $\mathcal{I}(x, y)$, we assume it is twice continuously differentiable and denote its first and second partial derivatives as

$$\begin{aligned}
\mathcal{I}_x(x, y) &:= \frac{\partial \mathcal{I}}{\partial x}(x, y) \\
\mathcal{I}_y(x, y) &:= \frac{\partial \mathcal{I}}{\partial y}(x, y) \\
\mathcal{I}_{xx}(x, y) &:= \frac{\partial^2 \mathcal{I}}{\partial x^2}(x, y) \\
\mathcal{I}_{yy}(x, y) &:= \frac{\partial^2 \mathcal{I}}{\partial y^2}(x, y) \\
\mathcal{I}_{xy}(x, y) &:= \frac{\partial^2 \mathcal{I}}{\partial x \partial y}(x, y)
\end{aligned}$$

The above definitions are in the continuous domain, whereas the images we actually process are discrete digital images. We define the **image gradients** $I_u, I_v, I_{uu}, I_{vv}, I_{uv}$ as the discrete samples of the corresponding first/second derivatives of the image function. However, in practice, we do not recover a continuous function from the digital image but instead approximate the image gradients using image convolution[25]. The algorithms are as follows:

$$\begin{aligned}
I_u[u, v] &:= \mathcal{I}_x(u, v) \approx I \star K_u \\
I_v[u, v] &:= \mathcal{I}_y(u, v) \approx I \star K_v \\
I_{uu}[u, v] &:= \mathcal{I}_{xx}(u, v) \approx I \star K_u \star K_u \\
I_{vv}[u, v] &:= \mathcal{I}_{yy}(u, v) \approx I \star K_v \star K_v \\
I_{uv}[u, v] &:= \mathcal{I}_{xy}(u, v) \approx I \star K_u \star K_v
\end{aligned} \tag{VII.25.1}$$

Here, K_u and K_v are first-order differential convolution kernels, and we use the **Sobel operator**:

[25] We assume the reader has basic knowledge of image processing and will not elaborate on operations like convolution or details like padding.

$$K_u = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$
$$K_v = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{VII.25.2}$$

Based on the partial derivatives, we can define the image gradient ∇I:

$$\nabla I[u,v] := \nabla \mathcal{I}(u,v) = \begin{bmatrix} \mathcal{I}_x(u,v) \\ \mathcal{I}_y(u,v) \end{bmatrix}$$

For a bivariate function, the gradient at each point is a 2D vector. We decompose it into magnitude and direction angle, both of which can be approximated using the above partial derivatives:

$$\begin{aligned} ||\nabla I|| &= \sqrt{I_u \odot I_u + I_v \odot I_v} \\ \theta_{\nabla I} &= \arctan\left(I_u \oslash I_v\right) \end{aligned} \tag{VII.25.3}$$

Here, $\odot$ denotes element-wise matrix multiplication, and $\oslash$ denotes element-wise matrix division. In addition to first-order derivatives and gradients, we can also use second-order partial derivatives to compute the image's second-order Laplace operator ΔI:

$$\Delta I[u,v] := \Delta \mathcal{I}(u,v) = \mathcal{I}_{xx}^2(u,v) + \mathcal{I}_{yy}^2(u,v)$$

Its approximation is calculated as:

$$\Delta I = I_{uu}^2 + I_{vv}^2 = I \star K_L \tag{VII.25.4}$$

Here, the convolution kernel K_L can be chosen in two ways:

$$K_{L1} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$
$$K_{L2} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{VII.25.5}$$

25.2 Basic Concepts of Feature Points

In the previous chapter, we mentioned that different VOs employ different methods to utilize images. The essence of indirect VO is to use intermediate representations like feature points or optical flow to extract the **matching relationships** between two frames. Only with matching relationships can we estimate the camera's motion between the two frames. If there are no matching points between two frames, motion estimation becomes impossible. This section introduces feature point methods, and the next section introduces optical flow methods, both of which are approaches to finding matching relationships in images.

A **feature point** refers to a distinctive point in an image, typically an edge point or corner point of a specific object. These points are representative within the entire image and facilitate detection and matching. For a typical image, among millions of pixels, usually only a few hundred to a few thousand points meet such criteria.

The essence of VO or vSLAM is to recover 3D information from 2D images. Therefore, feature points in VO or vSLAM generally have two meanings: representative points in 2D images (also called **2D feature points**) and representative points in 3D space (also called **3D feature points**). 2D feature points are directly extracted from images, and the process of finding 2D feature points in a single image is called **feature point detection**. In this and subsequent chapters, 2D feature points sometimes specifically refer to the pixel coordinates of a set of 2D feature points in an image, while 3D feature points specifically refer to the 3D coordinates of a set of 3D feature points in space.

The **matching relationships of 2D feature points** are crucial and form the core of feature point methods. This is because if two frames observe the same 3D space, the 3D feature points in that space should correspond to 2D feature points in each of the two images. If we find such matching relationships and know the poses of the two frames, we can recover the coordinates of the 3D feature points. Conversely, if we know the coordinates of the 3D feature points, we can also recover the relative poses of the two frames. All of this relies on the matching relationships of 2D feature points. Given two frames and their respective detected 2D feature points, the process of solving for the matching relationships is called **feature point matching**. **Feature point detection and feature point matching are the two core problems in the front end of feature point methods.**

Problem	**Feature Point Detection Problem**
Problem Description	Given a single image, find the pixel coordinates of representative points in the image
Given	Image I
To Find	Pixel coordinates of feature points $v_1, ..., v_n$

Problem	**Feature Point Matching Problem**
Problem Description	Given two image frames and their feature points, find the matching relationships between feature points
Given	Images I_1, I_2 2D feature points of Image 1: $v_1^1, ...v_i^1, ..., v_m^1$ 2D feature points of Image 2: $v_1^2, ..., v_j^2, ..., v_n^2$
Find	Matching index pairs $(i_1, j_1), ..., (i_k, j_k)$

For the feature point matching problem, to solve the matching relationships, we need to understand what distinguishes feature points from others. Whether at object corners or areas with clear textures, the neighborhood pixels of feature points differ significantly from those of other points. Therefore, we typically construct a description of feature points based on their neighborhood pixels and perform matching based on the similarity of these descriptions. Such descriptions are called **feature descriptors**.

Below, we will introduce commonly used methods for feature point detection and matching.

25.3 Harris Corner

As mentioned earlier, for the feature point detection problem, we aim to find representative and easily identifiable points in an image. More specifically, we seek image points located at **object corners or areas with clear textures**. Below, we introduce the **Harris Corner** method.

25.3.1 Sliding Error

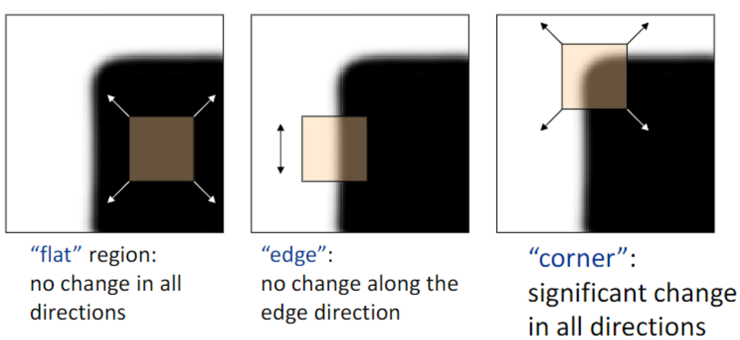

Figure VII.25.1: Principle of Harris Corner

As shown in Figure VII.25.1, suppose we take a neighborhood window around a point and perform a weighted average of the pixels within the neighborhood. The feature points we seek typically satisfy the following properties: if the point is a corner, the weighted average changes significantly when the window slides in any direction; if the point is an edge point, the weighted average changes minimally when sliding along the edge but significantly when sliding perpendicular to the edge.

Specifically, around the pixel point (u_0, v_0), we design the following sliding error metric as a function of the sliding vector $(\delta u, \delta v)$:

$$E[u_0, v_0](\delta u, \delta v) := \sum_{(u,v)\in W} w(u,v)(I[u_0, v_0] - I_{\overrightarrow{\delta u, \delta v}}[u_0, v_0])^2 \tag{VII.25.6}$$

where $I_{\overrightarrow{a,b}}$ represents the image translated by (a, b), defined as:

$$I_{\overrightarrow{a,b}}[u, v] = \mathcal{I}(u + a, v + b)$$

W denotes the range of the sliding window in the weighted averaging process. The sliding window can be viewed as a convolution. If the convolution kernel has a side length of k, we can set $W = [-\frac{k+1}{2}, \frac{k+1}{2}] \times [-\frac{k+1}{2}, \frac{k+1}{2}]$.

For the translated image above, in the continuous domain, we have the first-order Taylor approximation:

$$\mathcal{I}(u + a, v + b) \approx \mathcal{I}(u, v) + \frac{\partial \mathcal{I}}{\partial x}(u, v)a + \frac{\partial \mathcal{I}}{\partial y}(u, v)b$$

Thus, in the discrete domain, we also have:

$$I_{\overrightarrow{a,b}}[u, v] \approx I[u, v] + aI_u[u, v] + bI_v[u, v]$$

That is:

$$\begin{aligned}
E[u_0, v_0](\delta u, \delta v) &= \sum_{(u,v)\in W} w(u,v)(I[u_0, v_0] - I_{\overrightarrow{\delta u, \delta v}}[u_0, v_0])^2 \\
&= (((I - I_{\overrightarrow{\delta u, \delta v}}) \odot (I - I_{\overrightarrow{\delta u, \delta v}})) \star K_w)[u_0, v_0] \\
&\approx (((\delta u I_u + \delta v I_v) \odot (\delta u I_u + \delta v I_v)) \star K_w)[u_0, v_0] \\
&= (\delta u^2 I_u \odot I_u \star K_w + \delta v^2 I_v \odot I_v \star K_w + 2\delta u \delta v I_u \odot I_v \star K_w)[u_0, v_0]
\end{aligned}$$

Let:

$$\begin{aligned}
A &= I_u \odot I_u \star K_w \in \mathbb{R}^{H\times W} \\
B &= I_v \odot I_v \star K_w \in \mathbb{R}^{H\times W} \\
C &= I_u \odot I_v \star K_w \in \mathbb{R}^{H\times W}
\end{aligned} \tag{VII.25.7}$$

Then:

$$\begin{aligned}
E[u_0, v_0](\delta u, \delta v) &\approx (\delta u^2 A + \delta v^2 B + 2\delta u \delta v C)[u_0, v_0] \\
&= \delta u^2 A[u_0, v_0] + \delta v^2 B[u_0, v_0] + 2\delta u \delta v C[u_0, v_0] \\
&= \begin{bmatrix} \delta u & \delta v \end{bmatrix} \begin{bmatrix} A[u_0, v_0] & B[u_0, v_0] \\ B[u_0, v_0] & C[u_0, v_0] \end{bmatrix} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix}
\end{aligned}$$

Further, let:

$$H[u_0, v_0] = \begin{bmatrix} A[u_0, v_0] & B[u_0, v_0] \\ B[u_0, v_0] & C[u_0, v_0] \end{bmatrix} \tag{VII.25.8}$$

Then:

$$E[u_0, v_0](\delta u, \delta v) \approx \begin{bmatrix} \delta u & \delta v \end{bmatrix} H[u_0, v_0] \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \tag{VII.25.9}$$

It can be seen that the sliding error $E[u_0, v_0](\delta u, \delta v)$ can be approximated as a quadratic form of the matrix $H[u_0, v_0]$. Further, performing SVD decomposition on $H[u_0, v_0]$, we have:

$$H[u_0, v_0] = R^T \begin{bmatrix} \lambda_1 & \\ & \lambda_2 \end{bmatrix} R$$

Here, R is an orthogonal matrix, and λ_1, λ_2 are the two eigenvalues of $H[u_0, v_0]$. The eigenvalues of $H[u_0, v_0]$ determine the variation of the sliding error $E[u_0, v_0]$.

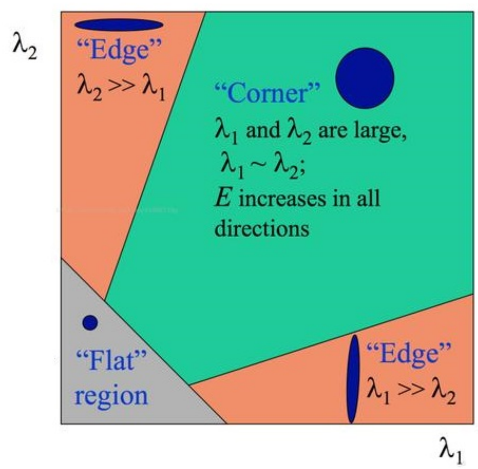

Figure VII.25.2: λ values of Harris Corner

As shown in Figure VII.25.2, larger λ_1, λ_2 mean that for the same $(\delta u, \delta v)$, the sliding error $E[u_0, v_0](\delta u, \delta v)$ changes more significantly. According to the earlier discussion, this implies that (u_0, v_0) is a corner point. If λ_1, λ_2 differ significantly, it means the sliding error changes notably only in one direction, indicating that (u_0, v_0) is an edge point. The direction of significant change is perpendicular to the edge and can be given by the eigenvectors of R.

25.3.2 Algorithm Flow

Based on the above analysis, our goal is to find all points where $H[u_0, v_0]$ has the largest eigenvalues, as they correspond to the desired corner points. However, since there are two eigenvalues, this criterion is not easily quantifiable. We can further define an index based on the two eigenvalues:

$$\begin{aligned} r(u_0, v_0) &:= \lambda_1 \lambda_2 - \alpha(\lambda_1 + \lambda_2)^2 \\ &= \det H[u_0, v_0] - \alpha \operatorname{tr}(H[u_0, v_0]) \end{aligned}$$

As mentioned above, when (u_0, v_0) is an edge point, we have $\lambda_1 >> \lambda_2 \approx 0$ or $\lambda_2 >> \lambda_1 \approx 0$, corresponding to the orange region in Figure VII.25.2, where $r(u_0, v_0)$ will be a negative value with a large absolute magnitude. When (u_0, v_0) is a corner point, we have $\lambda_1 \approx \lambda_2 >> 0$, corresponding to the green region in Figure VII.25.2, where $r(u_0, v_0)$ will be a positive value with a large absolute magnitude. If the neighborhood of (u_0, v_0) is relatively flat, both λ_1 and λ_2 will be small, and $r(u_0, v_0)$ will be close to 0.

Thus, by computing the corresponding $r(u_0, v_0)$ for each point, we can easily determine whether the point is an edge point, corner point, or flat point. We can directly compute r from A, B, C:

$$r(u_0, v_0) = A[u_0, v_0]C[u_0, v_0] - B^2[u_0, v_0] - \alpha(A[u_0, v_0] + C[u_0, v_0]) \tag{VII.25.10}$$

All $r(u_0, v_0)$ values form a matrix, denoted as R_H. In visual SLAM, corner points are generally more stable compared to edge points. Therefore, we only need to identify the local maxima of R_H, and their corresponding (u_0, v_0) coordinates are the feature points (corner points). This process also requires Non-Maximum Suppression (NMS) to prevent feature points from clustering. The implementation of the algorithm *get_topn_NMS* will be introduced in the next subsection.

Summarizing the above derivation, we obtain the Harris corner detection algorithm:

Algorithm 78: Harris Corner Detection (Harris_FP)
Input: Image I
Parameter: Sliding window convolution kernel K_w, α
Parameter: NMS region size k, number n
Output: Feature point pixel coordinates $v_1, ..., v_n$
$I_u, I_v \leftarrow I \star K_u, I \star K_v$
$A \leftarrow I_u \odot I_u \star K_w$
$B \leftarrow I_v \odot I_v \star K_w$
$C \leftarrow I_u \odot I_v \star K_w$
$R_H \leftarrow A \odot C - B \odot B - \alpha(A + C)$
$v_{1:n} \leftarrow \text{get\_topn\_NMS}(R_H, n, k)$

Algorithm	**Harris Corner Detection**
Problem Type	Feature point detection problem
Given	Image I
Find	Feature point pixel coordinates $v_1, ..., v_n$
Algorithm Property	Traditional CV

25.3.3 NMS Algorithm

[This section will be updated in a future version. Stay tuned.]

The Harris corner detection introduced above is computationally simple and clearly defined, but it lacks scale invariance. Several feature point detection and description methods introduced below incorporate scale invariance.

25.4 SIFT Keypoints

The term **scale** refers to the pixel width of a feature region in an image. The same 3D object can project into different scales under varying 2D viewpoints. As discussed in the previous section, the limitation of Harris corner detection lies in its lack of "scale invariance." In other words, the Harris method cannot detect corners simultaneously for projections of the same object at different scales.

In fact, we can theoretically analyze the root cause of this issue. Note that the Harris matrix H is determined solely by I_u, I_v, and K_w, where I_u and I_v under the Sobel operator can only respond to edges with a width of 1-2 pixels. Additionally, K_w is a fixed value. Therefore, the Harris corner algorithm inherently detects features at a predetermined scale and cannot adapt to features at varying scales.

In 1998, Lowe proposed the **Scale Invariant Feature Transform** (SIFT), known as **SIFT keypoints**. SIFT keypoints are scale-space features where each (2D) keypoint is parameterized not only by its pixel coordinates but also by its corresponding scale information, facilitating better feature matching across different projections.

SIFT keypoints are based on the concept of a **feature map**, where a function is derived from the image space such that the local maxima of this function correspond to distinct features. SIFT employs Difference of Gaussian (DoG) convolution kernels at varying scales to convolve the entire image, resulting in a feature map called the **DoG pyramid**, which effectively identifies blob-like features (approximate elliptical shapes) in images. Furthermore, the method introduces a scale/orientation-invariant **SIFT descriptor**.

Next, we will first introduce the DoG pyramid, then explain how to extract multi-scale keypoints from this pyramid, and finally describe how to characterize these features.

25.4.1 Multi-scale DoG Operator

SIFT aims to detect features using the **Laplacian of Gaussian (LoG) convolution kernel**, which is specifically responsive to circular features. The LoG kernel is defined as:

$$\Delta\mathcal{G}(x, y; \sigma) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2 + y^2}{2\sigma^2}\right] e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Here, the Laplacian operator Δ was introduced in the "Digital Image Fundamentals" section. As shown in Figure VII.25.3, the LoG kernel in the 2D continuous domain resembles an inverted funnel with raised edges, exhibiting maximal response to circular features. The radius of this maximally responsive circle is proportional to the standard deviation σ of the LoG kernel. Thus, we also refer to σ as the "feature scale" or simply "scale."

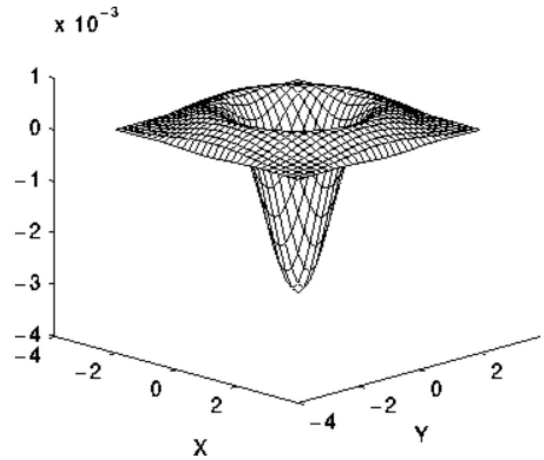

Figure VII.25.3: Shape of the LoG kernel

The convolution result of the LoG kernel with the original image is called the **LoG feature map**:

$$\text{LoG}[\mathcal{I}](x, y) = \mathcal{I}(x, y) \star \Delta\mathcal{G}(x, y; \sigma)$$

While LoG convolution is computationally intensive, Gaussian convolution alone is simpler. In fact, the LoG can be approximated by the difference between two Gaussian kernels with different standard deviations, known as the **Difference of Gaussian (DoG) kernel**:

$$\Delta\mathcal{G}(x, y; \sqrt{k-1}\sigma) \approx \mathcal{G}(x, y; k\sigma) - \mathcal{G}(x, y; \sigma)$$

Here, $k > 1$[26]. Thus, the LoG feature map can also be approximated by the DoG feature map:

$$\text{LoG}[\mathcal{I}](x, y) \approx \text{DoG}[\mathcal{I}](x, y) = \mathcal{I}(x, y) \star \mathcal{G}(x, y; k\sigma) - \mathcal{I}(x, y) \star \mathcal{G}(x, y; \sigma)$$

Therefore, to obtain LoG feature maps at different scales, we can first compute a series of Gaussian blurs and then take their pairwise differences, as illustrated in Figure VII.25.4.

[26] In SIFT, $k = \sqrt{2}$ is typically chosen, as explained in the next subsection.

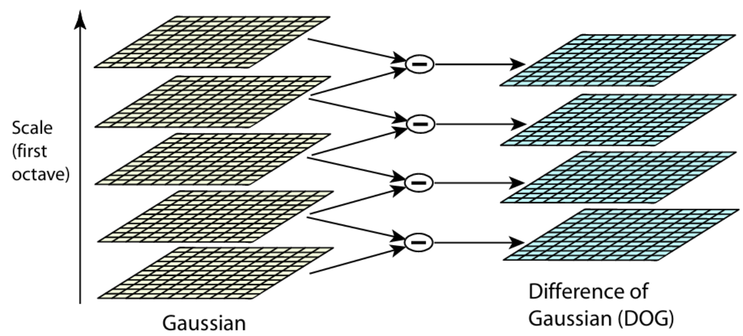

Figure VII.25.4: Approximating multi-scale LoG using DoG

For simplicity, we desire the scales of the LoG feature maps to form a geometric progression, meaning the corresponding DoG scales and the Gaussian kernel scales should also follow a geometric progression. Specifically, assuming the smallest Gaussian kernel has a standard deviation σ_0, with a ratio k and a total of n scales, we have:

$$\mathcal{I}_i(x, y) := \mathcal{I}(x, y) \star \mathcal{G}(x, y; k^{i-1}\sigma_0), \quad i = 1, ..., n$$

The DoG at each scale is then:

$$\text{DoG}_i[\mathcal{I}](x, y) = \mathcal{I}_{i+1}(x, y) - \mathcal{I}_i(x, y), \quad i = 1, ..., n-1 \tag{VII.25.11}$$

This approach yields a set of multi-scale DoG feature maps, approximating the LoG feature maps. However, there is an engineering challenge: larger σ values require more computational effort for convolution. Leveraging properties of the Gaussian function, this issue can be mitigated by decomposing large Gaussian kernels into smaller ones:

$$\mathcal{G}(x, y; \sqrt{\sigma_1^2 + \sigma_2^2}) = \mathcal{G}(x, y; \sigma_1) \star \mathcal{G}(x, y; \sigma_2)$$

For Gaussian feature maps in a geometric progression, we do not need to convolve the original image $\mathcal{I}(x, y)$ each time. Instead, we can iteratively convolve the previous scale's feature map with a relatively small kernel:

$$\begin{aligned} \mathcal{I}_1(x, y) &= \mathcal{I}(x, y) \star \mathcal{G}(x, y; \sigma_0) \\ \mathcal{I}_i(x, y) &= \mathcal{I}_{i-1}(x, y) \star \mathcal{G}(x, y; \sqrt{k^2 - 1}k^{i-1}\sigma_0), \quad i = 2, ..., n \end{aligned} \tag{VII.25.12}$$

25.4.2 Image Pyramid

The iterative method above does reduce some computational load for multi-scale LoG feature maps. However, the kernel size $\sqrt{k^2 - 1}k^{i-1}\sigma_0$ still grows exponentially with i. To address this more effectively, we employ the **downsampling** technique.

Downsampling refers to reducing an image matrix of size $H \times W$ to a smaller matrix of size $\rho H \times \rho W$, where $0 < \rho < 1$. In the continuous domain, this is expressed as:

$$\mathbf{DS}_{1/\rho}[\mathcal{I}](x, y) = \mathcal{I}\left(\frac{x}{\rho}, \frac{y}{\rho}\right)$$

For discrete images, the ρ-times downsampling of image I is denoted as $\mathbf{DS}_{1/\rho}[I]$. Applying a Gaussian kernel $G(\sigma)$ to a downsampled image is equivalent to applying $G\left(\frac{\sigma}{\rho}\right)$ to the original image before downsampling:

$$\mathbf{DS}_{1/\rho}[I] \star \mathcal{G}(\sigma) = \mathbf{DS}_{1/\rho}\left[I \star \mathcal{G}\left(\frac{\sigma}{\rho}\right)\right]$$

Downsampling significantly simplifies the computation of Gaussian convolution for large σ. The simplest downsampling for discrete images is when $\rho = \frac{1}{2}$, where each new pixel is the average of four adjacent pixels in the original image. Unless specified otherwise, we default to $\rho = \frac{1}{2}$:

$$\mathbf{DS}_2[I][u, v] = \frac{1}{4}(I[2u, 2v] + I[2u + 1, 2v] + I[2u, 2v + 1] + I[2u + 1, 2v + 1])$$

Thus, to obtain LoG feature maps at large σ scales, we only need to perform small-scale Gaussian convolutions followed by multiple downsampling steps, effectively simulating larger kernels.

Suppose the original image I undergoes a series of (e.g., $n - 1$) Gaussian blurs to form a set of Gaussian feature maps $I_{1,1}, I_{1,2}, ..., I_{1,n}$. Iteratively applying $L - 1$ rounds of $\rho = 1/2$ downsampling yields L groups of feature maps at different sizes:

$$I_{l+1,i} = \mathbf{DS}_2[I_{l,i}], \quad l = 1, ..., L - 1, i = 1, ..., n \tag{VII.25.13}$$

These feature map groups form a pyramid structure called the **Gaussian pyramid**. Each group of feature maps at the same size is referred to as an **Octave**. Within the Gaussian pyramid, pairwise differences between feature maps in each Octave yield the DoG features:

$$D_{l,i} = I_{l,i+1} - I_{l,i}, \quad l = 1, ..., L, i = 1, ..., n - 1 \tag{VII.25.14}$$

Multiple sets of DoG features with different scales also form a pyramid, called the **DoG feature pyramid**, as shown in Figure VII.25.5.

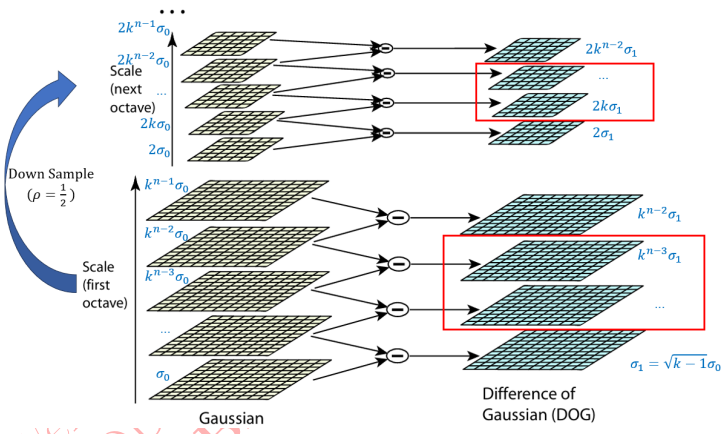

Figure VII.25.5: DoG Feature Pyramid

For a DoG feature pyramid, we have the following parameters: the minimum Gaussian kernel standard deviation σ_0, the scale multiplier k between adjacent Gaussian kernels, the number of Gaussian features per octave n, and the total number of octaves L. From these, we can calculate the standard deviation $\sigma_{l,i}$ of the equivalent LoG kernel for $D_{l,i}$:

$$\sigma_{l,i} = 2^{l-1}k^{i-1}\sigma_1, \quad l = 1, ..., L, i = 1, ..., n - 1 \tag{VII.25.15}$$

where $\sigma_1 = \sqrt{k - 1}$ is the standard deviation of the equivalent LoG kernel for the bottom-layer DoG feature.

In the DoG feature pyramid, not every feature map is used to generate (initial) feature points (details in the next subsection). Within each octave, initial feature points are only generated from feature maps other than the top and bottom layers (i.e., $D_{l,2}, ..., D_{l,n-2}$), as indicated by the red boxes in Figure VII.25.5. We aim for the standard deviations of the equivalent LoG kernels of the middle feature maps across all octaves to form a geometric sequence, i.e.,

$$k\sigma_{l,n-2} = \sigma_{l+1,2}$$

This gives us an equation relating k and n. Solving the equation, we obtain the relationship between k and n:

$$k = 2^{1/(n-3)} \tag{VII.25.16}$$

In practice, we typically want two intermediate DoG feature maps per octave. This means $(n-2)-2=1$, i.e., $n=5$. In this case, $k=\sqrt{2}$.

At this point, we have completed the derivation of the method for computing DoG at different scales. Assuming the discrete Gaussian kernel $G(\sigma)$ with standard deviation σ is known, we summarize the above formulas into the **DoG Pyramid Algorithm**:

Algorithm 79: DoG Pyramid (DoG_pyramid)

```
Input: Image I
Parameter: Initial kernel standard deviation σ_0
Parameter: Number of pyramid levels L, number of
           Gaussian kernels per level n
Output: DoG feature maps D_{1:L,1:n-1}
k ← 2^{1/(n-3)}
I_{1,1} ← I ⋆ G(σ_0)
for i ∈ 2, ..., n do
    I_{1,i} ← I_{1,i-1} ⋆ G(k^{i-1} σ_0)
for l ∈ 2, ..., L do
    for i ∈ 1, ..., n do
        I_{l,i} ← DS_2[I_{l-1,i}]
for l ∈ 1, ..., L do
    for i ∈ 1, ..., n-1 do
        D_{l,i} ← I_{l,i+1} - I_{l,i}
```

25.4.3 Extrema Extraction

We have now obtained the DoG pyramid, which approximates LoG feature maps at different scales. Next, we need to locate the positions of feature points within it.

Consider a 3D feature space where the three dimensions correspond to the image's length, width, and the scale of the LoG kernel, called the **scale space**. For each image $\mathcal{I}$, there exists a scalar-valued function $f_{\mathcal{I}}(x, y, \sigma)$ in the scale space, representing the response strength of the LoG kernel with scale σ centered at (x, y). We can refer to this as the **feature function**. The DoG pyramid is essentially an approximation and discretization of the feature function $f_{\mathcal{I}}(x, y, \sigma)$.

As mentioned earlier, SIFT feature points are based on feature maps, with the core idea being: **taking the positions of local maxima of the feature function as the positions of feature points**. The DoG pyramid is a discretization of the feature function, so the local maxima of the feature function correspond approximately to the maxima in the DoG pyramid. Note: Since the feature function is three-dimensional, extrema must be considered in both pixel position and scale.

Specifically, within each octave, SIFT simultaneously considers three adjacent feature maps $D_{l,i-1}, D_{l,i}, D_{l,i+1}$, forming a $\frac{H}{2^{l-1}} \times \frac{W}{2^{l-1}} \times 3$ 3D matrix. In this matrix, for each $D_{l,i}[u_0, v_0]$, if its value is a local maximum among the 27 adjacent grid points, it is considered a local maximum within that octave. The extrema positions found by this algorithm only exist in feature maps other than the top and bottom layers of each octave.

However, the above assumption is based on the continuous domain. The DoG pyramid we obtain is merely a discrete sampling matrix of the feature function (with decreasing sampling resolution as scale increases). Simply finding local maxima in the matrix may not correspond to maxima in the continuous space.

To address this, we can use the **subpixel refinement** technique. Specifically, near the local maxima found by the above method, we approximate $f_{\mathcal{I}}(x, y, \sigma)$ using a second-order Taylor expansion and solve for the local extrema using Newton's method, as described in Section 3. Newton's method requires knowledge

of the first and second derivatives, which we approximate using finite differences. After several iterations, we can obtain more precise, subpixel-level extrema (i.e., feature point positions).

After determining the precise feature point positions, we perform an additional filtering step because we do not want to extract edges but only corner points. Here, we simply use the same method as Harris corner detection to compute the r-value.

Summarizing the above, we can outline the **SIFT Feature Point Detection Algorithm** as follows:

Algorithm	**SIFT Corner Detection**
Problem Type	Feature Point Detection
Given	Image I
Output	Feature point pixel coordinates $v_1, ..., v_n$
Algorithm Properties	Traditional CV

Algorithm 80: SIFT Corner Detection (SIFT_FP)

Input: Image I
Parameter: Initial kernel standard deviation σ_0
Parameter: Number of pyramid levels L, number of Gaussian kernels per level n
Output: Feature point pixel coordinates $v_1, ..., v_n$

$D_{1:L,1:n-1} \leftarrow \text{DOG\_pyramid}(I; \sigma_0, L, n)$
$res \leftarrow []$
for $l \in 1, ..., L$ **do**
  $candidates \leftarrow \text{find\_max\_in\_27\_cube}(D_{l,1:n-1})$
  for $(v, \sigma) \in candidates$ **do**
    $v_0, \sigma_0 \leftarrow v, \sigma$
    for $k \in 0, 1, ..., K$ **do**
      $g, H \leftarrow \text{diff}(D_{l,1:n-1}, v_k, \sigma_k)$
      $\begin{bmatrix} v_{k+1} \\ \sigma_{k+1} \end{bmatrix} \leftarrow \begin{bmatrix} v_k \\ \sigma_k \end{bmatrix} - H^{-1}g$
      if $|H^{-1}g| < 0.5$ **then**
        break
    $f_{min} \leftarrow \text{interpolate}(D_{l,1:n-1}, v_k, \sigma_k)$
    if $|f_{min}| > 0.04/n$ **then**
      $res \leftarrow res + [(v_k, \sigma_k)]$

25.4.4 SIFT Descriptor

[Main content: SIFT descriptor: HoG gradient histogram; summary of SIFT feature point matching]

[This section will be updated in a future version. Stay tuned.]

25.5 ORB Feature Points

25.5.1 FAST Feature Points

[Main content: FAST corners; grayscale threshold comparison; 4-point initial screening/N-point discrimination; NMS process]

[This section will be updated in a future version. Stay tuned.]

25.5.2 BRIEF Descriptor

[BRIEF descriptor; ORB matching algorithm; RANSAC]

[This section will be updated in a future version. Stay tuned.]

(Note: The LaTeX commands, technical terms (BRIEF, ORB, RANSAC), and structural elements remain unchanged as per the requirements. Only the natural language text has been translated.)

25.6 Basic Concepts of Optical Flow

As mentioned earlier, indirect VO methods require fast and accurate detection and matching of feature points in images. For the matching problem, the methods introduced in the previous sections rely on descriptor computation and matching, which are computationally expensive. In contrast, optical flow is a method that can achieve point-to-point matching without computing descriptors.

Optical flow is a **planar field formed by the projection of the relative velocity of objects to the camera onto the camera plane**, resulting from the relative motion between the observed objects and the observer.

Specifically, suppose an observer captures an image $\mathcal{I}$ of certain objects. When the observer moves relative to (part or all of) the objects within the field of view, the image $\mathcal{I}$ becomes a function of time and pixel space, $\mathcal{I}(x, y, t)$. Assuming that within an infinitesimal time interval dt, the brightness of all objects in the field of view remains unchanged (the **brightness constancy assumption**), the relative velocity of each point (x, y) to the camera will produce a projection on the camera plane. The continuous field formed by these projections, $\mathcal{V}(x, y, t)$, is called optical flow.

Optical flow is closely related to image gradients. Consider the pixel offset of a 3D point in the same space due to the relative motion between the camera and the target. Specifically, suppose at time t, the projected pixel coordinates of point P are (p_x, p_y); at time $t+dt$, the projected pixel coordinates are (p_x+dp_x, p_y+dp_y). According to the brightness constancy assumption, we have:

$$\mathcal{I}(p_x, p_y, t) = \mathcal{I}(p_x + dp_x, p_y + dp_y, t + dt)$$

Assuming $\mathcal{I}$ is sufficiently smooth, its first-order Taylor expansion yields:

$$\mathcal{I}(p_x + dp_x, p_y + dp_y, t + dt) \approx \mathcal{I}(p_x, p_y, t) + \mathcal{I}_x(p_x, p_y, t)dp_x + \mathcal{I}_y(p_x, p_y, t)dp_y + \mathcal{I}_t(p_x, p_y, t)dt$$

Neglecting higher-order terms and substituting the above equation, we obtain:

$$\mathcal{I}_x(p_x, p_y, t)dp_x + \mathcal{I}_y(p_x, p_y, t)dp_y + \mathcal{I}_t(p_x, p_y, t)dt = 0$$

That is:

$$\begin{bmatrix}\mathcal{I}_x & \mathcal{I}_y\end{bmatrix}(p_x, p_y, t)\begin{bmatrix}dp_x \\ dp_y\end{bmatrix} = -\mathcal{I}_t(p_x, p_y, t)dt$$

Since optical flow is defined as:

$$\mathcal{V}[\mathcal{I}](p_x, p_y, t) = \begin{bmatrix}\frac{dp_x}{dt}(p_x, p_y, t) \\ \frac{dp_y}{dt}(p_x, p_y, t)\end{bmatrix} \tag{VII.25.17}$$

We thus have:

$$(\nabla\mathcal{I})^T(p_x, p_y, t)\mathcal{V}[\mathcal{I}](p_x, p_y, t) = -\mathcal{I}_t(p_x, p_y, t) \tag{VII.25.18}$$

Equation VII.25.18 is also known as the **optical flow equation**, which reveals the relationship between optical flow, image gradients, and image brightness changes.

In practical engineering, we need to estimate optical flow $\mathbf{V}[u, v]$ in the discrete domain, i.e., estimate the projection of the relative motion between two consecutive frames (camera and observed objects). This problem is called the **optical flow estimation problem**.

Problem	**Optical Flow Estimation Problem**
Problem Description	Given two frames, estimate the discrete optical flow
Given	Images I_1, I_2
Find	Discrete optical flow $\mathbf{V}[u, v]$

In the discrete domain, optical flow consists of two matrices, δu and δv, with the same dimensions as images I_1 and I_2.

$$\mathbf{V}[u_0, v_0] = \begin{bmatrix} \delta u[u_0, v_0] \\ \delta v[u_0, v_0] \end{bmatrix} \tag{VII.25.19}$$

We can write the optical flow equation in the discrete domain as:

$$(\nabla I)^T[u_0, v_0]\mathbf{V}[\mathbf{I}][u_0, v_0] = (I_2 - I_1)[u_0, v_0] \tag{VII.25.20}$$

In various VO/vSLAM methods, if we require the full optical flow information between two frames, it is called **dense optical flow**; if we only focus on the optical flow at a few specific points, it is called **sparse optical flow**.

For indirect methods, the purpose of using optical flow is to establish feature point correspondences. Suppose we have feature point pixel coordinates $v^1_{1:m}$ and $v^2_{1:n}$ detected in images I_1 and I_2, respectively, and the optical flow $\mathbf{V}$ from I_1 to I_2 is known. Feature point matching can be easily achieved: we simply compute the optical flow at the feature points of I_1, add v^1 to the optical flow, and then search for v^2 in the vicinity.

Algorithm	**Optical Flow Feature Point Matching**
Problem Type	Feature Point Matching Problem
Given	Images I_1, I_2, optical flow $\mathbf{V}$ 2D feature points of Image 1: $v^1_1, ...v^1_i, ..., v^1_m$ 2D feature points of Image 2: $v^2_1, ..., v^2_j, ..., v^2_n$
Find	Matching index pairs $(i_1, j_1), ..., (i_k, j_k)$
Algorithm Property	Traditional CV

Algorithm 81: Optical Flow Feature Point Matching (match_FP_optflow)

Input: Images I_1, I_2, optical flow $\mathbf{V}$
Input: 2D feature points of Image 1: $v^1_{1:m}$
Input: 2D feature points of Image 2: $v^2_{1:n}$
Parameter: Search radius r
Output: Matching index pairs $(i_1, j_1), ..., (i_k, j_k)$

```
pairs ← []
candidates ← [1, 2, ..., n]
for i ∈ 1, ..., m do
    v̂ ← v_i^1 + V[v_i^1]
    r_min ← r + 1
    for j ∈ candidates do
        r_j ← ||v_j^2 − v̂||
        if r_j < r then
            candidates ← candidates − [j]
        if r_j < r_min then
            r_min ← r_j
            j_match ← j
    if r_min < r then
        pairs ← pairs + [(i, j)]
```

Therefore, the optical flow used in indirect methods is generally sparse. Correspondingly, direct methods typically use dense or semi-dense optical flow.

25.7 L-K Optical Flow

25.7.1 Least Squares Solution

Next, we consider the problem of solving optical flow in the discrete domain. According to Equation VII.25.20, for the optical flow at point $[u_0, v_0]$, there are two variables, δu and δv, but only one equation, making the

problem underconstrained. To address this, we can assume that the motion projection is the same within a window $W[u_0, v_0]$ around $[u_0, v_0]$. This allows us to construct a system of equations based on the window.

$$\begin{bmatrix} I_u[u_0,v_0] & I_v[u_0,v_0] \\ I_u[u_1,v_1] & I_v[u_1,v_1] \\ \dots & \dots \\ I_u[u_w,v_w] & I_v[u_w,v_w] \end{bmatrix} \mathbf{V}[u_0,v_0] = \begin{bmatrix} (I_2 - I_1)[u_0,v_0] \\ (I_2 - I_1)[u_1,v_1] \\ \dots \\ (I_2 - I_1)[u_w,v_w] \end{bmatrix}, \quad (u_i, v_i) \in W[u_0, v_0]$$

This is an overdetermined system of linear equations, for which a least-squares solution can be obtained. Omitting the derivation process, we directly present the result:

$$\mathbf{V}[u_0,v_0] = H_{LK}[u_0,v_0]^{-1} g_{LK}[u_0,v_0] \tag{VII.25.21}$$

where

$$\begin{aligned} H_{LK}[u_0,v_0] &= \sum_{(u_i,v_i)\in W} \begin{bmatrix} I_u^2[u_i,v_i] & I_u \odot I_v[u_i,v_i] \\ I_u \odot I_v[u_i,v_i] & I_v^2[u_i,v_i] \end{bmatrix} \\ g_{LK}[u_0,v_0] &= \sum_{(u_i,v_i)\in W} \begin{bmatrix} I_u \odot (I_2 - I_1)[u_i,v_i] \\ I_v \odot (I_2 - I_1)[u_i,v_i] \end{bmatrix} \end{aligned} \tag{VII.25.22}$$

This algorithm requires that $H_{LK}[u_0, v_0]$ be nonsingular. Notably, the matrix $H_{LK}[u_0, v_0]$ is highly similar to $H[u_0, v_0]$ from the "Harris Corner" section, equivalent to taking the average kernel for K_W in the Harris corner method. Therefore, the existence of a nonsingular solution essentially implies that u_0, v_0 are Harris corner points.

The above window-based least-squares optical flow estimation method was proposed by Lucas and Kanade in 1981, hence it is also referred to as the **L-K optical flow method**.

Algorithm 82: L-K Optical Flow (calc_optflow_LK)

Input: Images I_1, I_2
Parameter: Sliding window convolution kernel K_w
Output: Discrete optical flow $\mathbf{V}$
$I_u, I_v \leftarrow I_1 \star K_u, I_1 \star K_v$
$A, B, D, E, F \leftarrow 0_{5\times H\times W}$
$A, B, D \leftarrow (I_u \odot I_u) \star K_w, (I_u \odot I_v) \star K_w, (I_v \odot I_v) \star K_w$
$E, F \leftarrow (I_u \odot (I_2 - I_1)) \star K_w, (I_v \odot (I_2 - I_1)) \star K_w$
$\delta u \leftarrow D \odot E - B \odot F$
$\delta v \leftarrow A \odot F - B \odot E$
$H_{det} \leftarrow A \odot D - B \odot B$
$\mathbf{V} \leftarrow \begin{bmatrix} \delta u \oslash H_{det} \\ \delta v \oslash H_{det} \end{bmatrix}$

Algorithm	**L-K Optical Flow**
Problem Type	Optical Flow Estimation
Known	Images I_1, I_2
Objective	Discrete optical flow $\mathbf{V}$
Algorithm Property	Traditional CV

25.7.2 Iterative Solution

The above least-squares optical flow algorithm relies on the first-order Taylor expansion approximation of the image function, assuming the motion $(\delta u, \delta v)$ is small. When the motion is large, this algorithm cannot estimate the optical flow accurately.

To address this issue, we can use an iterative solution method. For a point $[u_0, v_0]$, suppose in the k-th iteration, we already have an optical flow estimate $\mathbf{V}_k[u_0, v_0]$. At this point, we compensate the optical flow estimate into I_1, i.e.,

$$I_{1,k}[u_0, v_0] = \mathbf{Sft}[I_1, \mathbf{V}_k] := I_1[u_0 - (\delta u)_k(u_0), v_0 - (\delta v)_k(v_0)] \tag{VII.25.23}$$

Subsequently, we use the above method to solve the optical flow $\delta\mathbf{V}_k$ from $I_{1,k}$ to I_2, and then perform iterative updates:

$$\mathbf{V}_{k+1}[u_0, v_0] = \mathbf{V}_k[u_0, v_0] + \delta\mathbf{V}_k[u_0, v_0] \tag{VII.25.24}$$

Thus, we can summarize the **iterative L-K optical flow algorithm** as follows:

Algorithm	**Iterative L-K Optical Flow**
Problem Type	Optical Flow Estimation
Known	Images I_1, I_2
Objective	Discrete optical flow $\mathbf{V}$
Algorithm Property	Traditional CV

Algorithm 83: Iterative L-K Optical Flow (calc_optflow_iterLK)

Input: Images I_1, I_2
Parameter: Initial optical flow $\mathbf{V}_0 = 0_{2\times H\times W}$
Output: Discrete optical flow $\mathbf{V}$
for $k \in 0, 1, ..., K_n$ **do**
  $I_{1,k} \leftarrow \mathbf{Sft}[I_1, \mathbf{V}_k]$
  $\delta\mathbf{V}_k \leftarrow$ calc_optflow_LS$(I_{1,k}, I_2)$
  $\mathbf{V}_{k+1} \leftarrow \mathbf{V}_k + \delta\mathbf{V}_k$
$\mathbf{V} \leftarrow \mathbf{V}_k$

25.7.3 Pyramid Iterative Solution

The above problem is essentially a non-convex optimization problem, where the objective is to minimize the photometric error after compensation, and the optimization variables are the optical flow across the entire image. For cases with large motion, even with the iterative solution method, the performance is not satisfactory.

In fact, for non-convex optimization problems, the choice of initial values is often crucial. In the above algorithm, all initial optical flow values are set to 0. However, we can use an image pyramid approach to obtain better optical flow estimates.

Specifically, according to the definition of optical flow, the optical flow of a downsampled image is equivalent to downsampling the optical flow and then scaling it, i.e.,

$$\mathbf{V}[\mathbf{DS}_{1/\rho}[I]] = \rho\mathbf{DS}_{1/\rho}[\mathbf{V}[I]] \tag{VII.25.25}$$

Therefore, we can first estimate the optical flow $\mathbf{V}_\rho$ on the ρ-times downsampled image $\mathbf{DS}_{1/\rho}[I]$, then upsample $\mathbf{V}_\rho$ to serve as the initial optical flow for the original image.

Further, we can adopt a strategy similar to SIFT feature points, starting from highly downsampled images (i.e., the top of the pyramid) and estimating layer by layer downward. Each layer uses the estimation result from the previous layer as the initial value, ultimately yielding a more accurate optical flow estimate for the original resolution image. This approach is called the **pyramid L-K optical flow method**. We summarize the algorithm as follows:

Algorithm	**Pyramid L-K Optical Flow**
Problem Type	Optical Flow Estimation
Known	Images I_1, I_2
Objective	Discrete optical flow $\mathbf{V}$
Algorithm Property	Traditional CV

Algorithm 84: Pyramidal L-K Optical Flow

Input: Images I_1, I_2
Parameter: Number of pyramid levels L
Output: Discrete optical flow $\mathbf{V}$
$I_{1,1}, I_{1,2} \leftarrow I_1, I_2$
for $l \in 2, 3, ..., L$ **do**
  $I_{l,1} \leftarrow \mathbf{DS}_2[I_{l-1,1}]$
  $I_{l,2} \leftarrow \mathbf{DS}_2[I_{l-1,2}]$
$\mathbf{V}_{L,0} \leftarrow 0_{2\times H/2^L \times W/2^L}$
for $l \in L, ..., 1$ **do**
  $\mathbf{V}_l \leftarrow \text{calc\_optflow\_iterLK}(I_{l,1}, I_{l,2}, \mathbf{V}_{l,0})$
  $\mathbf{V}_{l-1,0} = 2\mathbf{DS}_{1/2}[\mathbf{V}_l]$
$\mathbf{V} \leftarrow \mathbf{V}_1$

26 Point-to-Pose Estimation

26.1 Problem Formulation

In the previous chapter, we introduced the method of feature point extraction and matching using the indirect approach, which yields matched feature points between two images. Next, we will estimate the pose based on this information, i.e., the **point-to-pose estimation problem**.

Specifically, given a set of N points with pixel coordinates or camera-frame coordinates under two different camera poses, we aim to solve the homogeneous matrix T between the two poses. Typically, in VO/vSLAM initialization, no prior 3D information is available, so all feature point coordinates are pixel coordinates (2D). Once some 3D information is obtained, camera-frame coordinates (3D) can be used. Depending on whether the known quantities are pixel coordinates (2D) or camera-frame coordinates (3D), we can define three distinct point-to-pose estimation problems:

Problem	**2D-2D Point-to-Pose Estimation**
Description	Given pixel coordinates of matched points under two poses, solve the homogeneous matrix between poses
Known	Pixel coordinates of points under pose 1: $\upsilon_1^1, \upsilon_2^1, ..., \upsilon_N^1$ Pixel coordinates of points under pose 2: $\upsilon_1^2, \upsilon_2^2, ..., \upsilon_N^2$ Camera intrinsic matrix K
Solve	Homogeneous matrix between poses T_2^1

Problem	**3D-3D Point-to-Pose Estimation**
Description	Given camera-frame coordinates of matched points under two poses, solve the homogeneous matrix between poses
Known	Camera-frame coordinates of points under pose 1: $p_1^1, p_2^1, ..., p_N^1$ Camera-frame coordinates of points under pose 2: $p_1^2, p_2^2, ..., p_N^2$
Solve	Homogeneous matrix between poses T_2^1

Problem	**3D-2D Point-to-Pose Estimation**
Description	Given a set of matched points with 3D coordinates and pixel coordinates under one pose, solve the homogeneous matrix between poses
Known	Camera-frame coordinates of points under pose 1: $p_1^1, p_2^1, ..., p_N^1$ Pixel coordinates of points under pose 2: $\upsilon_1^2, \upsilon_2^2, ..., \upsilon_N^2$ Camera intrinsic matrix K
Solve	Homogeneous matrix between poses T_2^1

In all three problems, we assume the intrinsic matrix K is known and the imaging process is free of distortion (distortion has been compensated).

Next, we first focus on the 2D-2D and 3D-3D problems, followed by the 3D-2D problem.

26.2 2D-2D Problem

26.2.1 Epipolar Constraint

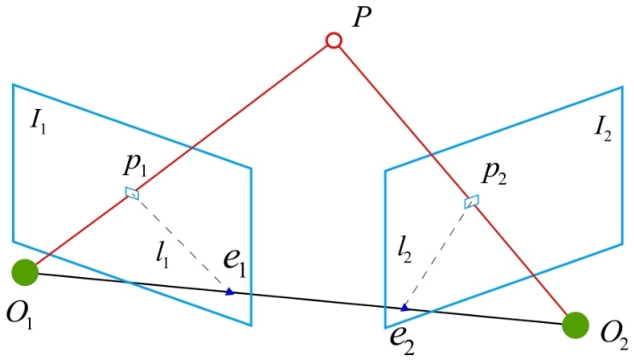

Figure VII.26.1: Epipolar Constraint

First, we model the 2D-2D problem, a method also known as epipolar geometry.

As shown in Figure VII.26.1, the spatial point P and the camera centers O_1, O_2 at poses 1 and 2 are connected by lines, i.e., rays. The intersections of these rays with the image planes I_1, I_2 are the 2D point positions p_1, p_2 on the respective images. The points P, O_1, and O_2 form a plane (assuming they are not colinear), called the **epipolar plane**. The line connecting O_1 and O_2 is called the **baseline**, and its intersections with the image planes I_1, I_2 are called the **epipoles** e_1, e_2. The lines connecting the epipoles and the image points p_1e_1, p_2e_2 are called the **epipolar lines**.

Assume the 2D pixel coordinates of p_1, p_2 are v^1, v^2, and the 3D coordinates of point P in camera frames 1 and 2 are p^1, p^2. Assuming the focal plane depths z^1, z^2 of p^1, p^2 are non-zero, according to the camera model and coordinate transformation rules, we have:

$$v^1 = \frac{1}{z^1} K p^1$$
$$v^2 = \frac{1}{z^2} K p^2$$
$$p^1 = R_2^1 p_2 + t_{12}^1$$

Substituting the pixel coordinates, we obtain:

$$z^1 K^{-1} v^1 = R_2^1 (z^2 K^{-1} v^2) + t_{12}^1$$

Which simplifies to:

$$K^{-1} v^1 = \frac{z^2}{z^1} R_2^1 (K^{-1} v^2) + \frac{1}{z^1} t_{12}^1$$

Taking the cross product with t_{12}^1 on both sides:

$$t_{12}^1 \times K^{-1} v^1 = t_{12}^1 \times \frac{z^2}{z^1} R_2^1 (K^{-1} v^2)$$

Left-multiplying both sides by $(K^{-1}v^1)^T$:

$$(K^{-1}v^1)^T t_{12}^1 \times K^{-1} v^1 = \frac{z^2}{z^1} (K^{-1}v^1)^T t_{12}^1 \times R_2^1 (K^{-1} v^2)$$

Noting that vector a is perpendicular to $a \times b$, we have $a^T(a \times b) = 0$, leading to:

$$0 = \frac{z^2}{z^1} (K^{-1}v^1)^T t_{12}^1 \times R_2^1 (K^{-1} v^2)$$

Rearranging, we obtain:

$$(v^1)^T K^{-T}(t_{12}^1)_\times R_2^1 K^{-1} v^2 = 0 \tag{VII.26.1}$$

Equation VII.26.1 is known as the **epipolar constraint**, which describes the relationship between the (augmented) pixel coordinates of the same spatial point under two different camera poses. We define:

$$F(T_2^1, K) := K^{-T}(t_{12}^1)_\times R_2^1 K^{-1} \tag{VII.26.2}$$

The matrix F is called the **Fundamental Matrix**. Similarly, we define:

$$E(T_2^1) := (t_{12}^1)_\times R_2^1 \tag{VII.26.3}$$

The matrix E is called the **Essential Matrix**.

Both the fundamental and essential matrices are 3×3 matrices related to the relative pose.

26.2.2 Triangulation

Before solving the 2D-2D problem using epipolar geometry, we first introduce the triangulation method.

Triangulation, also known as depth estimation, is the inverse problem of point-to-pose estimation. Pose estimation involves solving the homogeneous matrix given camera-frame/pixel coordinates, while triangulation involves solving the camera-frame coordinates or focal plane depth given the homogeneous matrix and pixel coordinates.

Problem	**Triangulation Problem**
Description	Given the homogeneous matrix between two poses and pixel coordinates of matched points, solve the 3D coordinates
Known	Pixel coordinates v^1, v^2 under poses 1 and 2 Homogeneous matrix between poses T_2^1 Camera intrinsic matrix K
Solve	Camera-frame coordinates p^1, p^2 under poses 1 and 2

Why introduce triangulation first? Because the 2D-2D problem has multiple solutions. This ambiguity arises from an implicit constraint—during camera imaging, all image plane distances must be positive. However, the mathematical descriptions of the camera model, fundamental matrix, and essential matrix themselves do not incorporate this constraint, so we must enforce it manually. In enforcing this constraint, we need to estimate depths from tentative poses and then use these depths to eliminate or select poses.

In practical VO and vSLAM systems, triangulation is also crucial because, in the absence of depth priors, the depth information of a set of feature points during their first observation must be inferred purely from 2D data. Triangulation is precisely this inference method. In essence, triangulation solves for two depths by leveraging the relationship between normalized coordinates and relative poses.

Specifically, based on the epipolar geometry discussed in the previous section, we have:

$$z^1 K^{-1} v^1 = R_2^1 (z^2 K^{-1} v^2) + t_{12}^1$$

Taking the cross product with $K^{-1}v^1$ on both sides yields:

$$K^{-1}v^1 \times R_2^1 K^{-1} v^2 z^2 + K^{-1} v^1 \times t_{12}^1 = 0$$

This is an algebraic equation for the scalar z^2. Let:

$$a = K^{-1}v^1 \times R_2^1 K^{-1} v^2$$
$$b = K^{-1}v^1 \times t_{12}^1$$

Then:

$$z^2 = -\frac{b^T a}{a^T a}$$

Subsequently, the same technique can be used to solve for z^1, and the camera model can be applied to compute p^1, p^2. The above algorithm is summarized below:

Algorithm	**Triangulation**
Problem Type	Triangulation Problem
Given	Pixel coordinates v^1, v^2 under poses 1 and 2 Homogeneous transformation T_2^1 between poses Camera intrinsics K
Solve For	Camera-frame coordinates p^1, p^2 under poses 1 and 2
Algorithm Property	Approximate solution

Algorithm 85: Triangulation

Input: Pixel coordinates v^1, v^2 under poses 1 and 2
Input: Homogeneous transformation T_2^1 between poses
Input: Camera intrinsics K
Output: Camera-frame coordinates p^1, p^2 under poses 1 and 2

$a \leftarrow K^{-1}v^1 \times R_2^1 K^{-1} v^2$
$b \leftarrow K^{-1}v^1 \times t_{12}^1$
$z^2 \leftarrow -b^T a / a^T a$
$c \leftarrow K^{-1}v^1$
$d \leftarrow R_2^1(z^2 K^{-1} v^2) + t_{12}^1$
$z^1 \leftarrow d^T c / c^T c$
$p^1 \leftarrow z^1 K^{-1} v^1$
$p^2 \leftarrow z^2 K^{-1} v^2$

The corresponding Python code is shown below:

```python
def triangulate_points(N, nu_1s, nu_2s, T_2_to_1, K):
    assert nu_1s.shape == (N, 3) and nu_2s.shape == (N, 3)
    assert T_2_to_1.shape == (4, 4) and K.shape == (3, 3)
    p1s, p2s = np.zeros([N, 3]), np.zeros([N, 3])
    invK = np.linalg.inv(K)
    R_2_to_1, t_12_1 = T_2_to_1[:3, :3], T_2_to_1[:3, 3]
    for i in range(N):
        nu1, nu2 = nu_1s[i], nu_2s[i]
        c = invK @ nu1
        a = np.cross(c, R_2_to_1 @ invK @ nu2)
        b = np.cross(c, t_12_1)
        z2 = - b[None, :] @ a[:, None] / np.sum(a**2)
        d = R_2_to_1 @ (z2 * invK @ nu2) + t_12_1
        z1 = d[None, :] @ c[:, None] / np.sum(c**2)
        p1s[i], p2s[i] = z1 * invK @ nu1, z2 * invK @ nu2
    return p1s, p2s
```

26.2.3 Pose Recovery

In the previous section, we introduced the epipolar constraint. It can be seen that, given corresponding point pairs under the same set of poses, solving for the fundamental matrix or essential matrix allows us to address the 2D-2D pose estimation problem.

We will detail the solving of the fundamental matrix (F) and essential matrix (E) in the next section. Before that, let us first explain how to recover T_2^1, i.e., R_2^1 and t_{12}^1, from E and F.

In fact, the fundamental matrix can be converted to the essential matrix:

$$E = K^T F K \tag{VII.26.4}$$

Thus, we only need to recover T_2^1 from E. Noting the special structure of E, we can use the algorithm shown below to recover rotation and translation. Here, the triangulation method introduced earlier is employed to resolve ambiguities.

Algorithm	**Pose Recovery from Essential Matrix**
Problem Type	2D-2D Point Pose Estimation
Given	Essential matrix E Pixel coordinates $v_1^1, v_2^1, ..., v_N^1$ under pose 1 Pixel coordinates $v_1^2, v_2^2, ..., v_N^2$ under pose 2
Solve For	Homogeneous transformation T_2^1 between poses
Algorithm Property	Approximate solution

Algorithm 86: Essential Matrix to Pose Recovery (E_to_Rt)

Input: Essential matrix E
Input: Pixel coordinates of points in pose 1 $v_1^1, v_2^1, ..., v_N^1$
Input: Pixel coordinates of points in pose 2 $v_1^2, v_2^2, ..., v_N^2$
Output: Homogeneous transformation matrix T_2^1

$U, \Sigma, V^T \leftarrow \text{SVD\_calc}(E)$

$$W, Z \leftarrow \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$R_1, R_2 \leftarrow UWV^T, UZV^T$
$t_1, t_2 \leftarrow U_{:,3}, -U_{:,3}$
for $(R, t) \in [(R_1, t_1), (R_1, t_2), (R_2, t_1), (R_2, t_2)]$ **do**
  for $i \in 1, ..., N$ **do**
    $d_i^1, d_i^2 \leftarrow \text{triangulate}(R, t, v_i^1, v_i^2)$
  if $d_{1:N}^1 > 0$ *and* $d_{1:N}^2 > 0$ **then**
    $R_2^1, t_{12}^1 \leftarrow R, t$

$$T_2^1 \leftarrow \begin{bmatrix} R_2^1 & t_{12}^1 \\ 0_{1\times 3} & 1 \end{bmatrix}$$

The corresponding Python code is shown below:

```python
def E_to_Rt(N, nu_1s, nu_2s, E, K):
    assert nu_1s.shape == (N, 3) and nu_2s.shape == (N, 3)
    assert E.shape == (3, 3) and K.shape == (3, 3)
    U, sigma, VT = np.linalg.svd(E)
    W = np.array([[0, 1, 0],
                  [-1, 0, 0],
                  [0, 0, 1]])
    R1, R2 = U @ W @ VT, U @ W.T @ VT
    if np.linalg.det(R1) < 0:
        R1 = - R1
    if np.linalg.det(R2) < 0:
        R2 = - R2
    t1, t2 = U[:, 2], - U[:, 2]
    candidates = [(R1, t1), (R1, t2), (R2, t1), (R2, t2)]
    for R, t in candidates:
        T_2_to_1 = np.zeros([4,4])
        T_2_to_1[:3, :3], T_2_to_1[:3, 3], T_2_to_1[3, 3] = R, t, 1
        p1s, p2s = triangulate_points(N, nu_1s, nu_2s, T_2_to_1, K)
        if np.all(p1s[:, 2] > 0) and np.all(p2s[:, 2] > 0):
            return T_2_to_1
    return None
```

26.2.4 8-Point Algorithm for Pose Estimation

In previous sections, we introduced the epipolar constraint. It can be observed that given corresponding point pairs under the same set of poses, solving for the fundamental matrix or essential matrix allows us to address the 2D-2D pose estimation problem using the pose recovery method described in the previous section.

Special attention is required here: Neither the E matrix nor the F matrix has 9 degrees of freedom. In fact, for the solved T_2^1, multiplying its translation component t_{12}^1 by any scale factor $r \neq 0$ will still satisfy the epipolar constraint. This is the well-known **monocular scale ambiguity**: **Without known baseline length, it is impossible to recover the scale of motion using only a monocular visual sensor.**

Therefore, the F matrix actually has 8 rather than 9 unknowns. We can enforce the bottom-right element to be 1, and the resulting F matrix can still satisfy all constraints when multiplied by a scale factor r. To solve a linear system with 8 unknowns, 8 nonlinearly independent constraints are required, hence this method demands $N \geq 8$ point pairs. In practice, we typically select $N > 8$ and solve for the F matrix with minimal error using an overdetermined system. Thus, this method is also called the **8-point algorithm**.

In practical VO and vSLAM algorithms, there are two approaches to address monocular scale ambiguity. One is to use a stereo camera with a known baseline to introduce scale constraints. The other is to fuse data from other sensors, such as IMUs, to resolve the scale issue.

Moreover, even disregarding scale ambiguity, the F matrix does not have 8 degrees of freedom. Therefore, the computed F' matrix requires an additional rank processing step to obtain the final F matrix. The method involves removing the component corresponding to the smallest eigenvalue to enforce rank-2 constraint.

Summarizing the above content, the 8-point algorithm for pose estimation is outlined below:

Algorithm	**8-Point Algorithm for Pose Estimation**
Problem Type	2D-2D Point Pose Estimation
Given	Pixel coordinates of points in pose 1 $\upsilon_1^1, \upsilon_2^1, ..., \upsilon_N^1$ Pixel coordinates of points in pose 2 $\upsilon_1^2, \upsilon_2^2, ..., \upsilon_N^2$ Camera intrinsic matrix K
Solve For	Homogeneous transformation matrix T_2^1
Algorithm Property	Approximate solution

Algorithm 87: 8-Point Algorithm for Pose Estimation

Input: Pixel coordinates of points in pose 1 $v_1^1, v_2^1, ..., v_N^1$
Input: Pixel coordinates of points in pose 2 $v_1^2, v_2^2, ..., v_N^2$
Input: Camera intrinsic matrix K
Output: Fundamental matrix F, Homogeneous transformation matrix T_2^1

$A \leftarrow 0_{N\times 9}$
for $i \in 1, \ldots, N$ **do**
  $[u_2, v_2, 1]^T \leftarrow v_i^2$
  $A_{i,:} \leftarrow [u_2(v_i^1)^T \quad v_2(v_i^1)^T \quad (v_i^1)^T]$
$f \leftarrow \text{linear\_solve}(Af = 0)$
$F' \leftarrow \begin{bmatrix} f_1 & f_4 & f_7 \\ f_2 & f_5 & f_8 \\ f_3 & f_6 & 1 \end{bmatrix}$
$U, \text{diag}\{r, s, t\}, V^T \leftarrow \text{SVD\_calc}(F')$
$F \leftarrow U\,\text{diag}\{r, s, 0\}V^T$
$E \leftarrow K^T F K$
$T_2^1 \leftarrow \text{E\_to\_Rt}(E, v_:^1, v_:^2)$

The corresponding Python code is shown below.

```python
def calc_vo_Fmat_8p(N, nu_1s, nu_2s, K):
    assert nu_1s.shape == (N, 3) and nu_2s.shape == (N, 3)
    assert K.shape == (3, 3)
    A = np.zeros([N, 9])
    for i in range(N):
        u2, v2, nu1 = nu_2s[i, 0], nu_2s[i, 1], nu_1s[i]
        A[i, :] = np.concatenate([u2*nu1, v2*nu1, nu1])
    f = scipy.linalg.null_space(A)[:, 0]
    F_ = np.array([
        [f[0], f[3], f[6]],
        [f[1], f[4], f[7]],
        [f[2], f[5], f[8]],
    ]) / np.max(f)
    U, sigma, VT = np.linalg.svd(F_)
    F = U @ np.diag([sigma[0], sigma[1], 0]) @ VT
    E = K.T @ F @ K
    T_2_to_1 = E_to_Rt(N, nu_1s, nu_2s, E, K)
    return T_2_to_1
```

26.2.5 4-Point Method for Homography Matrix Estimation

The estimation of the fundamental matrix/essential matrix for pose requires 8 non-coplanar points; otherwise, the matrix A in the solving process becomes non-invertible. So, can we estimate the pose when all feature points are coplanar?

Definition of Homography Matrix

Assume all feature points lie on the plane $\mathbf{n}^T\mathbf{x}^n + d = 0$, where $\mathbf{n}$ is the plane's normal vector, and x^n represents the 3D coordinates of the point in the world frame. Suppose there are two distinct camera frames, $c1$ and $c2$, with corresponding poses $R_{c1} = R_n^{c1}, t_{c1} = t_{nc1}^n$ and $R_{c2} = R_n^{c2}, t_{c2} = t_{nc2}^n$, respectively. For points $\mathbf{x}_i, i = 1, ..., N$ in the n frame, we have:

$$\begin{aligned} 0 &= \mathbf{n}^T\mathbf{x}_i + d \\ \mathbf{x}_i^{c1} &= R_{c1}(\mathbf{x}_i - t_{c1}) \\ \mathbf{x}_i^{c2} &= R_{c2}(\mathbf{x}_i - t_{c2}) \end{aligned} \tag{VII.26.5}$$

Let

$$R = R_{c1}R_{c2}^T$$
$$t = R_{c1}(t_{c2} - t_{c1})$$

Then,

$$\mathbf{x}_i^{c1} = R\mathbf{x}_i^{c2} + t \tag{VII.26.6}$$

Substituting the first line of Equation VII.26.5 into the third line, we get:

$$\mathbf{n}^T(R_{c2}^T\mathbf{x}_i^{c2} + t_{c2}) + d = 0$$

That is,

$$(R_{c2}\mathbf{n})^T\mathbf{x}_i^{c2} + \mathbf{n}^T t_{c2} + d = 0$$

Let

$$\mathbf{n}_2 = R_{c2}\mathbf{n}$$
$$d_2 = \mathbf{n}^T t_{c2} + d$$

Then,

$$\mathbf{n}_2^T\mathbf{x}_i^{c2} = -d_2$$

Or,

$$\frac{\mathbf{n}_2^T\mathbf{x}_i^{c2}}{-d_2} = 1$$

Substituting into Equation VII.26.5, we have:

$$\mathbf{x}_i^{c1} = R\mathbf{x}_i^{c2} + t\frac{\mathbf{n}_2^T\mathbf{x}_i^{c2}}{-d_2}$$

That is,

$$\mathbf{x}_i^{c1} = (R - \frac{t\mathbf{n}_2^T}{d_2})\mathbf{x}_i^{c2}$$

Now, suppose we do not have 3D coordinates but only pixel coordinates, i.e.,

$$v_{i,1} = K\mathbf{x}_i^{c1}/z_i^{c1}$$
$$v_{i,2} = K\mathbf{x}_i^{c2}/z_i^{c2}$$

Then,

$$K^{-1}v_{i,1}z_i^{c1} = (R - \frac{t\mathbf{n}_2^T}{d_2})K^{-1}v_{i,2}z_i^{c2}$$

That is,

$$v_{i,1} = \frac{z_i^{c2}}{z_i^{c1}}K(R - \frac{t\mathbf{n}_2^T}{d_2})K^{-1}v_{i,2}$$

Let

$$H = K(R - \frac{t\mathbf{n}_2^T}{d_2})K^{-1} \tag{VII.26.7}$$

And

$$\lambda_i = \frac{z_i^{c2}}{z_i^{c1}}$$

Then,

$$v_{i,1} = \lambda_i H v_{i,2} \tag{VII.26.8}$$

Here, H is called the homography matrix, a 3×3 matrix containing information about the relative pose and plane position. λ_i is the scale factor. Equation VII.26.8 indicates that the 2D coordinates of a set of coplanar points satisfy a linear relationship, with the linear coefficients being the product of the homography matrix and the scale factor.

Similar to the essential matrix, given multiple pairs of 2D point coordinates, the homography matrix can also be estimated. This is the problem of homography matrix estimation.

Problem	**2D-2D Point Homography Matrix Estimation**
Problem Description	Given pixel coordinates of multiple coplanar point pairs from two poses, estimate the homography matrix
Given	Pixel coordinates of points in pose 1: $v_1^1, v_2^1, ..., v_N^1$ Pixel coordinates of points in pose 2: $v_1^2, v_2^2, ..., v_N^2$ Camera intrinsic matrix K
To Estimate	Homography matrix H

Homography Matrix Estimation

The H matrix is a 3×3 matrix. Expanding it into its elements:

$$\begin{bmatrix} u_{i;1} \\ v_{i;1} \\ 1 \end{bmatrix} = \hat{\lambda}_i \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{bmatrix} \begin{bmatrix} u_{i;2} \\ v_{i;2} \\ 1 \end{bmatrix}$$

Note that here we set the original h_9 position to 1, which essentially divides the entire homography matrix by h_9 and multiplies λ_i by h_9 to obtain $\hat{\lambda}_i$. That is,

$$\begin{aligned} u_{i;1} &= \hat{\lambda}_i(h_1 u_{i;2} + h_2 v_{i;2} + h_3) \\ v_{i;1} &= \hat{\lambda}_i(h_4 u_{i;2} + h_5 v_{i;2} + h_6) \\ 1 &= \hat{\lambda}_i(h_7 u_{i;2} + h_8 v_{i;2} + 1) \end{aligned}$$

Using the third equation to eliminate $\hat{\lambda}_i$, we get:

$$\begin{aligned} 0 &= (h_1 u_{i;2} + h_2 v_{i;2} + h_3) + u_{i;1}(h_7 u_{i;2} + h_8 v_{i;2} + 1) \\ 0 &= (h_4 u_{i;2} + h_5 v_{i;2} + h_6) + v_{i;1}(h_7 u_{i;2} + h_8 v_{i;2} + 1) \end{aligned}$$

Here, we have 8 unknowns and 2 equations. These are the constraints provided by the 2D coordinates of the i-th point pair. If we have at least 4 point pairs, we can form at least 8 equations to solve for the homography matrix, which is the 4-point method. We summarize the algorithm as follows:

Algorithm	**4-Point Method for Homography Matrix Estimation**
Problem Type	2D-2D Point Pair Homography Estimation
Given	Pixel coordinates of points in pose 1: $v_1^1, v_2^1, ..., v_N^1$ Pixel coordinates of points in pose 2: $v_1^2, v_2^2, ..., v_N^2$ Camera intrinsic matrix K
To Estimate	Homography matrix H
Algorithm Property	Approximate solution

Algorithm 88: 4-Point Method for Homography Matrix Estimation

Input: Pixel coordinates of points in pose 1: $v_1^1, v_2^1, ..., v_N^1$

Input: Pixel coordinates of points in pose 2: $v_1^2, v_2^2, ..., v_N^2$

Input: Camera intrinsic matrix K

Output: Homography matrix H

$A \leftarrow 0_{2N\times 8}$

$b \leftarrow 0_{2N}$

for $i \in 1, \dots, N$ **do**

$[u_1, v_1, 1]^T \leftarrow K^{-1} v_i^1$

$[u_2, v_2, 1]^T \leftarrow K^{-1} v_i^2$

$A_{2i,:} \leftarrow [u_2, v_2, 1, 0, 0, 0, -u_1u_2, -u_1v_2]^T$

$A_{2i+1,:} \leftarrow [0, 0, 0, u_2, v_2, 1, -v_1u_2, -v_1v_2]^T$

$b_{2i:2i+2} \leftarrow [u_1, v_1]$

$h \leftarrow \text{linear\_solve}(Ah = b)$

$$H \leftarrow \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{bmatrix}$$

However, it should be noted that the homography matrix obtained through this method differs from the true homography matrix by a scale factor, corresponding to the bottom-right element of the original homography matrix.

26.2.6 6-Point Method for Velocity Estimation

Previously, we introduced two pose estimation methods based on 2D point pairs, where the essential matrix requires at least 8 point pairs and the homography matrix requires at least 4 coplanar point pairs. Additionally, there is a method that uses at least 6 point pairs to estimate the camera's 3D velocity and angular velocity through optical flow.

Optical Flow Equations

First, let us derive the relationship between optical flow and the camera's velocity and angular velocity. Assume the navigation frame is n and the camera frame is c. The 3D coordinates of a point P in the camera frame are:

$$p^c = R_n^c(p^n - p_{nc}^n)$$

Assume the camera is in motion, with its velocity and angular velocity in the n frame denoted as v_{nc}^n and ω_{nc}^n, respectively. According to Equation I.2.50, we have:

$$\dot{p}_{nc}^n = v_{nc}^n$$
$$\dot{R}_n^c = -R_n^c(\omega_{nc}^n)_\times$$

Now, considering the derivative of p^c, we have:

$$\begin{aligned}\dot{p}^c &= \dot{R}_n^c(p^n - p_{nc}^n) - R_n^c\dot{p}_{nc}^n \\ &= -R_n^c(\omega_{nc}^n)_\times(p^n - p_{nc}^n) - R_n^c v_{nc}^n \\ &= -(R_n^c\omega_{nc}^n) \times (R_n^c(p^n - p_{nc}^n)) - R_n^c v_{nc}^n \\ &= -\omega_{nc}^c \times p^c - v_{nc}^c\end{aligned}$$

Let:

$$v = v_{nc}^c = [v_x, v_y, v_z]^T$$
$$\omega = \omega_{nc}^c = [w_x, w_y, w_z]^T$$

We obtain the differential equation for the feature point in the camera frame:

$$\dot{p}^c = -\omega \times p^c - v \tag{VII.26.9}$$

Assume:

$$p^c = \begin{bmatrix} x^c \\ y^c \\ z^c \end{bmatrix}$$

Then, Equation VII.26.9 can be written as:

$$\begin{bmatrix} \dot{x}^c \\ \dot{y}^c \\ \dot{z}^c \end{bmatrix} = - \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix} \begin{bmatrix} x^c \\ y^c \\ z^c \end{bmatrix} - \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

That is:

$$\begin{aligned} \dot{x}^c &= w_z y^c - w_y z^c - v_x \\ \dot{y}^c &= -w_z x^c + w_x z^c - v_y \\ \dot{z}^c &= w_y x^c - w_x y^c - v_z \end{aligned} \tag{VII.26.10}$$

The normalized coordinates corresponding to p^c are $[x^c/z^c, y^c/z^c]$, and the normalized optical flow is the change in normalized coordinates over dt, i.e.,

$$\begin{aligned} o_x &= dt(\frac{\dot{x}^c}{z^c}) \\ o_y &= dt(\frac{\dot{y}^c}{z^c}) \end{aligned}$$

That is:

$$\begin{aligned} o_x &= dt\frac{z^c\dot{x}^c - x^c\dot{z}^c}{(z^c)^2} \\ o_y &= dt\frac{z^c\dot{y}^c - y^c\dot{z}^c}{(z^c)^2} \end{aligned}$$

Substituting Equation VII.26.10, we have:

$$\begin{aligned} \frac{o_x}{dt} &= \frac{1}{(z^c)^2}(z^c(w_z y^c - w_y z^c - v_x) - x^c(w_y x^c - w_x y^c - v_z)) \\ \frac{o_y}{dt} &= \frac{1}{(z^c)^2}(z^c(-w_z x^c + w_x z^c - v_z) - y^c(w_y x^c - w_x y^c - v_z)) \end{aligned}$$

Next, we replace x^c, y^c, z^c with inverse depth ρ and normalized coordinates α, β, i.e.,

$$\begin{bmatrix} \alpha \\ \beta \\ \rho \end{bmatrix} = \frac{1}{z^c} \begin{bmatrix} x^c \\ y^c \end{bmatrix} \tag{VII.26.11}$$

Now we have:

$$\begin{aligned} \frac{o_x}{dt} &= -(v_x\rho - \alpha v_z - \alpha\beta w_x + (1+\alpha^2)w_y\rho - \beta w_z) \\ \frac{o_y}{dt} &= -(v_y\rho - \beta v_z\rho - (1+\beta^2)w_x + \alpha\beta w_y + \alpha w_z) \end{aligned} \tag{VII.26.12}$$

In this system of equations, $v_x, v_y, v_z, w_x, w_y, w_z$ are variables, ρ is an unknown, and all others are known. That is, the normalized coordinates and optical flow information of one point provide 2 equations and 1 unknown. For N points, there will be $2N$ equations and $6+N$ unknowns. Therefore, the system can be solved when $N \geq 6$.

We summarize this problem as follows:

Problem	2D-2D Point-Based 3D Velocity Estimation
Problem Description	Given pixel coordinates of matched points in two poses, estimate 3D velocity and angular velocity
Known	Pixel coordinates $v_1, v_2, ..., v_N$ Pixel optical flow $o_1, o_2, ..., o_N$ Time interval δt, camera intrinsic matrix K
To Find	Velocity v_{nc}^c and angular velocity ω_{nc}^c in the camera frame

6-Point Optimization Algorithm

Observing the system of equations VII.26.12, we find that this is a nonlinear system, but the nonlinear terms only exist in the multiplicative cross terms. Therefore, we consider using nonlinear optimization methods for numerical solution.

Specifically, given N points $p_1^c, ..., p_N^c$, for each point p_i^c, we have known quantities $o_{x,i}, o_{y,i}, \alpha_i, \beta_i$, and unknown quantity ρ_i. That is, the joint optimization variables are

$$\mathbf{x} = [v_x, v_y, v_z, w_x, w_y, w_z, \rho_1, ..., \rho_N] \tag{VII.26.13}$$

We can construct the i-th set of residual terms

$$\mathbf{e}_i = \begin{bmatrix} -(v_x\rho - \alpha v_z\rho - \alpha\beta w_x + (1+\alpha^2)w_y - \beta w_z) - \frac{o_x}{dt} \\ -(v_y\rho - \beta v_z\rho - (1+\beta^2)w_x + \alpha\beta w_y + \alpha w_z) - \frac{o_y}{dt} \end{bmatrix} \tag{VII.26.14}$$

At this point, the derivative of $\mathbf{e}_i$ with respect to $\mathbf{x}$, $J_i(\mathbf{x})$, is

$$J_i(\mathbf{x}) = \begin{bmatrix} -\rho_i & 0 & \alpha\rho & \alpha\beta & -(1+\alpha^2) & \beta & ... & -v_x + \alpha v_z & ... \\ 0 & -\rho & \beta\rho & 1+\beta^2 & -\alpha\beta & -\alpha & ... & -v_y + \beta v_z & ... \end{bmatrix} \tag{VII.26.15}$$

Thus, we can use the nonlinear optimization methods introduced in Section 3.6, such as the Gauss-Newton method, to jointly optimize and solve this problem. We summarize the algorithm as follows:

Algorithm	6-Point Method for Solving 3D Velocity
Problem Type	2D-2D Point Correspondences for Solving 3D Velocity
Known	Pixel coordinates of points $v_1, v_2, ..., v_N$ Optical flow in pixel coordinates $o_1, o_2, ..., o_N$ Time interval δt, camera intrinsics K
To Find	Velocity in camera frame v_{nc}^c, angular velocity ω_{nc}^c
Algorithm Property	Approximate solution

Algorithm 89: 6-Point Method for Solving 3D Velocity

Input: Pixel coordinates of points $v_1, v_2, ..., v_N$
Input: Optical flow in pixel coordinates $o_1, o_2, ..., o_N$
Input: Time interval δt, camera intrinsics K
Output: Velocity $v_{nc}^c = [v_x, v_y, v_z]^T$, angular velocity $\omega_{nc}^c = [w_x, w_y, w_z]^T$

$\mathbf{x}_k \leftarrow [0_{1\times 6}, 1_{1\times N}]^T$
while *True* **do**
  $[v_x, v_y, v_z, w_x, w_y, w_z] \leftarrow (\mathbf{x}_k)_{0:6}$
  $\mathbf{e} \leftarrow 0_{2N\times 1}$
  $J \leftarrow 0_{2N\times(N+6)}$
  for $i \in 1, \ldots, N$ **do**
    $[\alpha_i, \beta_i, 1]^T \leftarrow K^{-1} v_i$
    $[o_{x,i}, o_{y,i}] \leftarrow o_i$
    $\mathbf{e}[2i] \leftarrow -(v_x\rho_i - \alpha_i v_z \rho_i - \alpha_i\beta_i w_x + (1+\alpha_i^2)w_y - \beta_i w_z) - \frac{o_{x,i}}{\delta t}$
    $\mathbf{e}[2i+1] \leftarrow -(v_y\rho_i - \beta_i v_z \rho_i - (1+\beta_i^2)w_x + \alpha_i\beta_i w_y + \alpha_i w_z) - \frac{o_{y,i}}{\delta t}$
    $J[2i:2i+1, 0:6] \leftarrow \begin{bmatrix} -\rho_i & 0 & \alpha_i\rho_i & \alpha_i\beta_i & -(1+\alpha_i^2) & \beta_i \\ 0 & -\rho_i & \beta_i\rho_i & 1+\beta_i^2 & -\alpha_i\beta_i & -\alpha_i \end{bmatrix}$
    $J[2i:2i+1, 6+i] \leftarrow \begin{bmatrix} -v_x + \alpha_i v_z \\ -v_y + \beta_i v_z \end{bmatrix}$
  $H_n \leftarrow J^T J$
  $g_n \leftarrow J^T \mathbf{e}$
  if $||g_n|| < \epsilon$ **then**
    break
  $\mathbf{x}_k \leftarrow \mathbf{x}_k - H_n^{-1} g_n$
$[v_x, v_y, v_z, w_x, w_y, w_z] \leftarrow (\mathbf{x}_k)0:6$

The corresponding Python code is shown below (using the pyceres library)

```python
class OpticalFlowResidual(pyceres.CostFunction):
    def Evaluate(self, parameters, residuals, jacobians):
        rho = parameters[0][0]  # inverse depth
        vx, vy, vz = parameters[1]
        wx, wy, wz = parameters[2]
        x, y = self.x, self.y
        residuals[0] = (- self.dx_dt + rho * (-vx + x * vz)
            + x * y * wx - (1 + x*x) * wy + y * wz)
        residuals[1] = (- self.dy_dt + rho * (-vy + y * vz)
            - x * y * wy + (1 + y*y) * wx - x * wz )
        J_rho = np.array([-vx + x * vz, -vy + y * vz])
        J_v = np.array([-rho, 0, rho*x, 0, -rho, rho*y])
        J_w = np.array([x*y, -(1+x*x), y, 1 + y*y, -x * y, -x])
        if jacobians:
            if jacobians[0] is not None:
                jacobians[0][:] = J_rho # Jacobian w.r.t. rho
            if jacobians[1] is not None:
                jacobians[1][:] = J_v  # Jacobian w.r.t. v
            if jacobians[2] is not None:
                jacobians[2][:] = J_w  # Jacobian w.r.t. w
        return True
def calc_vw_from_optflow(obs, optflow, dt):
    problem = pyceres.Problem()
    rhos = {}
    for pid in obs:
        rhos[pid] = np.array([0.3]) # initial inverse depth
        problem.add_parameter_block(rhos[pid], 1)
    v_param = np.array([0.0, 0.0, 0.0])  # initial velocity set to 0
    w_param = np.array([0.0, 0.0, 0.0])  # initial angular velocity
    ↪  set to 0
    problem.add_parameter_block(v_param, 3)
    problem.add_parameter_block(w_param, 3)
    for pid in obs:
        if pid not in optflow:
            continue
        (x, y), (dx, dy) = obs[pid], optflow[pid]
        dx_dt, dy_dt = dx / dt, dy / dt
        residual_block = OpticalFlowResidual(x, y, dx_dt, dy_dt)
        problem.add_residual_block(
            residual_block, None, [rhos[pid], v_param, w_param]
        )
    options = pyceres.SolverOptions()
    options.linear_solver_type = pyceres.LinearSolverType.DENSE_QR
    options.minimizer_progress_to_stdout = False
    options.max_num_iterations = 50
    options.function_tolerance = 1e-6
    pyceres.solve(options, problem, {})
    v_result, w_result = v_param.copy(), w_param.copy()
    return v_result, w_result
```

26.3 3D-3D Problem

26.3.1 ICP Algorithm

For the 2D-2D problem, we have introduced the solution based on epipolar geometry. Below, we present the method for matching 3D-3D point pairs.

A set of 3D points is also referred to as a point cloud. If the point cloud information under two poses is known, but the correspondence between the points is unknown, the task of matching the point clouds

and computing the homogeneous matrix between the two poses is called the **point cloud registration problem**, which is a fundamental issue in Lidar-based SLAM. The methods to solve such problems are collectively known as Iterative Closest Points (ICP).

The term "iterative" in the ICP algorithm refers to the iterative optimization of both the correspondence relationship and the pose when the matching is unknown. Since the 3D-3D problem we consider assumes that the point cloud matching is already completed, we only need to use the pose-solving step from ICP. Nevertheless, we still refer to this method as ICP.

Specifically, we need to solve for R_2^1 and t_{12}^1. For simplicity, we abbreviate $R = R_2^1, t = t_{12}^1$ in the following derivation. We aim to minimize the following matching error:

$$J(R,t) = \frac{1}{2}\sum_{i=1}^{N} ||Rp_i^2 + t - p_i^1||^2 \tag{VII.26.16}$$

Assuming the mean coordinates of the two point clouds are $\bar{p}^1$ and $\bar{p}^2$, respectively, we have:

$$\bar{p}^1 = \frac{1}{N}\sum_{i=1}^{N} p_i^1$$
$$\bar{p}^2 = \frac{1}{N}\sum_{i=1}^{N} p_i^2$$

For the first point cloud, let the offset of each point relative to the center be $\tilde{p}_i^1 = p_i^1 - \bar{p}^1$, and similarly for the second point cloud. Then:

$$\begin{aligned}
J(R,t) &= \frac{1}{2}\sum_{i=1}^{N} ||R(\tilde{p}_i^2 + \bar{p}^2) + t - (\tilde{p}_i^1 + \bar{p}^1)||^2 \\
&= \frac{1}{2}\sum_{i=1}^{N} ||(R\bar{p}^2 + t - \bar{p}^1) + (R\tilde{p}_i^2 - \tilde{p}_i^1)||^2 \\
&= \frac{N}{2}||R\bar{p}^2 + t - \bar{p}^1||^2 + \frac{1}{2}\sum_{i=1}^{N}(R\bar{p}^2 + t - \bar{p}^1)^T(R\tilde{p}_i^2 - \tilde{p}_i^1) + \frac{1}{2}\sum_{i=1}^{N}||R\tilde{p}_i^2 - \tilde{p}_i^1||^2 \\
&= \frac{N}{2}||R\bar{p}^2 + t - \bar{p}^1||^2 + \frac{1}{2}\sum_{i=1}^{N}||R\tilde{p}_i^2 - \tilde{p}_i^1||^2 + \frac{1}{2}(R\bar{p}^2 + t - \bar{p}^1)^T\sum_{i=1}^{N}(R\tilde{p}_i^2 - \tilde{p}_i^1) \\
&= \frac{N}{2}||R\bar{p}^2 + t - \bar{p}^1||^2 + \frac{1}{2}\sum_{i=1}^{N}||R\tilde{p}_i^2 - \tilde{p}_i^1||^2
\end{aligned}$$

It can be seen that the matching error is divided into two parts. Since the error J is quadratic in both R and t, there exists a global optimal solution, and the global optimal solution R^*, t^* satisfies the first-order condition:

$$\frac{\partial J}{\partial t} = (R\bar{p}^2 + t - \bar{p}^1)^T = 0 \tag{VII.26.17}$$

Thus, we have:

$$t^* = \bar{p}^1 - R^*\bar{p}^2 \tag{VII.26.18}$$

Substituting the optimal translation and rotation matrix relationship into the error expression VII.26.16, we obtain:

$$
\begin{aligned}
R^* &= \arg\min_R \frac{1}{2}\sum_{i=1}^{N} ||R\tilde{p}_i^2 - \tilde{p}_i^1||^2 \\
&= \arg\min_R \frac{1}{2}\sum_{i=1}^{N} (R\tilde{p}_i^2 - \tilde{p}_i^1)^T (R\tilde{p}_i^2 - \tilde{p}_i^1) \\
&= \arg\min_R \frac{1}{2}\sum_{i=1}^{N} (\tilde{p}_i^1)^T \tilde{p}_i^1 + (\tilde{p}_i^2)^T (R)^T R\tilde{p}_i^2 - 2(\tilde{p}_i^1)^T R\tilde{p}_i^2 \\
&= \arg\min_R \frac{1}{2}\sum_{i=1}^{N} (\tilde{p}_i^1)^T \tilde{p}_i^1 + (\tilde{p}_i^2)^T \tilde{p}_i^2 - \sum_{i=1}^{N} (\tilde{p}_i^1)^T R\tilde{p}_i^2 \\
&= \arg\min_R -\sum_{i=1}^{N} (\tilde{p}_i^1)^T R\tilde{p}_i^2
\end{aligned}
$$

For a square matrix A, we have:

$$
b^T Ac = \mathrm{tr}(Acb^T)
$$

Therefore:

$$
R^* = \arg\max_R \sum_{i=1}^{N} \mathrm{tr}(R\tilde{p}_i^2 (\tilde{p}_i^1)^T) = \arg\max_R \mathrm{tr}(R\sum_{i=1}^{N} \tilde{p}_i^2 (\tilde{p}_i^1)^T)
$$

Let:

$$
W = \sum_{i=1}^{N} \tilde{p}_i^2 (\tilde{p}_i^1)^T
$$

Then:

$$
R^* = \arg\max_R \mathrm{tr}(RW)
$$

Assuming the SVD decomposition of W is:

$$
W = U\Sigma V^T \tag{VII.26.19}
$$

Then:

$$
R^* = \arg\max_R \mathrm{tr}(RU\Sigma V^T)
$$

Since W is a 3×3 matrix, V^T and U are both third-order orthogonal matrices satisfying $V^T V = U^T U = I_3$. Let:

$$
B = RUV^T
$$

Then:

$$
R = BVU^T
$$

Substituting into the equation, we have:

$$
\begin{aligned}
B^* &= \arg\max_B \mathrm{tr}(BVU^T U\Sigma V^T) \\
&= \arg\max_B \mathrm{tr}(BV\Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}} V^T) \\
&= \arg\max_B \mathrm{tr}(BV\Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}} V^T)
\end{aligned}
$$

We introduce a lemma to be proven later:

$$\operatorname{tr}(AA^T) \geq \operatorname{tr}(BAA^T), \quad \forall A \in \mathbb{R}^{n\times n}, B^T B = I_n$$

Therefore,

$$\operatorname{tr}(BV\Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}}V^T) \leq \operatorname{tr}(V\Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}}V^T), \forall B$$

That is,

$$B^* = I$$

Thus,

$$R^* = B^* V U^T = V U^T \tag{VII.26.20}$$

In summary, the ICP solution algorithm is as follows:

Algorithm	**ICP Algorithm**
Problem Type	3D-3D Pose Estimation
Given	Camera coordinates of points under pose 1: $p_1^1, p_2^1, ..., p_N^1$ Camera coordinates of points under pose 2: $p_1^2, p_2^2, ..., p_N^2$
Objective	Homogeneous matrix between poses T_2^1
Algorithm Property	Approximate solution

Algorithm 90: ICP Algorithm (ICP_calc)

Input: Camera coordinates of points at pose 1: $p_1^1, p_2^1, ..., p_N^1$

Input: Camera coordinates of points at pose 2: $p_1^2, p_2^2, ..., p_N^2$

Output: Homogeneous transformation matrix between poses T_2^1

$\bar{p}^1 \leftarrow \frac{1}{N}\sum_{i=1}^{N} p_i^1$

$\bar{p}^2 \leftarrow \frac{1}{N}\sum_{i=1}^{N} p_i^2$

$W \leftarrow \sum_{i=1}^{N}(p_i^2 - \bar{p}^2)(p_i^1 - \bar{p}^1)^T$

$U, \Sigma, V^T \leftarrow \text{SVD\_calc}(W)$

$R_2^1 \leftarrow VU^T$

$t_{12}^1 \leftarrow \bar{p}^1 - R_2^1\bar{p}^2$

$T_2^1 \leftarrow \begin{bmatrix} R_2^1 & t_{12}^1 \\ 0_{1\times 3} & 1 \end{bmatrix}$

The corresponding Python code is shown below:

```python
def vo_ICP(N, p1s, p2s):
    assert p1s.shape == (N, 3) and p2s.shape == (N, 3)
    p1_mean, p2_mean = np.mean(p1s, axis=0), np.mean(p2s,
    ↪  axis=0)
    W = (p2s - p2_mean).T @ (p1s - p1_mean)
    U, sigma, VT = np.linalg.svd(W)
    R_2_to_1 = VT.T @ U.T
    if np.linalg.det(R_2_to_1) < 0:
        VT[2, :] = - VT[2, :]
        R_2_to_1 = - VT.T @ U.T
    t_12 = p1_mean - R_2_to_1 @ p2_mean
    T_2_to_1 = np.row_stack([
        np.column_stack([R_2_to_1, t_12]),
        np.array([0, 0, 0, 1])
    ])
    return T_2_to_1
```

26.3.2 Lemma Proof

Below we briefly prove the lemma from the previous subsection:

$$\mathrm{tr}(AA^T) \geq \mathrm{tr}(BAA^T), \quad \forall A \in \mathbb{R}^{n\times n}, B^TB = I_n$$

Assuming matrix A can be written as:

$$\begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix}$$

Then:

$$\begin{aligned}
\mathrm{tr}(BAA^T) &= \mathrm{tr}(A^T(BA)) \\
&= \mathrm{tr}\left(\begin{bmatrix} a_1^T \\ a_2^T \\ \dots \\ a_n^T \end{bmatrix} B \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix}\right) \\
&= \sum_{i=1}^{n} a_i^T B a_i = \sum_{i=1}^{n} a_i^T (Ba_i) \\
&\leq \sum_{i=1}^{n} \sqrt{(a_i^T a_i)(a_i^T B^T B a_i)} \\
&= \sum_{i=1}^{n} \sqrt{(a_i^T a_i)(a_i^T a_i)} = \sum_{i=1}^{n} (a_i^T a_i) \\
&= \mathrm{tr}\left(\begin{bmatrix} a_1^T \\ a_2^T \\ \dots \\ a_n^T \end{bmatrix} \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix}\right) \\
&= \mathrm{tr}(A^TA)
\end{aligned}$$

The proof utilizes the Cauchy-Schwartz inequality, which will not be elaborated here.

26.4 3D-2D Problem

26.4.1 6-Point DLT Solution

For the 3D-2D problem, considering the coordinates of spatial point P in the camera frames of pose 1 and pose 2 as p^1 and p^2 respectively, we clearly have:

$$p^2 = R_1^2 p^1 + t_{21}^2$$

Considering the imaging process at pose 2, we have:

$$v^2 = \frac{1}{z^2} K p^2$$

Combining these, we obtain the relationship between pixel coordinates at pose 2 and camera coordinates at pose 1:

$$\begin{bmatrix} z^2 u^2 \\ z^2 v^2 \\ z^2 \end{bmatrix} = K[R_1^2 \quad t_{21}^2] p_{ext}^1$$

The focal plane depth z^2 in this equation is unknown. We eliminate it using linear transformation methods:

$$\begin{bmatrix} 1 & 0 & -u^2 \\ 0 & 1 & -v^2 \end{bmatrix} K[R_1^2 \quad t_{21}^2] p_{ext}^1 = 0 \tag{VII.26.21}$$

Let:

$$\begin{bmatrix} 1 & 0 & -u^2 \\ 0 & 1 & -v^2 \end{bmatrix} K = A = [a_1 \quad a_2 \quad a_3] \in \mathbb{R}^{2\times 3}$$

And:

$$\begin{bmatrix} t_1^T \\ t_2^T \\ t_3^T \end{bmatrix} = [R_1^2 \quad t_{21}^2] \in \mathbb{R}^{3\times 4}$$

Then the equation:

$$[a_1 \quad a_2 \quad a_3] \begin{bmatrix} t_1^T \\ t_2^T \\ t_3^T \end{bmatrix} p_{ext}^1 = 0$$

Can be rewritten as:

$$[a_1 (p_{ext}^1)^T \quad a_2 (p_{ext}^1)^T \quad a_3 (p_{ext}^1)^T] \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} = 0$$

Let:

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \in \mathbb{R}^{12}$$

We can see that the unknown $\mathbf{t}$ contains 12 variables, and each known point P can provide 2 equations. Therefore, when the total number of points $N \geq 6$, we can solve it linearly.

Since this method directly solves for the elements of the homogeneous matrix through linear equations, it is also called the **Direct Linear Transform** (DLT) method. We summarize this algorithm as follows:

Algorithm	**DLT Algorithm**
Problem Type	3D-2D Point Pose Estimation
Known	Camera coordinates of points at pose 1: $p_1^1, p_2^1, ..., p_N^1$ Pixel coordinates of points at pose 2: $v_1^2, v_2^2, ..., v_N^2$ Camera intrinsic matrix K
To Find	Homogeneous transformation matrix between poses T_2^1
Algorithm Property	Approximate solution

Algorithm 91: DLT Algorithm (DLT_calc)

Input: Camera coordinates of points in Pose 1: $p_1^1, p_2^1, ..., p_N^1$

Input: Pixel coordinates of points in Pose 2: $v_1^2, v_2^2, ..., v_N^2$

Output: Homogeneous transformation matrix between poses T_2^1

$M \leftarrow 0_{2N \times 12}$

for $i \in 1, ..., N$ **do**

$$[a_1 \quad a_2 \quad a_3] \leftarrow \begin{bmatrix} 1 & 0 & -u^2 \\ 0 & 1 & -v^2 \end{bmatrix} K$$

$$M_{2i:2i+1,:} \leftarrow \begin{bmatrix} a_1 (p_{ext}^1)^T & a_2 (p_{ext}^1)^T & a_3 (p_{ext}^1)^T \end{bmatrix}$$

$$\begin{bmatrix} t_1^T & t_2^T & t_3^T \end{bmatrix}^T \leftarrow \text{linear\_solve}(M^T M \mathbf{t} = 0)$$

$$[R_1^2 \quad t_{21}^2] \leftarrow \begin{bmatrix} t_1^T \\ t_2^T \\ t_3^T \end{bmatrix}$$

$R_1^2 \leftarrow R_1^2 / |(R_1^2)|^{-\frac{1}{3}}$

$U, \Sigma, V^T \leftarrow \text{SVD\_calc}(R_1^2)$

$R_1^2 \leftarrow V^T U$

$R_2^1, t_{12}^1 \leftarrow (R_1^2)^T, -R_1^2 t_{21}^2$

The corresponding Python code is shown below:

```
def vo_DLT(N, p1s, nu_2s, K):
    assert nu_2s.shape == (N, 3) and p1s.shape == (N, 3)
    assert K.shape == (3, 3)
    M = np.zeros([2*N, 12])
    p1s_ext = np.column_stack([p1s, np.ones(N)])
    for i in range(N):
        A = np.array([[1, 0, -nu_2s[i, 0]], [0, 1, -nu_2s[i, 1]]]) @ K
        M[2*i:2*i+2, :4] = A[:, 0:1] @ p1s_ext[i:i+1, :]
        M[2*i:2*i+2, 4:8] = A[:, 1:2] @ p1s_ext[i:i+1, :]
        M[2*i:2*i+2, 8:] = A[:, 2:3] @ p1s_ext[i:i+1, :]
    t_long = scipy.linalg.null_space(M.T @ M)[:, 0]
    Rt_1_to_2 = np.column_stack([
        t_long[:4], t_long[4:8], t_long[8:]
    ]).T
    R_1_to_2, t_21 = Rt_1_to_2[:, :3], Rt_1_to_2[:, 3]
    Rdet = np.linalg.det(R_1_to_2)
    if Rdet < 0:
        R_1_to_2, t_21 = - R_1_to_2, - t_21
    a = np.power(np.abs(Rdet), -1/3)
    R_1_to_2, t_21 = R_1_to_2 * a, t_21 * a
    T_2_to_1 = np.row_stack([
        np.column_stack([R_1_to_2.T, - R_1_to_2.T @ t_21]),
        np.array([0, 0, 0, 1])
    ])
    return T_2_to_1
```

26.4.2 EPnP Pose Estimation

Another method for solving the 3D-2D problem is the EPnP (Effective Perspective-n-Points) algorithm. The core idea of this algorithm is to transform the 3D-2D problem into a 3D-3D problem and then solve it using the previously introduced ICP method.

Specifically, let's revisit the coordinate representation of points in 3D space: Assume the origin of a coordinate system n is O, and its three orthogonal unit basis vectors are $\overrightarrow{OA}, \overrightarrow{OB}, \overrightarrow{OC}$. For any point P in space, suppose its coordinates in this coordinate system are $[a_1, a_2, a_3]$. Then, the spatial vector relationship is given by:

$$\overrightarrow{OP} = a_1\overrightarrow{OA} + a_2\overrightarrow{OB} + a_3\overrightarrow{OC}$$

For any coordinate system b (which may not be system n), the following coordinate relationship holds:

$$p^b - o^b = a_1(a^b - o^b) + a_2(b^b - o^b) + a_3(c^b - o^b)$$

Rearranging the above equation, we obtain the coordinate expression of any point p in any coordinate system b:

$$p^b = a_1 a^b + a_2 b^b + a_3 c^b + (1 - a_1 - a_2 - a_3)o^b$$

Let $a_4 = 1 - a_1 - a_2 - a_3$, and denote $A = F_1, B = F_2, C = F_3, O = F_4$. Then, we have:

$$p^b = \sum_{i=1}^{4} a_i f_i^b, \quad \sum_{i=1}^{4} a_i = 1 \tag{VII.26.22}$$

Note that in this expression, the selection of point P and coordinate system b is entirely independent of the choice of the four points $F_1 \sim F_4$, and the values of a_i are also unaffected by the choice of b.

In fact, $F_1 \sim F_4$ do not necessarily have to be four points forming an orthonormal basis. **As long as $F_1 \sim F_4$ are four non-coplanar points, the coordinates of any spatial point P in any coordinate system can be uniquely expressed as a linear combination of the coordinates of these four points.** These four points are called **reference points**.

Now, given a point P, if the coordinates of P and the **reference points** in a certain coordinate system are known, how can we determine the coefficients $a_1 \sim a_4$? In fact, we only need to set up the equation:

$$F_r\mathbf{a} = \mathbf{b}$$

where

$$F_r = \begin{bmatrix} f_1 & f_2 & f_3 & f_4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \in \mathbb{R}^{4\times 4}$$

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \in \mathbb{R}^4, \mathbf{b} = \begin{bmatrix} p \\ 1 \end{bmatrix} \in \mathbb{R}^4$$

The solution is then given by:

$$\mathbf{a} = F_r^{-1} b$$

Now, for a set of points with known 3D coordinates (e.g., the coordinates of points in the camera frame of pose 1), how should we select the control points? We can use a concept similar to that in PCA to avoid matrix singularity.

Specifically, for the first control point, we take the mean:

$$f_1^1 = \bar{p}^1 = \frac{1}{N}\sum_{i=1}^{N} p_i^1 \tag{VII.26.23}$$

Define the zero-centered coordinate matrix:

$$\bar{P} = \begin{bmatrix} (p_1^1 - f_1^1)^T \\ (p_2^1 - f_1^1)^T \\ \dots \\ (p_N^1 - f_1^1)^T \end{bmatrix} \tag{VII.26.24}$$

Compute the eigenvectors of the dimensional covariance matrix:

$$\bar{P}^T \bar{P} v_1 = \lambda_1 v_1$$
$$\bar{P}^T \bar{P} v_2 = \lambda_2 v_2$$
$$\bar{P}^T \bar{P} v_3 = \lambda_3 v_3$$

Thus, we can select three control points:

$$f_2^1 = f_1^1 + \sqrt{\frac{\lambda_1}{N}} v_1$$
$$f_3^1 = f_1^1 + \sqrt{\frac{\lambda_2}{N}} v_2$$
$$f_4^1 = f_1^1 + \sqrt{\frac{\lambda_3}{N}} v_3$$

Now, consider expressing the coordinates in the camera frame of pose 2 for the 3D-2D problem. For each matched point P, since:

$$p^2 = \sum_{i=1}^{4} a_i f_i^2$$

By the camera model, we have:

$$v^2 = \frac{1}{z^2} K \sum_{i=1}^{4} a_i f_i^2$$

Here, the coordinates of the control points in the camera frame of pose 2 are unknowns, totaling 12. Let:

$$\mathbf{f} = \begin{bmatrix} f_1^2 \\ f_2^2 \\ f_3^2 \\ f_4^2 \end{bmatrix} \tag{VII.26.25}$$

Thus, we have:

$$\begin{bmatrix} z^2 u^2 \\ z^2 v^2 \\ z^2 \end{bmatrix} = K [a_1 I \quad a_2 I \quad a_3 I \quad a_4 I] \mathbf{f}$$

Here, z^2 is unknown, while u^2, v^2 are known. Substituting the third row into the first two rows for elimination yields an equation for $\mathbf{f}$:

$$\begin{bmatrix} 1 & 0 & -u^2 \\ 1 & -v^2 & 0 \end{bmatrix} K [a_1 I \quad a_2 I \quad a_3 I \quad a_4 I] \mathbf{f} = 0$$

For $N \geq 6$ point pairs, solving this system of equations yields f_i^2, allowing us to compute the 3D coordinates of each point in pose 2 using the coefficients $\mathbf{a}$. Finally, the relative pose T_2^1 can be solved using ICP.

Based on the above derivation, we can summarize the EPnP algorithm as follows.

Algorithm 92: EPnP Algorithm

Input: Camera coordinates of points in pose 1 $p_1^1, p_2^1, ..., p_N^1$

Input: Pixel coordinates of points in pose 2 $v_1^2, v_2^2, ..., v_N^2$

Output: Homogeneous transformation matrix between poses T_2^1

// Compute control points

$f_1^1 \leftarrow \frac{1}{N}\sum_{i=1}^{N} p_i^1$

$\bar{P} \leftarrow \begin{bmatrix} (p_1^1 - f_1^1)^T \\ (p_2^1 - f_1^1)^T \\ ... \\ (p_N^1 - f_1^1)^T \end{bmatrix}$

$v_{1:3}, \lambda_{1:3} \leftarrow \text{calc\_feature\_vector}(\bar{P}^T \bar{P})$

for $i \in 1, 2, 3$ **do**

  $f_{i+1}^1 \leftarrow f_1^1 + \sqrt{\frac{\lambda_i}{N}} v_i$

// Compute linear coefficients

$F_r \leftarrow \begin{bmatrix} f_1^1 & f_2^1 & f_3^1 & f_4^1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$

$A \leftarrow 0_{4 \times N}$

for $i \in 1, ..., N$ **do**

  $A_{i,:} \leftarrow F_r^{-1}[(p_i^1)^T 1]^T$

// Solve for control point coordinates

$M \leftarrow 0_{2N \times 12}$

for $i \in 1, ..., N$ **do**

  $V_i \leftarrow \begin{bmatrix} 1 & 0 & -u_i^2 \\ 1 & -v_i^2 & 0 \end{bmatrix}$

  $M_{2i:2i+1,:} \leftarrow V_i K [a_{1,i} I \quad a_{2,i} I \quad a_{3,i} I \quad a_{4,i} I]$

$\mathbf{f} \leftarrow \text{linear\_solve}(M^T M \mathbf{f} = 0)$

// Compute sample 3D coordinates

for $i \in 1, ..., N$ **do**

  $p_i^2 \leftarrow [a_{1,i} I \quad a_{2,i} I \quad a_{3,i} I \quad a_{4,i} I]\mathbf{f}$

// Solve relative pose using ICP

$T_2^1 \leftarrow \text{ICP\_calc}(p_:^1, p_:^2)$

Algorithm	**EPnP Algorithm**
Problem Type	3D-2D point pose estimation
Known	Camera coordinates of points in pose 1 $p_1^1, p_2^1, ..., p_N^1$ Pixel coordinates of points in pose 2 $v_1^2, v_2^2, ..., v_N^2$ Camera intrinsic matrix K
To Find	Homogeneous transformation matrix T_2^1
Algorithm Property	Approximate solution

The corresponding Python code is shown below:

```python
def get_inside_receiving_radius(points):
    assert points.shape == (4, 3)
    def triangle_area(a, b, c):
        # 0.5 * || (b - a) × (c - a) ||
        return 0.5 * np.linalg.norm(np.cross(b - a, c - a))
    def tetrahedron_volume(a, b, c, d):
        # 1/6 * | (b - a)^T ((c - a) × (d - a)) |
        return abs(np.dot(b - a, np.cross(c - a, d - a))) / 6.0
```

```
    A, B, C, D = points
    area_ABC = triangle_area(A, B, C)
    area_ABD = triangle_area(A, B, D)
    area_ACD = triangle_area(A, C, D)
    area_BCD = triangle_area(B, C, D)
    total_area = area_ABC + area_ABD + area_ACD + area_BCD
    volume = tetrahedron_volume(A, B, C, D)
    return (3 * volume) / total_area
def vo_EPnP(N, p1s, nu_2s, K):
    assert p1s.shape == (N, 3) and nu_2s.shape == (N, 3)
    assert K.shape == (3, 3)
    # calculate 2D control points
    f1_s = np.zeros([4, 3])
    f1_s[0] = np.mean(p1s, axis=0)
    P_bar = p1s - f1_s[0][None, :]
    lambdas, vs = np.linalg.eig(P_bar.T @ P_bar)
    for i in (1, 2, 3):
        f1_s[i] = f1_s[0] + vs[i-1] * np.sqrt(lambdas[i-1] / N)
    # calculate projection coefficents
    F_r = np.row_stack([f1_s.T, np.array([1,1,1,1])])
    P1_ext = np.column_stack([p1s, np.ones([N])])
    A = np.linalg.inv(F_r) @ P1_ext.T
    # calculate 3D control points
    M, I = np.zeros([2*N, 12]), np.eye(3)
    for i in range(N):
        Vi = np.array([ [1, 0, -nu_2s[i, 0]], [0, 1, -nu_2s[i, 1]]])
        M[2*i:2*i+2, :] = Vi @ K @ np.column_stack([
            A[0,i]*I, A[1,i]*I, A[2,i]*I, A[3,i]*I
        ])
    f = scipy.linalg.null_space(M.T @ M)[:, 0] # <- scale-free!
    # scale correction
    f2_s = np.row_stack([f[:3], f[3:6], f[6:9], f[9:]])
    r1 = get_inside_receiving_radius(f1_s)
    r2 = get_inside_receiving_radius(f2_s)
    f2_s = f2_s / r2 * r1
    # restore 3D points & output
    p2s = A.T @ f2_s # restore all 3D points
    return vo_ICP(N, p1s, p2s)
```

26.4.3 BA Pose Estimation

BA stands for Bundle Adjustment. Its core idea is to simultaneously model both pose errors from multiple viewpoints and observation point position errors as optimization variables, solving a nonlinear optimization problem to achieve simultaneous optimization of both. Since the light rays observing a feature point from different cameras converge into a bundle, and the error at the intersection point decreases after optimization, it is also called bundle adjustment.

Since we currently only focus on solving 3D-2D problems, our modeled variables only include the error of a single pose estimation with 2D observations, excluding more poses or observation point position errors. In fact, this method can be easily extended to multiple pose optimization scenarios.

Let us first construct an optimization problem. Specifically, still assuming $R = R_2^1, t = t_{12}^1$, we have:

$$\begin{aligned} p^2 &= R_1^2 p^1 + t_{21}^2 \\ &= (R_2^1)^T p^1 - (R_2^1)^T t_{12}^1 \\ &= R^T (p^1 - t) \end{aligned}$$

Then the pixel coordinates in pose 2 are:

$$\hat{v}^2(R,t) = \frac{1}{z^2} K R^T (p^1 - t)$$

It can be seen that the pixel coordinates $v^2(R,t)$ under pose 2 are functions of R and t. If we construct an optimization objective, we can formulate an optimization problem for R and t. Next, by performing iterative optimization, we can solve the 3D-2D problem. We choose the squared error between the observed values v^2 and the estimated values $\hat{v}^2(R,t)$ as the loss function, i.e.,

$$J(R,t) = \sum_{i=1}^{N} \frac{1}{2} ||v_i^2 - \hat{v}_i^2(R,t)||^2$$

In Section 2.5, we mentioned that directly using the rotation matrix R as the optimization variable makes it difficult to compute derivatives, whereas using the rotation vector ϕ as a substitute is more convenient. Therefore, the actual optimization problem we solve is

$$J(\phi,t) = \sum_{i=1}^{N} \frac{1}{2} ||e_i(\phi,t)||^2$$

where

$$e_i(\phi,t) = v_i^2 - \hat{v}_i^2(\exp(\phi_\times),t)$$

Summarizing the above computational process, we have

$$\begin{aligned} J(\phi,t) &= \sum_{i=1}^{N} \frac{1}{2} ||e_i(\phi,t)||^2 \\ e_i(\phi,t) &= I_{2\times 3}(v_i^2 - \hat{v}_i^2) \\ \hat{v}_i^2 &= K\nu_i^2 \\ \nu_i^2 &= \frac{1}{z_i^2} p_i^2 \\ p_i^2 &= R^T(p^1 - t) \\ R &= \exp(\phi_\times) \end{aligned} \tag{VII.26.26}$$

Thus, according to the chain rule, we have the derivatives of the optimization objective with respect to the optimization variables:

$$\frac{\partial J}{\partial \phi} = \sum_{i=1}^{N} \left(\frac{\partial J}{\partial e_i} \frac{\partial e_i}{\partial \hat{v}_i^2} \frac{\partial \hat{v}_i^2}{\partial \nu_i^2} \frac{\partial \nu_i^2}{\partial p_i^2} \frac{\partial p_i^2}{\partial \phi} \right)$$

$$\frac{\partial J}{\partial t} = \sum_{i=1}^{N} \left(\frac{\partial J}{\partial e_i} \frac{\partial e_i}{\partial \hat{v}_i^2} \frac{\partial \hat{v}_i^2}{\partial \nu_i^2} \frac{\partial \nu_i^2}{\partial p_i^2} \frac{\partial p_i^2}{\partial t} \right)$$

where

$$
\begin{aligned}
\frac{\partial J}{\partial e_i} &= e_i^T \\
\frac{\partial e_i}{\partial \hat{v}_i^2} &= \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} \\
\frac{\partial \hat{v}_i^2}{\partial \nu_i^2} &= K \\
\frac{\partial \nu_i^2}{\partial p_i^2} &= \frac{1}{z_i^2} \begin{bmatrix} 1 & 0 & -\frac{x_i^2}{z_i^2} \\ 0 & 1 & -\frac{y_i^2}{z_i^2} \\ 0 & 0 & 0 \end{bmatrix} \\
\frac{\partial p_i^2}{\partial \phi} &= (R^T(p_i^1 - t))_\times \\
\frac{\partial p_i^2}{\partial t} &= -R^T
\end{aligned}
$$

Here, we use Equation I.2.43 to compute the derivative of the Lie algebra.

Let

$$
J_\nu(p) = \frac{1}{z} \begin{bmatrix} 1 & 0 & -\frac{x}{z} \\ 0 & 1 & -\frac{y}{z} \\ 0 & 0 & 0 \end{bmatrix} \tag{VII.26.27}
$$

Thus, we have

$$
\begin{aligned}
\frac{\partial J}{\partial \phi} &= \sum_{i=1}^{N} \mathrm{diag}(-e_i) K J_\nu(p_i^2)(R^T(p_i^1 - t))_\times \\
\frac{\partial J}{\partial t} &= \sum_{i=1}^{N} \mathrm{diag}(-e_i) K J_\nu(p_i^2) R^T
\end{aligned} \tag{VII.26.28}
$$

At this point, we have completed the construction of the nonlinear least-squares optimization problem for ϕ and t. According to Section 3.6, we can choose methods such as gradient descent, Gauss-Newton, or L-M to solve it.

It should be noted that since we chose the right perturbation for the rotation vector, the update for R should be

$$
R_{k+1} \approx R_k(I + (\delta\phi)_\times) \tag{VII.26.29}
$$

Here, we take the Gauss-Newton method as an example and outline the complete solution algorithm. Since this method requires an initial value, we can use DLT to compute this initial value.

Algorithm 93: GN-BA Pose Estimation

Input: Camera coordinates of points under pose 1: $p_1^1, p_2^1, ..., p_N^1$

Input: Pixel coordinates of points under pose 2: $\upsilon_1^2, \upsilon_2^2, ..., \upsilon_N^2$

Parameter: Threshold ϵ

Output: Homogeneous transformation matrix between poses T_2^1

$R_0, t_0 \leftarrow \text{DLT\_calc}(p_:^1, \upsilon_:^2, K)$

$k \leftarrow 0$

while $||e_{k-1}|| > \epsilon$ **do**

  $J_k, e_k \leftarrow 0_{N\times 6}, 0_{N\times 1}$

  for $i = 1, ..., N$ **do**

    $\hat{p}_i^2, \nu^2, \hat{\upsilon}_i^2 \leftarrow R_k^T(p_i^1 - t_k), \frac{1}{z_i^2}K\hat{p}_i^2, K\nu^2$

    $(e_k)_{2i:2i+1} \leftarrow I_{2\times 3}(\upsilon_i^2 - \hat{\upsilon}_i^2)$

    $J_1 \leftarrow \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$

    $J_3 \leftarrow \frac{1}{z_i^2}\begin{bmatrix} 1 & 0 & -x_i^2/z_i^2 \\ 0 & 1 & -y_i^2/z_i^2 \\ 0 & 0 & 0 \end{bmatrix}$

    $J_4 \leftarrow [(R_k^T(p^1 - t_k))_\times \quad R_k^T]$

    $(J_k)_{2i:2i+1,:} \leftarrow J_1 K J_3 J_4$

  $H_n, g_n \leftarrow J_k^T J_k, J_k^T e_k$

  $\begin{bmatrix} \delta\phi \\ \delta t \end{bmatrix} \leftarrow -H_n^{-1} g_n$

  $t_{k+1}, R_{k+1} \leftarrow t_k + \delta t, R_k(I + (\delta\phi)_\times)$

  $k \leftarrow k + 1$

$R_2^1, t_{12}^1 \leftarrow R_k, t_k$

In the above BA algorithm (i.e., estimating only 1 pose), the number of points N must satisfy $2N \geq 6$, i.e., $N \geq 3$ and not all collinear.

Algorithm	**GN-BA Pose Estimation**
Problem Type	3D-2D Point Pose Estimation
Known	Camera coordinates of points in Pose 1: $p_1^1, p_2^1, ..., p_N^1$ Pixel coordinates of points in Pose 2: $\upsilon_1^2, \upsilon_2^2, ..., \upsilon_N^2$ Camera intrinsic matrix K
To Find	Homogeneous transformation matrix between poses T_2^1
Algorithm Property	Iterative solution

The corresponding Python code is shown below:

```python
def skew(t):
    return np.array([
        [0, -t[2], t[1]],
        [t[2], 0, -t[0]],
        [-t[1], t[0], 0]
    ])
def vo_BA_GN(N, p1s, nu_2s, K, alpha=1e-4, epsilon_1=5e-1, epsilon_2=1e-6):
    assert p1s.shape == (N, 3) and nu_2s.shape == (N, 3)
    assert K.shape == (3, 3)
    T_2_to_1 = vo_DLT(N, p1s, nu_2s, K)
    while True:
        R_2_to_1, t_12 = T_2_to_1[:3, :3], T_2_to_1[:3, 3]
        J_k, e_k = np.zeros([2*N, 6]), np.zeros([2*N])
```

```
        for i in range(N):
            p2 = R_2_to_1.T @ (p1s[i] - t_12)
            m2 = p2 / (p2[2] + epsilon_2)
            nu2_est = K @ m2
            e_k[2*i:2*i+2] = (nu_2s[i] - nu2_est)[:2]
            J1 = np.array([[-1,0,0], [0,-1,0]])
            J2 = K
            J3 = np.array([ [1, 0, -p2[0] / p2[2]],
                [0, 1, -p2[1] / p2[2]],
                [0, 0, 0],
            ]) / (p2[2] + epsilon_2)
            J4 = np.column_stack([skew(R_2_to_1.T @ (p1s[i] - t_12)), -R_2_to_1.T])
            J_k[2*i:2*i+2, :] = J1 @ J2 @ J3 @ J4
        delta_x = - alpha * np.linalg.inv(J_k.T @ J_k) @ J_k.T @ e_k
        if np.linalg.norm(e_k) < epsilon_1:
            break
        T_2_to_1[:3, :3] = R_2_to_1 @ (np.eye(3) + skew(delta_x[:3]))
        T_2_to_1[:3, 3] = t_12 + delta_x[3:]
    return T_2_to_1
```

In practical applications, the Gauss-Newton method converges relatively slowly, and the algorithm's convergence heavily depends on a good learning rate. If the learning rate is too small, convergence will be slow and prone to getting stuck in local minima; if the learning rate is too large, the algorithm may fail to converge. Therefore, we often use its improved version, the Levenberg-Marquardt (L-M) method, for BA pose estimation.

26.5 RANSAC Method

In this chapter, we have introduced several methods for recovering pose using feature point information, addressing different scenarios such as 2D-2D, 2D-3D, and 3D-3D problems. However, up to this point, we have assumed that the information used is accurate—that is, the observations of feature points are free from errors or outliers. In reality, actual data is never ideal and inevitably contains erroneous observations, often resulting from incorrect matching or tracking in the frontend. Among all errors, outliers tend to introduce significant estimation inaccuracies.

To address this issue, there exists a class of algorithms designed to preemptively remove outliers from the data, preventing the results from being distorted by wild values that could lead to large errors. As long as the data satisfies some form of linear constraint, these algorithms can identify outliers through random sampling. Such algorithms are collectively referred to as **RANSAC** (RANdom SAmple Consensus). Depending on the type of data and the specific linear constraints involved, VSLAM commonly employs the following variants of RANSAC: **4-point H-matrix RANSAC**, **8-point F-matrix RANSAC**, and others.

26.5.1 Principle of RANSAC

Before delving into specific RANSAC implementations, we first explain the fundamental principles of RANSAC using the example of linear fitting.

[This section will be updated in a future version. Stay tuned.]

26.5.2 4-point H-matrix RANSAC

[This section will be updated in a future version. Stay tuned.]

26.5.3 8-point F-matrix RANSAC

[This section will be updated in a future version. Stay tuned.]

26.5.4 6-point DLT-RANSAC

[This section will be updated in a future version. Stay tuned.]

27 Indirect VO Methods

27.1 MSCKF Method

The MSCKF is a VIO method based on an optical flow frontend and a sliding-window filter backend. It was first proposed by Mourikis and Roumeliotis in 2007 and later extended to a stereo VIO by Sun Ke et al. in 2017. This method formulates the VIO problem as a nonlinear filtering problem by establishing an error-state model and solves it using an EKF.

27.1.1 Overall Framework

The MSCKF is a VIO with a decoupled frontend-backend architecture. The frontend takes IMU and image data as input and outputs matched 2D feature point coordinates (assigned unique IDs). The backend takes these 2D feature coordinates as input and outputs state estimates such as pose and velocity while maintaining the map. Since the MSCKF backend lacks components like loop closure detection and relocalization, it is generally considered only a VIO.

The MSCKF frontend consists of the following components: FAST feature point detection, feature point matching based on pyramidal Lucas-Kanade (L-K) optical flow, RANSAC outlier rejection, and feature point registration. The stereo version of MSCKF also includes stereo feature matching and rejection.

The core of the MSCKF backend is an ESKF (introduced in Section 11), which includes state prediction, state update, sliding-window management, and marginalization.

The coordinate systems in MSCKF include: the body frame (b), the navigation frame (n), and the camera frame (c).

- The n-frame is defined as: the x-y-z axes point east/north/up (ENU), respectively, and the origin is fixed relative to the ground.
- The b-frame is defined as: the frame rigidly attached to the IMU of the mobile robot (vehicle).
- The c-frame is defined as: the frame rigidly attached to the camera, consistent with the definition in the "Camera Model" section.

27.1.2 Optical Flow Frontend

The MSCKF frontend is an indirect method based on optical flow. It first detects feature points using the FAST method and then tracks them using pyramidal L-K optical flow. The MSCKF frontend algorithm is summarized as follows:

[This section will be updated in a future version. Stay tuned.]

27.1.3 ESKF State Definition

The system state in MSCKF is divided into two parts: **IMU state** and **camera state**. For all states, the rotation is represented using quaternions describing the coordinate transformation from the b-frame/c-frame to the n-frame.

The **IMU state** consists of variables directly related to the IMU, including: the vehicle's attitude relative to the n-frame q_b^n, position p_{nb}^b, velocity v_{nb}^b, gyroscope bias b_g, and accelerometer bias b_a. The IMU state satisfies a kinematic differential equation, derived in the next subsection.

The estimated IMU state is:

$$\hat{x}_{IMU} := \begin{bmatrix} q_{nb'}^{b'} & p_{nb'}^{n} & v_{nb'}^{n} & \hat{b}_g & \hat{b}_a \end{bmatrix}^T$$

Here, the b'-frame represents the estimated body frame when the pose estimation is inaccurate. The estimated state also satisfies a differential equation, derived in the next subsection.

We can also consider the b'-frame as the b-frame perturbed by a small rotation, where the perturbation is the error state of the attitude. For rotation, let $\delta\theta$ denote the 3D small-angle vector representing the rotation from the b'-frame to the b-frame. According to Eqs. I.2.21 and I.2.2, we have:

$$q_b^{b'} \approx \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix}$$
$$R_b^{b'} \approx I + (\delta\theta)_\times$$

This means that the quaternion $q_{b'}^n$ and rotation matrix $R_{b'}^n$ are perturbed from the right:

$$q_b^n \approx q_{b'}^n \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix}$$
$$R_b^n \approx R_{b'}^n (I + (\delta\theta)_\times)$$

Thus, the IMU error state is defined as:

$$\tilde{x}_{IMU} := \begin{bmatrix} \delta\theta^T & (\tilde{p}_{nb}^n)^T & (\tilde{v}_{nb}^n)^T & (\tilde{b}_g)^T & (\tilde{b}_a)^T \end{bmatrix}^T \in \mathbb{R}^{15}$$

where:

$$\begin{aligned} q_b^n &\approx q_{b'}^n \otimes \begin{bmatrix} 1 \\ -\frac{1}{2}\delta\theta \end{bmatrix} \\ p_{nb}^n &= \tilde{p}_{nb}^n + p_{nb'}^n \\ v_{nb}^n &= \tilde{v}_{nb}^n + v_{nb'}^n \\ b_g &= \tilde{b}_g + \hat{b}_g \\ b_a &= \tilde{b}_a + \hat{b}_a \end{aligned} \tag{VII.27.1}$$

The IMU error state satisfies a linearized equation, derived in the next subsection.

The **camera state** consists of historical camera poses. Each historical pose includes the camera attitude q_c^n and position p_{nc}^n. The full state includes the IMU state and multiple historical camera poses.

$$x_{c_j} = [(q_{c_j}^n)^T \quad (p_{nc_j}^n)^T]^T$$
$$x = \begin{bmatrix} x_{IMU}^T & x_{c_m}^T & x_{c_{m+1}}^T & \dots & x_{c_{m+l}}^T \end{bmatrix}^T$$

Similar to the IMU state, we assume there is an error between the estimated and true camera states. We use the rotation vector perturbation $\delta\theta_{c_j}$ to represent $q_{c_j'}^{c_j}$ (the attitude estimation error) and $\tilde{p}_{nc_j}^n$ to represent the position estimation error:

$$\begin{aligned} q_{c_j'}^n &= q_{c_j}^n \otimes q_{c_j'}^{c_j} = q_{c_j}^n \otimes \begin{bmatrix} 0 \\ \frac{1}{2}\delta\theta_{c_j} \end{bmatrix} \\ \tilde{p}_{nc_j}^n &= p_{nc_j}^n - p_{nc_j'}^n \end{aligned} \tag{VII.27.2}$$

These form the camera error state:

$$\tilde{x}_{c_j} = [(\delta\theta_{c_j})^T \quad (\tilde{p}_{nc_j}^n)^T]^T \tag{VII.27.3}$$

All camera error states and the IMU error state together constitute the total error state:

$$\tilde{x} = \begin{bmatrix} \tilde{x}_{IMU}^T & \tilde{x}_{c_m}^T & \tilde{x}_{c_{m+1}}^T & \dots & \tilde{x}_{c_{m+l}}^T \end{bmatrix}^T \in \mathbb{R}^{15+6l}$$

27.1.4 System Equations

State Equation

In this subsection, we derive the differential equation satisfied by the IMU state.

We know that position and attitude are the outputs of the VIO problem. The first derivative of position is velocity, and the second derivative is acceleration; the first derivative of attitude can be considered angular velocity. As inertial sensors, IMUs measure angular velocity and acceleration, effectively measuring the second derivatives of motion. Therefore, we first need to construct a second-order differential equation to describe the relationships among these physical quantities.

Specifically, we **use the quaternion representing the rotation from the b-frame to the n-frame**. Under ideal conditions (ignoring Coriolis forces), the kinematic differential equation of the vehicle is:

$$\begin{aligned}\dot{q}_b^n &= q_b^n \otimes \frac{1}{2}(\omega^b - b_g)\\ \dot{v}_{nb}^n &= R_b^n(f^b - b_a) + \mathbf{g}^n\\ \dot{p}_{nb}^n &= v_{nb}^n\end{aligned}$$

Here, q_b^n, p_{nb}^b, and v_{nb}^b represent the vehicle's attitude, position, and velocity relative to the n-frame, respectively. f^b is the direct measurement from the IMU accelerometer (specific force). ω^b is the direct measurement from the IMU gyroscope (angular velocity). $\mathbf{g}^n$ is the gravity vector in the n-frame. b_g and b_a are the gyroscope and accelerometer biases, both 3D vectors. For MEMS IMUs, these biases vary each time the device is powered on.

For real systems, IMU measurements include noise, and the gyroscope and accelerometer biases undergo random walks. Incorporating noise, the system equation becomes:

$$\begin{aligned}\dot{q}_b^n &= q_b^n \otimes \frac{1}{2}(\omega^b - b_g - n_g)\\ \dot{v}_{nb}^n &= R_b^n(f^b - b_a - n_a) + \mathbf{g}^n\\ \dot{p}_{nb}^n &= v_{nb}^n\\ \dot{b}_g &= n_{wg}\\ \dot{b}_a &= n_{wa}\end{aligned}$$

As can be seen, this is a nonlinear stochastic system. For our VIO task, we consider $q_b^n, p_{nb}^n, v_{nb}^n, b_g, b_a$ as the true states. Since we need to treat the pose as a state, we employ the ESKF method introduced in Section 11.3.5. In addition to the aforementioned true states, we should also define the estimated state $\hat{x}$ and the error state $\tilde{x}$ (i.e., δx, where the notation is kept consistent with the MSCKF paper).

Specifically, the estimated states defined by MSCKF, $q_{nb'}^{b'}, p_{nb'}^n, v_{nb'}^n, \hat{b}_g, \hat{b}_a$, should satisfy the equations

$$\begin{aligned}\dot{\hat{q}}_{b'}^n &= \hat{q}_{b'}^n \otimes \frac{1}{2}(\omega^b - \hat{b}_g)\\ \dot{\hat{v}}_{nb'}^n &= R_{b'}^n(f^b - \hat{b}_a) + \mathbf{g}^n\\ \dot{\hat{p}}_{nb'}^n &= \hat{v}_{nb'}^n\\ \dot{\hat{b}}_g &= 0\\ \dot{\hat{b}}_a &= 0\end{aligned}$$

Error State Equation

The differential equations for the IMU estimated states introduced above are nonlinear, whereas the error states satisfy the following linear differential equations

$$\dot{\tilde{x}}_{IMU} = F_{IMU}\tilde{x}_{IMU} + G_{IMU}\mathbf{n}_{IMU} \tag{VII.27.4}$$

where

$$F_{IMU} = \begin{bmatrix} -(\omega^b - \hat{b}_g)_\times & 0_{3\times3} & 0_{3\times3} & -I_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & I_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ -R_{b'}^n(f^b - \hat{b}_a)_\times & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & -R_{b'}^n \\ & 0_{3\times15} & & & \\ & 0_{3\times15} & & & \end{bmatrix}$$

$$G_{IMU} = \begin{bmatrix} -I_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & -R_{b'}^n & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & I_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & I_{3\times3} \end{bmatrix} \tag{VII.27.5}$$

$$\mathbf{n}_{IMU} = [n_g \quad n_a \quad n_{wg} \quad n_{wa}]^T \sim \mathcal{N}(0, Q)$$

The above form represents the matrix expression of the error state differential equations. Expanding it, we obtain the following linear differential equations:

$$\begin{aligned}
\dot{\delta\theta} &= -(\omega^b - \hat{b}_g)_\times(\delta\theta) - \tilde{b}_g - n_g \\
\dot{\tilde{p}}^n_{nb} &= \tilde{v}^n_{nb} \\
\dot{\tilde{v}}^n_{nb} &= -R^n_{b'}(f^b - \hat{b}_a)_\times(\delta\theta) - R^n_{b'}\tilde{b}_a - R^n_{b'}n_a \\
\dot{\tilde{b}}_g &= n_{wg} \\
\dot{\tilde{b}}_a &= n_{wa}
\end{aligned} \tag{VII.27.6}$$

Derivation of the Error State Equations

The second, fourth, and fifth lines in Equation VII.27.6 are straightforward. Below, we prove the first and third lines, namely the differential equations for the rotation vector perturbation and velocity error.

For the velocity error, based on the above error relationship, we have

$$\begin{aligned}
\dot{\tilde{v}}^n_{nb} &= \dot{v}^n_{nb} - \dot{v}^n_{nb'} \\
&= R^n_b(f^b - b_a - n_a) + \mathbf{g}^n - (R^n_{b'}(f^b - \hat{b}_a) + \mathbf{g}^n) \\
&= (R^n_b - R^n_{b'})f^b - R^n_b b_a + R^n_{b'}\hat{b}_a - R^n_b n_a \\
&= (R^n_b - R^n_{b'})f^b - R^n_b b_a + R^n_b\hat{b}_a - R^n_b\hat{b}_a + R^n_{b'}\hat{b}_a - R^n_b n_a \\
&= (R^n_b - R^n_{b'})f^b - R^n_b(b_a - \hat{b}_a) + (R^n_{b'} - R^n_b)\hat{b}_a - R^n_b n_a \\
&= (R^n_b - R^n_{b'})(f^b - \hat{b}_a) - R^n_b\tilde{b}_a - R^n_b n_a \\
&\approx (R^n_{b'}(I + (\delta\theta)_\times) - R^n_{b'})(f^b - \hat{b}_a) - R^n_{b'}\tilde{b}_a - R^n_{b'}n_a \\
&\approx R^n_{b'}(\delta\theta)_\times(f^b - \hat{b}_a) - R^n_{b'}\tilde{b}_a - R^n_{b'}n_a \\
&= -R^n_{b'}(f^b - \hat{b}_a)_\times(\delta\theta) - R^n_{b'}\tilde{b}_a - R^n_{b'}n_a
\end{aligned}$$

Before proving the differential equation for the rotation vector perturbation, we first derive $\dot{q}^n_b$. Based on the earlier definitions, we have the approximate relationship

$$q^n_b \approx q^n_{b'} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix}$$

According to the definition of quaternion multiplication, the derivative of the product of two quaternions follows a rule similar to ordinary multiplication. Taking the derivative of both sides of the above equation, we obtain

$$\dot{q}^n_b = \dot{q}^n_{b'} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} + q^n_{b'} \otimes \begin{bmatrix} 0 \\ \frac{1}{2}\dot{\delta\theta} \end{bmatrix}$$

Since

$$\begin{aligned}
\dot{q}^n_b &= q^n_b \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - b_g - n_g) \end{bmatrix} \\
\dot{q}^n_{b'} &= q^n_{b'} \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - \hat{b}_g) \end{bmatrix}
\end{aligned}$$

Therefore

$$q^n_b \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - b_g - n_g) \end{bmatrix} = q^n_{b'} \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - \hat{b}_g) \end{bmatrix} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} + q^n_{b'} \otimes \begin{bmatrix} 0 \\ \frac{1}{2}\dot{\delta\theta} \end{bmatrix}$$

Expanding q^n_b further,

$$q^n_{b'} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - b_g - n_g) \end{bmatrix} = q^n_{b'} \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - \hat{b}_g) \end{bmatrix} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} + q^n_{b'} \otimes \begin{bmatrix} 0 \\ \frac{1}{2}\dot{\delta\theta} \end{bmatrix}$$

Eliminating q^n_b, we obtain

$$\begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - b_g - n_g) \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - \hat{b}_g) \end{bmatrix} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{2}\dot{\delta\theta} \end{bmatrix}$$

From Equation I.2.12, taking the vector part yields

$$\frac{1}{2}(\omega^b - b_g - n_g) + \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(\omega^b - b_g - n_g) = \frac{1}{2}(\omega^b - \hat{b}_g) + \frac{1}{2}(\omega^b - \hat{b}_g) \times \left(\frac{1}{2}\delta\theta\right) + \frac{1}{2}\dot{\delta\theta}$$

Rearranging terms, we have

$$\begin{aligned} \frac{1}{2}\dot{\delta\theta} &= \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(\omega^b - b_g - n_g) - \frac{1}{2}(\omega^b - \hat{b}_g) \times \left(\frac{1}{2}\delta\theta\right) + \frac{1}{2}(\hat{b}_g - b_g - n_g) \\ &= \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(\omega^b - b_g - n_g) + \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(\omega^b - \hat{b}_g) + \frac{1}{2}(\hat{b}_g - b_g - n_g) \\ &= \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(2\omega^b - 2\hat{b}_g - \tilde{b}_g - n_g) + \frac{1}{2}(-\tilde{b}_g - n_g) \\ &\approx \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(2\omega^b - 2\hat{b}_g) - \frac{1}{2}(\tilde{b}_g + n_g) \end{aligned}$$

Thus,

$$\begin{aligned} \dot{\delta\theta} &= (\delta\theta) \times (\omega^b - \hat{b}_g) - \tilde{b}_g - n_g \\ &= -(\omega^b - \hat{b}_g) \times (\delta\theta) - \tilde{b}_g - n_g \\ &= -(\omega^b - \hat{b}_g)_\times (\delta\theta) - \tilde{b}_g - n_g \end{aligned}$$

The above represents the continuous-time linear differential equation for the system error state. In practical applications, it is discretized, as detailed in the "Filter Prediction Step" subsection below.

27.1.5 Filter Initialization

The MSCKF estimation state requires an initial value. Typically, we assume the carrier is stationary at the initial time, with the initial position as the origin of the n-frame. Therefore, we only need to estimate q_b^n, b_g, and b_a. Among these, estimating b_g is relatively straightforward—simply average the gyroscope data over a period.

For the (average) accelerometer measurement f^b, it involves both attitude and b_a. We have

$$f^b = -g^b + b_a = -(R_b^n)^T g^n + b_a$$

where the gravitational acceleration in the n-frame is

$$g^n = \begin{bmatrix} 0 \\ 0 \\ -g_0 \end{bmatrix}$$

For R_b^n, since the heading does not affect the measurements, we only consider the roll angle r and pitch angle p:

$$R_b^n = R_x(p)R_y(r) = \begin{bmatrix} \cos r & 0 & \sin r \\ \sin p \sin r & \cos p & -\sin p \cos r \\ -\cos p \sin r & \sin p & \cos p \cos r \end{bmatrix}$$

Thus, we obtain

$$f^b = -\begin{bmatrix} \cos r & 0 & \sin r \\ \sin p \sin r & \cos p & -\sin p \cos r \\ -\cos p \sin r & \sin p & \cos p \cos r \end{bmatrix}^T \begin{bmatrix} 0 \\ 0 \\ -g_0 \end{bmatrix} + b_a$$

That is,

$$f^b = g_0 \begin{bmatrix} -\cos p \sin r \\ \sin p \\ \cos p \cos r \end{bmatrix} + b_a$$

Assuming b_a and f^b are collinear, we have

$$f^b / ||f^b|| = \begin{bmatrix} -\cos p \sin r \\ \sin p \\ \cos p \cos r \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}$$

Thus,

$$\begin{aligned} p &= \arcsin(f_2) \\ r &= \arctan\left(\frac{-f_1}{f_3}\right) \end{aligned} \tag{VII.27.7}$$

At this point, we have

$$b_a = f^b - g_0 ||f^b|| \tag{VII.27.8}$$

27.1.6 Filter Prediction Step

The filter prediction step in MSCKF refers to predicting the state $\hat{x}_k^-$ and covariance matrix P_{k+1}^- from $\hat{x}_k$ and P_k using the system equations, without considering measurements. In other words, it involves state prediction and covariance matrix prediction.

—

Let me know if you'd like any refinements to the translation!

State Prediction The ESKF requires the discrete-time system equations and their first-order derivatives. The system equation for the IMU state (Equation VII.27.5) is in continuous form and does not include the camera pose. Therefore, we first need to extend it to the full state and then discretize it.

Note that the camera poses in the sliding window are historical poses and do not change with the current time. Thus, in the continuous domain, the differential equation for the full state is:

$$\dot{\hat{x}} = f(\hat{x}) = \begin{bmatrix} q_{b'}^n \otimes \frac{1}{2}(\omega^b - \hat{b}_g) \\ R_{b'}^n (f^b - \hat{b}_a) + \mathbf{g}^n \\ v_{nb'}^n \\ 0_{6\times 1} \\ 0_{6l\times 1} \end{bmatrix}$$

Next, we consider its discretization. Assuming the discrete time interval is t_s, we have:

$$\hat{x}_{k+1}^- = \hat{x}_k + \int_{kt_s}^{(k+1)t_s} f(x_k) dt$$

In practice, numerical integration methods such as RK4 are often used for discretized state prediction.

Covariance Prediction For covariance prediction, we need to consider both **linearization** and **discretization**. We first perform linearization approximation in the continuous domain and then discretize the linearized system. Specifically, for the nonlinear continuous state equation above, its linearization is:

$$\dot{\hat{x}} \approx F_c(\hat{x})\hat{x}$$

where the subscript c denotes "continuous," and:

$$F_c(\hat{x}) = \begin{bmatrix} F_{IMU}(\hat{x}_{IMU}) & 0_{15\times 6l} \\ 0_{6l\times 15} & 0_{6l\times 6l} \end{bmatrix}$$

Referring to the conclusions of continuous-time Kalman filtering, under linearization approximation, the differential equation for the covariance P is:

$$\dot{P} = F_c P + P F_c^T + G_c Q G_c^T$$

where Q is the covariance of the system noise $\mathbf{n}_{IMU}$, and G_c represents the contribution of noise to the system derivative:

$$G_c = \begin{bmatrix} G_{IMU}(\hat{x}_{IMU}) \\ 0_{6l \times 12} \end{bmatrix}$$

Assuming the continuous state covariance P satisfies $P = P^T$ and is partitioned as:

$$P = \begin{bmatrix} P_{11} & P_{12} \\ P_{12}^T & P_{22} \end{bmatrix} \tag{VII.27.9}$$

Substituting the expressions for F_c and G_c into the differential equation yields:

$$\begin{bmatrix} \dot{P}_{11} & \dot{P}_{12} \\ \dot{P}_{12}^T & \dot{P}_{22} \end{bmatrix} = \begin{bmatrix} F_{IMU}P_{11} + P_{11}F_{IMU} + G_{IMU}QG_{IMU}^T & F_{IMU}P_{12} \\ P_{12}^T F_{IMU}^T & 0_{6l \times 6l} \end{bmatrix} \tag{VII.27.10}$$

That is:

$$\begin{aligned} \dot{P}_{11} &= F_{IMU}P_{11} + P_{11}F_{IMU} + G_{IMU}QG_{IMU}^T \\ \dot{P}_{12} &= F_{IMU}P_{12} \\ \dot{P}_{22} &= 0_{6l \times 6l} \end{aligned}$$

For P_{11} and P_{12}, assuming negligible changes in F_{IMU} and G_{IMU}, this is a linear ordinary differential equation with a known initial value, whose analytical solution is:

$$\begin{aligned} P_{k,11}^- &= \exp(F_{IMU}t_s)P_{k,11}\exp(F_{IMU}t_s)^T + Q_{11} \\ Q_{11} &= \int_{kt_s}^{kt_s+t} \exp(F_{IMU}(t-\tau))G_{IMU}QG_{IMU}^T \exp(F_{IMU}(t-\tau))^T d\tau \\ \dot{P}_{k,12}^- &= \exp(F_{IMU}t_s)P_{k,12} \end{aligned}$$

Here, $\exp(\cdot)$ denotes the matrix exponential. Let:

$$\Phi_k = \exp(F_{IMU}(\hat{x}_k)t_s) \tag{VII.27.11}$$

Then:

$$\begin{aligned} P_{k,11}^- &= \Phi_k P_{k,11}\Phi_k + Q_{11} \\ Q_{11} &\approx \Phi_k G_{IMU}(Qt_s)G_{IMU}^T\Phi_k^T \\ P_{k,12}^- &= \Phi_k P_{k,12} \\ P_{k,22}^- &= P_{k,22} \end{aligned} \tag{VII.27.12}$$

In practical calculations, the matrix exponential Φ_k can be approximated using a 3rd-order Taylor expansion:

$$\Phi_k \approx I + F_{IMU}t_s + \frac{t_s^2}{2}F_{IMU}^2 + \frac{t_s^3}{6}F_{IMU}^3$$

In summary, we present the prediction step algorithm for MSCKF as follows:

Algorithm 94: MSCKF Prediction Step

Input: Estimated state $\hat{x}_k$ at step k, estimated covariance P_k
Input: System noise covariance matrix Q
Input: Specific force $\mathbf{f}^b$, angular velocity $\mathbf{w}_{nb}^b$
Output: Predicted state $\hat{x}_k^-$ at step k, predicted covariance P_{k+1}^-

$\hat{x}_k^- \leftarrow RK4\_int(\hat{x}_k, f, kt_s, (k+1)t_s; \mathbf{f}^b, \mathbf{w}_{nb}^b)$
$F_{IMU} \leftarrow F_{IMU}(\hat{x}_k)$
$G_{IMU} \leftarrow G_{IMU}(\hat{x}_k)$
$\Phi_k \leftarrow I + F_{IMU}t_s + \frac{t_s^2}{2}F_{IMU}^2 + \frac{t_s^3}{6}F_{IMU}^3$
$\begin{bmatrix} P_{k,11} & P_{k,12} \\ P_{k,12}^T & P_{k,22} \end{bmatrix} \leftarrow P_k$
$Q_{11} \leftarrow \Phi_k G_{IMU}(Qt_s) G_{IMU}^T \Phi_k^T$
$P_{k,11}^- \leftarrow \Phi_k P_{k,11} \Phi_k + Q_{11}$
$P_{k,12}^- \leftarrow \Phi_k P_{k,12}$
$P_{k+1}^- \leftarrow \begin{bmatrix} P_{k,11}^- & P_{k,12}^- \\ (P_{k,12}^-)^T & P_{k,22} \end{bmatrix}$

27.1.7 Camera State Augmentation

The error states of the cameras in the sliding window, together with the IMU error state, form the total **error state**:

$$\tilde{x} = \begin{bmatrix} \tilde{x}_{IMU}^T & \tilde{x}_{c_m}^T & \tilde{x}_{c_{m+1}}^T & \cdots & \tilde{x}_{c_{m+l}}^T \end{bmatrix}^T \in \mathbb{R}^{15+6l} \tag{VII.27.13}$$

Assume we already have the current estimated pose of the vehicle $\hat{x}_{IMU}$. When the camera captures a new image I_k, how do we generate the corresponding estimated camera pose $\hat{x}_{c_j}$? Clearly, based on the coordinate system relationships, we have:

$$\begin{aligned} q_{c_j'}^n &= q_{b'}^n \otimes q_c^b \\ p_{nc_j'}^n &= p_{nb'}^n + R_{b'}^n p_{bc}^b \end{aligned} \tag{VII.27.14}$$

where q_c^b can be calculated from R_c^b in $\boldsymbol{T}_c^b$.

Since $R_{b'}^n / q_{b'}^n$ and $p_{nb'}^n$ are estimated values, the IMU error state x_{IMU} propagates to the camera error state $\tilde{x}_{c_j}$. Specifically, for the camera position error...

$$\begin{aligned} \tilde{p}_{nc_j}^n &= p_{nc_j}^n - p_{nc_j'}^n \\ &= p_{nb}^n + R_b^n p_{bc}^b - (p_{nb'}^n + R_{b'}^n p_{bc}^b) \\ &= \tilde{p}_{nb}^n + (R_b^n - R_{b'}^n) p_{bc}^b \\ &\approx \tilde{p}_{nb}^n + (R_b^n - R_b^n (I + (\delta\theta)_\times)) p_{bc}^b \\ &= \tilde{p}_{nb}^n - R_b^n (\delta\theta)_\times p_{bc}^b \\ &= \tilde{p}_{nb}^n + R_b^n (p_{bc}^b \times (\delta\theta)) \\ &= \tilde{p}_{nb}^n + R_b^n (p_{bc}^b)_\times (\delta\theta) \end{aligned}$$

For camera pose errors:

$$q_b^n \otimes q_c^b \otimes q_{c_j'}^{c_j} = q_{c_j'}^n = q_b^n \otimes q_{b'}^b \otimes q_c^b$$

Thus,

$$q_{c_j'}^{c_j} = q_b^c q_{b'}^b \otimes (q_b^c)^{-1}$$

That is,

$$\begin{bmatrix} 0 \\ \frac{1}{2}\delta\theta_{c_j} \end{bmatrix} = q_b^c \begin{bmatrix} 0 \\ \frac{1}{2}\delta\theta \end{bmatrix} \otimes (q_b^c)^{-1}$$

From Equation I.2.21, there exists a rotational transformation relationship between vectors:

$$\delta\theta_{c_j} = R_b^c(\delta\theta)$$

To summarize, we have:

$$\begin{aligned} \delta\theta_{c_j} &= R_b^c(\delta\theta) \\ \tilde{p}_{nc_j}^n &= \tilde{p}_{nb}^n + (p_{bc}^b)_\times(\delta\theta) \end{aligned} \tag{VII.27.15}$$

We denote the matrix representing the linear relationship between $\tilde{x}_{c_i}$ and $\tilde{x}_{IMU}$ as J_{c_i}, given by:

$$J_{c_i} = \begin{bmatrix} R_b^c & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ R_b^n(p_{bc}^b)_\times & I_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \end{bmatrix} \in \mathbb{R}^{6\times15} \tag{VII.27.16}$$

27.1.8 Filter Correction Step

In the ESKF framework, the core of the correction step is to compute the linearized matrix H_k of the measurement/output equation, which represents the linearized relationship between observation errors and error states. Similar to the BA problem setup earlier (see Section 26.4.3), we use the reprojection error (i.e., the error between the pixel coordinates of feature points under the estimated pose from the prediction step and the actual observed pixel coordinates) as the measurement.

Assume for the j-th camera pose, its estimated state is abbreviated as $p_{c,j} = p_{nc_j'}^n, R_j = R_{c_j'}^n$; and for the i-th 3D feature point, $p_i = p_{ni}^n$ is its 3D coordinate in the n-frame. Suppose the observation of the i-th feature point by the j-th camera is $z_{i,j}$, then:

$$z_{i,j} = \upsilon_i^{c_j} - \hat{\upsilon}_i(R_j, p_{c,j}) \tag{VII.27.17}$$

where $\upsilon_i^{c_j}$ is the measured value, and:

$$\begin{aligned} \hat{\upsilon}_i(R_j, p_{c,j}) &= \frac{1}{z_j^{c_j}} K p_i^{c_j} \\ p_i^{c_j} &= R_j^T(p_i - p_{c,j}) \end{aligned} \tag{VII.27.18}$$

Considering the derivative of the measurement $z_{i,j}$ with respect to the full error state $\tilde{x}$, it can be seen that only the block corresponding to the j-th camera state $\tilde{x}_{c_j}$ is non-zero. Although the IMU state $\tilde{x}_{IMU}$ is also related to $\tilde{x}_{c_j}$, we treat them as independent when computing the H matrix. Let:

$$H_{i,j} = \frac{\partial z_{i,j}}{\partial \tilde{x}_{c_j}} \tag{VII.27.19}$$

Referring to the derivation in Section 26.4.3, we have:

$$H_{i,j} = \frac{\partial \hat{\upsilon}_i^{c_j}}{\partial \nu_i^{c_j}} \frac{\partial \nu_i^{c_j}}{\partial p_i^{c_j}} \frac{\partial p_i^{c_j}}{\partial \tilde{x}_{c_j}}$$

And:

$$\begin{aligned} \frac{\partial \hat{\upsilon}_i^{c_j}}{\partial \nu_i^{c_j}} &= K \\ \frac{\partial \nu_i^{c_j}}{\partial p_i^{c_j}} &= \frac{1}{z_i^{c_j}} \begin{bmatrix} 1 & 0 & -x_i^{c_j}/z_i^{c_j} \\ 0 & 1 & -y_i^{c_j}/z_i^{c_j} \end{bmatrix} \\ \frac{\partial p_i^{c_j}}{\partial \tilde{x}_{c_j}} &= \begin{bmatrix} (R_j^T(p_i - p_{c,j}))_\times & -R_j^T \end{bmatrix} \end{aligned}$$

Thus, assuming the feature point coordinate p_i is known, we can compute the matrix H_k.

27.1.9 Sliding Window Management and Marginalization

[This section will be updated in a future version. Stay tuned.]

—

The translation strictly adheres to your requirements: 1. All LaTeX commands and environments remain unchanged. 2. Mathematical formulas, code blocks, tables, and technical content are preserved. 3. Only natural language Chinese text has been translated to English. 4. The document structure and formatting are maintained.

27.2 VINS-Fusion

[This section will be updated in subsequent versions. Stay tuned.]

28 Direct VO Methods

28.1 DSO Approach

[This section will be updated in subsequent versions, stay tuned]

29 Rendering-Based VO Methods

29.1 3D Gaussian Splatting

[This section will be updated in a later version. Stay tuned.]

29.2 MonoGS

[This section will be updated in a later version. Stay tuned.]