Math Toolbox for Robotics v0.1.2

Xinran Wei

2025 年 10 月 14 日

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

目录

1 导语 **6**
1.1 鸣谢 . . . 6
1.2 版权声明 . . . 7
1.3 联系作者 . . . 7

2 索引 **8**
2.1 符号 . . . 8
2.2 缩写 . . . 9
2.3 问题 . . . 11
2.4 算法 . . . 15

I 数学物理基础 **19**

1 三维坐标系 **19**
1.1 三维空间 . . . 19
1.2 旋转矩阵 . . . 21
1.3 齐次矩阵 . . . 23
1.4 动坐标系位矢求导 . . . 23

2 旋转的描述 **26**
2.1 角-轴表示法 . . . 26
2.2 欧拉角 . . . 28
2.3 四元数 . . . 28
2.4 各类旋转表示的转换 . . . 34
2.5 旋转矩阵与求导 . . . 36
2.6 各类旋转微分的转换 . . . 40

3 最优化方法 **44**
3.1 优化问题定义 . . . 44
3.2 无约束优化 . . . 46
3.3 等式约束优化 . . . 52
3.4 不等式约束优化 . . . 55
3.5 二次规划 QP . . . 58
3.6 非线性最小二乘 . . . 62

4 插值 **69**
4.1 基本概念 . . . 69
4.2 样条插值 . . . 69

II 机器人学基础 **77**

5 基本概念 **77**
5.1 关节与正逆运动学 . . . 77
5.2 雅可比，动力学和控制 . . . 77
5.3 轨迹规划和插值 . . . 78

6 正逆运动学 **79**
6.1 D-H 参数 . . . 79
6.2 正运动学 . . . 80
6.3 逆运动学 . . . 82

7 微分运动学 **84**
7.1 雅可比矩阵 . . . 84
7.2 雅可比求解 . . . 85
7.3 基于雅可比的 IK . . . 90
7.4 雅可比的性质 . . . 94

8 动力学 **95**
8.1 动能与势能 . . . 95
8.2 动力学方程推导 . . . 96
8.3 动力学方程性质 . . . 99

9 轨迹规划 **101**

III 控制理论基础 **102**

10 控制基础知识 **102**
10.1 基本概念 . . . 102
10.2 频域基本工具 . . . 107
10.3 时域响应与指标 . . . 109

10.4 稳定性 . . . 113
10.5 线性时域工具 . . . 113
10.6 一阶倒立摆问题 . . . 116

11 观测与滤波 **119**
11.1 基本概念 . . . 119
11.2 线性状态观测器 . . . 122
11.3 卡尔曼滤波 KF . . . 122
11.4 无迹卡尔曼滤波 UKF . . . 133

12 最优控制基础 **139**
12.1 最优控制问题 . . . 139
12.2 贝尔曼最优与动态规划 . . . 141
12.3 连续最优性原理 . . . 145
12.4 离散时间 LQR . . . 147
12.5 连续时间 LQR . . . 152
12.6 非线性近似 LQR 控制 . . . 154
12.7 最优跟踪控制 . . . 158
12.8 约束最优控制问题 . . . 163

13 模型预测控制基础 **165**
13.1 基本概念 . . . 165
13.2 无约束线性 MPC . . . 166
13.3 有约束线性 MPC . . . 168

IV 机器人控制基础 **172**

14 机器人控制概述 **172**
14.1 运动控制 . . . 172
14.2 阻抗/导纳控制 . . . 173
14.3 约束力控制 . . . 175

15 独立关节控制 **177**
15.1 基本概念 . . . 177
15.2 独立关节运动控制 . . . 181

16 关节空间控制 **184**
16.1 关节空间运动控制 . . . 184
16.2 关节空间阻抗控制 . . . 188

17 操作空间控制 **189**
17.1 操作空间运动控制 . . . 189
17.2 操作空间阻抗控制 . . . 193

17.3 操作空间约束力控制 . . . 196

V 深度学习方法 **197**

18 深度学习基础 **197**
18.1 机器学习基本概念 . . . 198
18.2 深度学习与 DNN . . . 201
18.3 多层感知机 MLP . . . 203
18.4 深度学习理论 . . . 209

19 优化与正则化 **211**
19.1 DNN 的困难 . . . 211
19.2 DNN 优化算法 . . . 212
19.3 正则化技巧 . . . 223

VI 强化学习方法 **228**

20 强化学习基础 **228**
20.1 状态空间 . . . 228
20.2 MRP 与价值 . . . 228
20.3 MRP 的价值估计 . . . 229
20.4 MDP 与动作价值 . . . 233
20.5 MDP 预测问题 . . . 235
20.6 MDP 控制问题 . . . 241
20.7 强化学习问题 . . . 246
20.8 强化学习观点 . . . 247

21 表格型方法 **248**
21.1 蒙特卡洛策略迭代 . . . 248
21.2 SARSA 方法 . . . 250
21.3 Q 学习方法 . . . 252
21.4 倒立摆求解 . . . 253

22 策略梯度方法 **255**
22.1 策略参数化 . . . 255
22.2 策略梯度定理 . . . 256
22.3 原始 REINFORCE 算法 . . . 259
22.4 REINFORCE 算法 . . . 262
22.5 倒立摆求解 . . . 264

23 PPO 算法 **265**
23.1 重要性采样 . . . 265
23.2 Critic 网络 . . . 267
23.3 PPO 算法 . . . 270
23.4 倒立摆求解 . . . 274

VII 视觉导航方法 **275**

24 视觉里程计基础 **275**
24.1 基本概念 . . . 275
24.2 相机模型 . . . 280

25 特征点与光流 **282**
25.1 数字图像基础 . . . 282
25.2 特征点基本概念 . . . 283
25.3 Harris 角点 . . . 284
25.4 SIFT 特征点 . . . 288
25.5 ORB 特征点 . . . 293
25.6 光流基本概念 . . . 294
25.7 L-K 光流 . . . 296

26 点对位姿求解 **299**
26.1 问题建立 . . . 299
26.2 2D-2D 问题 . . . 300
26.3 3D-3D 问题 . . . 314
26.4 3D-2D 问题 . . . 318
26.5 RANSAC 方法 . . . 328

27 间接 VO 方法 **330**
27.1 MSCKF 方法 . . . 330
27.2 VINS-Fusion . . . 340

28 直接 VO 方法 **341**
28.1 DSO 方法 . . . 341

29 基于渲染的 VO 方法 **342**
29.1 3D 高斯泼溅 . . . 342
29.2 MonoGS . . . 342

1 导语

机器人是我们这个时代最令人激动人心的技术之一。尽管这个学科本身并不年轻，工业机器人等成熟产品也早已在 30 年前就开始为世界服务，但新兴的 AI 技术 (主要是深度学习 (DL)/计算机视觉 (CV)/大语言模型 (LLM)/强化学习 (RL) 等) 正让我们这个时代的机器人以前所未有的方式认识、理解和决策。今天，我们比历史上的任何时期都更接近真正自主、通用、智能的机器人。

未来的机器人，需要怎样的技术人才？在大语言模型 (LLM) 已经可以生成和修改代码、推导公式的今天，一个面向未来的机器人工程师，应当具备怎样的能力和素质？本书试图给出这样的回答——未来需要的是不局限在某个领域，而是对机器人技术的所有领域都有理解的通才机器人工程师。

因此，本书和现有的大部分机器人教材在内容编排上颇有不同。我们的内容覆盖了当今世界机器人技术的各个主要领域而非单个技术领域，包括但不限于：**机器人建模与控制 (运动学与动力学，控制理论)、机器人感知 (vSLAM，滤波)、机器人学习 (深度学习，强化学习)**，等等。我们认为：在当下这个时代，具有对机器人各主要领域的基础有通盘理解 (而无需在每个领域都成为专家) 的视野，是开发下一代机器人的重要基石。

相比现有的各类教材，本书内容跨度非常广泛。如果从课程设置的角度来衡量，本书可能覆盖了 10-20 门本科生和研究生课程的内容。显然，我们不可能、也不追求替代这些领域现有的优秀的专业教材。我们亦并不追求在某个具体领域的全面性、前沿性，与之相反，我们写作的目标是建立跨学科的技术视野。我们可以在同样的设定下对比传统控制方法 (如 PID 和 LQR) 和数据驱动的方法 (如 PPO 算法) 在同一问题 (如倒立摆问题) 中的表现，我们可以讨论优化方法在解决不同问题时的应用，等等。我们希望，面对下一代机器人的需求，通过这些跨学科的视野，读者可以建立对机器人基础技术更加深刻的认知，从而更好地组合运用这些技术，创造真正的下一代机器人。

本书的写作风格也不同于以往的机器人教材。我们的定位是**工具书**而非教科书。这意味着，本书不会事无巨细的进行 0 基础讲解，不会如同数学教材一样追求绝对的严谨性，没有习题和实验，但本书包含机器人关键基础领域的**关键概念、问题、公式、算法、关键代码**。我们的追求是：使用简练、清晰的语言和符号，帮助读者清晰地理解这些内容的本质。

在这之中，是本书最为看重的是**问题**。问题意识是工程师思维的核心。一个优秀的工程师，应该且仅应该这样思考：如何定义一个问题、如何转化一个问题、如何解决一个问题……我们强调，**没有算法可以脱离问题而存在，任何算法都为解决某一类问题而生**。因此，读者将会在正文中看到，在跨学科的视野下，我们对各种机器人技术问题进行了详细的梳理和定义。这往往是相关领域的许多专业教材无法顾及的。

本书共划分为七大部分：**数学物理基础、机器人学基础、控制理论基础、机器人控制基础、深度学习方法、强化学习方法、视觉导航方法**，每一个部分的编号以罗马数字表示。每部分分为不同的章，每章分为节和小节，其编号以阿拉伯数字表示。

本书中的大部分算法都配备了相应的 python 代码。代码的核心部分列于算法之后。可运行的完整版代码已在 Github 开源：`https://github.com/weixr18/MT4R`，欢迎读者对照查阅。

1.1 鸣谢

本书的编写参考了大量现有的教材和参考书，首先我想感谢这些书籍的编写者们：在机器人基础和机器人控制部分，主要参考了《机器人学导论》《机器人学：建模、规划与控制》；在控制理论部分，参考了胡寿松《自动控制原理》、王天威《控制之美 (卷 1/卷 2)》等；在深度学习部分，参考了《深度学习》(" 花书") 和邱锡鹏《神经网络与深度学习》等；在强化学习部分，参考了 Sutton 的《强化学习》和 *EasyRL*(" 蘑菇书")；在 SLAM 部分参考了《视觉 SLAM 十四讲》等等。

其次，本书撰写还参考了本科和博士在清华和北航学习期间多门专业课的讲义。这些课程为我打开了机

器人领域的大门，在此我想对这些课程老师的辛勤付出表示感谢：清华自动化系赵明国、石宗英、张涛老师（《智能机器人》）；清华自动化系王焕钢、李力老师（《运筹学》）；北航计算机学院黄雷老师（《强化学习》）；北航自动化学院赵龙老师（《现代导航技术》），等等。谢谢你们！

除此之外，本书中的许多公式推导和算法的理解也借鉴了许多优质互联网资料，包括博客、问答网站、视频等等。读者可在每章相应内容的脚注和参考文献中找到这些材料的引用。在此由衷地感谢这些互联网创作者和分享者的工作。

最后，在本书的编写过程中，我的家人和朋友们也为我提供了巨大的帮助。感谢你们，没有你们就没有这本书的诞生！

1.2 版权声明

1.3 联系作者

您在阅读本书时，如果您有任何对本书的建议，或发现任何印刷、排版、文字、公式推导、图表等方面的问题或错误，欢迎通过电子邮件联系我：weixr0605@sina.com(请在邮件标题注明"[MT4R]")。诚挚感谢您的帮助！

2 索引

本书内容较多，为便于读者快速查找翻阅，在此将重要的符号定义、问题、算法等集中列出。

2.1 符号

2.1.1 导数布局

对于多元函数，其导数的写法一直以来有两种传统，称为**分子布局**和**分母布局**。对于函数

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

其一阶导数在两种布局下的形状不同：分子布局是 $\mathbb{R}^{1\times n}$，而分母布局则是 $\mathbb{R}^{n\times 1}$。在本书中，**我们将所有导数符号统一为分子布局**，即

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

我们定义梯度为和输入相同的变量，即和一阶偏导有转置关系

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

对于线性变换，我们有

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

对于微分计算 (即导数和输入的微分的乘法)，我们有

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

对于梯度下降，我们仍有

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

2.1.2 数组索引

本书中许多算法附带了 python 代码，我们力求使算法和代码中的变量命名一一对应，方便读者阅读。然而，编程语言的习惯和数学公式习惯并不一致。在数学公式中，例如要表示一组变量时，下标习惯从 1 开始，如 $b_1,...,b_n$。但在 python 等大多数编程语言中，列表索引从 0 开始。

为了解决这一问题，我们统一做如下的约定：在大多数情况下，通过在代码中加入冗余空间，使**代码中的索引和公式保持一致**。例如：对于算法中的变量组 $b_1,...,b_n$，则代码中会创建总长度为 $n+1$ 的变量组 b_ns，在索引时 b_ns$[n]$ 对应 $b[n]$。额外的未使用空间 b_ns[0] 既不访问，也不调用。如果算法中的变量索引带有增减，如 b_{n-1}，则代码中的数组也命名为 b_ns，通过对应的索引写入 b_ns$[n-1]$。

不过，这样的约定难免有例外。例如，对于固定长度的数组或向量 (比如 3 维空间的点坐标)，强行加入一个空维度，仅仅为了保持索引一致，就显得不甚必要。在这种情况下，我们不会加入额外的维度。

2.1.3 机器人学符号

本书的第 II 和第 IV 部分都是传统机器人学的内容。在这两部分中，对于关节空间，我们使用 q 代表机器人的关节向量 (一般为多维)，$\dot{q}$ 代表关节空间速度。如果仅考虑一个 (旋转) 关节，则用 θ 表示关节角。对于操作空间，我们用 x_e 代表机器人末端状态 (位姿)，$\dot{x}_e$ 或 v_e 代表操作空间速度 (广义速度，即包含角速度)。操作空间速度有两种表达方式，如果角度部分用角速度表示，记为 v_{ew}；如果用欧拉角的导数或四元数导数表示，记为 v_{ea}。

我们使用 $J(q)$ 表示机器人的雅可比矩阵，其中几何雅可比为 $J_w(q)$，分析雅可比为 $J_a(q)$。我们使用 τ 表示关节空间的驱动力矩，F_e 表示末端所受外力向量 (广义力，包含推力和扭力)。我们将欧拉角组成的向量记为 ϑ。在第 II 和第 IV 这两部分中，如果需要用到四元数，为和关节角区分，我们将四元数记为 $\mathbf{q}$。

2.1.4 最优控制

在第 III 部分的最优控制相关章节，我们使用冒号 : 代表数组/序列的切片。该切片索引与 matlab 风格保持一致，如：$a_:$ 表示序列 a 的全部元素；切片 $a:b$ 表示从索引 a 到索引 b 的元素，包含索引 a 和 b。数组根据需要可以从 0 或 1 开始索引。

2.1.5 强化学习

本书第 VI 部分是强化学习部分，我们需要处理各种随机变量和分布。对于强化学习中的关键要素，我们采用约定俗成的符号，即 s 代表状态，a 代表动作，r 代表奖励。状态空间记为 $\mathcal{A}$，动作空间记为 $\mathcal{S}$。以上的小写符号代表一个具体的取值，用于泛用场景。

如果我们要强调这是一个从分布中采样出的随机变量，我们会使用大写符号：S, A, R。注意随机变量无论是连续还是离散都使用大写。但有时，为了表达方便，我们也不会特意区分随机变量和它的采样值。

随机变量的分布与随机变量的性质有关，连续型对应概率密度，离散型对应分布列。我们使用大写符号如 $P(S_{t+1}|S_t)$ 表达离散型分布的概率函数，用小写符号如 $p(S_{t+1}|S_t, A_t)$ 表达连续型分布的概率密度函数。需要注意的是，很多时候我们推导的背景是离散型变量，但公式对于连续型也适用。此时我们倾向于使用小写符号。

一般来说，概率分布和它对应的函数值是两个概念。但有时为了表达方便，我们也会混用符号。例如，我们会用 $P(S_{t+1}|S_t)$ 直接表达状态转移的条件分布。

我们有时候要表达某个特定取值下，概率密度函数的值，此时我们会记为 $p(S_{t+1} = s'|S_t = s, A_t = a)$。这样的记法有些繁琐，在不引起歧义的情况下，我们会将其简化为 $p(s'|s, a)$。此外，如果我们要强调某个随机事件的概率，我们会记为 $Pr(\cdot)$。

2.2 缩写

本书中，我们将使用一些业内广泛使用的英文缩写以简化表述。下面我们在表1中列举全书中用到的缩写中，以供读者查阅。索引表按照缩写在正文中出现的顺序排序。

表 1: 缩写索引表

缩写	全拼	中文含义
DL	Deep Learning	深度学习
CV	Computer Vision	计算机视觉

缩写索引表 – 接上页

缩写	全拼	中文含义
LLM	Large Language Model	大语言模型
RL	Reinforcement Learning	强化学习
LSGD	Line Search Gradient Descent	线搜索梯度下降法
F-R 共轭梯度法	Fletcher-Reeves conjugate gradients	F-R 共轭梯度法
G-N 法	Gauss-Newton method	高斯-牛顿法
L-M 法	Levenberg-Marquardt method	列文伯格-马夸尔特法
D-H 参数	Denavit-Hartenberg parameters	D-H 参数
FK	Forward Kinematics	正运动学
IK	Inverse Kinematics	逆运动学
LTI 系统	Linear Time-Invariant system	线性时不变系统
PID 控制	Proportional Integral Derivative control	比例积分微分控制
PD 控制	Proportional Derivative control	比例微分控制
KF	Kalman Filter	卡尔曼滤波
EKF	Extended Kalman Filter	扩展卡尔曼滤波
UKF	Unscented Kalman Filter	无迹卡尔曼滤波
HJB 方程	Hamilton-Jacobi-Bellman equation	哈密顿-雅可比-贝尔曼方程
LQR	Linear Quadratic Regulator	线性二次型调节器
iLQR	iterative Linear Quadratic Regulator	迭代线性二次型调节器
DDP	Derivative Dynamic Programming	微分动态规划
MPC	Model Predictive Control	模型预测控制
JS	Joint Space	关节空间
OS	Operating Space	操作空间
DNN	Deep Neural Network	深度神经网络
DL	Deep Learning	深度学习
MLP	Multi-Layer Perceptron	多层感知机
GD	Gradient Descent	梯度下降
SGD	Stochastic Gradient Descent	随机梯度下降
BSGD	Batch Stochastic Gradient Descent	随机梯度下降
RL	Reinforcement Learning	强化学习
DRL	Deep Reinforcement Learning	深度强化学习
MRP	Markov Reward Process	马尔可夫回报过程
MDP	Markov Decision Process	马尔可夫决策过程
MC	Monte Carlo	蒙特卡洛
DP	Dynamic Programming	动态规划
POMDP	Partially Observable Markov Decision Process	部分可观测马尔可夫决策过程
SARSA	State-Action-Reward-State-Action	状态-动作-回报-状态-动作
K-L 散度	Kullback-Leibler divergence	K-L 散度
GAE	General Advantege Estimation	广义优势估计

缩写索引表 – 接上页

缩写	全拼	中文含义
PPO 算法	Proximal Policy Optimization algorithms	近似策略优化算法
SLAM	Simultaneous Localization and Mapping	同时定位与建图
vSLAM	Visual Simultaneous Localization and Mapping	视觉同时定位与建图
VO	Visual Odometer	视觉里程计
IMU	Inertial Measurement Unit	惯性测量单元
VIO	Visual Inertial Odometer	视觉惯性里程计
NMS	Non-Maximum Suppression	非极大值抑制
LoG	Laplacian of Gaussian	高斯拉普拉斯算子
DoG	Difference of Gaussian	高斯差分算子
HoG	Histogram of Gradient	梯度直方图
SIFT	Scale Invariant Feature Transform	尺度不变特征变换
SURF	Speeded Up Robust Features	加速鲁棒特征
FAST	Features from Accelerated Segment Test	加速分割检测特征
BRIEF	Binary Robust Independent Elementary Features	二值鲁棒独立元素特征
ORB	Oriented FAST and Rotated BRIEF	原生 FAST 和旋转 BRIEF
DLT	Direct Linear Transform	直接线性变换
EPnP	Effective Perspective n-Point	有效透视 n 点法
BA	Bundle Adjustment	光束平差法
MSCKF	Multi-State Constraint Kalman Filter	多约束状态卡尔曼滤波器
VINS	Visual-Inertial Navigation System	视觉惯性导航系统

2.3 问题

如导言部分所述，对于机器人工程师来说，**问题意识**非常重要。为强调这一点，我们在本书中明确定义了每一种算法解决的问题，集中列出于下表，以供读者查阅。

表 2: 问题索引表

领域	问题名	章节	简述
最优化	一般非线性优化	3.1	在等式和不等式约束下，求使目标函数取最小值的优化变量
最优化	等式约束非线性优化	3.1	在等式约束下，求使目标函数取最小值的优化变量
最优化	不等式约束非线性优化	3.1	在不等式约束下，求使目标函数取最小值的优化变量
最优化	无约束非线性优化	3.1	求使目标函数取最小值的优化变量

问题索引表 – 接上页

领域	问题名	章节	简述
最优化	无约束最小二乘	3.1	求使非线性二次型目标函数取最小值的优化变量
最优化	无约束 QP 问题	3.1	求使二次型目标函数取最小值的优化变量
最优化	等式约束 QP 问题	3.1	在等式约束下，求使二次型目标函数取最小值的优化变量
最优化	一般 QP 问题	3.1	在等式和不等式约束下，求使二次型目标函数取最小值的优化变量
最优化	线搜索问题	3.2.3	给定目标函数、起点和更新方向，求使目标函数取最小值的步长
插值	C1 连续插值	4.1	已知一元标量函数采样点序列，求区间内一阶连续可导的插值函数
插值	C2 连续插值	4.1	已知一元标量函数采样点序列，求区间内二阶连续可导的插值函数
机器人学	正运动学求解	6.2	在已知固定参数的情况下，根据关节向量求解末端状态
机器人学	逆运动学求解	6.3	在已知固定参数的情况下，根据关节向量求解末端状态
控制理论	调节控制问题	10.1.3	已知开环系统，设计控制器使输出稳定在参考值，且系统稳定
控制理论	跟踪控制问题	10.1.3	已知开环系统，设计控制器使输出跟随参考值，且系统稳定
控制理论	状态观测问题	11.1.1	已知系统状态方程和输出方程，设计观测器使观测误差收敛
控制理论	线性状态观测问题	11.1.1	已知线性系统状态方程和输出方程，设计观测器使观测误差收敛
控制理论	去噪状态估计 (滤波) 问题 [离散]	11.1.2	已知离散系统状态方程和输出方程，设计滤波器，使状态估计误差小
控制理论	最优滤波问题 [离散]	11.1.2	已知离散系统状态方程和输出方程，设计滤波器，使状态估计误差小
控制理论	去噪状态估计 (滤波) 问题 [连续]	11.1.2	已知连续系统状态方程和输出方程，设计滤波器，使状态估计误差小
控制理论	最优滤波问题 [连续]	11.1.2	已知系统状态方程和输出方程，设计滤波器，使状态估计误差小
控制理论	线性滤波问题 [离散]	11.3.1	已知离散线性系统状态方程和输出方程，设计滤波器，使状态估计误差小
控制理论	线性最优滤波问题 [离散]	11.3.1	已知离散线性系统状态和输出方程，设计滤波器，使状态估计方差最小

问题索引表 – 接上页

领域	问题名	章节	简述
控制理论	线性滤波问题 [连续]	11.3.1	已知连续线性系统状态方程和输出方程，设计滤波器，使状态估计误差小
控制理论	线性最优滤波问题 [连续]	11.3.1	已知连续线性系统状态和输出方程，设计滤波器，使状态估计方差最小
最优控制	(无约束) 一般最优调节 [离散]	12.1	已知离散系统、代价函数，求最优的调节控制输入
最优控制	(无约束) 一般最优跟踪 [离散]	12.1	已知离散系统、代价函数、参考轨迹，求最优的跟踪控制输入
最优控制	(无约束) 二次型最优调节 [离散]	12.1	已知离散系统，求使二次型代价最小的调节控制输入
最优控制	(无约束) 二次型最优跟踪 [离散]	12.1	已知离散系统，求使二次型代价最小的跟踪控制输入
最优控制	(无约束) 一般最优调节 [连续]	12.3	已知连续系统、代价函数，求最优的调节控制输入
最优控制	(无约束) 一般最优跟踪 [连续]	12.3	已知连续系统、代价函数、期望轨迹，求最优的跟踪控制输入
最优控制	LQR 调节控制 [离散]	12.4.1	已知线性离散系统，求使二次型代价最小的调节控制输入
最优控制	无限时间 LQR 调节控制 [离散]	12.4.1	已知线性离散系统，求使无限时间二次型代价最小的控制输入
最优控制	LQR 调节控制 [连续]	12.5	已知连续线性系统，求使二次型代价最小的调节控制输入
最优控制	LQR 跟踪控制 [离散]	12.7.1	已知线性离散系统，求使二次型代价最小的跟踪控制输入
最优控制	LQR 平滑跟踪控制 [离散]	12.7.2	已知线性离散系统，求使二次型代价最小的平滑跟踪控制输入
最优控制	等式约束最优调节 [连续]	12.8	已知连续系统、代价函数，求等式约束下最优的调节控制输入
最优控制	微分约束最优调节 [连续]	12.8	已知连续系统、代价函数，求微分约束下最优的调节控制输入
最优控制	积分约束最优调节 [连续]	12.8	已知连续系统、代价函数，求积分约束下最优的调节控制输入
最优控制	有约束 LQR 调节控制 [离散]	13.3	已知线性离散系统和约束，求使二次代价最小的调节控制输入
机器人控制	关节空间调节控制	14.1	已知机器人参数，设计控制器使机器人关节角稳定在参考值
机器人控制	关节空间跟踪控制	14.1	已知机器人参数，设计控制器使机器人关节角跟随参考值变化

问题索引表 – 接上页

领域	问题名	章节	简述
机器人控制	操作空间调节控制	14.1	已知机器人参数，设计控制器使机器人末端稳定在参考值
机器人控制	操作空间跟踪控制	14.1	已知机器人参数，设计控制器使机器人末端位姿跟随参考值变化
机器人控制	关节空间阻抗控制	14.2	已知机器人参数和状态，设计控制器使机器人关节呈现阻抗特性
机器人控制	操作空间阻抗控制	14.2	已知机器人参数和状态，设计控制器使机器人末端呈现阻抗特性
机器人控制	操作空间零力控制	14.2	已知机器人参数和状态，设计控制器使机器人末端呈现零力特性
机器人控制	关节空间导纳控制	14.2	已知机器人参数和外力矩，设计轨迹使机器人关节呈现阻抗特性
机器人控制	操作空间导纳控制	14.2	已知机器人参数和外力，设计轨迹使机器人末端呈现阻抗特性
机器人控制	约束力控制	14.3	已知机器人参数，设计控制器使机器满足约束且跟踪给定力和速度轨迹
机器人控制	关节抗扰调节控制	15.1.4	已知电机参数，设计控制器使关节角稳定在参考值，且抑制干扰
机器人控制	关节抗扰跟踪控制	15.1.4	已知电机参数，设计控制器使关节角跟随参考值变化，且抑制干扰
深度学习	回归问题	18.1.1	已知样本集、标签集，求最优变换以拟合样本到标签映射
深度学习	分类问题	18.1.1	已知样本集、标签集，求最优变换以拟合样本到分类标签映射
深度学习	降维问题	18.1.1	已知数据集，求编解码变换，最小化重构误差
深度学习	聚类问题	18.1.1	已知数据集，求类内差异小、类间差异大的聚类划分方法
深度学习	MLP 梯度求解	18.3.2	已知 MLP 损失函数、输入、参数，求解各参数梯度
深度学习	MLP 参数学习	18.3.2	已知数据集、MLP 损失函数，求解最佳的参数
强化学习	MRP 价值估计 [依赖模型]	20.3	MRP 已知环境模型，求各状态的状态价值
强化学习	MRP 价值估计 [无模型]	20.3	MRP 未知环境模型，求各状态的状态价值
强化学习	MDP 预测问题 [依赖模型]	20.5	MDP 已知环境模型和策略，求各状态的状态价值
强化学习	MDP 预测问题 [无模型]	20.5	MDP 未知环境模型，求某策略下各状态的状态价值

问题索引表 – 接上页

领域	问题名	章节	简述
强化学习	MDP 控制问题 [依赖模型]	20.6	MDP 已知环境模型，求最优策略
强化学习	MDP 控制问题 [无模型]	20.7	MDP 未知环境模型，求最优策略
强化学习	POMDP 控制问题 [无模型]	20.7	POMDP 部分可观测，未知环境模型，求最优策略
强化学习	MDP 控制问题 [无模型参数化]	22.1	MDP 未知环境模型，求最优参数化策略
视觉 SLAM	单目视觉里程计	24.1.1	已知图像序列，求机器人位姿序列
视觉 SLAM	双目视觉里程计	24.1.2	已知双目基线、双目图像序列，求机器人位姿序列
视觉 SLAM	RGBD 视觉里程计	24.1.2	已知光度和深度图像序列，求机器人位姿序列
视觉 SLAM	(单目) 视觉惯性里程计	24.1.2	已知图像序列、IMU 测量序列，求机器人位姿序列
视觉 SLAM	特征点检测问题	25.2	已知单帧图像，求图像中有代表性点的像素坐标
视觉 SLAM	特征点匹配问题	25.2	已知两帧图像及其特征点，求特征点匹配关系
视觉 SLAM	光流估计问题	25.6	已知两帧图像，求离散光流
视觉 SLAM	2D-2D 点对位姿求解	26.1	已知两个位姿的匹配点像素坐标，求位姿间齐次矩阵
视觉 SLAM	3D-3D 点对位姿求解	26.1	已知两个位姿的匹配点相机系坐标，求位姿间齐次矩阵
视觉 SLAM	3D-2D 点对位姿求解	26.1	已知一组匹配点的 3D 坐标和某位姿下像素坐标，求位姿间齐次矩阵
视觉 SLAM	三角测量问题	26.2.2	已知两位姿间齐次矩阵、一组匹配点的像素坐标，求 3D 坐标
视觉 SLAM	2D-2D 点求解单应矩阵	26.2.5	已知两个位姿的多对共面点像素坐标，求单应矩阵
视觉 SLAM	2D-2D 点求解 3 维速度	26.2.6	已知两个位姿的匹配点像素坐标，求 3 维速度和角速度

2.4 算法

本书中介绍了大量机器人相关的算法。我们除了给出算法流程外，还通过算法要素表提炼算法的关键要素 (如算法针对的问题、输入、输出等)。为方便读者查找，我们将本书中介绍的所有算法列在这里。

本书中的大部分算法都在正文中给出了 python 代码实现 (核心部分)。表中的 ∘ 符号就表示该算法已给出 python 代码；† 符号表示对应 python 代码已通过运行测试。可运行的完整版代码已在 Github 开源：`https://github.com/weixr18/MT4R`，欢迎读者对照查阅。

表 3: 算法索引表

算法	编号	任务	解类型
梯度下降法 †	1	无约束非线性优化	迭代解
牛顿迭代法 †	2	无约束非线性优化	迭代解
黄金分割法 †	3	线搜索问题	迭代解
线搜索梯度下降法 †	4	无约束非线性优化	迭代解
阻尼牛顿法 †	5	无约束非线性优化	迭代解
F-R 共轭梯度法 †	6	无约束非线性优化	迭代解
拉格朗日乘子法 †	7	等式约束非线性优化	迭代解
障碍函数 LSGD†	8	不等式约束非线性优化	迭代解
无约束 QP 解析解 †	9	无约束 QP 问题	解析解
等式约束 QP 解析解 †	10	等式约束 QP 问题	解析解
QP 障碍函数法 †	11	一般 QP 问题	迭代解
梯度下降法 †	12	无约束最小二乘	迭代解
高斯-牛顿法 †	13	无约束最小二乘	迭代解
L-M 法 †	14	无约束最小二乘	迭代解
二次样条插值 †	15	C1 连续插值	解析解
三次样条插值 †	16	C2 连续插值	解析解
正运动学解析解 ◦	17	正运动学求解	解析解
分析雅可比解析解 ◦	18	分析雅可比求解	解析解
质心雅可比解析解 ◦	19	质心雅可比求解	解析解
梯度下降法 IK◦	20	逆运动学求解	迭代解
高斯-牛顿法 IK◦	21	逆运动学求解	迭代解
阻尼最小二乘法 IK◦	22	逆运动学求解	迭代解
动力学解析解	23	动力学项求解	解析解
离散卡尔曼滤波 †	24	线性最优滤波问题 [离散]	滤波器，递推
连续卡尔曼滤波	25	线性最优滤波问题 [连续]	滤波器，递推
扩展卡尔曼滤波 †	26	最优滤波问题 [离散]	滤波器，递推
误差状态卡尔曼滤波 †	27	最优滤波问题 [离散]	滤波器，递推
无迹卡尔曼滤波 ◦	28	最优滤波问题 [离散]	滤波器，递推
最优控制-动态规划求解	29	(无约束) 一般最优调节 [离散]	model-based, DP, 解析推导
离散 LQR 迭代求解 ◦	30	LQR 调节控制 [离散]	model-based, DP, 解析解
连续 LQR 解析解	31	LQR 调节控制 [连续]	model-based, DP, 解析解
非线性最优控制-DDP	32	(无约束) 一般最优调节 [离散]	model-based, DP, 迭代解
非线性最优控制-iLQR	33	(无约束) 一般最优调节 [离散]	model-based, DP, 迭代解
LQR 跟踪控制 ◦	34	LQR 跟踪控制 [离散]	model-based, DP, 迭代解
LQR 输入增量控制	35	LQR 平滑跟踪控制 [离散]	model-based, DP, 解析解
无约束线性 MPC	36	LQR 调节控制 [离散]	model-based, MPC, 解析解
不等式约束线性 MPC	37	有约束 LQR 调节控制 [离散]	model-based, MPC, 迭代解

算法索引表 – 接上页

算法	编号	任务	解类型
独立关节 PI 控制	38	关节抗扰调节控制	model-based, 解析解
独立关节 PID 控制	39	关节抗扰跟踪控制	model-based, 解析解
JS 重力补偿 PD 控制	40	关节空间调节控制	model-based, 解析解
JS 逆动力学 PD 控制	41	关节空间跟踪控制	model-based, 解析解
OS 重力补偿 PD 控制	42	操作空间调节控制	model-based, 解析解
OS 逆动力学 PD 控制	43	操作空间跟踪控制	model-based, 解析解
OS 阻抗控制	44	操作空间阻抗控制	model-based, 解析解
OS 零力控制	45	操作空间零力控制	model-based, 解析解
MLP 前向传播 †	46	MLP 推理	解析解
MLP 反向传播 †	47	MLP 梯度求解	解析解
GD 优化 †	48	MLP 参数学习	迭代解
SGD 优化 †	49	MLP 参数学习	迭代解
BSGD 优化 †	50	MLP 参数学习	迭代解
学习率衰减 BSGD 优化 †	51	MLP 参数学习	迭代解
RMSProp 优化 †	52	MLP 参数学习	迭代解
动量 BSGD 优化 †	53	MLP 参数学习	迭代解
Adam 优化 †	54	MLP 参数学习	迭代解
MLP-BN 前向传播 †	55	MLP 推理	解析解
MLP-BN 反向传播 †	56	MLP 梯度求解	解析解
MRP-价值解析解	57	MRP 价值估计 [依赖模型]	model-based, 表格型
MRP-DP 价值估计	58	MRP 价值估计 [依赖模型]	model-based, 表格型, 自举
MRP-MC 价值估计	59	MRP 价值估计 [无模型]	model-free, 表格型, MC
MDP-DP 策略评估 ○	60	MDP 预测问题 [依赖模型]	model-based, 表格型, 自举
MDP-MC 策略评估 ○	61	MDP 预测问题 [无模型]	model-free, 表格型, MC
MDP-TD(0) 策略评估 ○	62	MDP 预测问题 [无模型]	model-free, 表格型, TD
MDP-TD(n) 策略评估 ○	63	MDP 预测问题 [无模型]	model-free, 表格型, TD
MDP-DP 策略迭代 ○	64	MDP 控制问题 [依赖模型]	model-based, 表格型, 自举
MDP-确定性价值迭代 ○	65	MDP 控制问题 [依赖模型]	model-based, 表格型, 自举
MDP-MC 策略迭代 ○	66	MDP 控制问题 [无模型]	value-based, on-policy?, 表格型
MDP-SARSA 算法 ○	67	MDP 控制问题 [无模型]	value-based, on-policy, 表格型
MDP-SARSA(n) 算法 ○	68	MDP 控制问题 [无模型]	value-based, on-policy, 表格型
MDP-Q 学习算法 ○	69	MDP 控制问题 [无模型]	value-based, off-policy, 表格型
原始 REINFORCE○	70	MDP 控制问题 [无模型]	policy-based, on-policy, 策略梯度
REINFORCE 算法 ○	71	MDP 控制问题 [无模型]	policy-based, on-policy, 策略梯度
Actor-Critic 算法	72	MDP 控制问题 [无模型]	policy-based, on-policy, 策略梯度
Actor-Critic 算法-GAE	73	MDP 控制问题 [无模型]	policy-based, on-policy, 策略梯度
原始 PPO 算法	74	MDP 控制问题 [无模型]	policy-based, on-policy, 策略梯度
PPO-Penalty 算法	75	MDP 控制问题 [无模型]	policy-based, on-policy, 策略梯度

算法索引表 – 接上页

算法	编号	任务	解类型
PPO-Clip 算法	76	MDP 控制问题 [无模型]	policy-based, on-policy, 策略梯度
Harris 角点检测	78	特征点检测问题	传统 CV
SIFT 角点检测	80	特征点检测问题	传统 CV
光流特征点匹配	81	特征点匹配问题	传统 CV
L-K 光流	82	光流估计问题	传统 CV
迭代 L-K 光流	83	光流估计问题	传统 CV
金字塔 L-K 光流	84	光流估计问题	传统 CV
三角测量 †	85	三角测量问题	近似解
本质矩阵恢复位姿 †	86	2D-2D 点位姿求解	近似解
8 点法位姿求解 †	87	2D-2D 点位姿求解	近似解
4 点法求解单应矩阵 ∘	88	2D-2D 点求解单应矩阵	近似解
6 点法求解三维速度 †	89	2D-2D 点求解三维速度	近似解
ICP 算法 †	90	3D-3D 点位姿求解	近似解
DLT 算法 †	91	3D-2D 点位姿求解	近似解
EPnP 算法 †	92	3D-2D 点位姿求解	近似解
GN-BA 位姿求解 †	93	3D-2D 点位姿求解	迭代解

Part I

数学物理基础

1 三维坐标系

1.1 三维空间

1.1.1 矢量与坐标

我们生活在一个三维世界中。为了清楚地表达三维世界的空间关系，我们需要建立三维笛卡尔坐标系。

取 3 个两两垂直的单位矢量 $\vec{\mathbf{o_1}}, \vec{\mathbf{o_2}}, \vec{\mathbf{o_3}}$，作为笛卡尔坐标系的坐标基，这一组坐标基张成整个三维空间。这些坐标基矢量的指向，习惯上也分别称为 x 轴、y 轴、z 轴。

三维空间存在手性问题。按照惯例，**本书中的所有三维坐标系均为右手系**。即：右手手掌伸直，四指指向为 x 轴，垂直掌心向外的指向为 y 轴；右手四指握拳，大拇指保持竖起，此时大拇指方向为 z 轴。

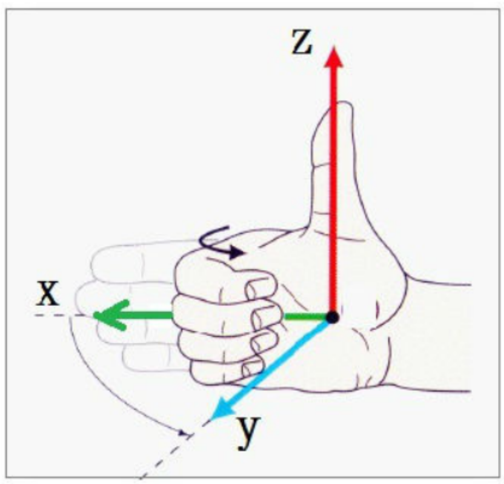

图 I.1.1: 右手坐标系

矢量是有大小和方向的量。在三维空间中，任何两个空间点的有向连线也可以成为矢量。在物理学中，许多矢量有自身的物理意义。一些矢量可以任意平移而不改变物理意义，比如速度、加速度、动量等；一些矢量可以沿轴滑动而不改变物理意义，比如刚体上的力和力矩；一些矢量不能移动，例如位矢。本章中，我们讨论的主要是坐标系中的抽象矢量，或称为三维向量。在本章中，所有矢量都会用箭头符号表示，以强调其和标量的区别。

三维空间中的任何矢量 $\mathbf{a}$ 可以正交分解到一个坐标系 o 的一组坐标基上：

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

也就是说，任意一个矢量可以写为坐标基的线性组合。在给定坐标基的情况下，这组线性组合的系数可以用来代表这个坐标系中的这个向量。我们称之为**向量的坐标**，记作

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

于是，我们有矢量、坐标以及坐标基之间的关系

$$\vec{\mathbf{a}} = [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}] \begin{pmatrix} a_1^o \\ a_2^o \\ a_3^o \end{pmatrix} = [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\mathbf{a}^o \tag{I.1.3}$$

这里请读者注意：矢量并不依赖于坐标而存在，一个矢量的一组坐标一定对应一组特定的坐标基。在后续表示中，为表述方便，在不引起歧义的情况下，我们可能会用矢量的坐标代替矢量。在这种情况下，我们有时称这种默认坐标表示的矢量为**向量**。

如前所示，我们将坐标系写作矢量坐标的右上角标。取一组坐标基 $\mathbf{o_1}, \mathbf{o_2}, \mathbf{o_3}$ 为**默认坐标基**，坐标系称为 **o 系**。我们在这里约定：如果是在默认坐标系中表示矢量，则坐标可以省略右上角标。

1.1.2 矢量与叉乘

三维空间中存在矢量叉乘。三维矢量叉乘的结果是另一个三维矢量，且与坐标系选取无关。叉乘后矢量的大小是两个矢量张成平行四边形的面积，方向同时垂直于两个输入矢量，按照叉乘先后顺序用右手定则判定，如图I.1.2所示。

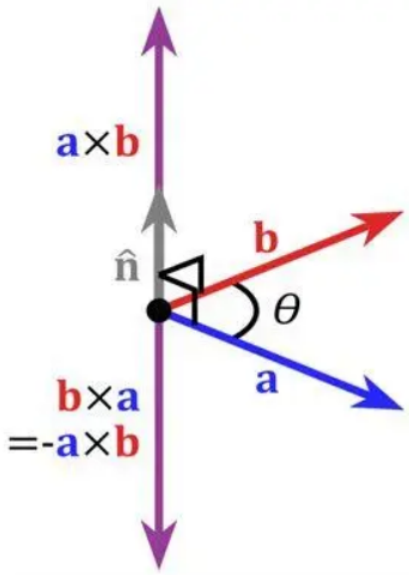

图 I.1.2: 矢量叉乘

在同一坐标系中，矢量叉乘时有如下坐标关系

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

我们将向量 $\mathbf{a}$ 张成的 3×3 矩阵定义为 $\mathbf{a}$ 的**反对称阵**，即

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

因此，三维向量的叉乘就可以写为矩阵乘法形式

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

1.2 旋转矩阵

1.2.1 旋转的两种表述

旋转是相对的运动。想象一位正在舞台上旋转的芭蕾舞者。假设舞台静止，固联在舞台上的坐标系为 n 系；固联在芭蕾舞者身体上的坐标系为 b 系。适当的选取一个空间中的点，同时作为两个坐标系的原点（避免平移的麻烦）。两个坐标系中的”上”的定义是一致的 (重力的反方向)。

假如我们作为观众，在舞台上方的观众席看舞者正在逆时针旋转，这时我们可以讲：b 系正在以一定的角速度相对 n 系旋转，且角速度指向上方。而在芭蕾舞者的视角，她看到的整个世界都在相对她的身体而旋转。也就是说，在 b 系看来，n 系正在以一定的角速度相对 b 系旋转，且角速度指向下。**对同一个旋转，至少存在两个不同的视角对其进行观察**。这也符合我们对于相对运动的直觉认知。

现在，我们以观众视角进行观察。假设我们截取一段舞者的旋转，我们可以如何表述这种旋转呢？我们有两种方法。

- **向量旋转**：固联在芭蕾舞者身体上的每个向量，都相对 n 系进行了一定的旋转。
- **坐标系旋转**：坐标系 b 相对坐标系 n 进行了一定的旋转。

需要注意的是，这两种表示方法的区别仅仅在使用的语言不同。它们所描述的实质都是三维世界的同一个旋转运动。事实上，**对同一个旋转运动，我们永远可以同时使用两种表述方法进行描述**。

那么，这两种表述有什么联系呢？

在**向量旋转**表述下，取和芭蕾舞者身体固联的某个向量 $\mathbf{a}$，其在 n 系下的坐标为 $\mathbf{a}^n$。假设经过旋转，$\mathbf{a}$ 转到了 $\mathbf{a}'$ 的位置。$\mathbf{a}'$ 在 n 系下的坐标为 $\mathbf{a}^{'n}$。对于这一旋转，对任意的舞者固联向量 $\mathbf{a}$，都存在一个矩阵 R_v，满足

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

在**坐标系旋转**表述下，取和舞台固联的某个向量 $\mathbf{c}$，其在 n 系下的坐标为 $\mathbf{c}^n$。假设旋转开始时，b 系和 n 系重合。经过这一旋转，b 系和 n 系不再重合。此时向量 $\mathbf{c}$ 在 b 系下的坐标为 $\mathbf{c}^b$。对于这一旋转，对任意的舞台固联向量 $\mathbf{c}$，都存在一个矩阵 R_n^b，满足

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

在实际使用中，这两种表述中的 R 矩阵都被称为**旋转矩阵**。依据不同的需求，我们会灵活切换两种不同的表述。不过，**在大多数时候，我们会使用坐标系旋转表述**。事实上，这两个矩阵 R_v 和 R_n^b 存在一定的关系

$$R_v^T = R_n^b$$

从直觉上，这也很好理解。舞者相对舞台的旋转，和舞台相对于舞者的旋转，应该是“大小相等，方向相反”的。当使用满足正交性的旋转矩阵表示时，二者就呈现互相转置/互为逆矩阵的关系。

接下来，我们使用坐标系旋转表述，正式地定义旋转矩阵。

1.2.2 旋转矩阵定义

在三维空间中，我们可以定义不同的坐标系。如果两个坐标系的原点重合，那么这两个坐标系之间就是三维旋转关系，即一个坐标系可以通过一定的旋转步骤，变为另一个坐标系。

为了描述不同坐标系之间的旋转关系，我们可以写出两组坐标基两两内积组成的矩阵。假设除 O 系的坐标基外，我们还有一组坐标基 $\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \vec{\mathbf{e_3}}$，它们构成另一个坐标系 E 系。我们如下定义**从 O 系到 E 系的旋转矩阵** ${}^{e}_{o}R$：

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

这个定义有几个关键点。首先，所有的旋转矩阵都是从某系到某系的旋转矩阵，必须有两个坐标系，并且两个坐标系有先后顺序。其次，为了明确表示这两个坐标系，我们将其写在矩阵的左侧，并且原坐标系在下，新坐标系在上。

由此，这个矩阵可以用于处理**同一个矢量从 O 系到 E 系的坐标变换**：

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

即

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

一方面，对于同一个矢量，旋转矩阵可以将一个坐标系中的坐标变为另一个坐标系中的坐标。另一方面，旋转矩阵还可以将一组坐标基变成另一组坐标基

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

事实上，根据上面的定义，旋转矩阵 ${}^{e}_{o}R$ 的各列即为 O 系的坐标基在 E 系中的坐标

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

如果有多个坐标系 e_1, e_2, e_3，那么旋转矩阵可以通过直接相乘的方式进行合成

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

由旋转矩阵的定义，易知任意旋转矩阵 R 存在以下性质

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

我们默认旋转矩阵对应的两个坐标系都是右手系，因此还有

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

在后续章节中，不引起歧义的情况下，我们也将 ${}^e_oR$ 记为 R^e_o。

1.3 齐次矩阵

在三维坐标系中，除了旋转关系，两个坐标系之间还可能存在平移关系。为了表达这种关系，我们定义平移向量

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

平移向量一般在坐标系中表示，如 t^e_{eo} 表示从 E 系原点到 O 系原点的位矢在 E 系下的坐标表示。

通过平移向量 $\vec{t}^e_{eo}$ 和旋转矩阵 R^e_o，我们可以表达任意两个三维空间中的三维坐标系的位置关系；也可以对任意向量做这两个坐标系中的坐标转换。例如，对于任意向量 $\mathbf{a}^o$，有

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

在两个坐标系之间进行平移转换，会用到两个不同的平移向量：t^e_{eo} 和 t^o_{oe}。它们之间的关系还涉及两个坐标系间的旋转矩阵

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

为了使坐标转换的表示更简洁，我们可以定义**齐次矩阵**

$$T^e_o = \begin{bmatrix} R^e_o & t^e_{eo} \\ 0_{1\times3} & 1 \end{bmatrix} \tag{I.1.19}$$

同时，我们也对三维坐标进行增广，**增广坐标**之间的转换是齐次的，表示起来更加简洁

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

当需要在多个坐标系间进行连续的旋转平移混合坐标变换时，使用增广坐标和齐次矩阵连乘非常简单

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

1.4 动坐标系位矢求导

如上节所述，空间中两个坐标系之间的位置关系包括相对平移和相对旋转。同样的，两个坐标系之间的相对运动也分为平移运动和旋转运动。齐次矩阵可以解决相对静止的坐标系间的矢量坐标变换问题，但无法解决动坐标系的位矢求导问题。

具体来说，如前所述，矢量分为不可移动矢量、滑移矢量和可移动矢量。对于不可移动矢量 (主要是位矢)，其在动坐标系中的求导和坐标系之间的运动相关，比可移动矢量更为复杂。

在 19 世纪，科学家科里奥利 (Gaspard Gustave de Coriolis) 研究了这个问题，并总结了一个重要公式。在接下来的讨论中，我们取静坐标系为默认坐标系 o 系，运动坐标系为 e 系，其原点分别为 O 和 E。

假设空间中存在不依赖坐标系的点 P。其在 o 系和 e 系中的 (相对原点的) 位矢记为

$$\overrightarrow{OP} = [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\mathbf{p}^o$$
$$\overrightarrow{EP} = [\vec{\mathbf{e}_1}, \vec{\mathbf{e}_2}, \vec{\mathbf{e}_3}]\mathbf{p}^e$$

注意：在此处的记号中，$\mathbf{p}^o$ 和 $\mathbf{p}^e$ 是**不同的空间矢量在不同坐标系中的投影**，而非同一空间矢量在不同坐标系中的投影。因此，它们不满足旋转坐标变换 (式I.1.10) 和齐次坐标变换。

此外，我们记两个坐标系间的平移矢量为

$$\overrightarrow{OE} = [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\mathbf{t}_{oe}^o$$

对三维空间点，我们可以写出三角矢量式

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

使用坐标基和坐标进行表达，即

$$[\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\mathbf{p}^o = [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\mathbf{t}_{oe}^o + [\vec{\mathbf{e}_1}, \vec{\mathbf{e}_2}, \vec{\mathbf{e}_3}]\mathbf{p}^e$$

现在，假设 e 系对 o 系有相对平移和相对旋转。相对平移的瞬时速度为 $\mathbf{v}$，相对旋转的瞬时角速度为 ω。这样，我们有两个基本的关系式。

$$\begin{aligned} \frac{d}{dt}\mathbf{t}_{oe}^o &= \mathbf{v}_{oe}^o \\ \frac{d}{dt}\vec{\mathbf{e}_\mathbf{i}} &= \vec{\omega} \times \vec{\mathbf{e}_\mathbf{i}} \end{aligned} \tag{I.1.22}$$

将上面三角矢量式的展开形式对时间进行求导，有：

$$[\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\frac{d}{dt}\mathbf{p}^o = [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\frac{d}{dt}\mathbf{t}_{oe}^o + \frac{d}{dt}([\vec{\mathbf{e}_1}, \vec{\mathbf{e}_2}, \vec{\mathbf{e}_3}]\mathbf{p}^e)$$

根据求导法则，我们可以对后一项进行展开

$$[\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\frac{d}{dt}\mathbf{p}^o = [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\frac{d}{dt}\mathbf{t}_{oe}^o + [\vec{\mathbf{e}_1}, \vec{\mathbf{e}_2}, \vec{\mathbf{e}_3}]\frac{d}{dt}\mathbf{p}^e + \frac{d}{dt}([\vec{\mathbf{e}_1}, \vec{\mathbf{e}_2}, \vec{\mathbf{e}_3}])\mathbf{p}^e$$

代入基本关系式，即

$$[\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\frac{d}{dt}\mathbf{p}^o = [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\frac{d}{dt}\mathbf{t}_{oe}^o + [\vec{\mathbf{e}_1}, \vec{\mathbf{e}_2}, \vec{\mathbf{e}_3}]\frac{d}{dt}\mathbf{p}^e + \vec{\omega} \times ([\vec{\mathbf{e}_1}, \vec{\mathbf{e}_2}, \vec{\mathbf{e}_3}])\mathbf{p}^e$$

由坐标基变换关系

$$[\vec{\mathbf{e}_1}, \vec{\mathbf{e}_2}, \vec{\mathbf{e}_3}] = [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\,{}_e^oR$$

我们有

$$[\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\frac{d}{dt}\mathbf{p}^o = [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\frac{d}{dt}\mathbf{t}_{oe}^o + [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\,{}_e^oR\frac{d}{dt}\mathbf{p}^e + \vec{\omega} \times [\vec{\mathbf{o}_1}, \vec{\mathbf{o}_2}, \vec{\mathbf{o}_3}]\,{}_e^oR\mathbf{p}^e$$

消去坐标基，我们有

$$\frac{d}{dt}\mathbf{p}^o = \frac{d}{dt}\mathbf{t}_{oe}^o + {}_e^oR\frac{d}{dt}\mathbf{p}^e + \omega_{oe}^o \times {}_e^oR\mathbf{p}^e \tag{I.1.23}$$

这就是**位矢的一阶变率关系**。

在此基础上，在相对角速度 ω 恒定的条件下，我们还能得到**位矢的二阶变率关系**

$$\frac{d^2}{dt^2}\mathbf{p}^o = \frac{d^2}{dt^2}\mathbf{t}_{oe}^o + {}_e^oR\frac{d^2}{dt^2}\mathbf{p}^e + 2\omega_{oe}^o \times {}_e^oR\frac{d}{dt}\mathbf{p}^e + \omega_{oe}^o \times (\omega_{oe}^o \times {}_e^oR\mathbf{p}^e) \tag{I.1.24}$$

在二阶变率关系式中，等号右侧的第一项称为**牵连加速度**，是坐标系相对变速运动造成的变率；第二项称为**相对加速度**，是绝对变率的投影；后两项称为**哥氏加速度**，是相对旋转造成的加速度。

(参考资料：《现代导航技术》)

2 旋转的描述

上一章中，我们简单介绍了基于坐标系的旋转和平移的数学表示。对于机器人学，这些表示非常重要，因为机器人的运动是三维空间中的运动。为了准确高效地对机器人运动进行描述，我们一般在机器人的各类活动部件上固联三维坐标系。这样，我们就可以使用上一章中的数学表述，来精确描述机器人的运动状态。

对于三维空间中的平移运动，它包括 3 个自由度，我们也使用 3 维平移向量进行描述。对于三维空间中的旋转运动，它也包括 3 个自由度，但在上一章中，我们介绍的旋转矩阵使用了 9 个元素进行描述。这 9 个元素需要满足正交矩阵的额外约束，使得这种方法在表达上不够简洁，运算上也不够直观。因此，我们需要引入其他的表示坐标系旋转的方法。

本章中，我们会介绍**角轴** (**旋转向量**)、**欧拉角**和**四元数**三种更简洁的旋转表示方法，并介绍它们和旋转矩阵之间的相互转换关系。

2.1 角-轴表示法

任何一个三维旋转都可以写成围绕某个轴旋转一定角度的形式，这种表述旋转的方法称为**角-轴表示法** (组合起来也称为**旋转矢量**)。在角-轴表示法中，我们使用一个 3 维**单位矢量** $\mathbf{n}$ 描述旋转轴，标量 θ 代表旋转的角度。

下面，我们介绍角轴表示法和旋转矩阵表示法之间的关系。

2.1.1 角轴转旋转矩阵

角-轴表示法可以转换为旋转矩阵。下面我们考虑：在**向量旋转表述**下，用旋转矩阵 R 表示角轴旋转 $\mathbf{n}, \theta$。

考虑任意非 0 三维向量 a，其总有唯一的分解方法，将其分解为平行于 $\mathbf{n}$ 的 $a_{\parallel}$ 和垂直于 $\mathbf{n}$ 的 $a_{\perp}$。

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

其中

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

考虑 $\mathbf{n}, \theta$ 代表的旋转对二者的作用：旋转后的 $a_{\parallel}$ 仍为 $a_{\parallel}$。而旋转后的 $a_{\perp}$ 满足两个性质：垂直于 $a_{\parallel}$，且与 $a_{\perp}$ 夹角为 θ(以 $\mathbf{n}$ 为夹角正方向)。我们记这个向量为 $a'_{\perp}$。有

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

如何数学表达 $a'_{\perp}$ 满足的性质呢？假设 $a_{\perp}$ 不为 0，让我们建立一个新的坐标系，其 x, y, z 轴方向分别为 $a_{\perp}, \mathbf{n}, a \times \mathbf{n}$ 的方向。写出各轴的单位向量的表达式

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

显然，$a'_{\perp}$ 在 $x - z$ 平面内。根据右手系，$a'_{\perp}$ 的方向满足

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

代入上述表达式，即

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

由于

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

有

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

因此

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

即

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

这个公式叫做**罗德里格斯公式**。

在**坐标系旋转表述**下，假设从 n 系到 b 系需要绕 $\mathbf{n}$ 旋转 θ 角，则有

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

2.1.2 旋转矩阵转角轴

注意到，如果对罗德里格斯公式左右两边取迹，则有

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

因此

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

此外，$\mathbf{n}$ 是矩阵 $R(\mathbf{n}, \theta)$ 的特征值 1 对应的特征向量

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

需要注意，由于角轴 $(\mathbf{n}m\theta + 2\pi n)$ 对应的旋转矩阵完全一样，因此角轴到旋转矩阵的映射是满射，一个旋转矩阵可对应多组角轴。上述反变换仅给出了一个满足条件的” 主值”。

2.2 欧拉角

欧拉角是另一种描述三维旋转的方法。

考虑坐标系旋转问题：假设将 n 系沿 x 轴旋转角度 p 得到 bx 系, 将 n 系沿 y 轴旋转角度 r 得到 by 系，将 n 系沿 z 轴旋转角度 y 得到 bz 系。根据式I.2.2，我们写出旋转矩阵 $R_{bx}^n, R_{by}^n, R_{bz}^n$。

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

这里，我们定义：绕 x 轴的旋转角称为俯仰角 (pitch)，符号为 p; 绕 y 轴的旋转角称为横滚角 (roll)，符号为 r; 绕 z 轴的旋转角称为航向角 (yaw)，符号为 y。定义旋转向量和各轴正方向重合时，为各角的正方向。

对于几乎任意的三维旋转，我们都可以将其等效分解为连续的 3 次绕轴旋转。具体来说，我们首先定义一个轴的顺序，如"Z-X-Y"。那么，几乎任意一个从 n 系到 b 系的三维旋转，都可以唯一地等效于“先绕 z 轴转 y 角，再绕 x 轴转 p 角，再绕 y 轴转 r 角”。即

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

这里的 r, p, y 就称为**欧拉角**。

一组欧拉角必须要明确旋转的轴的顺序。正如矩阵乘法没有交换律，同一组欧拉角交换顺序后的旋转是不同的旋转。**在本书中，如无特殊说明，我们默认欧拉角旋转顺序为 ZXY。**

对 ZXY 顺序的欧拉角，其旋转矩阵和欧拉角的关系为

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

在上面的定义中，我们使用了“几乎”。这是因为，任何一种欧拉角顺序都有对应的万向锁问题。所谓万向锁，就是说对于某一种特殊的旋转，欧拉角表达会出现冗余或奇异。对于 ZXY 顺序的欧拉角，万向锁发生在俯仰角为 $90°$ 或 $-90°$ 的情况。

如果想尽量避免万向锁问题，可以采用下面介绍的四元数来表示旋转。

2.3 四元数

四元数包含向量部分和标量部分，目前有两种广泛使用的定义方法，分别是 JPL 四元数 (向量在前) 和 Hamilton 四元数 (向量在后)。Hamilton 四元数在机器人领域的应用相对更加广泛。因此，本书中我们一律默认使用 Hamilton 定义的四元数。

2.3.1 四元数定义

四元数是由一个实部和三个正交虚部组成的数

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

其中 i, j, k 是三个正交的虚数单位。满足：

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

Hamilton 定义的四元数满足右手定则

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

这样的定义，我们称为**四元数的虚数定义**。
有时，我们也将四元数记为

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

这样的记法，我们称为**四元数的向量记法**。在这种记法下，我们可以定义四元数的模长

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

模长为 1 的四元数称为**单位四元数**。
此外，我们还可以将四元数写为标量部分和向量部分

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

这样的记法，我们称为**四元数的标量向量记法**。

2.3.2 四元数基本运算

在标量向量记法下，任意四元数的加法定义为

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

在标量向量记法下，任意四元数的乘法为

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

或

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

在向量记法下，四元数乘法为

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

注意，由于叉乘不满足交换律，**四元数乘法不可交换**。不过，四元数乘法是满足结合律的

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

2.3.3 四元数表示旋转

旋转的四元数表示与旋转的角轴表示有着密切的关系。假设一个旋转的角轴表示为 $\mathbf{n}, \theta$，在 Hamilton 定义下，可以用标量向量记法将其表示为一个 (单位) 四元数

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

注意，单位四元数满足模长为 1

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

考虑旋转的坐标系旋转表述，即坐标系从 n 系旋转到 b 系是绕 $\mathbf{n}$ 旋转了 θ。向量 $\mathbf{a}$ 在 n 系和 b 系下分别表示为 $\mathbf{a}^n$ 和 $\mathbf{a}^b$。此时，我们计算下列表达式

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

根据式I.2.12，我们首先可以计算乘积的实部

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

其次是虚部

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

注意到向量叉乘满足**三重积公式**

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

因此

$$a\times(b\times a) = (a^TaI - aa^T)b = (a\times b)\times a$$

代入上式中，有

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

根据式I.2.2，有

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

因此，我们有基于四元数的坐标系旋转公式

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

记

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

则

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

这就是用四元数进行坐标变换的方法。

2.3.4 四元数转旋转矩阵

四元数和旋转矩阵都可以用于旋转的坐标变换。事实上，四元数 q_b^n 可以直接表示出旋转矩阵 R_b^n。具体来说，由四元数与角轴关系，有

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

因此有

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

以及

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

以及

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

根据罗德里格斯公式，R_b^n 为以上三项之和，因此

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

2.3.5 四元数微分

两个坐标系之间的旋转一般都是连续的，因此四元数表示可以作为时间的函数。其微分和角速度间满足一定的关系。

首先，我们考虑小角度时的四元数近似。当 θ 很小时，有 $\sin\theta \approx \theta$, $\cos\theta \approx 1$，即

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

考虑用四元数 $q_{b'}^{n}$ 表示从 b' 系到 n 系的旋转。现假设在极短时间 δt 内，b' 系又旋转到 b 系，有

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

由小角度近似，$q_b^{b'}$ 可表示为

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

其中 $\delta\theta$ 表示 b 系下**从 b' 系到 b 系旋转的三维小角度**，ω^b 表示 b 系下的旋转角速度，即

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

当 $\delta t \to 0$，有

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

即：b 系角速度应该在关于 q_b^n 的微分方程中**右乘**到估计状态的四元数上.

参考资料：论文《误差状态四元数》

2.4 各类旋转表示的转换

在表示**坐标系之间的旋转**时，以上介绍的几种旋转表示方法可以互相转换。

本节中，我们假设从 n 系可以通过 ZXY 顺序的欧拉角 y,p,r 旋转到 b 系；也可以通过绕 $\mathbf{n}$ 旋转 θ 角转到 b 系；旋转矩阵 $R=R_b^n$；四元数 $q=q_b^n$。

旋转矩阵与角轴

根据罗德里格斯公式，有

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

根据前述推导，还有

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

旋转矩阵与欧拉角

欧拉角转旋转矩阵的公式为

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

具体展开形式为

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

如果想从旋转矩阵转欧拉角，可以利用上述公式。记 r_{ij} 为矩阵 R_b^n 的第 i 行第 j 列的元素。

在非万向锁情况下 ($|r_{32}|<1$)，有

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

在万向锁情况下，我们需要指定 r 和 y 其中之一。在这里我们指定 r 为 0.

如果 $r_{32}=1$，有

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

如果 $r_{32}=-1$，有

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

四元数与角轴

四元数的旋转表示就来源于角轴，因此互相转换非常简单

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

当 $\sin\theta$ 不为 0 时，有

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

四元数与旋转矩阵

四元数转旋转矩阵，上面已经推到过，有

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

如需要将旋转矩阵转为四元数，应考虑将旋转矩阵转为角轴/欧拉角，再将角轴/欧拉角转为四元数。

四元数与欧拉角

根据欧拉角定义与式I.2.21，有

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

其中

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

因此，有

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

如需要将四元数转为欧拉角，应考虑将四元数转为旋转矩阵，再将旋转矩阵转为欧拉角。

2.5 旋转矩阵与求导

2.5.1 旋转矩阵变率

角速度是旋转变化的最佳表示。前文中我们已经推导了四元数微分和角速度的关系。下面，在坐标系旋转表述下，我们推导旋转矩阵的微分和角速度的关系。

仍然使用芭蕾舞者的例子。假设 n 系与舞台固联，b 系与芭蕾舞者固联。芭蕾舞者相对舞台的旋转角速度在 n 系下记为 ω_{nb}^n。从 n 系到 b 系的坐标变换旋转矩阵为 R_n^b。

根据式I.1.12，我们有

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

因此，如果想求 $\dot{R}_n^b$，我们只需求出 $\vec{\mathbf{b}}_i^n$ 的导数。更一般地，我们考虑任取一和 b 系固联的向量 $\mathbf{a}$，求出其在 n 系下坐标的导数 $\dot{\mathbf{a}}^n$. 这样，我们就将坐标系旋转表述下的旋转矩阵求导转化为了向量旋转表述下的旋转矩阵求导。

具体来说，考虑将 $\mathbf{a}$ 分解为平行于 ω_{nb}^n 的 $a_\parallel$ 和垂直于 ω_{nb}^n 的 $a_\perp$

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

ω_{nb} 的旋转产生的线速度仅与 $a_\perp$ 有关。其方向为 $\omega_{nb} \times \mathbf{a}_\perp$ 的方向，其大小为

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

因此，有

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

由于

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

有

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

即：任一与 b 系固联向量的变率都是角速度叉乘该向量

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

因此，有

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

即

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

2.5.2 李群李代数简介

在机器人中，我们会遇到将 (相对) 位置和姿态作为优化变量求解优化问题的情况 (关于优化问题求解，详见第3章)。在这个过程中，我们需要求解优化目标对姿态的导数。

如果我们用旋转矩阵表示旋转，则上述问题基本可以归结为下列导数的求解

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

旋转矩阵是一个 3×3 矩阵。然而，我们不能将其直接作为普通矩阵，运用一般的对矩阵求导法则。这是因为，3 维旋转只有 3 个自由度，为了确保这一点，R 的 9 个元素需要满足 2 个正则化约束 $R^T R = I$ 和 $det(R) = 1$。对普通矩阵的求导法则并不能满足这一点，求出的导数是错误的。

事实上，旋转矩阵 R 并非存在于 9 维的欧几里得空间中，而是处于这个空间的一个**流形**上。流形可以看作一种高维空间中的低维超曲面，这个曲面的表面一般有一定的光滑性[1]。对变量的约束方程就是曲面的解析方程。典型的流形如二维空间中的圆环 (1 维流形)、三维空间中的球面 (2 维流形)。一般来说，流形和对应维度的欧几里得空间并不存在一一对应的关系。

那么，对于流形上的变量，应该如何进行求导呢？事实上，求导的本质是在自变量附近构造一个扰动，这个扰动会通过函数映射关系，带来值空间中的扰动。忽略高阶小项后，函数值的扰动与自变量之间的扰动会形成近似线性的关系，也就是函数的局部线性化。我们不希望扰动发生在在超曲面上，但假如我们可以在某点附近构造流形的局部线性化——也就是高维曲面的**切平面**——就可以**用扰动在切平面上的投影，来代替曲面上的扰动**。

其实，旋转矩阵不仅是一个流形，还因为其一定的对称性组成一个**群**。旋转矩阵组成的群有一个特定的名字：**特殊正交群**，写作 $SO(3)$。$SO(3)$ 既是流形也是群，因此是一种**李群**。作为流形的李群，其" 切平面" 构成一个类似线性空间的结构，我们称之为**李代数**[2]。每一种李群都有一种对应的李代数，反之亦然。

关于流形、群、李群、李代数的严谨定义，读者可参考其他的代数学教材，这里不再赘述。

2.5.3 指数对数映射

每一种李群都有与之对应的李代数。对于旋转矩阵组成的李群 $SO(3)$，其李代数为

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

事实上，这正是我们前面介绍的角轴表示法中角和轴的乘积。我们将其称为**旋转矢量**。

旋转矢量是角速度对时间的积分。假设 b 系相对 n 系匀速旋转。记在 n 系下 b 系的旋转角速度为 $\omega = \omega_{nb}^b$。记旋转矩阵 $R(t) = R_b^n$。旋转矩阵对应的旋转矢量 $\phi = \phi_{nb}^b$。有

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

回顾上一节中，我们得到的 R 满足的微分方程

[1] 非严谨定义

[2] 非严谨定义

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

假设微分方程初值为 $R(0) = I$，即初始时刻 n 系与 b 系重合。根据矩阵微分方程求解方法，我们有

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

即

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

回顾2.1节中推导的**罗德里格斯公式** (式I.2.1)，我们有

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

这一公式揭示了 $SO(3)$ 元素和 $so(3)$ 元素之间的**指数关系**，具体地给出了在已知李代数元素 ϕ 时，求解李群元素 R 的方法[3]。事实上，**对于任意一个李代数中的元素，都可以找到唯一的李群元素与之对应**。由于符合矩阵指数的形式，这一对应关系也称为**指数映射**。式I.2.39即为 $so(3)$ 到 $SO(3)$ 的指数映射。

与之相反，**对数映射**是指，给定一个李群中的元素，求与之对应的李代数的元素。不难发现，对于 $\phi = \mathbf{n}(\theta_0 + k\theta), k \in \mathbb{N}$，其对应的旋转矩阵均为同一个矩阵。也就是说，对数映射存在多解。类似于复变函数中的处理方式，我们在所有满足 $R = \exp((\phi)_\times)$ 的 ϕ 中选出一个值，作为对数映射的**主值**。对于 $SO(3)$ 到 $so(3)$ 对数映射 (主值) 的具体求解方法，在2.1节亦有说明 (即旋转矩阵转角轴)，此处不再赘述。

2.5.4 扰动与求导

上面我们介绍了，对于构成高维空间流形的旋转矩阵李群 $SO(3)$，可以找到李群元素 $R \in SO(3)$ 和切空间李代数元素 $\phi \in so(3)$ 的对应关系。这种关系建立在微小扰动的假设上。那么，我们如何利用这样的关系，从扰动的角度解决“对旋转矩阵求导”的难题呢？

在这里，我们首先要区分**左扰动**和**右扰动**。我们知道，旋转矩阵乘法没有交换性，即在一般情况下，

$$R_1 R_2 \neq R_2 R_1$$

因此，对于待求导的旋转矩阵 $R \in SO(3)$ 以及我们希望施加的扰动 $\Delta R \in SO(3)$，也有

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

我们将 $(\Delta R)R$ 形式的扰动称为**左扰动**，而 $R(\Delta R)$ 形式的扰动称为**右扰动**。考虑其物理意义，我们不难发现，假设 $R = R_b^n$ 表示从 b 系到 n 系的旋转矩阵，则左扰动 $(\Delta R)R$ 表示在 n 系上施加扰动，而右扰动 $R(\Delta R)$ 表示在 b 系上施加扰动。

下面我们先以左扰动为例，考虑**将旋转矩阵 (李群) 的变化量转化为旋转向量 (李代数) 的变化量**。假设对于旋转矩阵 R，其对应的旋转向量为 ϕ，有

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

现在对于旋转矩阵 R 施加左扰动 ΔR，假设 ΔR 对应的旋转向量为 $\Delta\phi$，即

[3]其实，从矩阵指数函数的泰勒展开，也可以推导出上述公式，即给出了罗德里格斯公式的第二种推导方法。《视觉 SLAM 十四讲》中给出了推导过程，此处不再赘述

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

那么，总的旋转为

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

这样，我们就可以用对李代数元素 ϕ 的求导代替对李群元素 R 的求导。根据扰动模型，有

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

对于此处的 $\exp((\Delta\phi)_\times) - I$，由于我们已经假设 $\Delta\phi$ 为小量，可以使用矩阵函数的一阶泰勒展开进行近似，即

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

因此，我们有

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

以上是左扰动情况。对于右扰动，有

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

即

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

对于转置的情况，有

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

即：左扰动转置后等效于右扰动取反。右扰动也同理。因此，我们还有如下结论

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

值得注意的是，这里的李代数扰动 $\Delta\phi$，基本可以理解为第2.3.5小节中的三维小角度 $\delta\theta$。

2.5.5 扰动雅可比

在上述扰动模型中需要注意一点：这里的指数函数是矩阵函数，对矩阵函数而言 $\exp(A)\exp(B) \neq \exp(A+B)$。因此，$(\Delta R)R$ 的旋转向量不能写为 $\exp((\Delta\phi+\phi)_\times)$，而需要进行一个变换

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

即

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

矩阵 $J_l(\phi)$ 称为左乘时的**扰动雅可比**，它是一个 3×3 的矩阵，依赖于原旋转向量 ϕ。其具体计算方式为

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

同理，对于右乘情况，有

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

以及

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

这些公式的证明不再赘述，读者可以参考其他教材。

（参考资料：《视觉 SLAM 十四讲》）

2.6 各类旋转微分的转换

到目前为止，我们已经介绍了 4 类旋转表示的方法：旋转矩阵、四元数、欧拉角、角轴。这四类表示都可以作为随时间变化的变量，从而有其微分量。不过，更一般的情况下，我们会用**角速度**表示旋转的变化。这些旋转表示的微分和角速度之间存在转换关系，我们在这里统一进行总结。

对于角速度的表示，一共有四种可能的表示方式，分别是 n 系和 b 系下 n 系相对 b 系以及 b 系相对 n 系的旋转。我们常用的两种是 ω_{nb}^n 和 $\omega^b = \omega_{nb}^b$。其中，ω^b 是陀螺仪的直接测量量。它们的转换关系如下所示

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

在本节中，我们仍假设从 n 系可以通过 ZXY 顺序的欧拉角 y, p, r 旋转到 b 系，欧拉角的微分记为 $\dot{y}, \dot{p}, \dot{r}$。$n$ 系通过绕轴 $\mathbf{n}$ 旋转 θ 角转到 b 系，旋转向量 $\phi = \theta\mathbf{n}$，满足 $R_b^n = \exp((\phi)_\times)$；其微分为 $\dot{\phi}$。旋转矩阵 R_b^n，其微分为 $\dot{R}_b^n$。四元数 q_b^n，其微分为 $\dot{q}$。

旋转矩阵微分

在2.5.1小节中，我们推导了以下结论

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

四元数微分

在2.3.5小节中，我们推导了以下结论

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

欧拉角微分

欧拉角微分和角速度的关系可以使用角速度相加的思路求解。即：分别考虑每个欧拉角的变化导致的角速度 $\omega_z, \omega_x, \omega_y$，再将其相加，即

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

其中，每个欧拉角导致的角速度就是当时的坐标轴向量和欧拉角微分的乘积

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

由于在 ZXY 顺序下，我们有如下欧拉角和旋转矩阵的关系

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

因此坐标轴向量分别为

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

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

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

代入以上表达式，我们有角速度的表达式

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

其中

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

即

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

旋转向量微分

旋转向量 ϕ 是旋转矩阵构成的李群对应的李代数。微小旋转带来的旋转矩阵变化，可以近似线性表示为旋转向量的变化。在2.5.5小节中，对于右扰动，我们有这样的结论

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

其中

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

进行一阶近似，即

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

假设时间极短，则旋转向量的变化可表示为

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

即

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

因此，当 $\lim_{\delta t \to 0}$，有

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

又由于

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

因此

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

即

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

同理，对于左扰动，有

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

3 最优化方法

在机器人领域中，很多问题都可以转化为优化问题求解，包括机器人逆运动学求解 (见第7章)、MPC(见第13章) 和 vSLAM(见第26章) 等。是一种常见的问题解决手段和问题转化的思路。

相比于专业的最优化教材，我们不会介绍凸优化、凸性、线性规划等基础内容，而只关注机器人领域的基础应用。

3.1 优化问题定义

优化 (规划) 问题，是指在一定的约束条件下，寻找关于某个变量的某个函数最值的问题。这个变量称为**优化变量**，可以是一个向量，我们用 $\mathbf{x}$ 表示。我们关心的这个函数称为**目标函数**，一般记为 $f(\mathbf{x})$，是一个标量值函数。我们一般要求目标函数对于优化变量 $\mathbf{x}$ 连续可导。

优化问题的约束条件既可以是等式条件，也可以是不等式条件。一般写为 $\mathbf{g}(\mathbf{x}) = 0, h(\mathbf{x}) \leq 0$ 的形式。$\mathbf{g}(\mathbf{x})$ 和 $\mathbf{h}(\mathbf{x})$ 一般是向量值函数，即包含多个约束条件。

一般的非线性优化问题定义如下

问题	一般非线性优化
问题简述	在等式和不等式约束下，求使目标函数取最小值的优化变量
已知	目标函数 $f(\mathbf{x})$ 等式约束函数 $\mathbf{g}(\mathbf{x})$ 不等式约束函数 $\mathbf{h}(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$ $s.t. \quad \mathbf{g}(\mathbf{x}) = 0, \mathbf{h}(\mathbf{x}) \leq 0$

对于上述问题，如果去除不等式约束，我们称为**等式约束的非线性优化问题**

问题	等式约束非线性优化
问题简述	在等式约束下，求使目标函数取最小值的优化变量
已知	目标函数 $f(\mathbf{x})$ 等式约束函数 $\mathbf{g}(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$ $s.t. \quad \mathbf{g}(\mathbf{x}) = 0$

相应的，如果去除等式约束，我们称为**不等式约束的非线性优化问题**

问题	不等式约束非线性优化
问题简述	在不等式约束下，求使目标函数取最小值的优化变量
已知	目标函数 $f(\mathbf{x})$ 不等式约束函数 $\mathbf{h}(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$ $s.t. \quad \mathbf{h}(\mathbf{x}) \leq 0$

如果去除全部约束，我们就得到**无约束非线性优化问题**

问题	无约束非线性优化
问题简述	求使目标函数取最小值的优化变量
已知	目标函数 $f(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$

在非线性优化问题中，有一类问题被称为最小二乘问题。它的目标函数是一个 (可微) 向量值函数的二次型 $f(\mathbf{x}) = \mathbf{e}^T(\mathbf{x})H\mathbf{e}(\mathbf{x})$。例如，我们可以定义无约束最小二乘问题

问题	无约束最小二乘
问题简述	求使非线性二次型目标函数取最小值的优化变量
已知	目标函数 $\mathbf{e}^T(\mathbf{x})H\mathbf{e}(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} \mathbf{e}^T(\mathbf{x})H\mathbf{e}(\mathbf{x})$

对于上述几类非线性优化问题，如果函数 $f(\mathbf{x})$ 满足凸性，则称该问题为**凸优化问题**。凸优化问题有很多好的性质，例如从任一一点出发进行累积优化都能达到全局极小值。

最简单的凸优化问题是**二次规划问题** (QP, Quadratic Programming)。QP 问题是指：目标函数是关于优化变量的二次型和线性函数、约束条件均为优化变量的线性函数的优化问题。根据是否有约束，我们可以将 QP 问题分为无约束 QP 问题、等式约束 QP 问题、不等式约束 QP 问题

问题	无约束 QP 问题
问题简述	求使二次型目标函数取最小值的优化变量
已知	半正定矩阵 H, 向量 $\mathbf{g}$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x}$

问题	等式约束 QP 问题
问题简述	在等式约束下，求使二次型目标函数取最小值的优化变量
已知	半正定矩阵 H, 向量 $\mathbf{g}$ 矩阵 M_{eq}, 向量 b_{eq}
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x}$ $s.t.\quad M_{eq}\mathbf{x} = b_{eq}$

问题	一般 QP 问题
问题简述	在等式和不等式约束下，求使二次型目标函数取最小值的优化变量
已知	半正定矩阵 H, 向量 $\mathbf{g}$ 矩阵 M_{eq}, 向量 b_{eq} 矩阵 M, 向量 b
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x}$ $s.t.\quad M_{eq}\mathbf{x} = b_{eq}, M\mathbf{x} \leq b$

3.2 无约束优化

3.2.1 梯度下降法

首先，我们考虑无约束优化问题。我们假设目标函数均可微。对无约束优化问题，最直接的迭代求解方法是沿梯度的反方向迭代求解，即**梯度下降法**。其算法总结如下

算法	梯度下降法
问题类型	无约束非线性优化
已知	可微目标函数 $f(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
算法性质	迭代解

Algorithm 1: 梯度下降法

Input: 可微目标函数 $f(\mathbf{x})$, 初值 $\mathbf{x}_0$

Parameter: 阈值 ϵ, 步长 α

Output: 最优解 $\mathbf{x}^*$

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

while $||\nabla_{\mathbf{x}} f(\mathbf{x}_k)|| > \epsilon$ **do**

  $\mathbf{x}_k \leftarrow \mathbf{x}_k - \alpha \nabla_{\mathbf{x}} f(\mathbf{x}_k)$

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

对应的 python 代码如下所示

```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 牛顿迭代法

梯度下降法需要预先给定两个参数：阈值 ϵ 和步长 α。步长在深度学习社区有时也称为学习率。步长过大或过小都可能导致收敛缓慢。另一方面，对于凸优化问题，梯度下降方法可能由于震荡收敛导致收敛缓慢。

为解决这一问题，我们可以使用二阶的**牛顿法**。具体来说，我们考虑将函数 $f(\mathbf{x})$ 在 $\mathbf{x}_k$ 附近做二阶泰勒展开

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

式中，$H_f(\mathbf{x})$ 是函数 f 对 x 的二阶导数，在 x 为向量、f 为标量值时是一个方阵，称为 **Hessian 矩阵**。

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

我们忽略二阶小项，用其余的二次型作为原目标函数 $f(\mathbf{x})$ 的近似，有

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

考虑在每步优化中，直接取近似目标函数 $\hat{f}(x)$ 的极小值。由于 $\hat{f}(x)$ 是二次型，必为凸函数，在 H_f 正定的条件下，假设

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

有

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

即

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

假设 $H_f(\mathbf{x}_k)$ 可逆，有

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

因此，我们可以总结**牛顿迭代法**如下

算法	**牛顿迭代法**
问题类型	无约束非线性优化
已知	可微目标函数 $f(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
算法性质	迭代解

Algorithm 2: 牛顿迭代法

Input: 二阶可微目标函数 $f(\mathbf{x})$, 初值 $\mathbf{x}_0$
Parameter: 阈值 ϵ
Output: 最优解 $\mathbf{x}^*$

$\mathbf{x}_k \leftarrow \mathbf{x}_0$
while $||\nabla_{\mathbf{x}} f(\mathbf{x}_k)|| > \epsilon$ **do**
  $\mathbf{x}_k \leftarrow \mathbf{x}_k - H_f^{-1}(\mathbf{x}_k)\nabla_{\mathbf{x}} f(\mathbf{x}_k)$
$\mathbf{x}^* \leftarrow \mathbf{x}_k$

对应的 python 代码如下所示

```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
```

相比梯度下降法，牛顿迭代法有更快的收敛速度，但也有更苛刻的要求：目标函数必须二阶可导，并且 Hessian 矩阵必须可逆，还要在每个循环中都进行矩阵求逆的计算。那么，能否在不计算二阶导数的情况下，也获得更快的收敛速度呢？

3.2.3 线搜索方法

一种思路是：将迭代步骤分为两个问题：首先寻找最好的下降方向，其次沿着该方向寻找最好的步长。这样的优化方法我们称为**基于线搜索的优化方法**。

线搜索就是其中第二步的名称，即沿着某条多维空间中的直线寻找最优的函数值。我们简单定义一下线搜索问题

问题	线搜索问题
问题简述	给定目标函数、起点和更新方向，求使目标函数取最小值的步长
已知	目标函数 $f(\mathbf{x})$ 搜索起点 $\mathbf{x}_0 \in \mathbb{R}^n$ 搜索方向 $\mathbf{t} \in \mathbb{R}^n$
求	最优步长 $h^* = \arg\min_h f(\mathbf{x}_0 + h\mathbf{t})$

线搜索最常用的方法是黄金分割法，即在给定搜索区间后，每次搜索都将搜索区间缩短为上一区间的 $\phi = 0.5(\sqrt{5}-1)$ 倍。黄金分割算法的具体实现如下所示

算法	黄金分割法
问题类型	线搜索问题
已知	目标函数 $f(\mathbf{x})$ 搜索起点 $\mathbf{x}_0 \in \mathbb{R}^n$ 搜索方向 $\mathbf{t} \in \mathbb{R}^n$
求	最优步长 $h^* = \arg\min_h f(\mathbf{x}_0 + h\mathbf{t})$
算法性质	迭代解

Algorithm 3: 黄金分割法 (GR_line_search)

Input: 目标函数 $f(\mathbf{x})$

Input: 搜索起点 $\mathbf{x}_0$, 搜索方向 t

Parameter: 初始步长 h_0, 阈值 r

Output: 最优步长 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**
  - $a, b \leftarrow p, b$

$h^* \leftarrow (a + b)/2$

对应的 python 代码如下所示

```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
```

基于上述线搜索方法，我们可以改进梯度下降法。改进的方法称为线搜索梯度下降法 (Line Search Gradient Decent, LSGD)。请读者务必注意，其中的搜索方向应为梯度负方向。

算法	**线搜索梯度下降法**
问题类型	无约束非线性优化
已知	可微目标函数 $f(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
算法性质	迭代解

Algorithm 4: 线搜索梯度下降法 (opt_nc_LSGD)

Input: 可微目标函数 $f(\mathbf{x})$

Parameter: 阈值 ϵ, 初值 $\mathbf{x}_0$

Output: 最优解 $\mathbf{x}^*$

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

while $||\nabla_{\mathbf{x}} f(\mathbf{x}_k)|| > \epsilon$ **do**

  $\mathbf{t} \leftarrow -\nabla_{\mathbf{x}} f(\mathbf{x}_k)$

  $h \leftarrow \text{GR\_line\_search}(f, \mathbf{x}_k, \mathbf{t})$

  $\mathbf{x}_k \leftarrow \mathbf{x}_k + h\mathbf{t}$

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

对应的 python 代码如下所示

```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
```

同理，我们也可以将线搜索和牛顿迭代法结合。改进的牛顿迭代法也称为**阻尼牛顿法**。

算法	**阻尼牛顿法**
问题类型	无约束非线性优化
已知	可微目标函数 $f(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
算法性质	迭代解

Algorithm 5: 阻尼牛顿法

Input: 可微目标函数 $f(\mathbf{x})$

Parameter: 阈值 ϵ, 初值 $\mathbf{x}_0$

Output: 最优解 $\mathbf{x}^*$

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

while $||\nabla_{\mathbf{x}} f(\mathbf{x}_k)|| > \epsilon$ **do**

  $\mathbf{t} \leftarrow -H_f^{-1}(\mathbf{x}_k)\nabla_{\mathbf{x}} f(\mathbf{x}_k)$

  $h \leftarrow \text{GR\_line\_search}(f, \mathbf{x}_k, \mathbf{t})$

  $\mathbf{x}_k \leftarrow \mathbf{x}_k + h\mathbf{t}$

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

对应的 python 代码如下所示

```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　共轭梯度法

共轭梯度来源于 QP 问题，其核心是在迭代搜索的过程中，使上一步迭代的梯度方向和下一步迭代的梯度方向关于 Hessian 矩阵共轭。所谓共轭，就是对于方阵 A，满足 $u^T Av = 0$ 的两个向量 (u, v) 称为共轭向量。研究者发现，相比梯度下降法，共轭梯度法可以在 QP 问题中最快速地收敛，且无需像牛顿法一样需要实际计算 Hessian 矩阵。

共轭梯度法的具体推导略去，详见其他的优化理论教材。我们仅给出其核心的梯度方向搜索公式。使用该公式的共轭梯度法也称为 F-R 共轭梯度法，其中"F-R" 是两位发明者 (Fletcher-Reeves) 的姓氏缩写。

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

共轭梯度法也是一种基于线搜索的优化方法，共轭梯度提供搜索方向，具体步长使用线搜索求解。实际使用中，每经过 n 轮迭代，会重置搜索方向为梯度方向。

F-R 共轭梯度算法总结如下

算法	F-R 共轭梯度法
问题类型	无约束非线性优化
已知	可微目标函数 $f(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
算法性质	迭代解

Algorithm 6: F-R 共轭梯度法

Input: 可微目标函数 $f(\mathbf{x})$, 初值 $\mathbf{x}_0$

Parameter: 阈值 ϵ, 重置轮数 n

Output: 最优解 $\mathbf{x}^*$

$k \leftarrow 0$

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

    $\mathbf{t}_k \leftarrow -\mathbf{g}_k + ||\mathbf{g}_k||/||\mathbf{g}_{k-1}||\mathbf{t}_{k-1}$

  $h \leftarrow \text{GR\_line\_search}(f, \mathbf{x}_k, \mathbf{t}_k)$

  $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + h\mathbf{t}_k$

  $k \leftarrow k + 1$

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

对应的 python 代码如下所示

```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
```

(参考资料：清华《运筹学》课件)

3.3 等式约束优化

3.3.1 Lagrange 条件

上一节中，我们给出了很多求解无约束非线性优化问题的算法。对于具有等式约束的情况，我们应该如何处理呢？一种思路是：考虑最优解需要满足的必要条件，该条件构成关于优化变量的方程，随后对该方程进行求解。

具体来说，考虑约束非线性优化问题可行域中的点 $\mathbf{x}$。若从该点出发，向某方向移动一微小距离后仍在可行域中，称该方向 $\mathbf{p}$ 为**可行方向**。假设目标函数和约束条件均连续可微，那么可以得到这样的直观的必要性结论：**在最优解处，目标函数梯度方向不能是可行方向。**

首先考虑较为简单的情形，即只有等式约束 $\mathbf{g}(\mathbf{x})=0$ 的凸优化问题。可以想象，在决策变量所在的 n 维欧氏空间中，一个等式约束 $g_i(\mathbf{x})=0$ 定义了一个 $n-1$ 维超平面。在任何可行解处，可行方向 $\mathbf{p}$ 在超平面内，也就是可行方向垂直于约束的梯度

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

对于 m 个约束而言，这些约束共同作用，取交集。这样，就要求可行方向垂直于各超平面法向量张成的空间；而超平面的法向量方向，正是约束函数的梯度方向，及

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

之前我们从直觉上推导出了这样的必要性结论：最优解处，目标函数梯度方向不能是可行方向。用数学语言描述，就是 ∇f 必须和可行方向垂直，即

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

结合两个垂直关系，我们就知道，目标函数的梯度 $\nabla_{\mathbf{x}}f(\mathbf{x})$ 一定位于各超平面法向量张成的空间，即

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

上面的结论可以写成等价的较为规范的形式。记 **Lagrange 函数**为

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

则有等式约束非线性优化问题最优解 $\mathbf{x}^\star$ 的必要条件

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

式I.3.8称为 **Lagrange 条件**。系数 λ 称为 **Lagrange 乘子**。将约束最优化问题转换为 Lagrange 条件进行求解的方法称为 **Lagrange 乘子法**。注意：Lagrange 条件不等价于 L 函数的最值。

可以看到，从优化问题的角度来看，Lagrange 乘子法相当于是构造了一个新的无约束优化问题，将原来的优化变量 $\mathbf{x}$ 扩维成了 $[\mathbf{x}^T,\lambda^T]^T$。因此，适用于无约束问题的求解算法也可以用于等式约束的情况。

对于**凸优化问题**，Lagrange 条件是一个最优解存在前提下满足的必要条件。对于非凸情况，在 f 和 g 满足一定的规范性条件的情况下，Lagrange 条件也是最优解必要条件。在此我们不再赘述。

3.3.2 等式约束的优化算法

如上所述，使用 Lagrange 条件对问题进行转换，就可以从无约束优化算法得到针对等式约束优化问题的算法。我们以线搜索梯度下降法为例，其他算法可类推，不再赘述。

记

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

则有

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

根据 Lagrange 条件，我们需要求解 $\nabla_{\bar{\mathbf{x}}} L(\bar{\mathbf{x}}) = 0$。构造新的优化目标

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

我们仍记

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

因此，新优化目标的梯度为

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

可以看到，为了求解等式约束优化问题，我们需要用到 f, g 函数的二阶导。经过这一转换，约束优化问题就转化为了无约束优化问题，可以通过上节介绍的各类算法求解。

我们以线搜索梯度下降作为外层优化算法，总结等式约束优化问题的拉格朗日乘子法如下

Algorithm 7: 拉格朗日乘子法

Input: 二阶可微目标函数 $f(\mathbf{x})$, 初值 $\mathbf{x}_0$
Input: 二阶可微等式约束函数 $\mathbf{g}(\mathbf{x})$
Parameter: 阈值 ϵ, 乘子初值 λ_0
Output: 最优解 $\mathbf{x}^*$

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

算法	拉格朗日乘子法
问题类型	等式约束非线性优化
已知	二阶可微目标函数 $f(\mathbf{x})$ 二阶可微等式约束函数 $\mathbf{g}(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$ $s.t. \quad \mathbf{g}(\mathbf{x}) = 0$
算法性质	迭代解

对应的 python 代码如下所示

```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 不等式约束优化

3.4.1 KKT 条件

相比于等式约束，不等式约束的处理方法略微复杂一些。

首先，并不是每个不等式约束都会对可行解/最优解起作用。对于一个不等式约束 $h_j(\mathbf{x}) \leq 0$，若 $\mathbf{x}$ 的取值使其取等，说明 $\mathbf{x}$ 处于可行域的由约束 $h_j(\mathbf{x}) \leq 0$ 定义的边界上，此时称该约束为**起作用约束**。反之，称其为**不起作用约束**。

我们仍然从最优解可行方向满足的必要条件的角度来思考问题。对于一个可行解 $\mathbf{x}$ 而言，不起作用约束对应的超平面和该解很远，不会影响可行方向的选择。然而，如果约束条件 $h_j(\mathbf{x}) \leq 0$ 是起作用约束，那么可行方向不能是 h_j 减小的方向，但可以是 h_j 增加的方向。

因此，对于含有不等式约束的优化问题3.1，最优解处每一个起作用约束梯度和可行方向的内积都要为正。也就是说，目标函数的梯度需要在起作用约束梯度负方向的线性组合半空间内。结合等式约束，我们可以写出如下的梯度关系：

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

这里的 $c(x)$ 表示：在可行解 $\mathbf{x}$ 处，起作用约束的编号集合。

我们将这一必要条件其整理为更常见的形式，称为 **KKT 条件**

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

其中

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

式I.3.13中的第四行称为互补松弛条件。它的意思是，若第 j 个不等式约束为起作用约束，则有 $h_j = 0$，此时 v_j 可以为正，可行方向不能为 h'_j 的方向。若第 j 个不等式约束不起作用，则 $h_j < 0$，v_j 必须为 0，可行方向和 h'_j 无关。

对于凸优化问题，KKT 条件是必要而不充分的条件，满足 KKT 的解不一定是最优解。对于非凸问题，如果问题中的 $f, \mathbf{g}, \mathbf{h}$ 满足约束线性无关条件 (Linear Independence Constraint Quality, LICQ)，那么 KKT 条件也是最优解的必要 (不充分) 条件。这些证明，读者可以查阅相应的最优化教材，这里不再赘述。

3.4.2 障碍函数法

由于 KKT 条件并不是充分条件，我们无法将其直接应用于不等式约束问题的求解算法。不过，我们可以将不等式约束条件以另一种方式加入目标函数中。

具体来说，我们定义一个在 x 负半轴定义的一元函数，该函数在 x 越接近 0 时，函数值越大，$x = 0$ 时取值为无穷大。这样的函数称为**障碍函数**。一种典型的障碍函数是

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

这样，我们可以将不等式约束条件加入目标函数中，有

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

其中，λ 为惩罚项因子。此时有新的目标函数梯度

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

这样，不同的 λ 值会将约束优化问题转化为不同的无约束优化问题。显然，无约束优化问题的目标函数 f_b 和原目标函数 f 并不相同。当 λ 值较大，f_b 表达的是可行域本身，求出的最优解和原目标函数 f 的最优解差距较大。当 λ 值较小，约束的作用会变弱，因此 f_b 的最优解会靠近 f。

这样，我们设置一组不断减小的 λ，运行多次无约束优化求解，每一次求解的初值是上一次的最优解，就可以在满足约束的条件下迭代求出原函数的最小值。为简单起见，我们的乘子可选取如下的衰减律

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

其中，k 表示迭代运行无约束优化问题的次数。

上述障碍函数法是一种**内点法**。基于无约束的 LSGD 方法，我们总结障碍函数 LSGD 算法如下

算法	**障碍函数 LSGD**
问题类型	不等式约束非线性优化
已知	目标函数 $f(\mathbf{x})$ 不等式约束函数 $\mathbf{h}(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$ $s.t.\quad \mathbf{h}(\mathbf{x}) \leq 0$
算法性质	迭代解

Algorithm 8: 障碍函数 LSGD

Input: 可微目标函数 $f(\mathbf{x})$, 初值 $\mathbf{x}_0$

Input: 不等式约束函数 $\mathbf{h}(\mathbf{x})$

Parameter: 阈值 ϵ, 乘子初值 λ_0, 乘子衰减率 γ

Output: 最优解 $\mathbf{x}^*$

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

while *True* **do**

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

  $f_b \leftarrow f - \lambda_k \mathbf{1}^T\mathbf{h}$

  $\nabla_{\mathbf{x}} f_b \leftarrow \nabla_{\mathbf{x}} f(\mathbf{x}) - \lambda_k \sum_{j=1}^{l} \nabla_{\mathbf{x}} h_j(\mathbf{x})/h_j(\mathbf{x})$

  $\mathbf{x}_k \leftarrow \text{opt\_nc\_LSGD}(\mathbf{x}_k, f_b, \nabla_{\mathbf{x}} f_b)$

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

对应的 python 代码如下所示

```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 二次规划 QP

二次规划问题已经在第 1 节中介绍，其核心是二次的目标函数和线性的等式/不等式约束。

3.5.1 无约束和等式约束

对于 QP 问题，无约束和等式约束情况都可以直接求出解析解。具体来说，对于无约束情况，在介绍牛顿迭代法时，我们已经求解出二次型函数的全局极小值。即满足

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

的最优解为

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

上述最优解析解存在的条件是 H 矩阵可逆。

算法	无约束 QP 解析解
问题类型	无约束 QP 问题
已知	半正定矩阵 H, 向量 $\mathbf{g}$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x}$
算法性质	解析解

Algorithm 9: 无约束 QP 解析解

Input: 半正定矩阵 H, 向量 $\mathbf{g}$

Output: 最优解 $\mathbf{x}^*$

$\mathbf{x}^* \leftarrow H^{-1}\mathbf{g}$

对于等式约束问题，我们使用 Lagrange 乘子法，可知

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

由最优解必要条件，我们有

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

因此，如果矩阵

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

可逆，则有解析形式的最优解

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

算法	等式约束 QP 解析解
问题类型	等式约束 QP 问题
已知	半正定矩阵 H, 向量 $\mathbf{g}$ 矩阵 M_{eq}, 向量 b_{eq}
求	最优解 $\mathbf{x}^*=\arg\min_{\mathbf{x}}\frac{1}{2}\mathbf{x}^T H\mathbf{x}+\mathbf{g}^T\mathbf{x}$
算法性质	解析解

Algorithm 10: 等式约束 QP 解析解

Input: 半正定矩阵 H, 向量 $\mathbf{g}$

Input: 矩阵 M_{eq}, 向量 b_{eq}

Output: 最优解 $\mathbf{x}^*$

$$\begin{bmatrix}\mathbf{x}^*\\ \lambda^*\end{bmatrix}\leftarrow\begin{bmatrix} H & M_{eq}^T \\ M_{eq} & 0 \end{bmatrix}^{-1}\begin{bmatrix}-\mathbf{g}\\ b_{eq}\end{bmatrix}$$

对应的 python 代码如下所示

```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 不等式约束 QP

对于含有不等式约束的一般 QP 问题，我们可以使用上文介绍的障碍函数法进行求解。具体来说，对比一般非线性优化问题，一般 QP 问题中的三个函数分别为

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

设

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

根据前面介绍的障碍函数法，为优化目标加入不等式约束障碍函数，有

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

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

再加上拉格朗日乘子，扩充优化变量，有

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

对扩充后的 $\mathbf{x}$ 求导，有

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

其中

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

根据 Lagrange 条件，我们需要求解 $L_b'(\bar{\mathbf{x}}) = 0$。新的优化目标为

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

因此，新优化目标的梯度为

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

其中

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

由此，基于 LSGD 算法，总结针对一般 QP 问题的障碍函数法如下

算法	QP 障碍函数法
问题类型	一般 QP 问题
问题简述	在等式和不等式约束下，求使目标函数取最小值的优化变量
已知	半正定矩阵 H, 向量 g, 可行初值 $\mathbf{x}_0$ 矩阵 M_{eq}, 向量 b_{eq} 矩阵 M, 向量 b
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{g}^T\mathbf{x}$ $s.t. \quad M_{eq}\mathbf{x} = b_{eq}, M\mathbf{x} \leq b$
算法性质	迭代解

Algorithm 11: QP 障碍函数法 (solve_QP_barrier)

Input: 半正定矩阵 H, 向量 g, 可行初值 $\mathbf{x}_0$
Input: 矩阵 M_{eq}, 向量 b_{eq}
Input: 矩阵 M, 向量 b
Parameter: 阈值 ϵ_1, ϵ_2, 衰减系数 γ
Parameter: 乘子初值 λ_0, λ_{b0}
Output: 最优解 $\mathbf{x}^*$

$\bar{\mathbf{x}}_k \leftarrow [\mathbf{x}_0^T \quad \lambda_0^T]^T$
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
$\mathbf{x}^*, \lambda^* \leftarrow \bar{\mathbf{x}}_k$

对应的 python 代码如下所示

```python
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]
```

这样，只要我们能够将问题写成一般 QP 形式，我们就可以使用上述算法进行求解。

3.6 非线性最小二乘

3.6.1 梯度下降法

对于最小二乘问题的优化，我们首先考虑无约束情况。还是从较为简单的梯度下降法入手。假设向量值函数 $\mathbf{e}(\mathbf{x})$ 满足一阶可微，记

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

假设 $H^T = H$，则有

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

我们可以用梯度下降法进行迭代求解

算法	梯度下降法
问题类型	无约束最小二乘
已知	可微函数 $\mathbf{e}(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
算法性质	迭代解

Algorithm 12: 梯度下降法

Input: 对称阵 H, 可微向量值函数 $\mathbf{e}(\mathbf{x})$

Parameter: 阈值 ϵ, 步长 α, 初值 $\mathbf{x}_0$

Output: 最优解 $\mathbf{x}^*$

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

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

  $\mathbf{x}_k \leftarrow \mathbf{x}_k - \alpha\nabla_x f(\mathbf{x}_k)$

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

对应的 python 代码如下所示

```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 高斯-牛顿法

梯度下降法仍有收敛慢等问题。我们仍使用类似牛顿法的思路，在 $\mathbf{x}_k$ 附近进行展开。由于目标函数本身就是二次型，我们仅需展开到一阶，即

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

则有

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

其中 $\delta = \mathbf{x} - \mathbf{x}_k$。该近似函数取最小值的充分必要条件为

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

即

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

因此有

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

这一方法也称为**高斯-牛顿法**。我们总结该算法如下

算法	**高斯-牛顿法**
问题类型	无约束最小二乘
已知	可微函数 $\mathbf{e}(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
算法性质	迭代解

Algorithm 13: 高斯-牛顿法 (Gauss_Newton)

Input: 对称阵 H, 可微向量值函数 $\mathbf{e}(\mathbf{x})$
Parameter: 阈值 ϵ, 初值 $\mathbf{x}_0$
Output: 最优解 $\mathbf{x}^*$

```
x_k ← x_0
while True do
    H_n ← J^T(x_k) H J(x_k)
    g_n ← J^T(x_k) H e(x_k)
    if ||g_n|| < ε then
        break
    x_k ← x_k − H_n^{-1} g_n
x* ← x_k
```

对应的 python 代码如下所示

```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 列文伯格-马夸尔特法

高斯-牛顿法是一种牛顿法的近似。记优化变量的更新

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

事实上，在高斯牛顿法中，我们求出的 $\delta \mathbf{x}_k$ 是使 $\hat{f}(\cdot)$ 下降最快的 $\delta \mathbf{x}$，但未必是使 $f(\cdot)$ 下降最快的 $\delta \mathbf{x}$。$\hat{f}(\cdot)$ 只是 $f(\cdot)$ 在 $\mathbf{x}_k$ 的某个邻域内的二阶近似，距离 $\mathbf{x}_k$ 越远则二者偏差越大。

对此，我们可以考虑在优化问题中，加入更新量的约束。具体来说，对于每一步迭代，我们构造一个新的优化问题，该优化问题的目标是在优化 $\hat{f}$ 的基础上，**使约束更新量尽量小**。具体来说，我们的优化目标是

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

式中，λ 是一个惩罚项。λ 越大，意味着我们希望更新量越小，反之亦然。对于每步迭代，我们应当根据 $\hat{f}(\cdot)$ 的更新量和 $f(\cdot)$ 更新量的差距，自适应调节 λ 的值。

具体来说，我们希望当 $\delta f(x)$ 和 $\delta \hat{f}(x)$ 的差距较大时，应当增大 λ；在差距较小时，可以适当减小 λ。我们采用用如下的指标来量化 $\delta f(x)$ 和 $\delta \hat{f}(x)$ 的差距

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

通过设置阈值 ρ_{min} 和 ρ_{max}，我们可以对 λ 进行自适应调节。

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

新的优化问题是关于 $\mathbf{x}_k$ 的二次规划，存在解析解。求损失函数 L 对 $\mathbf{x}_k$ 的导数，有

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

记

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

令导函数为 0，有

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

这一算法也称为**列文伯格-马夸尔特算法**，简称为 **L-M 法**。该算法总结如下

算法	**L-M 法**
问题类型	无约束最小二乘
已知	可微函数 $\mathbf{e}(\mathbf{x})$
求	最优解 $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$
算法性质	迭代解

Algorithm 14: L-M 法 (Levenberg_Marquardt)

Input: 可微向量值函数 $\mathbf{e}(\mathbf{x})$, 优化初值 $\mathbf{x}_0$
Input: 对称阵 H
Parameter: 优化阈值 ϵ, 差距阈值 ρ_{min}, ρ_{max}
Parameter: 惩罚项初值 λ_0
Output: 最优解 $\mathbf{x}^*$

$k \leftarrow 0$
while *True* **do**
  // 优化变量更新
  $H_l \leftarrow J^T(\mathbf{x}_k) H J(\mathbf{x}_k) + \lambda_k I$
  $g_l \leftarrow J^T(\mathbf{x}_k) H \mathbf{e}(\mathbf{x}_k)$
  if $||g_l|| < \epsilon$ **then**
    break
  $\delta \mathbf{x}_k \leftarrow -H_l^{-1} g_l$
  $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \delta \mathbf{x}_k$
  // 惩罚项更新
  $f_k \leftarrow \mathbf{e}^T(\mathbf{x}_k) H \mathbf{e}(\mathbf{x}_k)$
  $f_{k+1} \leftarrow \mathbf{e}^T(\mathbf{x}_{k+1}) H \mathbf{e}(\mathbf{x}_{k+1})$
  $\rho_k = |f_k - f_{k+1}| / ||J(\mathbf{x}_k)\delta \mathbf{x}_k||$
  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
    $\lambda_{k+1} \leftarrow \lambda_k$
  $k \leftarrow k + 1$
$\mathbf{x}^* \leftarrow \mathbf{x}_k$

对应的 python 代码如下所示

```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 补

在一些序列最小二乘优化问题中，我们会面临这样的**降维**需求：原先优化问题的多维优化变量 $\mathbf{x}_k$ 需要分为 $\mathbf{x}_r$ 和 $\mathbf{x}_d$ 两个部分，$\mathbf{x}_d$ 删除，$\mathbf{x}_r$ 保留。

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

如果优化问题中 $\mathbf{x}_r$ 和 $\mathbf{x}_d$ 相关的优化项、约束项可以完全分开，那么这个问题非常简单，直接丢弃 $\mathbf{x}_d$ 对应的优化和约束项即可。然而，实际绝大部分问题中，优化目标和约束项都是各种变量关联耦合在一起的。此时强行剥离 $\mathbf{x}_d$ 相关项，会导致下一轮优化的误差较大。那么，如何才能分开这个问题，既能去除 $\mathbf{x}_d$，又能保留一些必要的信息呢？

一种方法是，在需要降维的 $\mathbf{x}_k$ 局部进行二阶展开，则原本的非线性优化问题近似为二次优化问题。此时取最值相当于求解线性方程 $Ax = b$。使用线性代数的矩阵分块，就可以生成一个和 $\mathbf{x}_d$ 相关，但不包含 $\mathbf{x}_d$ 的关于 $\mathbf{x}_r$ 的约束。将该约束作为下一轮优化的一部分，可以一定程度减小误差。

具体来说，类似上一小节 L-M 法中的假设，在序贯优化中的某个时刻，优化 $\mathbf{x}$ 实际上是在优化一阶展开的 $\mathbf{x}_k$ 邻域内的 $\delta\mathbf{x}_k$

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

将二次型展开，取最值，我们有

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

设

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

可知 $A^T = A$。假设 $\delta\mathbf{x}_k$ 也分为保留部分和丢弃部分

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

则将 A, b 对应分块，有

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

这是一个线性方程组，下面我们通过 Schur 消元消去 $\delta\mathbf{x}_{d,k}$。首先，根据方程组的下式，有

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

移项，假设 A_{dd} 可逆，有

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

代入方程组的上式，有

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

整理，我们有关于 $\delta\mathbf{x}_{r,k}$ 的线性约束

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

其中

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

需要注意的是：Schur 补仅仅是在 $\mathbf{x}_k$ 邻域内起作用的线性约束。当经过一轮更新，变量开始远离，该约束就不再有效。

(参考资料：清华《运筹学》课件《控制之美 • 卷 2》《视觉 SLAM 十四讲》)

4 插值

4.1 基本概念

人类对世界的数学描述存在两套方法：**连续**和**离散**。在物理学中，绝大多数宏观现象 (运动和力、热、光、电等等) 都由连续的方程描述，遵守连续的物理法则。另一方面，现代机器人技术十分依赖数字计算机进行运算和处理，但数字计算机只擅长处理离散信息。为了在连续和离散间搭起桥梁，人类开发了许多数学工具。

在这之中，**插值**就是一类将离散表示转化为连续表示的重要工具。对于一个函数 $y = f(x)$，计算机可以方便地存储、处理、传输其离散化的采样值 $(x_0, y_0), ..., (x_N, y_N)$。而插值就是将这些采样值按照一定的要求，恢复为连续函数 $f(x)$ 的过程。

对于不同的应用场景，我们往往有不同的插值要求，体现为对函数及其各阶导数连续性的关注。例如，当我们有一组机器人位置的离散值，我们希望将其恢复为一条连续的机器人运动轨迹，此时我们应当保证其对时间的一阶导数 (即速度) 连续可导，不发生突变。

由此，我们可以定义 **C1 连续插值问题**。注意我们此处仅针对 $f(\cdot) : \mathbb{R} \to \mathbb{R}$ 的一元标量值函数进行插值。

问题	**C1 连续插值**
问题简述	已知一元标量函数采样点序列，求区间内一阶连续可导的插值函数
已知	采样点 $(x_i, y_i)_{0:N}$ 插值区间 $[a, b]$
求	函数 $f \in C^1[a, b]$ $s.t. f(x_i) = y_i, i = 0, ..., N$

相应的，如果我们继续增加连续性要求，希望得到的插值函数二阶连续可导，我们也有 **C2 连续插值问题**

问题	**C2 连续插值**
问题简述	已知一元标量函数采样点序列，求区间内二阶连续可导的插值函数
已知	采样点 $(x_i, y_i)_{0:N}$ 插值区间 $[a, b]$
求	函数 $f \in C^2[a, b]$ $s.t. f(x_i) = y_i, i = 0, ..., N$

对于插值问题，我们一般假设采样点已经过排序，即

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

我们仅关心最小和最大的采样点自变量之间的区间，称为**插值区间** $[a, b]$。我们一般考虑 $a = x_0, b = x_N$。

在本章中，我们将介绍机器人中最常用的样条插值方法。

4.2 样条插值

样条是一种用于手工绘图的柔软长木条。在手工作业时代，图纸绘制往往需要在平面上画出经过某几个给定点的光滑曲线。此时绘图员在每个经过点上打入一个钉子，将柔软的木条卡在这一系列钉子中间，木条自

然弯曲就会呈现出光滑曲线。

在信息时代，我们仍然借鉴这一思路，解决上述连续插值问题。

4.2.1 二次样条插值

首先，我们来看比较简单的 C1 连续插值问题。在数学中，有许多函数满足一阶连续，其中最简单的就是二次函数。然而，二次函数只有 3 个可配置参数，无法保证通过要求的 $N+1$ 个给定点 (一般远大于 3)。因此，我们可以使用分段函数的思想，借助给定点的 x_i 值，将整个插值区间 $[a,b]$ 分为许多小区间，在每个小区间内拟合一条二次函数，并注意保持一阶导连续。这样求出的函数曲线，非常类似于上文介绍的样条曲线，因此也称为**二次样条插值**。

具体来说，我们将插值区间分为 $N-1$ 个小区间。对于每个小区间 $[x_i, x_{i+1}], i=0,...,N-1$，$f_i$ 都是一个二次函数，所有的分段 f_i 组成函数 f，即

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

N 个分段的二次函数包含 $3N$ 个未知的参数，而插值条件就是求解这些未知数的方程。我们有

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

上述约束条件一共产生 $3N-1$ 个方程。为使方程数和未知数个数相同，我们需要添加一个条件，例如

$$f_0'(x_0) = 0$$

将二次函数代入上述方程，有

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

可以看到，上述方程均为关于未知系数的线性方程。将所有未知数写成标准线性方程的形式，我们有

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

其中

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

求解上述方程，我们即可得到所需的 $f(x)$。这里我们求出的是单输入单输出函数的插值结果。如果需要插值单输入多输出函数，只需要将输出的每个维度作为一元函数单独插值即可。

Algorithm 15: 二次样条插值

Input: 采样点 $(x_i, y_i)_{0:N}$

Output: 插值函数 $f \in C^1[a,b]$

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

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

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

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

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

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

算法	二次样条插值
问题类型	C1 连续插值
已知	采样点 $(x_i, y_i)_{0:N}$ 插值区间 $[a,b]$
求	函数 $f \in C^1[a,b]$ $s.t. f(x_i) = y_i, i = 0, ..., N$
算法性质	迭代解

对应的 python 代码如下所示

```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 三次样条插值

对于 C2 连续插值问题，我们可以延续上面二次样条的思路，只是将每个区间的连续函数变为三次函数，即**三次样条插值**。具体来说，我们有

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

这样，我们需要求解的未知数有 $4N-4$ 个。对于每个区间，我们的约束变成了二阶连续可导，即

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

此时共有 $4N-6$ 个方程，仿照上述二次样条的思路，我们可以限定左右边界的一阶导为 0[4]

$$\begin{aligned}
f_0'(x_0) &= 0 \\
f_{N-1}'(x_N) &= 0
\end{aligned}$$

将三次函数代入上述方程，有

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

此时，我们有线性方程

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

其中

[4]此处额外条件的选取并不唯一，例如我们还可以现代左右边界二阶导为 0，或左右边界一二阶导相等。实际使用中，可根据插值函数的额外已知信息进行选取

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

且

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

求解上述方程，我们即可得到所需的 $f(x)$。

Algorithm 16: 三次样条插值

Input: 采样点 $(x_i, y_i)_{0:N}$

Output: 插值函数 $f \in C^1[a, b]$

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

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

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

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

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

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

算法	三次样条插值
问题类型	C2 连续插值
已知	采样点 $(x_i, y_i)_{0:N}$ 插值区间 $[a, b]$
求	函数 $f \in C^1[a, b]$ $s.t. f(x_i) = y_i, i = 0, ..., N$
算法性质	迭代解

对应的 python 代码如下所示


```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
```

(参考资料：清华《数值分析》课件)

Part II

机器人学基础

5 基本概念

在本部分中，我们将介绍**机器人学** (Robotics) 中最基础的概念和理论：运动学、动力学、雅可比等。这些理论是后续所有机器人控制、学习等算法的基础。

5.1 关节与正逆运动学

机器人分为固定基座机器人和移动机器人两类。无论属于哪一类，机器人 (机械臂) 都可以建模为一系列通过关节连接的刚体。我们给不同的关节驱动器发送指令，控制整个机器人 (尤其是末端) 实现各种简单或复杂的运动。在本部分的章节中，我们的主要讨论对象是**固定基座机器人**[5]。

在数学模型中，关节分为转动关节和移动关节，分别实现相对旋转运动和相对平移运动。转动关节可以产生力矩和角速度，移动关节可以产生力和线速度。**在本书中，如无特殊说明，我们讨论的关节均为转动关节。**

目前，各类机器人中最常用的关节驱动机构是永磁同步电机。我们给关节提供电压/电流指令，配合减速机构后，就可以实现精细的扭矩和转动角度的控制，详见第15.1节的介绍。除电机外，还有液压、绳驱等关节驱动方式。

无论转动关节还是移动关节，都可以将其参数化表示，详见第6.1节。关节的参数分为固定参数和可变参数，固定参数不可控，可变参数对应着运动的**自由度**。自由度是描述机器人系统的重要参数。在没有闭链的情况下，自由度就是驱动关节的数量。

机器人的可变参数组成**关节变量** q(也称**关节向量**)。关节变量构成的空间称为**关节空间**，由关节角度 (转动关节) 组成。关节空间对机器人控制非常重要。

除了关节角度，我们还非常关注机器人的末端。机器人的末端一般安装有工具或机械手，统称为**执行器**。末端位姿记为 x_e，包括三维位置和三维姿态。末端位姿其所在的空间称为**操作空间**。相比关节空间，操作空间和机器人的任务，也就是用户的需求直接相关。

在给定关节参数 (固定和可变) 的条件下，根据空间几何关系，我们可以写出 x_e 和 q 的表达式 $x_e = k(q)$。这就是机器人的**正运动学**。**正运动学将关节空间位置转换为操作空间位置**。

与此相反，如果用末端位姿 x_e 计算关节变量 q，就称为机器人**逆运动学**。**逆运动学将操作空间位置转换为关节空间位置**。关于正逆运动学的具体介绍，详见第6章。

逆运动学和正运动学的关键区别是：逆运动学可能有多解甚至无穷多解，也就是一个需求有不同的方案，这意味着机器人的冗余；逆运动学可能面临奇异，也就是一个需求处于机器人的极限；逆运动学还可能无解，这意味着需求超过了机器人的能力范围。在实际应用中，妥善处理这样的问题是非常重要的。

5.2 雅可比，动力学和控制

上面我们介绍的正逆运动学是关节空间和操作空间位置关系的描述，而**雅可比矩阵**是关节空间和操作空间速度关系的描述，也就是微分关系的描述。其中，**正向雅可比**是指 v_e 和 $\dot{q}$ 的关系，即 $v_e = J(q)\dot{q}$。雅可比矩阵可能会随 q 的变化出现奇异，即不满秩。研究雅可比的奇异性非常重要。

[5]这些结论稍加修改，就可以同样适用于双足机器人等没有固定基座的移动机器人

此外，雅可比矩阵还和失重状态下的机械臂静力关系有关。在这种假设下，雅可比转置可以将操作空间的力转化为关节空间的力矩，而无需求解复杂的逆运动学。以上内容详见第7章。

动力学是指机器人的操作空间位置/速度/加速度和各关节力/力矩之间的关系。其中，**逆动力学问题**是指：根据操作空间的位置/速度/加速度和控制目标，来计算每个关节要给多大力/力矩。它们对机器人的控制非常重要，一般从末端向前求解。此外，在仿真器中还需要知道施加力/力矩后的速度/加速度，即**正动力学问题**。正运动学求解一般从基座到末端求解。以上内容详见第8章。

机器人 (机械臂) 的控制是一个关键的问题，也是控制理论的一个典型应用场景。开环的控制就是解算逆动力学，然后将指令直接给电机，但这样会造成精度等问题。合理的闭环控制，考虑了用户的指令、环境的影响、建模的精度等等方面，能够实现精准的跟踪和调节 (镇定)。关于机器人控制，我们在本书第 IV 部分集中介绍。

5.3 轨迹规划和插值

机器人控制需要关节空间或操作空间指令，一般是位置指令。指令从何而来呢？

较为初级的机器人，可以由人类指定运动轨迹，例如运动末端点或一系列离散的轨迹点 x_d。许多固定环境中工作的工业机械臂都属于这一类型。

较为自主的机器人，需要在规划轨迹点的过程中，考虑避碰等问题。这个过程称为**轨迹规划**，详见第9章。目前的许多自动驾驶车辆、复杂环境中使用的工业机械臂属于这一类型。

更加自主的机器人，则要根据人类的需要，处理图像/声音/文字等不同模态的指令和传感器信息，随后通过自主程序规划动作指令。当下非常热门的 VLA(Vision-Language-Action) 大模型就是这样的技术。这样的机器人已经可以表现出一定的智能，是未来机器人的发展方向。关于机器人学习的相关内容，将在后续章节进行讨论。

无论轨迹来源如何，最初给定的轨迹指令一般是离散的 (频率较低)。因此，需要有一个过程，将最初给定的较为离散的轨迹，插值成较为连续的轨迹，同时满足二阶导连续的可行性约束。这个过程称为**轨迹插值**。轨迹插值既可以在操作空间完成，也可以在关节空间完成。实际的机器人系统中，会根据需要安排轨迹插值模块。关于插值的内容，详见第？章。

6 正逆运动学

6.1 D-H 参数

如上章所述，机器人 (机械臂) 可以建模为一系列通过关节连接的刚体，即连杆系统。为简单起见，以下叙述我们仅考虑单端固定的单链转动关节机器人。

连杆系统总有一端固定于一较稳定的平台上，这一端称为**基座**，也被编号为连杆 L_0。连杆系统的另一端可以在一定的限度内自由活动，称为**末端**。基座到末端之间有 n 个关节，由基座到末端依次编号为 $1\sim N$。关节 J_n 连接着 L_{n-1} 和 L_n 两个连杆，靠近基座的为连杆 L_{n-1}，靠近末端的为连杆 L_n。相应的，**连杆 L_n 连接着关节 J_n 和关节 J_{n+1}**。

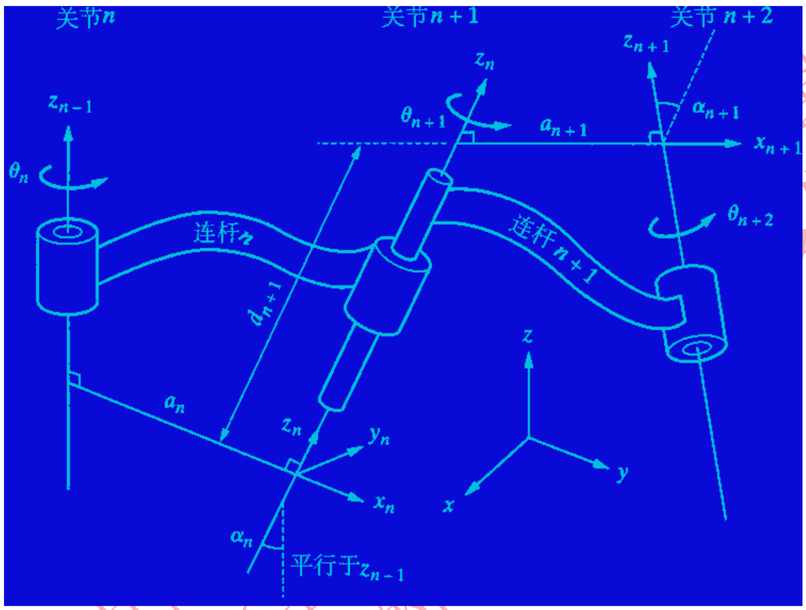

图 II.6.1: D-H 参数定义

对于每一连杆 L_n，我们可以定义两个**连杆坐标系**：n' 系和 n 系。n' 系在前，n 系在后。n' 系和 n 系都固联在 L_n 上。n' 系对应关节 J_n，n 系对应关节 J_{n+1}。

每个坐标系的 z 轴由关节转轴定义。对于旋转关节，每个关节的转动轴是固定的。关节 J_{n+1} 的转动轴是 n 系和 $n+1'$ 系的 z 轴。注意 z_n 不是 J_n 的转动轴而是 J_{n+1} 的转动轴。

每个坐标系的 x 轴由连杆公垂线定义。对于连杆 L_n，关节 J_n 和关节 J_{n+1} 的 z 轴分别是空间中的两条有向直线。不考虑两直线重合的情况，其相对位置关系有如下三种可能：平行、相交、异面。对于非平行的情况，总能找到唯一的一条和 z_n 轴以及 z_{n+1} 轴同时垂直且相交的直线，称为连杆 n 的公垂线。这两个交点定义为 O'_n 和 O_{n+1}。若两轴平行，则中间的关节不提供额外自由度，被视为冗余关节，可跳过该关节。

若 z_n 轴和 z_{n+1} 轴异面，则指定公垂线 O'_n 到 O_{n+1} 的方向为坐标系 n 和 $n+1'$ 的 x 轴方向，即 x_n 和 $x_{n+1'}$ 方向。若 z_n 轴和 z_{n+1} 轴相交，则 O_n 和 $O_{n+1'}$ 重合，且可任取一个公垂线方向为 x_n 和 $x_{n+1'}$ 方向。随后，由右手系定义，可定义出每个坐标系的 y 轴。这样，对每个连杆 L_n，n' 系和 n 系的原点及各坐标轴都定义完毕。

对上述坐标系定义，对于每个连杆 L_n，我们定义以下四个参数：前端关节平移量 d_n、连杆平移量 a_n、连杆偏移角 α_n、关节旋转角 θ_n。具体定义为：

- 前端关节平移量 d_n：O_{n-1} 到 $O_{n'}$ 的距离

- 关节旋转角 θ_n：x_{n-1} 轴和 x_n 轴的夹角
- 连杆平移量 a_n：$O_{n'}$ 到 O_n 的距离
- 连杆偏移角 α_n：z_{n-1} 轴和 z_n 轴的夹角

上述参数组称为机器人系统的 **D-H 参数**。

这样，我们就完成了连杆系统的参数化。对于旋转关节构成的连杆系统，d_n, a_n, α_n 均为固定值，出厂时经标定得到。θ_n 是可变值，也简称为关节角，也称为广义的关节空间位置。所有的 θ_n 组合在一起，形成关节向量 q，即

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

6.2 正运动学

通过上述参数定义，我们可以表达出末端相对于基座的位姿，即坐标系 N 到 0 的齐次坐标变换 T_N^0。我们一般定义关节系 0 为基座坐标系 b；定义关节系 N 为末端坐标系 e。因此，末端相对于基座的位姿也可写作 T_e^b。

如第 I.1 章所述，齐次矩阵可分为旋转矩阵和平移向量两部分

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

而旋转矩阵 R_e^b 可被等效的欧拉角或四元数替代。我们可以将平移向量和某一种旋转表示写在一起，紧凑的表示末端相对于基座的位姿，即末端状态 $\mathbf{x}_e$。即

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

在已知全部固定参数的情况下，根据关节向量 q 求解末端状态 $\mathbf{x}_e$ 的问题，称为**正运动学求解**。

问题	正运动学求解
问题简述	在已知固定参数的情况下，根据关节向量求解末端状态
已知	关节向量 q
求	末端状态 $\mathbf{x}_e$

根据上述定义，正运动学可以解析求解。由于从 T_e^b 到 $\mathbf{x}_e$ 的转换是唯一的，我们仅考虑求解 T_e^b，或 T_N^0。具体来说，让我们首先考虑 T_n^{n-1}。易知

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

根据 a_n 和 α_n 定义，有

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

根据 d_n 和 θ_n 定义，有

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

这里我们为简单起见，将 $\cos(\alpha), \sin(\alpha)$ 简写为 C_α, S_α。
因此，我们有连杆系转换公式

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

因此，正运动学公式为

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

我们有

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

总结上述公式，得到正运动学的解析算法

Algorithm 17: 运动学解析解 (robot_fk)

Input: D-H 参数 $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: 关节向量 $q = \theta_{1:N}$
Output: 末端位姿 x_e
$T_0^0 \leftarrow I_{4\times4}$
for $n \in 1, \ldots, N$ **do**
  $T_n^{n-1} \leftarrow \begin{bmatrix} C_{\theta_n} & -S_{\theta_n}C_{\alpha_n} & S_{\theta_n}S_{\alpha_n} & a_nC_{\theta_n} \\ S_{\theta_n} & C_{\theta_n}C_{\alpha_n} & -C_{\theta_n}S_{\alpha_n} & a_nS_{\alpha_n} \\ 0 & S_{\alpha_n} & C_{\alpha_n} & d_n \\ 0 & 0 & 0 & 1 \end{bmatrix}$
  $T_n^0 \leftarrow T_{n-1}^0 T_n^{n-1}$
$R_0^N, t_{0N}^N \leftarrow (T_n^0)^{-1}$
$\vartheta_e \leftarrow \text{rotmat\_to\_eular}(R_0^N)$
$x_e \leftarrow [(t_{0N}^N)^T \quad (\vartheta_e)^T]^T$

算法	运动学解析解
问题类型	正运动学求解
已知	D-H 参数 d_n, a_n, α_n, 关节向量 q
求	末端位姿 x_e
算法性质	解析解

对应的 python 代码如下所示

```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])
```

除了末端位置 x_e，每一连杆的质心位置 $p_n(q) = p^b_{b,l_n}(q)$ 也有相似的正运动学关系

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

其中，$p^n_{l_n}$ 表示在坐标系 n 下，连杆 n **质心**的位置，是常量。

6.3 逆运动学

和正运动学相反，逆运动学是指，在已知固定参数的情况下，根据末端状态 $\mathbf{x}_e$ 求解关节向量 q.

问题	逆运动学问题
问题简述	在已知固定参数的情况下，根据关节向量求解末端状态
已知	末端状态 $\mathbf{x}_e$
求	关节向量 q

相比正运动学，逆运动学的求解有很大不同。首先，正运动学一定有唯一解，逆运动学则不然。如果机器人的位形位于冗余空间，会出现逆运动学**多解**。如果位于操作空间边界或内部奇异点，会出现逆运动学**奇异解**。如何处理多解和识别奇异解是逆运动学算法的重要问题。

其次，逆运动学没有一般形式的解析解。在实际使用中，要么根据特定的机器人推导特定的**解析法**求解方程；要么使用**数值法**进行求解。数值法的基本原理是借助雅可比矩阵，通过将运动学方程求根问题转化为优化问题，从而迭代求解。我们将在7.3节中，详细介绍数值法逆运动学求解方法。

(参考资料：清华《智能机器人》课件, 《机器人学：建模、规划与控制》)

7 微分运动学

7.1 雅可比矩阵

7.1.1 几何与分析雅可比

正逆运动学给出了关节位置和末端位置间的相互关系。那么关节速度和末端速度是否有关系呢？这就是雅可比矩阵。

对于正运动学公式 $x_e = k(q)$，考虑极短时间 δt 内的变化，有

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

两边同时除以时间小量 δt，有

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

即

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

该式中的 v_e 是广义末端速度，它包含两部分：位置对时间的导数和姿态对时间的导数。这里，我们定义两种不同的 v_e。如果 v_e 的角度部分是角速度 ω_e，则记 v_e 为 v_{ew}。如果 v_e 的角度部分是四元数或者欧拉角的微分，则记 v_e 为 v_{ea}。

根据这两种不同的记法，我们可以定义两种不同的雅可比矩阵。v_{ew} 对 $\dot{q}$ 的雅可比记为 $J_w(q)$，称为**几何雅可比**。v_{ea} 对 $\dot{q}$ 的雅可比记为 $J_a(q)$，称为**分析雅可比**。我们有

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

式II.7.1将机器人的关节空间速度转化为操作空间速度，由于它与正运动学方向一致，也称为**正向雅可比**。

由于**欧拉角和四元数的导数不等于角速度**，几何雅可比和分析雅可比一般并不相同。二者的数学关系将在下一节中推导。

在两种雅可比中，几何雅可比最常见的应用是机器人动力学方程 (见下文的推导)，而分析雅可比可用于操作空间控制中观测速度。

7.1.2 雅可比与力

一些时候，我们会考虑分析失重状态下的机械臂静力关系。注意到，在**失重状态下，忽略各种非末端阻力**，机器人所有关节的功率之和就是末端的功率。

记末端的广义力为 F_e(三维扭力和三维平动力)，机器人关节力矩组成的向量为 τ。由上述功率关系，我们有

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

代入式II.7.1，有

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

即

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

也就是说，**失重状态下，雅可比的转置可以将操作空间的末端力直接转化为关节力矩**。对于有外力的控制(见第 4 章)，这样的关系非常重要，可以避免计算复杂的逆动力学。

7.1.3 逆向雅可比

对于六自由度机械臂，正向雅可比矩阵是方阵。假设该雅可比矩阵可逆，我们就可以得到**从操作空间速度到关节空间速度的转化关系**。（使用哪种雅可比取决于给出的操作空间速度表达形式）

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

式II.7.4也称为**逆向雅可比**。它可以将操作空间速度转换为关节空间速度。在第 IV 部分介绍的操作空间控制中，逆向雅可比十分重要。

7.1.4 二阶微分运动学

此外，如果对式II.7.1进一步求导，我们还可获得二阶微分运动学关系

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

7.2 雅可比求解

雅可比矩阵可以通过解析的方式计算。

7.2.1 末端速度和角速度

假设我们仅考虑只含有转动关节的机器人。我们首先考虑连杆 n 相对于 $n-1$ 的运动。记坐标系 n 的原点相对基坐标系的位置为 p_n，速度为 $\dot{p}_n$，角速度为 ω_i，即

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

首先考虑连杆 n 的相对角速度，它的大小是关节 n 的角度变化速度，方向是 $n-1$ 系的 z 轴方向

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

而其相对于上一个关节的线速度为

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

因此，我们有 p_i 和 ω_i 的递推关系式

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

对于末端状态，由于

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

我们有

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

以及

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

7.2.2 几何雅可比推导

对于几何雅可比，广义速度由线速度和角速度构成

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

因此，$J_w(q)$ 也由两部分构成

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

此处的 $\dot{q}$ 即为各关节转角对时间的微分 $\dot{\theta}_n$，因此我们可以写出
即

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

这里需要注意，每一个 $p_{n-1,N}^b$ 和 z_{n-1}^b 都是 q 的函数。它们的表达式是

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

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

7.2.3 两个雅可比的转换

对于分析雅可比，我们以欧拉角 ϑ_e 表示末端姿态，有

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

假设 T_ϑ 表示从欧拉角微分 $\dot{\vartheta}$ 到角速度 ω 的变换矩阵，即

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

则有

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

其中 T_w^a 表示从 J_w 到 J_a 的变换矩阵

$$T_w^a = \begin{bmatrix} I_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & T_\vartheta^{-1} \end{bmatrix} \in \mathbb{R}^{6\times6} \tag{II.7.9}$$

因此

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

变换矩阵 $T_\vartheta \in \mathbb{R}^{3\times3}$ 没有统一的形式。欧拉角定义的顺序不同，该矩阵的求法也不同。在 ZXY 欧拉角下，其表达式为

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

注意：欧拉角表示存在万向锁问题。在奇点附近，矩阵 T_ϑ 不可逆，此时分析雅可比出现奇异。

总结上述公式，我们有分析雅可比的解析算法

Algorithm 18: 分析雅可比解析解 (robot_jacobian_a)

Input: D-H 参数 $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: 关节向量 $q = \theta_{1:N}$, 欧拉角 ϑ
Output: 分析雅可比矩阵 J_a

$T_0^0, T_N^N \leftarrow I_{4\times4}, I_{4\times4}$
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)
$J_a \leftarrow \mathrm{diag}\{I_{3\times3}, T_\vartheta^{-1}\} J_w$

算法	分析雅可比解析解
问题类型	分析雅可比求解
已知	D-H 参数 d_n, a_n, α_n 关节向量 q, 欧拉角 ϑ
求	分析雅可比矩阵 J_a
算法性质	解析解

对应的 python 代码如下所示

```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 质心雅可比推导

上节我们提到，每个连杆的质心也有自己的运动学关系 $p_n(q)$。因此，我们也可以计算每个连杆质心的速度和角速度对关节角速度的偏导，即质心雅可比矩阵

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

其中

$$p^b_{i,l_n}(q) = R^0_i(q)(R^i_n(q)p^n_{l_n} + t^i_n(q))$$

其中，$p^n_{l_n}$ 表示在坐标系 n 下，连杆 n **质心**的位置，是常量。
并且有

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

这样，我们就有

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

总结上述公式，得到质心雅可比的解析算法

Algorithm 19: 质心雅可比解析解 (robot_j_centroid)

Input: D-H 参数 $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: 关节向量 $q = \theta_{1:N}$, 连杆质心位置 $p^n_{l_n}$
Output: 质心雅可比矩阵 $J_{w,n}, n = 1, ..., N$

$T^0_0, T^N_N \leftarrow I_{4\times4}, I_{4\times4}$
for $n \in 1, \ldots, N$ **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 $m \in n-1, \ldots, 0$ **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 $m \in 0, \ldots, n-1$ **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 $m \in n+1, \ldots, N$ **do**
    $(J_{w,n})_{:,m} \leftarrow 0_{6\times1}$

算法	**质心雅可比解析解**
问题类型	质心雅可比求解
已知	D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$ 关节向量 $q = \theta_{1:N}$ 连杆质心位置 $p^n_{l_n}, n = 1, ..., N$
求	质心雅可比矩阵 $J_{w,n}, n = 1, ..., N$
算法性质	解析解

对应的 python 代码如下所示

```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 基于雅可比的 IK

在上一章中我们提到，雅可比也可以用于机器人逆运动学的求解，也就是 IK 的数值求解方法。具体来说，我们需要求解运动学方程

$$x_e = k(q)$$

其中 x_e 表示给定的末端位姿。事实上，任何方程求解可以等效为一个最小二乘优化问题

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

其中 $H \in \mathbb{R}^{6\times6}$ 为对称正定矩阵。在逆运动学有解的条件下，优化问题的最优解 q^* 就是运动学方程的解。

通过这样的转换，我们就可以使用第3章介绍的非线性最小二乘问题优化算法求解逆运动学了。这些优化算法都需要误差 $\mathbf{e}$ 对优化变量的导数。注意到，如果我们假设给定的位姿是欧拉角形式，这个导数正是分析雅可比矩阵取负

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

因此，只需给定一个初值，我们就可以使用最小二乘算法求解 IK。例如，我们可以使用梯度下降法 (也称最速下降法) 进行求解。这一方法在一些其他资料中，也称为**雅可比转置法**。

算法	**梯度下降法 IK**
问题类型	逆运动学求解
已知	末端状态 $\mathbf{x}_e$
求	关节向量 q
算法性质	迭代解

Algorithm 20: 梯度下降法 IK

Input: D-H 参数 $d_{1:N}, a_{1:N}, \alpha_{1:N}$

Input: 末端位姿 x_e, 关节初值 q_0

Parameter: 阈值 ϵ, 步长 α_p, 对称阵 H

Output: 最优解 q^*

$q_k \leftarrow q_0$

while *True* **do**

$J_a \leftarrow \text{robot\_jacobian\_a}(q_k, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$

$x_{e,k} \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q_k)$

$\nabla_x J \leftarrow -J_a^T H(x_e - x_{e,k})$

if $||x_{e,k} - x_e|| < \epsilon$ **then**

break

$q_k \leftarrow q_k - \alpha_p \nabla_x J$

$q^* \leftarrow q_k$

对应的 python 代码如下所示

```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
```

同样的，我们也可以使用高斯牛顿法，得到**基于高斯-牛顿法的逆运动学算法**。在一些其他的教科书中，这一方法也称为**雅可比伪逆法**。

算法	高斯-牛顿法 IK
问题类型	逆运动学求解
已知	末端状态 $\mathbf{x}_e$
求	关节向量 q
算法性质	迭代解

Algorithm 21: 高斯-牛顿法 IK

Input: D-H 参数 $d_{1:N}, a_{1:N}, \alpha_{1:N}$

Input: 末端位姿 x_e, 关节初值 q_0

Parameter: 阈值 ϵ, 对称阵 H

Output: 最优解 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

  $H_n \leftarrow J_a^T H J_a$

  $g_n \leftarrow J_a^T H(x_e - x_{e,k})$

  $q_k \leftarrow q_k - H_n^{-1} g_n$

$q^* \leftarrow q_k$

对应的 python 代码如下所示

```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
```

此外，我们还可以使用 L-M 法求解该最小二乘问题，该方法也称为**阻尼最小二乘法**。

算法	阻尼最小二乘法 IK
问题类型	逆运动学求解
已知	末端状态 $\mathbf{x}_e$
求	关节向量 q
算法性质	迭代解

Algorithm 22: 阻尼最小二乘法 IK

Input: D-H 参数 $d_{1:N}, a_{1:N}, \alpha_{1:N}$

Input: 末端位姿 x_e, 关节初值 q_0

Parameter: 优化阈值 ϵ, 差距阈值 ρ_{min}, ρ_{max}

Parameter: 惩罚项初值 λ_0, 对称阵 H

Output: 最优解 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

    $\lambda_{k+1} \leftarrow \lambda_k$

  $k \leftarrow k + 1$

$q^* \leftarrow q_k$

对应的 python 代码如下所示

```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 雅可比的性质

[主要内容：运动学奇异/运动学冗余]

[本部分内容将在后续版本中更新，敬请期待]

(参考资料：清华《智能机器人》课件，《机器人学：建模、规划与控制》)

8 动力学

动力学是机器人的力矩与关节空间位置 (速度，加速度) 的关系。它常常写成动力学方程的形式。

显然，动力学涉及机器人的力和力矩。根据牛顿运动定律，力/力矩和物体的质量/转动惯量、加速度/角加速度相关。因此，相比于运动学方程，动力学方程的已知量除了包含 D-H 参数，还需要包含机器人的质量以及质量分布。

机器人的动力学方程可以根据牛顿-欧拉方程，从基座/末端开始推导；也可以从分析力学的欧拉-拉格朗日方程进行推导。这里我们采取后一种推导方式。

8.1 动能与势能

根据动能定义，我们可以写出连杆动能表达式[6]

$$\mathcal{T}_n(q,\dot{p}_n,\omega_n)=\frac{1}{2}m_n\dot{p}_n^T\dot{p}_n+\frac{1}{2}\omega_n^T R_n(q)\mathbf{I}_n^n R_n^T(q)\omega_n^T \tag{II.8.1}$$

其中，$R_n = R_n^b$ 表示从 n 系到基坐标系的旋转矩阵。$\mathbf{I}_n^n$ 表示在 n 系下连杆 n 的惯量矩阵，为常值。$p_n = p_{b,l_n}^b$ 表示在基坐标系下连杆**质心**的位置，满足

$$p_n = p_{b,l_n}^b = T_n^0(q)p_{l_n}^n$$

如前所述，使用几何雅可比矩阵，可将动能表达式写为关节空间速度的表达式

$$\mathcal{T}_n(q,\dot{q})=\frac{1}{2}m_n\dot{q}^T J_{p,n}^T(q)J_{p,n}(q)\dot{q}+\frac{1}{2}\dot{q}^T J_{\theta,n}^T(q)R_n(q)\mathbf{I}_n^n R_n^T(q)J_{\theta,n}(q)\dot{q}$$

事实上，可定义机器人在关节空间的惯量矩阵

$$B(q)=\sum_{n=1}^{N} m_n J_{p,n}^T(q)J_{p,n}(q)+J_{\theta,n}^T(q)R_n(q)\mathbf{I}_n^n R_n^T(q)J_{\theta,n}(q) \tag{II.8.2}$$

则机器人的总动能为

$$\mathcal{T}(q,\dot{q})=\frac{1}{2}\dot{q}^T B(q)\dot{q}=\sum_{j=1}^{N}\sum_{k=1}^{N} b_{ij}(q)\dot{q}_j\dot{q}_k \tag{II.8.3}$$

随后，在不考虑弹性形变的情况下，我们可以写出总势能

$$\mathcal{U}(q)=-\sum_{n=1}^{N} m_n g_0^T p_n(q) \tag{II.8.4}$$

其中，g_0 为基坐标系中的重力加速度向量，一般取

$$g_0=\begin{bmatrix}0\\0\\-g\end{bmatrix} \tag{II.8.5}$$

[6]事实上，在更精细的建模中，需要单独考虑机器人驱动电机的动能（质心和惯量）。本书为简化推导过程，仅考虑每个连杆。

8.2 动力学方程推导

根据分析力学，拉格朗日量是动能减去势能

$$\mathcal{L}(q,\dot{q}) = \mathcal{T}(q,\dot{q}) - \mathcal{U}(q) \tag{II.8.6}$$

因此，在刚体机器人系统中的拉格朗日量为

$$\mathcal{L}(q,\dot{q}) = \frac{1}{2}\dot{q}^T B(q)\dot{q} - \sum_{n=1}^{N} m_n g_0^T p_n(q) \tag{II.8.7}$$

根据拉格朗日方程，有

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}} - \frac{\partial \mathcal{L}}{\partial q} = \xi \tag{II.8.8}$$

这里的 ξ 是广义力。对于仅含转动关节的机器人，广义力就是关节转矩。关节转矩包括关节电机施加的驱动力矩 τ、摩擦力矩 τ_f、外力矩 τ_F 等。

接下来，我们推导理想机器人 (无摩擦、弹性形变) 在不受外力作用下的动力学方程。

将拉格朗日量代入机器人方程，同时取 $\xi = \tau$，有

$$\frac{d}{dt}\frac{\partial \mathcal{T}}{\partial \dot{q}} - \frac{\partial \mathcal{T}}{\partial q} - \frac{\partial \mathcal{U}}{\partial q} = \tau$$

$$\frac{d}{dt}\left(B(q)\dot{q}\right) - \frac{1}{2}\dot{q}^T\frac{\partial B(q)}{\partial q}\dot{q} - \frac{\partial \mathcal{U}}{\partial q} = \tau$$

先来处理第三项，注意到

$$\begin{aligned} -\frac{\partial \mathcal{U}}{\partial q} &= -\left(-\sum_{n=1}^{N} m_n g_0^T \frac{\partial}{\partial q} p_n(q)\right) \\ &= \sum_{n=1}^{N} m_n g_0^T J_{p,n}(q) \end{aligned}$$

记

$$g(q) = \sum_{n=1}^{N} m_n g_0^T J_{p,n}(q) \tag{II.8.9}$$

于是我们有

$$-\frac{\partial \mathcal{U}}{\partial q} = g(q)$$

对于第一项，我们有

$$\frac{d}{dt}\left(B(q)\dot{q}\right) = B(q)\ddot{q} + \dot{B}(q)\dot{q}$$

因此

$$\frac{d}{dt}\frac{\partial \mathcal{T}}{\partial \dot{q}} - \frac{\partial \mathcal{T}}{\partial q} = B(q)\ddot{q} + \dot{B}(q)\dot{q} - \frac{1}{2}\dot{q}^T\frac{\partial B(q)}{\partial q}\dot{q}$$

考虑第 n 个关节变量 q_n(对于转动关节即 θ_n)，有

$$\frac{d}{dt}\frac{\partial \mathcal{T}}{\partial \dot{q}_i}-\frac{\partial \mathcal{T}}{\partial q_i}=\sum_{j=1}^{N} b_{nj}(q)\ddot{q}_j+\sum_{j=1}^{N}\dot{b}_{nj}(q)\dot{q}_j-\frac{1}{2}\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{jk}(q)}{\partial q_n}\dot{q}_j\dot{q}_k$$

注意到

$$\begin{aligned}\dot{b}_{nj}(q)&=\frac{\partial b_{nj}(q)}{\partial q}\dot{q}\\&=\sum_{k=1}^{N}\frac{\partial b_{nj}(q)}{\partial q_k}\dot{q}_k\end{aligned}$$

于是前两项可简化为

$$\begin{aligned}\frac{d}{dt}\frac{\partial \mathcal{T}}{\partial \dot{q}_i}-\frac{\partial \mathcal{T}}{\partial q_i}&=\sum_{j=1}^{N} b_{nj}(q)\ddot{q}_j+\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{nj}(q)}{\partial q_k}\dot{q}_k\dot{q}_j-\frac{1}{2}\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{jk}(q)}{\partial q_n}\dot{q}_j\dot{q}_k\\&=\sum_{j=1}^{N} b_{nj}(q)\ddot{q}_j+\sum_{j=1}^{N}\sum_{k=1}^{N}\left(\frac{\partial b_{nj}(q)}{\partial q_k}-\frac{1}{2}\frac{\partial b_{jk}(q)}{\partial q_n}\right)\dot{q}_k\dot{q}_j\end{aligned}$$

记

$$h_{njk}(q)=\frac{\partial b_{nj}(q)}{\partial q_k}-\frac{1}{2}\frac{\partial b_{jk}(q)}{\partial q_n}$$

且

$$c_{njk}(q)=\frac{1}{2}\left(\frac{\partial b_{nj}}{\partial q_k}+\frac{\partial b_{nk}}{\partial q_j}-\frac{\partial b_{jk}}{\partial q_n}\right)\tag{II.8.10}$$

且

$$c_{nj}(q,\dot{q})=\sum_{k=1}^{N}c_{njk}(q)\dot{q}_k$$

则有

$$\begin{aligned}\sum_{j=1}^{N}c_{nj}(q,\dot{q})\dot{q}_j&=\sum_{j=1}^{N}\sum_{k=1}^{N}c_{njk}(q)\dot{q}_k q_j\\&=\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{1}{2}\left(\frac{\partial b_{nj}}{\partial q_k}+\frac{\partial b_{nk}}{\partial q_j}-\frac{\partial b_{jk}}{\partial q_n}\right)\dot{q}_k\dot{q}_j\\&=\frac{1}{2}\left(\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{nj}}{\partial q_k}\dot{q}_k\dot{q}_j+\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{nk}}{\partial q_j}\dot{q}_k\dot{q}_j-\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{jk}}{\partial q_n}\dot{q}_k\dot{q}_j\right)\\&=\frac{1}{2}\left(\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{nj}}{\partial q_k}\dot{q}_k\dot{q}_j+\sum_{k=1}^{N}\sum_{j=1}^{N}\frac{\partial b_{nj}}{\partial q_k}\dot{q}_j\dot{q}_k-\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{jk}}{\partial q_n}\dot{q}_k\dot{q}_j\right)\\&=\left(\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{nj}}{\partial q_k}\dot{q}_k\dot{q}_j-\frac{1}{2}\sum_{j=1}^{N}\sum_{k=1}^{N}\frac{\partial b_{jk}}{\partial q_n}\dot{q}_k\dot{q}_j\right)\end{aligned}$$

(注意：第 3 行到第 4 行，对第 2 项使用了交换下标技巧–把上一行的 k 叫做 j，j 叫做 k，不影响这一项的求和，也不影响其他项)

注意到现在的求和即为 $h_{njk}(q)$ 的形式

$$\sum_{j=1}^{N} c_{nj}(q,\dot{q})\dot{q}_j = \sum_{j=1}^{N}\sum_{k=1}^{N}\left(\frac{\partial b_{nj}(q)}{\partial q_k} - \frac{1}{2}\frac{\partial b_{jk}(q)}{\partial q_n}\right)\dot{q}_k\dot{q}_j$$

于是，动力学方程的前两项可写为

$$\frac{d}{dt}\frac{\partial \mathcal{T}}{\partial \dot{q}_i} - \frac{\partial \mathcal{T}}{\partial q_i} = \sum_{j=1}^{N} b_{nj}(q)\ddot{q}_j + \sum_{j=1}^{N} c_{nj}(q,\dot{q})\dot{q}_j$$

将上述构造的 $c_{nj}(q,\dot{q})$ 记为 C 矩阵，即

$$C(q,\dot{q}) = [c_{nj}(q,\dot{q})]_{N\times N} \tag{II.8.11}$$

我们就有理想条件下的动力学方程

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau \tag{II.8.12}$$

在动力学方程的各项中，$B(q)\ddot{q}$ 是惯性项；$C(q,\dot{q})\dot{q}$ 代表科里奥利力，或者牵连力；$g(q)$ 是重力项。更完整的动力学方程为

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f(q,\dot{q})\dot{q} + g(q) = \tau - J^T(q)F_e \tag{II.8.13}$$

在这个版本中，新增了 $F_f(q,\dot{q})\dot{q}$ 代表的摩擦阻力项（包括滑动摩擦力和静摩擦力）以及 $-J^T(q)F_e$ 代表的外力项。F_e 表示关节末端在操作空间受到的广义外力，包含平动力和扭力。

总结前述各式，得到方程中各项的计算方法如下

$$\begin{aligned}
C(q,\dot{q}) &= \left[\sum_{k=1}^{N} c_{njk}(q)\dot{q}_k\right]_{N\times N}\\
c_{njk}(q) &= \frac{1}{2}\left(\frac{\partial b_{nj}(q)}{\partial q_k} + \frac{\partial b_{nk}(q)}{\partial q_j} - \frac{\partial b_{jk}(q)}{\partial q_n}\right)\\
B(q) &= [b_{ij}(q)]_{N\times N} = \sum_{n=1}^{N} m_n J_{p,n}^T(q)J_{p,n}(q) + J_{\theta,n}^T(q)R_n(q)\mathbf{I}_n^n R_n^T(q)J_{\theta,n}(q)\\
g(q) &= \sum_{n=1}^{N} m_n g_0^T J_{p,n}(q)\\
J_{\theta,n}(q) &= \begin{bmatrix} z_0^b & \dots & z_n^b & 0_{3\times1} & \dots & 0_{3\times1}\end{bmatrix}\\
J_{p,n}(q) &= \begin{bmatrix} z_0^b \times p_{0,l_n}^b & \dots & z_{n-1}^b \times p_{n-1,l_n}^b & 0_{3\times1} & \dots & 0_{3\times1}\end{bmatrix}
\end{aligned} \tag{II.8.14}$$

可见，完整的机器人动力学方程计算是比较复杂的。理想条件下的 $B(q), C(q,\dot{q}), g(q)$ 项，均涉及连杆质心的几何雅可比。C 矩阵的计算还需要对雅可比进一步求导。

我们总结上述动力学算法如下

算法	动力学解析解
问题类型	动力学项求解
已知	D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$ 关节向量 q 及其微分 $\dot{q}$ 连杆参数 $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$
求	动力学项 $B(q), C(q, \dot{q}), g(q)$
算法性质	解析解

Algorithm 23: 机器人动力学 (robot_dyn)

Input: 连杆参数 $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$

Input: D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$

Input: 关节向量 q 及其微分 $\dot{q}$

Output: 动力学项 $B(q), C(q, \dot{q}), g(q)$

$p_e, \vartheta, R_n^b \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q)$

$J_{p,n}, J_{\theta,n} \leftarrow \text{robot\_j\_centroid}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q, p_{l_n}^n)$

$B \leftarrow \sum_{n=1}^{N} m_n J_{p,n}^T J_{p,n} + J_{\theta,n}^T R_n^b \mathbf{I}_n^n (R_n^b)^T J_{\theta,n}$

$g \leftarrow \sum_{n=1}^{N} m_n g_0^T J_{p,n}(q)$

for $n \in 1, ..., N$ **do**

  for $j \in 1, ..., N$ **do**

    for $k \in 1, ..., N$ **do**

      $c_{njk} \leftarrow \frac{1}{2}\left(\frac{\partial b_{nj}}{\partial q_k} + \frac{\partial b_{nk}}{\partial q_j} - \frac{\partial b_{jk}}{\partial q_n}\right)$

    $C_{n,j} \leftarrow \sum_{k=1}^{N} c_{njk}\dot{q}_k$

实际使用时，机器人动力学一般使用成熟的代码库进行计算，无需手动求解；其中的求偏导步骤通过自动微分实现。

(参考资料：清华《智能机器人》课件，《机器人学：建模、规划与控制》《机器人学导论》)

8.3 动力学方程性质

对式II.8.11引入的 C 矩阵，有一个非常重要的性质。记

$$N(q, \dot{q}) = \dot{B}(q) - 2C(q, \dot{q}) \tag{II.8.15}$$

考虑其中的一个具体元素，将其展开，有

$$\begin{aligned} n_{ij}(q, \dot{q}) &= \dot{b}_{ij}(q) - 2c_{ij}(q, \dot{q}) \\ &= \dot{b}_{ij}(q) - 2\sum_{k=1}^{N} c_{ijk}(q)\dot{q}_k \\ &= \sum_{k=1}^{N} \frac{\partial b_{ij}}{\partial q_k}\dot{q}_k - 2\sum_{k=1}^{N} \frac{1}{2}\left(\frac{\partial b_{ij}}{\partial q_k} + \frac{\partial b_{ik}}{\partial q_j} - \frac{\partial b_{jk}}{\partial q_i}\right)\dot{q}_k \\ &= -\sum_{k=1}^{N}\left(\frac{\partial b_{ik}}{\partial q_j} - \frac{\partial b_{jk}}{\partial q_i}\right)\dot{q}_k \end{aligned}$$

交换下标，显然有

$$\begin{aligned} n_{ji}(q,\dot{q}) &= -\sum_{k=1}^{N}\left(\frac{\partial b_{jk}}{\partial q_i}-\frac{\partial b_{ik}}{\partial q_j}\right)\dot{q}_k \\ &= -n_{ij}(q,\dot{q}) \end{aligned}$$

即：**矩阵 $N(q,\dot{q})$ 为反对称阵**

$$N(q,\dot{q})+N^T(q,\dot{q})=0 \tag{II.8.16}$$

对于反对称阵，其二次型恒为 0，因为

$$x^TNx=\frac{1}{2}x^TNx+\frac{1}{2}(x^TNx)^T=\frac{1}{2}(x^TNx+x^TN^Tx)=0$$

即

$$x^T(\dot{B}(q)-2C(q,\dot{q}))x=0 \tag{II.8.17}$$

这一性质将在后续章节机器人控制的推导中发挥关键作用。

9 轨迹规划

[主要内容：轨迹规划问题；PRM 算法、RRT 算法、双向 RRT、RRT*]
[本部分内容将在后续版本中更新，敬请期待]
(参考资料：清华《智能机器人》课件, 《机器人学：建模、规划与控制》)

Part III

控制理论基础

10 控制基础知识

机器人的控制是非常关键的问题。在上一部分中，我们介绍了对机器人进行运动学和动力学建模的方法。接下来，我们还需要根据一定的需求、依据一定的理论对其进行控制。在本部分中，我们将介绍基础的控制理论；第 IV 部分将具体介绍机器人控制的基本方法。

在本章中，我们将介绍控制理论基本概念、频域工具、时域工具等。这些工具将在后续章节机器人控制器的设计中发挥关键作用。此外，我们还将介绍倒立摆问题。这一问题贯穿本书的多个部分多个章节，有助于读者思考不同技术之间的区别和联系。

10.1 基本概念

和许多控制教材不同，在本节中，我们将首先介绍反馈、开环与闭环、调节与跟随等控制理论的核心概念。我们认为，对这些概念建立基本但清晰的认识是理解控制理论中其他重要内容的基础。

10.1.1 反馈与闭环

控制，是一种人类的朴素愿望。所谓控制，就是希望某一现实中的对象按照人类希望的方式进行运动和变化。

为实现这一目标，被控对象需要能被外界干预。被控对象能够被干预的量称为**控制量**，也称为**输入量**。控制必有其关注的目标，和控制目标紧密相关的量称为**输出量**。输出量一定可被观测，拥有控制量和输出量的系统称为**受控系统**。

记控制量为 u，输出量为 y，则最简单的受控系统形式为

$$y = f(u)$$

控制的目标，一般表现为希望输出量按照某一预设的方式稳定或变化。这一预设目标称为**参考输入**，一般用 r 表示。当系统的输出和参考输入不相等，就会产生**误差**，记为 e。于是，我们有

$$e = r - u \tag{III.10.1}$$

最朴素的控制目标，就是希望通过调节控制量，使误差变为 0。为此，需要用某种方式计算控制量。这种计算应当考虑参考输入和输出量，我们称之为**控制器**，即

$$u = c(r, y)$$

在这种情况下，我们称控制器和受控系统构成**反馈**回路。反馈是最朴素的控制想法，是一切控制理论的开端。

更多的情况下，我们的控制器会显式地对误差进行运算，即

$$u = c(r, y, e) \tag{III.10.2}$$

引入反馈后，受控系统和控制器同时组成**闭环系统**，即

$$y = f(c(r, y, e))$$

10.1.2 微分方程

在现实世界中，许多受控系统的数学模型不是代数方程，而是微分方程。

例如，在一个有摩擦的斜面上，试图控制一个物体的位置。控制量为对物体施加的力。根据牛顿第二定律，力和加速度成正比，也就是和位置的二阶导数成正比。在这样的情况下，受控系统的实际形式为

$$\dot{y} = f(y, y^{(2)}, ..., u)$$

例如，对上面的摩擦斜面例子，我们可以列出方程

$$\ddot{y} = \frac{F}{m} - g\sin\theta - \mu g\cos\theta\,\mathrm{sgn}(\dot{y})$$

其中 y 表示斜面上物体的位置，F 表示沿斜面的外力。和之前引入的受控系统形式相比，等号左边并不是一阶导。没关系，我们可以通过状态定义将高阶方程转化为一阶方程。记

$$x = \begin{bmatrix} y \\ \dot{y} \end{bmatrix}$$

则有

$$\begin{bmatrix} \dot{y} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y \\ \dot{y} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{F}{m} \end{bmatrix} + \begin{bmatrix} 0 \\ -g\sin\theta - \mu g\cos\theta\,\mathrm{sgn}(\dot{y}) \end{bmatrix}$$

取

$$u = \begin{bmatrix} 0 \\ \frac{F}{m} \end{bmatrix}$$

并记

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

$$g(x) = \begin{bmatrix} 0 \\ -g\sin\theta - \mu g\cos\theta\,\mathrm{sgn}(x[1]) \end{bmatrix}$$

则有

$$\dot{x} = Ax + g(x) + u \tag{III.10.3}$$

事实上，大部分含有高阶项的受控系统微分方程，都可以按照类似的思路进行化简，得到一个含有线性和/或非线性项的一阶微分方程。因此，我们写出一个通用的受控系统微分方程形式

$$\begin{aligned} \dot{x} &= f(x, u) \\ y &= g(x) \end{aligned} \tag{III.10.4}$$

形如式III.10.4的方程，我们称为**开环系统方程**。式中的 x 称为**状态** (状态变量/状态向量)，有时也会强调其时变特性而写成 $x(t)$。事实上，在绝大部分机器人领域的应用场景下，**系统**一词指的就是**描述某个对象所需的状态量及其满足的微分方程**。

类似于上节，我们可以在开环系统方程中引入反馈控制 (式III.10.2)，即构成闭环系统

$$
\begin{aligned}
\dot{x} &= f(x, u) \\
y &= g(x) \\
e &= r - y \\
u &= c(r, y, e)
\end{aligned}
$$

假设 r 的各阶导数已知，则可忽略 x 和 y 的区别，取 $y = x$，则闭环系统可写为

$$
\begin{aligned}
\dot{x} &= f(x, u) \\
e &= r - x \\
u &= c(r, x, e)
\end{aligned}
$$

使用 $x = r - e$ 消去开环系统状态 x，并假设 r 完全已知，我们就会得到

$$\dot{e} = f_c(e) := f(e, c(e)) \tag{III.10.5}$$

这样，我们就有了仅关于系统误差 e 的微分方程。对于此类形式的微分方程，我们称之为**闭环系统方程**。该方程不含额外的输入项，是一种**自治系统**，而非受控系统。此处的误差 e 也称为系统的**误差状态**。

10.1.3 控制目标

前面我们已经介绍了，现实世界中的被控对象往往用微分方程描述，即开环系统；通过反馈控制，可以使开环系统的输出满足某一目标，而这一目标和参考输入密切相关。

具体来说，我们可以考虑两类参考输入。一类参考输入为恒定不变的已知量，记为 r 或 x_d。另一类参考输入为动态变化的已知量，记为 $r(t)$ 或 $x_d(t)$。

对于希望系统输出恒定不变的任务，我们称为**调节**任务。例如，我们希望倒立摆和竖直方向的夹角维持在 $0°$ 或某一特定角度。而对于希望系统输出跟随某个参考信号的任务，我们称为**跟随**任务。例如，我们希望汽车车轮的指向跟随方向盘的运动。

无论是调节问题还是跟随问题，都需要在**保证闭环系统稳定** (详见下节) 的前提下设计控制方法。因此，我们总结这两个问题如下。

问题	**调节控制问题**
问题简述	已知开环系统，设计控制器使输出稳定在参考值，且系统稳定
已知	开环系统 $\dot{x} = f(x, u)$, 参考输入 x_d
求	控制器 $c(e, x_d, x)$

问题	**跟踪控制问题**
问题简述	已知开环系统，设计控制器使输出跟随参考值，且系统稳定
已知	开环系统 $\dot{x} = f(x, u)$, 参考输入 $x_d(t)$
求	控制器 $c(e, x_d, x)$

在实际应用中，调节控制器设计一般相对简单，跟随控制器相对更复杂。调节问题的控制器有时也可用于跟随问题，但性能一般不如跟随控制器，尤其是目标轨迹的导数较大的情况。

除了调节和跟随，**抗扰**也是重要的控制目标。实际系统中可能存在各种扰动，例如系统建模的误差、传感器噪声、非线性耦合项、各类近似忽略的干扰项等等。在机器人中需要精细控制的场景下，干扰往往需要特别处理。

具体来说，含有一般扰动项的开环受控系统方程为

$$\begin{aligned}\dot{x}(t) &= f(x(t), u(t), d(t), t)\\ y(t) &= g(x(t))\end{aligned}$$

闭环系统方程为

$$\begin{aligned}\dot{x} &= f(x, u, d)\\ e &= r - x\\ u &= c(r, x, e)\end{aligned}$$

一般的调节和跟踪控制器都有一定程度的抗扰作用。但无疑扰动会影响收敛速度、跟踪误差等定量指标。如果需要更好的控制效果，则需针对干扰进行精细建模、估计、补偿等，称为抗干扰控制。

10.1.4 基本动态系统分类

动态系统是一切状态随时间变化的系统的总称。正如上文介绍，连续变化的动态系统，一般使用**微分方程**来描述其变化的规律。并且，根据系统是否有可人为控制的输入，动态系统可分为自治系统和受控系统两类。

典型的自治系统是

$$\begin{aligned}\dot{x}(t) &= f(x(t), t)\\ y(t) &= g(x(t))\end{aligned} \tag{III.10.6}$$

而典型的受控系统是

$$\begin{aligned}\dot{x}(t) &= f(x(t), u(t), t)\\ y(t) &= g(x(t))\end{aligned} \tag{III.10.7}$$

这里，我们将系统状态显示写为时间 t 的函数，并且强调了系统微分方程也可含有时间 t。如果确实含有，这样的系统我们称为**时变系统**。而式III.10.4和式III.10.5所表示的系统，则是典型的**时不变系统**。

对于一个系统，一个特定的输入对应的输出，称为这个系统对该输入的**响应**。

如果系统输入 $u(t)$ 和输出 $y(t)$ 均为 1 维，即标量，我们称该系统为**单输入单输出系统** (SISO 系统)。单输入单输出系统往往可以写为单个多阶微分方程的形式

$$f_y(y^{(m-1)}(t), y^{(m-2)}(t), ..., y'(t), y(t), t) = f_u(u^{(n-1)}(t), u^{(n-2)}(t), ..., u'(t), u(t), t)$$

在所有受控系统中，最简单的系统是**线性时不变系统** (LTI 系统)。所谓线性，就是对系统输入进行线性变换，其系统输出也是原输出的相同线性变换。所谓时不变，就是微分方程中不显含 t。即：

$$a_{m-1}y^{(m-1)}(t)+a_{m-2}y^{(m-2)}(t)+...+a_1y'(t)+a_0y(t) = u^{(n-1)}(t)+b_{n-2}u^{(n-2)}(t)+...+b_1u'(t)+b_0u(t) \tag{III.10.8}$$

典型的线性时不变系统例如式III.10.3，不过该系统并不是线性系统。以下是一个线性二阶时不变系统的例子。

$$a_2y''(t) + a_1y'(t) + a_0y(t) = u(t) \tag{III.10.9}$$

在力学和电学的背景下，典型二阶系统的各参数拥有清晰的物理意义。考虑一个光滑面上一维运动的物体，沿其运动方向有外力 F，阻尼系统 (阻力大小和速度成正比，阻尼系数为 c)，弹簧 (劲度系数为 k)。以弹簧原厂为位移 0 点，设位移为输出量 y，我们有

$$m\ddot{y} = F - c\dot{y} - ky$$

即

$$m\ddot{y} + c\dot{y} + ky = F \tag{III.10.10}$$

可以看到，上述二阶时不变系统通式中的 a_0, a_1, a_2 分别对应于劲度系数 k, 阻尼系数 c, 质量 m。

仿照摩擦斜面例子，仍定义

$$x = \begin{bmatrix} y \\ \dot{y} \end{bmatrix}$$

我们有

$$\dot{x} = Ax + Bu$$

其中

$$A = \frac{1}{m}\begin{bmatrix} 0 & m \\ -c & -k \end{bmatrix}, \quad B = \frac{1}{m}\begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad u = \begin{bmatrix} 0 \\ F \end{bmatrix}$$

可以看到，LTI 系统通过定义多阶微分量为状态，可以写成连续时域下的线性形式。这样的表达简洁直观，本章后续还将介绍时域线性系统的更多性质。

10.1.5 PID 控制

如前所述，反馈控制器的通用形式是式III.10.2。实际使用中，并不是所有控制器都需要 r 和 x。需要 r 的部分称为**前馈**，需要 x 的部分称为**内环反馈**。前馈和内环反馈，往往用于使非线性系统线性化。

实际使用中，和误差 e 相关的部分是最重要的反馈部分。该部分的设计往往直接决定闭环系统 $\dot{e} = f_c(e)$ 的性质。无论是调节问题还是跟随问题，都有一种简单而强大的误差反馈控制方法，这就是**比例-积分-微分控制**，即 **PID 控制**.

PID 控制器仅使用误差项、误差项微分、误差项积分的线性组合作为控制量，即

$$u = K_Pe + K_I\int_0^t edt + K_D\dot{e} \tag{III.10.11}$$

有时，可以省略积分项，而仅使用比例微分项，称为 PD 控制

$$u = K_Pe + K_D\dot{e} \tag{III.10.12}$$

在连续场景中使用 PID 控制需要注意：误差微分项需要单独观测，而不能是误差的简单数值微分，否则会造成稳定性和可实现性问题。工程中，一般考虑使用某种滤波器进行观测。

如果在离散场景中使用 PID 控制，则控制量可以写为

$$u(t) = K_p e(t) + K_I \sum_{\tau=0}^{t} e(\tau) + K_D(e(t) - e(t-1)) \tag{III.10.13}$$

对该式进行差分，定义

$$\Delta u(t) := u(t) - u(t-1)$$

可得

$$\Delta u(t) = K_p(e(t) - e(t-1)) + K_I e(t) + K_D(e(t) - 2e(t-1) + e(t-2))$$

即

$$\Delta u(t) = (K_p + K_I + K_D)e(t) - (K_p + 2K_D)e(t-1) + K_D e(t-2) \tag{III.10.14}$$

式III.10.14也称为**增量式 PID**。相比于直接离散化的 PID 控制，增量式 PID 无需记忆此前的误差，具有常数空间复杂度。

PID 控制十分简洁直观，在机器人控制中应用广泛。在许多精度要求不高的场景下，PID 控制即可较好地完成调节和跟踪任务。对于一般的扰动，PID 也能实现抗扰控制。在后续章节中，我们还会继续分析 PID 控制器。

10.2 频域基本工具

在本节中，我们介绍一些频域的基本工具。我们的讨论范围仅限于 SISO 的 LTI 系统。

10.2.1 卷积变换

即使是最简单的受控系统，即 SISO 的 LTI 系统，要想直接研究其性质也不容易。

考虑一个单位冲激函数 $\delta(t)$，定义为

$$\delta(t) = 0, t \neq 0$$
$$\int_{-\infty}^{\infty} \delta(t)dt = 1$$

在连续时间系统中，任何输入可分解为不同时刻的单位冲激函数叠加。

$$u(t) = \int_{-\infty}^{\infty} u(\tau)\delta(t-\tau)d\tau$$

因此，对于 LTI 系统，根据其线性和时不变性，我们可以取其对 $\delta(t)$ 的响应 $h(t)$，则任意输入 $u(t)$ 下的响应 $y(t)$ 可分解为

$$y(t) = \int_{-\infty}^{\infty} u(\tau)h(t-\tau)d\tau$$

这一积分过程也称为**卷积**，即

$$y(t) = u(t) * h(t) := \int_{-\infty}^{\infty} u(\tau)h(t-\tau)d\tau$$

10.2.2 传递函数

即使已经获取系统的单位冲激响应 $h(t)$，对于不同的输入 $u(t)$，还是需要多次计算卷积，来知道不同的 $y(t)$。这样的描述对系统而言既不直观，也不简洁。

因此，我们引入**拉普拉斯变换**，将时域函数 $f(t)$ 写为复频域函数 $F(s)$

$$F(s) = \mathcal{L}(f(t)) := \int_0^{\infty} f(t)e^{-st}dt$$

注意，该积分对于不同的 $f(t)$ 有不同的收敛条件，称为**收敛域**.

对于 SISO 的 LTI 系统，经过拉普拉斯变换后，输入输出的卷积关系就可以变为乘法关系

$$Y(s) = H(s)U(s)$$

即

$$H(s) = \frac{Y(s)}{U(s)}$$

我们把此处的 $H(s)$ 称为系统的传递函数。

对于式III.10.8中的 SISO LTI 系统，其传递函数形式为

$$H(s) = \frac{s^{m-1} + a_{m-2}s^{m-2} + ... + a_1 s + a_0}{b_{n-1}s^{n-1} + ... + b_1 s + b_0}$$

例如，对于式III.10.9中的系统，其传递函数形式为

$$H(s) = a_2 s^2 + a_1 s + a_0$$

10.2.3 反馈与闭环传递函数

对于受控系统 $H(s)$，最简单的反馈控制是**单位负反馈**。即

$$u(t) = e(t) = r(t) - y(t)$$
$$y(t) = u(t) * h(t)$$

此时我们定义开环传递函数 $H_o(s)$ 和闭环传递函数 $H_c(s)$ 为

$$H_o(s) = H(s) = \frac{Y(s)}{U(s)}$$
$$H_c(s) = \frac{Y(s)}{R(s)}$$

经推导，我们有单位负反馈下的传递函数

$$H_c(s) = \frac{H_o(s)}{1 + H_o(s)} = \frac{Y(s)}{U(s) + Y(s)}$$

若反馈控制器为一 LTI 的微分方程 (单位反馈是其特例)，其单位冲激响应为 $c(t)$，即

$$
\begin{aligned}
u(t) = e(t) = r(t) - c(t) * y(t) \\
y(t) = u(t) * h(t)
\end{aligned}
$$

此时有

$$H_c(s) = \frac{H_o(s)}{1 + C(s)H_o(s)}$$

假设开环系统和控制器分子分母均为多项式，即

$$
\begin{aligned}
H_o(s) &= \frac{H_A(s)}{H_B(s)} \\
C(s) &= \frac{C_A(s)}{C_B(s)}
\end{aligned}
$$

则闭环系统传递函数为

$$H_c(s) = \frac{C_B(s)H_A(s)}{C_B(s)H_B(s) + C_A(s)H_A(s)}$$

可以看到，加入反馈控制器 $C(s)$ 后，闭环系统的特征多项式为 $C_B(s)H_B(s) + C_A(s)H_A(s)$，和开环系统的特征多项式并不相同。

10.2.4 零极点和阶数

对 SISO LTI 系统，其传递函数的分子分母均为多项式。我们可以将其进行复因式分解，写为

$$H(s) = \frac{K(s - z_1)(s - z_2)...(s - z_m)}{s^k(s - p_1)(s - p_2)...(s - p_n)}$$

式中，常系数 K 称为增益，分子多项式的根 z_i 称为**零点**，分母多项式的根 p_i 称为**极点**，分母多项式的最高次数 $k + n$ 称为系统的阶数。

对于 SISO LTI 系统，微分方程的解可能包含的项，正是由分母多项式的根决定。相比于分子多项式，分母多项式更能决定一个系统的特性。我们也将分母多项式称为**特征多项式**。令分母多项式为 0，称为**特征方程**。

对于上节所述闭环反馈系统，可以看到：闭环系统的分母多项式和开环系统并不相同。事实上，频域控制的核心就是：通过一定的方法设计控制器 $C(s)$，将开环系统的特征多项式改变为闭环系统的特征多项式，从而改变系统的特性 (极点/阶数/各类指标/...)，达到一定的控制目的。

(参考资料：《控制之美 • 卷 1》《自动控制原理》)

10.2.5 离散与连续

[主要内容：时域连续系统/时域离散系统/频域连续系统/频域离散系统]

[本部分内容将在后续版本中更新，敬请期待]

10.3 时域响应与指标

对于控制系统，可以定义各种指标，对人类关注的各方面性能进行量化。这些指标有些是时域的，有些是频域的。我们首先介绍时域的指标。

10.3.1 单位阶跃响应

在实际系统中，对单位阶跃输入的响应非常重要。单位阶跃输入就是在 $t=0$ 时跳变为 1 的函数，可以看作是单位冲激响应的积分

$$u_{step}(t) = 1(t) := \int_{-\infty}^{t} \delta(\tau) d\tau$$

单位阶跃函数的拉普拉斯变换为 $\frac{1}{s}$

$$U_{step}(s) = \frac{1}{s}$$

系统对单位阶跃输入的响应称为单位阶跃响应。其频域表达式为

$$Y_{step}(s) = \frac{1}{s} H(s)$$

之所以单位阶跃响应非常重要，是因为许多时候我们需要突然改变控制输入。对线性系统而言，复杂的输入就相当于多个不同单位阶跃输入的叠加。通过单位阶跃响应，我们可以直观的在时域定义系统的延迟特性、超调特性等我们关注的量化控制目标。

10.3.2 负反馈与阶跃响应

考虑一个非常简单的微分方程

$$u = \dot{y}$$

该开环系统的其传递函数为

$$H_1(s) = \frac{1}{s}$$

这样的系统在现实中可以找到。例如，对物体施力可以控制速度。取 $u(t)$ 为合作用力，$y(t)$ 为速度。则这个系统就符合上述传递函数。

对于这一系统，我们可知其单位阶跃响应为

$$\begin{aligned} y_{step,1}(t) &= \mathcal{L}^{-1}(Y_{step}(s)) \\ &= \mathcal{L}^{-1}(\frac{1}{s} H_1(s)) \\ &= \mathcal{L}^{-1}(\frac{1}{s^2}) = t \end{aligned}$$

因此，有

$$\lim_{t\to\infty} y_{step,1}(t) = +\infty \tag{III.10.15}$$

即，开环系统的单位阶跃响应最终是发散的。结合前述的单位阶跃响应的意义，这样的开环系统并非理想。

如想要该系统的单位阶跃响应收敛，可考虑加入负反馈。简单起见，我们首先加入单位负反馈，根据上一节的结论，我们有

$$H_{1c}(s) = \frac{1}{s+1}$$

此时对应的微分方程为

$$u - y = \dot{y}$$

再次计算单位阶跃响应，有

$$\begin{aligned} y_{step,1c}(t) &= \mathcal{L}^{-1}(Y_{step,1c}(s)) \\ &= \mathcal{L}^{-1}\left(\frac{1}{s}H_{1c}(s)\right) \\ &= \mathcal{L}^{-1}\left(\frac{1}{s(s+1)}\right) \\ &= 1(t) - e^{-t} \end{aligned}$$

此时，已经有

$$\lim_{t\to\infty} y_{step,1c}(t) = 1$$

我们可以看到，负反馈可以将形如 $\frac{1}{s}$ 的开环系统变为形如 $\frac{1}{s+1}$ 的系统，且使其单位阶跃响应由发散变为收敛。

事实上，我们有更好的方法来计算阶跃响应的最终值。我们需要用到拉普拉斯变换的**终值定理**

$$\lim_{t\to+\infty} f(t) = \lim_{s\to 0+} sF(s) \tag{III.10.16}$$

对任意系统 $H(s)$，在满足终值定理条件时，其单位阶跃响应的终值

$$\begin{aligned} y_{step,c}(+\infty) &= \lim_{s\to 0+} sY_{step,c}(s) \\ &= \lim_{s\to 0+} s\frac{1}{s}H_c(s) \\ &= \lim_{s\to 0+} H_c(s) \end{aligned} \tag{III.10.17}$$

对闭环系统 $H_{1c}(s)$，我们有

$$\begin{aligned} y_{step,1c}(+\infty) &= \lim_{s\to 0+} H_{1c}(s) \\ &= \lim_{s\to 0+} \frac{1}{s+1} = 1 \end{aligned}$$

10.3.3 典型简单系统

上述的两个例子实际上是最简单的两类系统，或者复杂系统中的最简单的组成模块/子系统。这样的子系统也称为”环节”。

对应于微分方程 $u = T\dot{y}$ 的子系统，称为**积分环节**，或一阶积分系统、积分器。其传递函数为

$$H_1(s) = \frac{1}{Ts}$$

其单位阶跃响应为

$$Y_{step,1}(s) = t$$

对应于微分方程 $u - y = T\dot{y}(T > 0)$ 的子系统，称为**惯性环节**，或惰性环节、时延环节、一阶环节。其传递函数为

$$H_2(s) = \frac{1}{Ts + 1} \quad (T > 0)$$

其单位阶跃响应为

$$Y_{step,2}(s) = 1(t) - e^{-\frac{t}{T}} \quad (T > 0)$$

对应于微分方程 $u - y = T^2\ddot{y} + 2\zeta T\dot{y}$ 的子系统，称为**震荡环节**，或惰性环节、时延环节、二阶环节。其传递函数为

$$H_3(s) = \frac{1}{T^2s^2 + 2\zeta Ts + 1}$$

其单位阶跃响应为

$$Y_{step,3}(s) = 1(t) - \frac{1}{\sqrt{1-\zeta^2}} e^{-\frac{\zeta}{T}t} \sin(\omega_d t + \theta)$$

其中，称 ω_d 为**阻尼振荡角频率**，θ 为**阻尼振荡角**，有

$$\omega_d = \frac{1}{T}\sqrt{1-\zeta^2}$$

$$\theta = \arctan\frac{\sqrt{1-\zeta^2}}{\zeta}$$

在以上三种环节中，后两者的单位阶跃响应收敛，前者不收敛。

惯性环节中，我们默认 $T > 0$，因为这样的系统的微分方程的解才是收敛的。如果 $T < 0$，微分方程就会发散。我们称之为**一阶不稳定环节**

$$H_4(s) = \frac{1}{Ts + 1} \quad (T < 0)$$

其单位阶跃响应为

$$Y_{step,4}(s) = 1(t) - e^{-\frac{t}{T}} \quad (T < 0)$$

积分环节的逆，称为**微分环节**或一阶微分子系统，即对应于微分方程 $\dot{u} = Ty$ 的子系统。其传递函数为

$$H_5(s) = \frac{s}{T}$$

其单位阶跃响应为

$$Y_{step,5}(s) = \delta(t)$$

在连续域中，微分环节是无法单独实现的。实际系统中如果需要实现微分环节，会使用其他环节进行近似。

所有复杂的线性系统，都可以由线性环节和以上几种环节，经过串并联而构成。

10.4 稳定性

10.4.1 Lyapunov 稳定性

之前，我们仅介绍性的给出了系统稳定的说明，并指出稳定性是所有控制器设计需要遵守的原则，但并未精确给出系统稳定性的定义。本节中，我们正式给出系统稳定性的定义及 Lyapunov 判据。

注意：本节中，讨论的系统仅限于自治系统。受控系统不在本节讨论范围中。

Lyapunov 稳定性定义

平衡点：x_0 是自治系统 $\dot{x} = f(x)$ 平衡点，当且仅当 $x(0) = x_0$ 时有 $x(t) = x_0, \forall t > 0$

Lyapunov 稳定：假设对于自治系统 $\dot{x} = f(x,t)$, $x = 0$ 是其平衡点，若 $\forall \epsilon > 0, \forall t_0, \exists \delta > 0, s.t. \forall x(t_0) = x_0 \in B(0,\delta)$, 有 $||x(t)|| < \epsilon$，则称自治系统 $\dot{x} = f(x,t)$ 是 Lyapunov 稳定的，或在 Lyapunov 意义下稳定。

渐进稳定：假设对于自治系统 $\dot{x} = f(x,t)$, $x = 0$ 是其平衡点，若 $\forall \epsilon > 0, \forall t_0, \exists \delta > 0, s.t. \forall x(t_0) = x_0 \in B(0,\delta)$, 有 $\lim_{t\to\infty} ||x(t)|| = 0$，则称自治系统 $\dot{x} = f(x,t)$ 是渐进稳定的。

Lyapunov 判据

Lyapunov 稳定判据：假设对于自治系统 $\dot{x} = f(x,t)$, $x = 0$ 是其平衡点。若存在函数 $V(x)$，满足 $V(0) = 0$, 且 $V(x) \geq 0, \forall x$，且 $\forall x(0) = x_0, \dot{x} = f(x,t), \dot{V}(x) \leq 0$，则自治系统 $\dot{x} = f(x,t)$ 是 Lyapunov 稳定的。

渐进稳定判据：假设对于自治系统 $\dot{x} = f(x,t)$, $x = 0$ 是其平衡点。若存在函数 $V(x)$，满足 $V(0) = 0$, 且 $V(x) > 0, \forall x \neq 0$，且 $\forall x(0) = x_0, \dot{x} = f(x,t), \dot{V}(x) < 0$，则自治系统 $\dot{x} = f(x,t)$ 是渐进稳定的。

上述两条判据中的 $V(x)$ 也称为该系统的 Lyapunov 函数。若某系统符合某种稳定性判据，可能同时存在多种 Lyapunov 函数。

Lyapunov 判据又名 Lyapunov 第二法。对其的直观理解是：我们需要找到一个状态空间中的半正定函数 $V(x)$。当从任一初始位置出发，自治系统 $\dot{x} = f(x,t)$ 决定了系统演化的轨迹。如果沿轨迹的 $V(x)$ 是不增的，那么自治系统是 Lyapunov 稳定的。如果函数 $V(x)$ 是正定的，且沿轨迹的 $V(x)$ 是单调下降的，那么自治系统是渐近稳定的。

上述直观理解展现了 Lyapunov 函数 $V(x)$ 的一种特性，即其可被视作一种系统的能量。如果系统能量不增，说明系统可以保持不发散。如果系统能量持续降低，那么其总会回到平衡点 0 点。对于一些有实际背景的系统，Lyapunov 函数可能确实具有物理意义。

(参考资料：《自动控制原理》)

10.5 线性时域工具

现实世界中的很多受控动态系统都是满足 LTI 的 SISO 系统，例如控制某个位移量、某个气体的压力值、某个温度值，等等。前面我们介绍的传递函数及相关控制器设计方法，已经可以较好地满足这类需求。

然而，随着人类科技的进步，对系统控制的需求也变得复杂化。有些时候我们需要同时控制一个系统中的多个目标状态，而非单个目标状态。这些状态往往不是独立的，而是通过微分方程紧密关联。例如，在一个运动系统中，同时使速度、航向、位置跟踪给定的输入。面对这样的需求，传递函数等频域设计工具使用起来就会变得麻烦，且不够直观。

面对这样的需求，我们一般会使用**时域分析工具**进行系统分析与控制器设计。具体来说，我们直接对状态方程进行分析，结合上一节提到的李雅普诺夫稳定性准则，我们也设计了一系列方法和工具来满足指定的需求。

和频域情况一样，在所有的动态系统中，线性系统是最简单的系统。因此，本节主要针对线性系统的描述及性质进行介绍，同时介绍简单的镇定控制器的设计。

10.5.1 线性系统状态方程

前面我们说过，动态系统分为自治系统和受控系统两类，由微分方程描述。这两类系统中，最简单的都是微分方程为线性的情况，即**线性自治系统**和**线性受控系统**。其中，线性自治系统表示为

$$\begin{aligned} \dot{x}(t) &= A(t)x(t) \\ y(t) &= C(t)x(t) \end{aligned} \tag{III.10.18}$$

而线性受控系统表示为

$$\begin{aligned} \dot{x}(t) &= A(t)x(t) + B(t)u(t) \\ y(t) &= C(t)x(t) + D(t)u(t) \end{aligned} \tag{III.10.19}$$

这里我们表现的是时变的线性系统。如果系统的微分方程系数不随时间变化，则称为线性时不变系统 (LTI 系统)。这一概念在频域中已经介绍过。LTI 自治系统表示为

$$\begin{aligned} \dot{x}(t) &= Ax(t) \\ y(t) &= Cx(t) \end{aligned} \tag{III.10.20}$$

而 LTI 受控系统表示为

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned} \tag{III.10.21}$$

在以上系统中，A 矩阵称为**系统矩阵**，反映系统本身的性质；B 矩阵称为**输入矩阵**，反映输入对系统的影响；C 矩阵称为**输出矩阵**，反映系统输出和状态的关系；D 矩阵称为**前馈矩阵**，是系统中输入和输出关系中不依赖微分方程的部分。

10.5.2 线性系统稳定性

对于 LTI 自治系统，我们可以使用上节介绍的 Lyapunov 稳定性进行稳定性判断。我们介绍两种判定 LTI 自治系统 Lyapunov 稳定的工具：**特征值判据**和 **Lyapunov 方程判据**。

特征值判据：对于 LTI 自治系统 $\dot{x} = Ax$，平衡点 $x = 0$ 是 Lyapunov 稳定的，当且仅当矩阵 A 的所有特征值均在复平面左半平面和虚轴上，且虚轴特征值重数不大于 1.

特征值判据成立的关键在于，LTI 自治系统的微分方程 $\dot{x}(t) = Ax(t)$ 的解析解为 $x(t) = \exp(At)x_0$。而 (A 可对角化的情况下) 矩阵指数 $\exp(At)$ 满足

$$\exp(At) = Pexp(\Lambda t)P = P \begin{bmatrix} e^{\lambda_1 t} & & \\ & \ddots & \\ & & e^{\lambda_n t} \end{bmatrix} P^{-1}$$

其中 P 是对角分解时产生的单位矩阵。因此，如果矩阵 A 的所有特征值均在复平面左半平面和虚轴上，且虚轴特征值重数不大于 1，则 $exp(\Lambda t)$ 的每一块都是收敛为 0 的，因此 $x(t)$ 一定有界。

Lyapunov 方程判据：对于 LTI 自治系统 $\dot{x} = Ax$，平衡点 $x = 0$ 是渐进稳定的，当且仅当 $\forall Q > 0, Q^T = Q$，$\exists! P > 0, P^T = P$，满足

$$A^T P + PA = -Q \tag{III.10.22}$$

证明. **充分性：**

对于系统 $\dot{x}=Ax$，考虑如下的二次型 Lyapunov 函数

$$V(x)=\frac{1}{2}x^TPx$$

由于 $P>0$，从而 $V(0)=0$, 且 $V(x)>0,\forall x\neq 0$。此时，Lyapunov 函数的导数为

$$\begin{aligned}\dot{V}(x)&=\dot{x}^TPx+x^TP\dot{x}\\&=x^TA^TPx+x^TPAx\\&=x^T(A^TP+PA)x\\&=-x^TQx<0\end{aligned}$$

根据 Lyapunov 稳定性判据，系统 $\dot{x}=Ax$ 是 Lyapunov 稳定的。

必要性：

对于任意 $Q>0,Q^T=Q$，考虑矩阵函数

$$X(t)=\exp(A^Tt)Q\exp(At)$$

易知 $X(0)=Q$. 对其求导，有

$$\begin{aligned}\dot{X}(t)&=A^T\exp(A^Tt)Q\exp(At)+\exp(A^Tt)Q\exp(At)A\\&=A^TX(t)+X(t)A\end{aligned}$$

对上式由 $t=0$ 到 $t=+\infty$ 进行积分，有

$$X(+\infty)-X(0)=A^T\int_0^{+\infty}X(t)dt+\int_0^{+\infty}X(t)dtA$$

由于系统渐近稳定，$\lim_{t\to+\infty}\exp(At)=0$，则 $\lim_{t\to+\infty}X(t)=0$。因此我们有

$$-Q=A^T\int_0^{+\infty}X(t)dt+\int_0^{+\infty}X(t)dtA$$

设

$$P=\int_0^{+\infty}\exp(A^Tt)Q\exp(At)dt \tag{III.10.23}$$

则 $-Q=A^TP+PA$。由于 $P^T=P,P>0$，该值即为所要求的 P，即这样的 P 是存在唯一的。

□

(参考资料：郑大钟《线性系统理论 (第 2 版)》)

10.5.3 能控性

在本章的第一节，我们给出了反馈控制的最一般形式。通过反馈控制，我们可以将一个受控系统转换为一个自治系统。上节中，我们讨论了自治 LTI 系统的稳定性问题。那么，对于一个给定的受控 LTI 系统 $\dot{x}=Ax+Bu$，我们能否将其转换为一个稳定的自治 LTI 系统呢？

这个问题实际上分成两个部分。首先，我们需要知道，是否所有的 LTI 自治系统都能满足这样的需求？其次，如果能满足这个需求，我们具体应该怎么做？在本小节中，我们首先解决第一个问题，即定义系统的能控性。

状态的能达性：对于受控 LTI 系统 $\dot{x} = Ax+Bu$，若存在一个时刻 $t_1 > 0$ 以及一个控制信号 $u(t), t \in [0, t_1]$，使系统状态可以从 $x(0) = x_0 \neq 0$ 转移到 $x(t_1) = 0$，则称状态 x_0 为能达的。

系统的能控性：对于受控 LTI 系统 $\dot{x} = Ax + Bu$，若任意非 0 状态 x_0 都是能达的，则称系统是能控的。

如何判别一个系统是能控的呢？在此，我们不加证明地给出能控性的秩判据

能控性秩判据：对于受控 LTI 系统 $\dot{x} = Ax + Bu$，系统完全能控当且仅当下列定义的能控性矩阵的秩等于系统状态 x 的维数 n

$$Q_c = \begin{bmatrix} B & AB & \dots & A^{n-1}B \end{bmatrix} \tag{III.10.24}$$

(参考资料：郑大钟《线性系统理论 (第 2 版)》)

10.5.4 镇定控制器

通过能控性判据，我们已经可以预知一个 LTI 受控系统是否有通过反馈变为稳定自治系统的能力。那么，我们具体应该如何设计控制器，才能实现需求呢？

在本章的第一部分，我们介绍了反馈控制器，它将控制信号 u 描述为输出量的函数 $u = c(y)$。事实上，对于状态方程描述的系统，更简单的思路是将 u 作为状态量 x 的函数。对于线性系统，我们一般选择这个函数为线性函数，即

$$u = -Kx \tag{III.10.25}$$

上述关系称为系统的**状态反馈**。事实上，只要系统能控，则必定可以通过状态反馈将其转化为稳定自治系统，即**系统镇定**。记

$$A_a = A - BK \tag{III.10.26}$$

则镇定后的系统为

$$\dot{x} = A_a\dot{x} = (A - BK)x \tag{III.10.27}$$

由线性系统稳定的特征值判据，我们只要找到一个合适的 K，使得 $A_a = A - BK$ 的特征值均在复平面左半平面，就能使闭环系统 Lyapunov 稳定。这种方法称为**极点配置**。

极点配置的原理是：将 A 矩阵线性变换为可控规范型，寻找将规范型极点配置为需求极点的矩阵 $\tilde{K}$，再将 $\tilde{K}$ 通过反变换得到 K。具体原理此处不再赘述，读者可以参考 matlab 的 place 函数文档或其他资料。

(参考资料：《自动控制原理》)

10.6 一阶倒立摆问题

倒立摆是一种经典的非线性受控系统，本书的大量章节将使用一阶倒立摆作为分析对象，在此进行介绍。

10.6.1 问题建模

首先，我们来对一阶倒立摆进行定义。

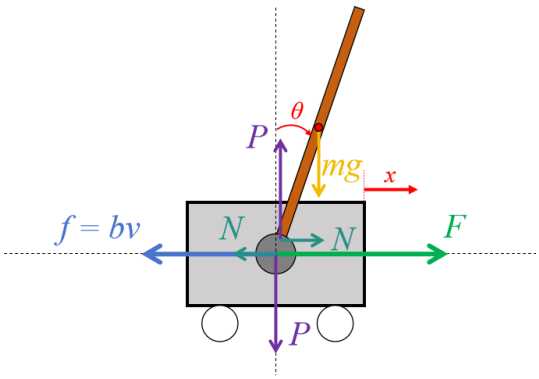

图 III.10.1: 一阶倒立摆

一阶倒立摆：如图 X 所示，平面内有一水平轨道，轨道上有一可滑动的小车，其质量为 M。定义轨道向右为正方向，轨道某位置为原点。在滑动时，小车的位置 x 和速度 $\dot{x}$ 可测，摩擦力 $f = -\mu\dot{x}$。小车上通过一自由旋转关节连接一匀质棒，棒质量为 m_1，棒长度为 $2l_1$。棒和竖直方向的夹角 θ 以及角速度 $\dot{\theta}$ 可测，角度以棒右倾为正。可通过某种方式给小车施加水平方向的连续力 F，限定 $-F_{max} \le F \le F_{max}$。记 $\mathbf{x} = [x, \dot{x}, \theta, \dot{\theta}]$。控制目标：对于一定范围内的系统初始状态 $x_{min} < x(0) < x_{max}$，设计控制器，使系统尽量维持状态 $x(t) = 0$

基于牛顿定律进行分析，可以得到以下方程

$$
\begin{aligned}
M\ddot{x} &= F - \mu\dot{x} - N_x \\
m_1(\ddot{x} + l_1(-\sin\theta\dot{\theta}^2 + \cos\theta\ddot{\theta})) &= N_x \\
-m_1 l_1(\sin\theta\ddot{\theta} + \cos\theta\dot{\theta}^2) &= -m_1 g + N_y \\
\frac{1}{3}m_1 l_1^2\ddot{\theta} &= N_y l_1 \sin\theta - N_x l_1 \cos\theta
\end{aligned}
$$

化简，有

$$
\begin{aligned}
(M + m_1)\ddot{x} - m_1 l_1 \sin\theta\dot{\theta}^2 + m_1 l_1 \cos\theta\ddot{\theta} + \mu\dot{x} &= F \\
\frac{4}{3}l_1\ddot{\theta} - g\sin\theta + \ddot{x}\cos\theta &= 0
\end{aligned}
\tag{III.10.28}
$$

可以看到，这是一个比较复杂的非线性微分方程组。考虑小角度近似 $\theta \to 0$ 时有 $\sin\theta \to \theta, \cos\theta \to 1, \dot{\theta}^2 \to 0$，我们有近似的一阶倒立摆方程

$$
\begin{aligned}
(M + m_1)\ddot{x} + \mu\dot{x} + m_1 l_1\ddot{\theta} &= F \\
\frac{4}{3}l_1\ddot{\theta} - g\theta + \ddot{x} &= 0
\end{aligned}
$$

整理，得

$$
\begin{aligned}
\ddot{x} &= -\mu k_1 x + k_2\theta + k_1 F \\
\ddot{\theta} &= -\mu\dot{x} + k_3\theta + k_4 F
\end{aligned}
\tag{III.10.29}
$$

其中

$$
\begin{aligned}
k_1 &= \frac{4}{4M + m_1} \\
k_2 &= -\frac{3m_1 g}{4M + m_1} \\
k_3 &= \frac{3(M + m_1)g}{l_1(4M + m_1)} \\
k_4 &= -\frac{3}{l_1(4M + m_1)}
\end{aligned}
\tag{III.10.30}
$$

设 $u = F$，写成矩阵形式，我们有

$$
\dot{\mathbf{x}} = A\mathbf{x} + Bu = \begin{bmatrix} 0 & -\mu k_1 & 0 & k_2 \\ 0 & 1 & 0 & 0 \\ -\mu & 0 & 0 & k_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{x} + \begin{bmatrix} k_1 \\ 0 \\ k_4 \\ 0 \end{bmatrix} u \tag{III.10.31}
$$

10.6.2 稳定性和能控性分析

[本部分内容将在后续版本中更新，敬请期待]

10.6.3 PID 控制倒立摆

对于上述倒立摆问题，我们可以用 PID 实现一个简单的控制器，并在仿真环境中验证其性能。
[本部分内容将在后续版本中更新，敬请期待]

11 观测与滤波

11.1 基本概念

观测和滤波是控制理论和工程中两个非常重要的问题。它们有一定的联系，也有区别。本节中我们将具体地定义这两类问题。

11.1.1 状态观测问题

在上一节的各类推导中，尤其是时域线性控制的推导中，我们往往假设系统的状态量已知。然而在现实中，一些实际系统的状态并不容易得知。我们可能只能得到由输出方程定义的输出量/**观测量** $z(t)$[7]。

其连续形式为

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t)) \\ z(t) &= h(x(t), u(t)) \end{aligned} \tag{III.11.1}$$

离散形式为

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) \\ z_k &= h(x_k, u_k) \end{aligned} \tag{III.11.2}$$

状态观测问题，就是指系统在满足一定条件 (能观性) 时，虽然系统的状态变量不可测，但可以基于系统的输出 z、系统状态方程 f、系统输出方程 h，设计一些算法恢复状态变量，并实现闭环误差系统的稳定。

问题	状态观测问题
问题简述	已知系统状态方程和输出方程，设计观测器使观测误差收敛
已知	系统方程 $\dot{x} = f(x, u)$ 观测方程 $z = h(x, u)$ 系统输入输出 u, z
求	观测器 $o(u, z, \hat{x})$

最简单的状态观测问题是线性系统方程和观测方程下的状态观测问题，即**线性状态观测问题**。

问题	线性状态观测问题
问题简述	已知线性系统状态方程和输出方程，设计观测器使观测误差收敛
已知	线性系统方程 $\dot{x} = Ax + Bu$ 线性观测方程 $z = Cx$ 系统输入输出 u, z
求	观测器 $o(u, z, \hat{x})$

11.1.2 滤波问题

理想系统是确定性系统，实际系统可能含有各种**随机噪声**。噪声是一类较小但影响系统内部和输出信号的随机变量。最常见的噪声是**高斯噪声**，其本质是一个 n 维的随机向量 v，满足分布 $v \sim \mathcal{N}(\mu, \Sigma)$，对应的概率密度函数为

[7]滤波领域的习惯是用 $z(t)$ 代表观测量，而控制社区更习惯使用 $y(t)$，本章中我们均采用 $z(t)$

$$p(v;\mu,\Sigma) = \frac{1}{\sqrt{(2\pi)^k|\Sigma|}} \exp\left(-\frac{1}{2}(v-\mu)^T\Sigma^{-1}(v-\mu)\right) \tag{III.11.3}$$

如果高斯噪声的均值 $\mu = 0$，我们就称为**高斯白噪声**，其分布对应的概率密度函数如下

$$p(v;0,\Sigma) = \frac{1}{\sqrt{(2\pi)^k|\Sigma|}} \exp\left(-\frac{1}{2}v^T\Sigma^{-1}v\right) \tag{III.11.4}$$

和干扰类似，噪声是我们希望去除的，这一过程称为**去噪**。由于历史原因，**滤波**一词代表了两种和去噪相关的含义。一方面，滤波代表**去噪信号恢复**，即在已知/部分已知/未知噪声和信号特性的条件下，从有噪声的信号中恢复原始信号。通信、信号处理、图像处理等领域中的滤波一词，多为这一”信号恢复”的含义。另一方面，滤波也代表**去噪状态估计**。即：在有噪声的系统中，根据输出对状态进行观测，尝试恢复状态。

可以看到，后者的意思更接近去噪意义下的状态观测，而前者可以看作是后者的特例[8]。在机器人领域，对滤波一词，我们更为关注后者的含义。**后续本书中所有涉及”滤波”的表述，除特殊说明外，均指去噪状态估计。**

去噪状态估计问题一般在离散时间域中表达，其形式为

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) + w_k \\ z_k &= h(x_k, u_k) + v_k \end{aligned} \tag{III.11.5}$$

即

问题	去噪状态估计（滤波）问题 [离散]
问题简述	已知离散系统状态方程和输出方程，设计滤波器，使状态估计误差小
已知	离散系统 $x_{k+1} = f(x_k, u_k) + w_k$ 观测方程 $z_k = h(x_k, u_k) + v_k$ 系统输入输出 u_k, z_k
求	滤波器 $\hat{x}_k = o(u_k, z_k, \hat{x}_{k-1})$

当系统含有随机噪声时，真实系统状态 x_k、滤波器估计的状态 $\hat{x}_k$ 均为随机变量。如果想衡量滤波的好坏，我们可以考虑定义估计误差

$$e_k = x_k - \hat{x}_k \tag{III.11.6}$$

此时，我们就可以定义“最优”的估计——**平方误差** (期望) 最小

$$\begin{aligned} \min_{o} \quad & \mathbb{E}[e_k^T e_k] \\ s.t. x_{k+1} &= f(x_k, u_k) + w_k \\ z_k &= h(x_k, u_k) + v_k \\ \hat{x}_k &= o(u_k, z_k, \hat{x}_{k-1}) \\ e_k &= x_k - \hat{x}_k \end{aligned} \tag{III.11.7}$$

我们将上述问题总结为最小平方误差意义下的**最优滤波问题**

[8] 即输出量 z 和状态量 x 相同

问题	最优滤波问题 [离散]
问题简述	已知离散系统状态方程和输出方程，设计滤波器，使状态估计误差小
已知	离散系统 $x_{k+1} = f(x_k, u_k) + w_k$ 观测方程 $z_k = h(x_k, u_k) + v_k$ 系统输入输出 u_k, z_k
求	滤波器 $\hat{x}_k = o(u_k, z_k, \hat{x}_k)$ 使得 $\mathbb{E}[e_k^T e_k]$ 最小

上述问题也可以在连续域中进行表示。注意在连续域下，噪声 $w(t), v(t)$ 都是连续时间的白噪声，它们满足

$$\begin{aligned} \mathbb{E}[w(t)] &= 0 \\ \mathbb{E}[w(t)w(\tau)^T] &= Q(t)\delta(t-\tau) \\ \mathbb{E}[v(t)] &= 0 \\ \mathbb{E}[v(t)v(\tau)^T] &= R(t)\delta(t-\tau) \\ \mathbb{E}[w(t)v(\tau)] &= 0 \end{aligned} \tag{III.11.8}$$

其中的 $\delta(\cdot)$ 函数是第10.2节介绍的单位冲激函数。

由此，我们可以总结连续时间的滤波问题如下

问题	去噪状态估计 (滤波) 问题 [连续]
问题简述	已知连续系统状态方程和输出方程，设计滤波器，使状态估计误差小
已知	连续系统 $\dot{x} = f(x, u) + w$ 观测方程 $z = h(x, u) + v$ 系统输入输出 u, z
求	滤波器 $o(u, z, \hat{x})$

和离散域类似，在连续域下，我们也定义平方误差期望意义下的最优滤波问题

$$\begin{aligned} \min_{o} \quad & \mathbb{E}[e(t)^T e(t)] \\ s.t. \dot{x}(t) &= f(x(t), u(t)) + w(t) \\ z(t) &= h(x(t), u(t)) + v(t) \\ \dot{\hat{x}}(t) &= o(u(t), z(t), \hat{x}(t)) \\ e(t) &= x(t) - \hat{x}(t) \end{aligned} \tag{III.11.9}$$

我们将这一最优滤波问题总结如下

问题	最优滤波问题 [连续]
问题简述	已知连续系统状态方程和输出方程，设计滤波器，使状态估计误差小
已知	连续系统 $\dot{x} = f(x, u, w)$ 观测方程 $z = h(x, u) + v$ 系统输入输出 u, z
求	滤波器 $\hat{x} = o(u, z, \hat{x})$ 使得 $\mathbb{E}[e(t)^T e(t)]$ 最小

11.2 线性状态观测器

首先，在无噪声的情况下，我们来考虑线性系统的状态观测问题。

11.2.1 能观性

和前一章类似，在讨论具体的状态观测算法前，我们首先要判断系统是否可观。仍然，我们直接给出两个关键定义

状态的不能观测性：对于 LTI 系统 $\dot{x} = Ax + Bu$，$y = Cx + Du$，若存在某个非零状态 $x_0 \neq 0$，使得对于任意时刻 $t_1 > 0$，当初始状态 $x(0) = x_0$ 时，系统的输出响应 $y(t) \equiv 0$（对所有 $t \in [0, t_1]$），则称状态 x_0 是不能观测的。

系统的完全能观性：对于 LTI 系统 $\dot{x} = Ax + Bu$，$y = Cx + Du$，若系统中不存在任何不能观测的非零状态，即任意非零初始状态 $x_0 \neq 0$ 都会产生非零的输出响应，则称系统是完全能观的。

系统的部分能观性：若系统不是完全能观的，即存在至少一个不能观测的非零状态，则称系统是部分能观的。

和能控性对偶，对于能观性，我们也有秩判据

能观性秩判据：对于 LTI 系统 $\dot{x} = Ax + Bu$，$y = Cx + Du$，系统完全能观当且仅当下列定义的能观性矩阵的秩等于系统状态 x 的维数 n

$$Q_o = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1} \end{bmatrix} \tag{III.11.10}$$

11.2.2 龙伯格观测器

[主要内容：龙伯格观测器；线性解耦原理；输出反馈极点配置]

[本部分内容将在后续版本中更新，敬请期待]

(参考资料：郑大钟《线性系统理论 (第 2 版)》)

11.3 卡尔曼滤波 KF

11.3.1 线性滤波问题

前文已经介绍过去噪状态估计 (滤波) 问题。一般非线性系统的状态估计是比较难的，如果叠加噪声则更加复杂。因此，我们首先考虑简单的线性系统情景，噪声也为加性噪声，即**线性滤波问题**。

问题	线性滤波问题 [离散]
问题简述	已知离散线性系统状态方程和输出方程，设计滤波器，使状态估计误差小
已知	离散线性系统 $x_{k+1} = Ax_k + Bu_k + w_k$ 观测方程 $z_k = Cx_k + v_k$ 系统输入输出 u_k, z_k
求	滤波器 $\hat{x}_k = o(z_k, \hat{x}_{k-1}, u_k)$

随机噪声是滤波问题的难点，不同的噪声有不同的分布特性。在所有噪声中，最简单的噪声是高斯白噪声。例如，我们可以假设系统噪声和输出噪声均满足高斯白噪声

$$\begin{aligned} w_k &\sim \mathcal{N}(0, Q_k) \\ v_k &\sim \mathcal{N}(0, R_k) \end{aligned} \tag{III.11.11}$$

这里的 R_k, Q_k 均为对称正定矩阵。如上所述，在平方误差最小意义下，我们也可以定义**线性最优滤波问题**

问题	**线性最优滤波问题 [离散]**
问题简述	已知离散线性系统状态和输出方程，设计滤波器，使状态估计方差最小
已知	离散线性系统 $x_{k+1} = Ax_k + Bu_k + w_k$ 观测方程 $z_k = Cx_k + v_k$ 系统输入输出 u_k, z_k 系统噪声分布 $w_k \sim \mathcal{N}(0, Q_k)$ 输出噪声分布 $v_k \sim \mathcal{N}(0, R_k)$
求	滤波器 $\hat{x}_k = o(z_k, \hat{x}_{k-1}, u_k)$ 使得 $\mathbb{E}[e_k^T e_k]$ 最小

同样的，我们也可以将这两个问题表达在连续域中

问题	**线性滤波问题 [连续]**
问题简述	已知连续线性系统状态方程和输出方程，设计滤波器，使状态估计误差小
已知	连续线性系统 $\dot{x} = Ax + Bu + w$ 观测方程 $z = Cx + v$ 系统输入输出 u, z
求	滤波器 $\hat{x} = o(z, \hat{x}, u)$

问题	**线性最优滤波问题 [连续]**
问题简述	已知连续线性系统状态和输出方程，设计滤波器，使状态估计方差最小
已知	连续线性系统 $\dot{x} = Ax + Bu + w$ 观测方程 $z = Cx + v$ 系统输入输出 u, z 系统噪声分布 $w(t) \sim \mathcal{N}(0, Q(t))$ 输出噪声分布 $v(t) \sim \mathcal{N}(0, R(t))$
求	滤波器 $\hat{x} = o(z, \hat{x}, u)$ 使得 $\mathbb{E}[e(t)^T e(t)]$ 最小

11.3.2 离散卡尔曼滤波

考虑上述离散系统的线性最优滤波问题。我们定义滤波器的估计状态为 $\hat{x}_k$。根据系统方程，我们可以对 $k+1$ 步的状态做一个先验预测，即中间状态 $\hat{x}_k^-$

$$\hat{x}_k^- = A\hat{x}_k + Bu_k$$

此时，我们定义状态估计误差

$$
\begin{aligned}
e_k &:= x_k - \hat{x}_k \\
e_{k+1}^- &:= x_{k+1} - \hat{x}_k^-
\end{aligned}
\tag{III.11.12}
$$

则有

$$
\begin{aligned}
e_{k+1}^- &= x_{k+1} - \hat{x}_k^- \\
&= Ax_k + Bu_k + w_k - (A\hat{x}_k + Bu_k) \\
&= A(x_k - \hat{x}_k) + w_k \\
&= Ae_k + w_k
\end{aligned}
$$

同时我们定义误差的协方差

$$
\begin{aligned}
P_k &:= \mathbb{E}[e_k e_k^T] \\
P_{k+1}^- &:= \mathbb{E}[(e_{k+1}^-)(e_{k+1}^-)^T]
\end{aligned}
\tag{III.11.13}
$$

则我们有误差协方差间的关系式[9]

$$
\begin{aligned}
P_{k+1}^- &= \mathbb{E}[(e_{k+1}^-)(e_{k+1}^-)^T] \\
&= \mathbb{E}[(Ae_k + w_k)(Ae_k + w_k)^T] \\
&= \mathbb{E}[Ae_k e_k^T A^T] + \mathbb{E}[w_k w_k^T] \\
&= A\mathbb{E}[e_k e_k^T]A^T + \mathbb{E}[w_k w_k^T] \\
&= AP_k A^T + Q_k
\end{aligned}
$$

现在，我们考虑根据观测量 z_k 对中间状态 $\hat{x}_k^-$ 进行修正。根据中间状态 $\hat{x}_k^-$ 和系统输出方程，可以得到预测的输出为 $C\hat{x}_k^-$。将观测量与之做差，可以得到**观测残差** $\tilde{z}_k$，也称为**新息**

$$
\tilde{z}_k = z_k - C\hat{x}_k^- \tag{III.11.14}
$$

我们大胆假设：最优滤波器的状态修正量和观测残差呈线性关系，即

$$
\hat{x}_{k+1} - \hat{x}_k^- = K\tilde{z}_k
$$

这里 K 是表示观测残差和状态修正量之间线性关系的矩阵。因此我们得到残差的递推表达式

$$
\begin{aligned}
e_{k+1} &= x_{k+1} - \hat{x}_{k+1} \\
&= x_{k+1} - \hat{x}_k^- - K\tilde{z}_k \\
&= x_{k+1} - \hat{x}_k^- - K(Cx_k + v_k - C\hat{x}_k^-) \\
&= (I - KC)(x_{k+1} - \hat{x}_k^-) - Kv_k \\
&= (I - KC)e_{k+1}^- - Kv_k
\end{aligned}
$$

因此，残差的递推关系可以总结如下

$$
\begin{aligned}
e_{k+1}^- &= Ae_k + w_k \\
e_{k+1} &= (I - KC)e_{k+1}^- - Kv_k
\end{aligned}
\tag{III.11.15}
$$

[9]此处使用了 w_k 与 e_k 独立的结论

我们继续推导残差的协方差矩阵的递推关系

$$\begin{aligned}
P_{k+1} &= \mathbb{E}[(e_{k+1})(e_{k+1})^T] \\
&= \mathbb{E}[((I-KC)e_{k+1}^- - Kv_k)((I-KC)e_{k+1}^- - Kv_k)^T] \\
&= (I-KC)\mathbb{E}[(e_{k+1}^-)(e_{k+1}^-)^T](I-KC)^T + K\mathbb{E}[v_k v_k^T]K^T \\
&= (I-KC)P_{k+1}^-(I-KC)^T + KR_kK^T
\end{aligned}$$

因此，残差协方差矩阵递推关系也可总结如下

$$\begin{aligned}
P_{k+1}^- &= AP_kA^T + Q_k \\
P_{k+1} &= (I-KC)P_{k+1}^-(I-KC)^T + KR_kK^T
\end{aligned} \tag{III.11.16}$$

由于我们想求解最优滤波器，我们关注的是优化目标 $\mathbb{E}[e_k^T e_k]$

$$\begin{aligned}
\mathbb{E}[e_{k+1}^T e_{k+1}] &= \mathrm{tr}(\mathbb{E}[e_{k+1}e_{k+1}^T)] = \mathrm{tr}(\mathbb{E}[P_{k+1}]) \\
&= \mathrm{tr}((I-KC)P_{k+1}^-(I-KC)^T + KR_kK^T) \\
&= \mathrm{tr}(P_{k+1}^- - KCP_{k+1}^- - P_{k+1}^-C^TK^T + K(CP_{k+1}^-C^T + R_k)K^T) \\
&= \mathrm{tr}(V(K))
\end{aligned}$$

可以看到，我们将优化目标表达为了 K 的函数。使该函数最小的 K_k 就是我们所求的最优滤波对应的 K，即

$$K_k = \min_K \mathrm{tr}(V(K))$$

$\mathrm{tr}(V(K))$ 是关于变量 K 的二次型，对 K 二阶可导，且存在全局唯一极小值，当取该极小值时，一阶导数为 0，即

$$\left.\frac{\partial \mathrm{tr}(V(K))}{\partial K}\right|_{K_k} = 0$$

由于

$$\frac{d\mathrm{tr}(BAC)}{dA} = B^TC^T$$

有

$$\begin{aligned}
\frac{\partial}{\partial K}\mathrm{tr}(K(CP_{k+1}^-C^T + R_k)K^T) &= 2K(CP_{k+1}^-C^T + R_k) \\
\frac{\partial}{\partial K}\mathrm{tr}(-KCP_{k+1}^- - P_{k+1}^-C^TK^T) &= -2P_{k+1}^-C^T
\end{aligned}$$

因此，最优滤波器的滤波增益 K_k 需要满足如下线性方程

$$2K_k(CP_{k+1}^-C^T + R_k) - 2P_{k+1}^-C^T = 0$$

整理得

$$K_k(CP_{k+1}^-C^T + R_k) = P_{k+1}^-C^T$$

即

$$K_k = P_{k+1}^- C^T (C P_{k+1}^- C^T + R_k)^{-1} \tag{III.11.17}$$

整理上述公式，我们有最优滤波算法。该算法也称为**卡尔曼滤波** (Kalman Filter, KF)，K_k 称为**卡尔曼增益**。卡尔曼滤波的核心为五个公式

$$\begin{aligned}
\hat{x}_k^- &= A\hat{x}_k + Bu_k \\
P_{k+1}^- &= AP_kA^T + Q_k \\
K_k &= P_{k+1}^- C^T (C P_{k+1}^- C^T + R_k)^{-1} \\
\hat{x}_{k+1} &= \hat{x}_k^- + K_k(z_k - C\hat{x}_k^-) \\
P_{k+1} &= K_k R_k K_k^T + (I - K_kC)P_{k+1}^-(I - K_kC)^T
\end{aligned} \tag{III.11.18}$$

我们将离散卡尔曼滤波的完整算法整理如下

算法	**离散卡尔曼滤波**
问题类型	线性最优滤波问题 [离散]
已知	离散线性系统 $x_{k+1} = Ax_k + Bu_k + w_k$ 观测方程 $z_k = Cx_k + v_k$ 系统输入输出 u_k, z_k 系统噪声分布 $w_k \sim \mathcal{N}(0, Q_k)$ 输出噪声分布 $v_k \sim \mathcal{N}(0, R_k)$
求	滤波器 $o(z, \hat{x}, u)$ 使得 $\mathbb{E}[\hat{x}]$ 最小
算法性质	滤波器，递推

Algorithm 24: 离散卡尔曼滤波

Input: 线性系统矩阵 A, B, C

Input: 对称正定矩阵列 Q_k, R_k，对称正定矩阵 P_0

Input: 系统输入序列 u_k，系统输出序列 z_k

Input: 系统初值 $\hat{x}_0$

Output: 估计状态 $\hat{\mathbf{x}}$

for $k \in 0, 1, \ldots, N_K$ **do**

- $\hat{x}_k^- \leftarrow A\hat{x}_k + Bu_k$
- $P_{k+1}^- \leftarrow AP_kA^T + Q_k$
- $K_k \leftarrow P_{k+1}^- C^T (C P_{k+1}^- C^T + R_k)^{-1}$
- $\hat{x}_{k+1} \leftarrow \hat{x}_k^- + K_k(z_k - C\hat{x}_k^-)$
- $P_{k+1} \leftarrow K_k R_k K_k^T + (I - K_kC)P_{k+1}^-(I - K_kC)^T$

对应的 python 代码如下所示

```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 连续卡尔曼滤波

对于连续系统的线性最优滤波问题，我们可以将优化问题写为

$$\begin{aligned} &\min_{o} \mathbb{E}[e(t)^T e(t)] \\ &s.t. \dot{x} = Ax + Bu + w \\ &z = Cx + v \\ &\hat{x} = o(u, z, \hat{x}) \\ &e(t) = x - \hat{x} \end{aligned} \tag{III.11.19}$$

仿照离散情况，我们假设最优滤波器仍为新息更新的形式

$$\dot{\hat{x}} = A\hat{x} + Bu + K(z - C\hat{x}) \tag{III.11.20}$$

因此我们有误差微分方程

$$\begin{aligned} \dot{e}(t) &= \dot{x}(t) - \dot{\hat{x}}(t) \\ &= Ax(t) + Bu(t) + w(t) - (A\hat{x}(t) + Bu(t) + K(Cx(t) + v(t) - C\hat{x}(t))) \\ &= A(x(t) - \hat{x}(t)) - KC(x(t) - \hat{x}(t)) + w(t) - Kv(t) \\ &= (A - KC)(x(t) - \hat{x}(t)) + w(t) - Kv(t) \\ &= (A - KC)e(t) + w(t) - Kv(t) \end{aligned}$$

假设误差协方差为 P，即

$$P(t) = \mathbb{E}[e(t)e(t)^T] \tag{III.11.21}$$

则 $P(t)$ 的微分方程为[10]

[10]此处忽略了二阶项中和 P 相关的部分

$$
\begin{aligned}
\dot{P}(t) &= \mathbb{E}[\dot{e}(t)e(t)^T + e(t)\dot{e}(t)^T + \dot{e}(t)\dot{e}(t)^T] \\
&= \mathbb{E}[(A-KC)e(t)e(t)^T] + \mathbb{E}[e(t)((A-KC)e(t))^T] + \mathbb{E}[(w(t)-Kv(t))(w(t)-Kv(t))^T] \\
&= (A-KC)P(t) + P(t)(A-KC)^T + Q(t) + KR(t)K \\
&= -KCP(t) - P(t)C^TK + KR(t)K + AP(t) + P(t)A^T + Q(t) \\
&= V(K) + AP(t) + P(t)A^T + Q(t)
\end{aligned}
$$

为最小化 $\mathbb{E}[e(t)^Te(t)]$，我们有如下等价关系

$$
\begin{aligned}
\min_K \mathbb{E}[e(t)^Te(t)] &= \min_K \mathbb{E}[\mathrm{tr}(e(t)e(t)^T)] \\
&= \min_K \mathbb{E}[\mathrm{tr}(P(t))] \\
&= \min_K \mathbb{E}[\mathrm{tr}(\dot{P}(t))] \\
&= \min_K \mathbb{E}[\mathrm{tr}(V(K))]
\end{aligned}
$$

其中我们用到了 $\dot{P}(t) > 0$ 的结论。因此，最优的增益 K 是满足下列偏导为 0 的 K

$$
\frac{\partial V}{\partial K} = \frac{\partial}{\partial K}(-KCP(t) - P(t)C^TK + KR(t)K) = 0
$$

即

$$
-2P(t)C^T + 2KR(t) = 0
$$

因此

$$
K = P(t)C^TR^{-1}(t) \tag{III.11.22}
$$

此时

$$
\begin{aligned}
V(K) &= -KCP(t) - P(t)C^TK + KR(t)K^T \\
&= -2P(t)C^TR^{-1}(t)CP(t) + P(t)C^TR^{-1}(t)CP(t) \\
&= -P(t)C^TR^{-1}(t)CP(t)
\end{aligned} \tag{III.11.23}
$$

因此有

$$
\dot{P}(t) = -P(t)C^TR^{-1}(t)CP(t) + AP(t) + P(t)A^T + Q(t) \tag{III.11.24}
$$

综上，我们总结连续卡尔曼滤波算法的公式如下

$$
\begin{aligned}
\dot{\hat{x}} &= A\hat{x} + Bu + K(z - C\hat{x}) \\
\dot{P}(t) &= -P(t)C^TR^{-1}(t)CP(t) + AP(t) + P(t)A^T + Q(t) \\
K &= P(t)C^TR^{-1}(t)
\end{aligned} \tag{III.11.25}
$$

在实际应用中，需要给定两个微分方程的初值 $\hat{x}(0)$ 和 $P(0)$，随后不断求解这两个微分方程。

算法	连续卡尔曼滤波
问题类型	线性最优滤波问题 [连续]
已知	连续线性系统 $\dot{x} = Ax + Bu + w$ 观测方程 $z = Cx + v$ 系统输入输出 u, z 系统噪声分布 $w(t) \sim \mathcal{N}(0, Q(t))$ 输出噪声分布 $v(t) \sim \mathcal{N}(0, R(t))$
求	滤波器 $o(z, \hat{x}, u)$ 使得 $\mathbb{E}[\hat{x}]$ 最小
算法性质	滤波器，递推

Algorithm 25: 连续卡尔曼滤波

Input: 线性系统矩阵 A, B, C

Input: 对称正定矩阵函数 $Q(t), R(t)$，对称正定矩阵 $P(0)$

Input: 系统输入序列 u_k，系统输出序列 z_k

Input: 系统初值 $\hat{x}(0)$

Output: 估计状态 $\hat{\mathbf{x}}$

$\dot{\hat{x}} = A\hat{x} + Bu + K(z - C\hat{x})$

$\dot{P}(t) = -P(t)C^T R^{-1}(t)CP(t) + AP(t) + P(t)A^T + Q(t)$

$K = P(t)C^T R^{-1}(t)$

11.3.4 扩展卡尔曼滤波 EKF

上小节中，我们介绍了对线性滤波问题方差最小的卡尔曼滤波方法。

对于非线性的滤波问题，噪声协方差的变化会非常复杂，卡尔曼滤波无法处理。具体来说，状态方程 f 和观测方程 h 都为滤波带来了难度。然而，由于卡尔曼滤波是迭代更新的，当采样时间足够小，每一时间步中的状态变化就比较小。此时，在该状态附近，非线性的 f 和 h 就可以通过一阶泰勒展开的方式进行线性化，将非线性的方差变化近似为线性系统的变化，从而继续使用卡尔曼滤波的方法。

具体来说，假设离散系统方程为

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) \\ z_{k+1} &= h(x_k) \end{aligned} \tag{III.11.26}$$

假设

$$\begin{aligned} F_k &= \frac{\partial f}{\partial x}(x_k) \\ H_k &= \frac{\partial h}{\partial x}(x_k) \end{aligned} \tag{III.11.27}$$

则上述方程可以近似为

$$\begin{aligned} x_{k+1} &\approx f(x_k, u_k) + F_k(x_{k+1} - x_k) \\ z_{k+1} &\approx H_k x_k \end{aligned} \tag{III.11.28}$$

即：卡尔曼滤波中的 A, B, C 可以被系统一阶导数 F_k, B_k, H_k 近似。因此，我们有**扩展卡尔曼滤波** (Extended Kalman Filter, EKF) 算法。

$$
\begin{aligned}
\hat{x}_k^- &= f(\hat{x}_k, u_k) \\
P_{k+1}^- &= F_k P_k F_k^T + Q_k \\
K_k &= P_{k+1}^- H_k^T (H_k P_{k+1}^- H_k^T + R_k)^{-1} \\
\hat{x}_{k+1} &= \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, u_k)) \\
P_{k+1} &= K_k R_k K_k^T + (I - K_k H_k) P_{k+1}^- (I - K_k H_k)^T
\end{aligned}
\tag{III.11.29}
$$

EKF 是 KF 在非线性系统的扩展。相比 KF，EKF 的预测和更新使用非线性方程，而方差更新、卡尔曼增益求解仍使用线性化的矩阵。

Algorithm 26: 扩展卡尔曼滤波

Input: 可微系统和输出方程 f, h
Input: 对称正定矩阵列 Q_k, R_k，对称正定矩阵 P_0
Input: 系统输入序列 u_k，系统输出序列 z_k
Input: 系统初值 $\hat{x}_0$
Output: 估计状态 $\hat{\mathbf{x}}$
for $k \in 0, 1, \ldots, K$ **do**
$\hat{x}_k^- \leftarrow f(\hat{x}_k, u_k)$
$F_k \leftarrow \frac{\partial f}{\partial x}(\hat{x}_k)$
$P_{k+1}^- \leftarrow F_k P_k F_k^T + Q_k$
$H_k \leftarrow \frac{\partial h}{\partial x}(\hat{x}_k)$
$K_k \leftarrow P_{k+1}^- H_k^T (H_k P_{k+1}^- H_k^T + R_k)^{-1}$
$\hat{x}_{k+1} \leftarrow \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, u_k))$
$P_{k+1} \leftarrow K_k R_k K_k^T + (I - K_k H_k) P_{k+1}^- (I - K_k H_k)^T$

算法	**扩展卡尔曼滤波**
问题类型	最优滤波问题 [离散]
已知	离散系统 $x_{k+1} = f(x_k, u_k, w_k)$ 观测方程 $z_k = h(x_k, u_k, v_k)$ 系统输入输出 u_k, z_k 系统噪声分布 $w_k \sim \mathcal{N}(0, Q_k)$ 输出噪声分布 $v_k \sim \mathcal{N}(0, R_k)$
求	滤波器 $\hat{x} = o(u, z, \hat{x})$ 使得 $\mathrm{Var}[\hat{x}]$ 最小
算法性质	滤波器，递推

对应的 python 代码如下所示

```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 误差状态卡尔曼滤波 ESKF

对于很多非线性问题，扩展卡尔曼滤波都是一种不错的方法。然而，在状态自身受到约束的情况下，这种方法可能造成一些问题。

例如，机器人中常见的需求是要表示末端/本体的姿态。姿态一般使用 9 个元素的旋转矩阵 R 或 4 个元素的四元数 q 表示。然而，3 维旋转只有 3 个自由度，冗余的维数意味着存在归一化约束，即 $R^TR = I$ 或 $||q|| = 1$。

在这种情况下，使用上述 EKF 方法，计算对状态的导数 $\frac{\partial f}{\partial x}, \frac{\partial h}{\partial x}$ 并利用其进行状态更新，可能会造成破坏归一化、收敛较慢等问题。

事实上，在2.5节中，我们已经讨论过这一问题，并给出了利用旋转向量小扰动 $\Delta\phi$ 代替 ΔR 的方法 (Δq 同理)。其本质上是利用切空间的扰动代替流形上的扰动，可以看作是流形自身的一种线性化。

更一般地，针对滤波问题中 x 可能为流形、存在自身约束的情况，我们可以使用其切空间上的扰动 δx 作为滤波优化的变量。δx 在滤波问题中可以视为估计量和真值的误差。使用这样的思想构成的滤波方法，我们称为**误差状态卡尔曼滤波** (Error State Kalman Filter, ESKF)。其中，估计状态亦称为**标称状态** (Nominal State)。

具体来说，我们认为系统存在真实状态 x，而滤波算法能得到的仅仅是估计值 $\hat{x}$。x 与 $\hat{x}$ 是某个流形上的点，而且相隔很近。在 $\hat{x}$ 附近的流形切平面上 (如旋转矩阵对应的旋转矢量) 可以定义一个扰动 δx，我们希望找到满足下列关系的 δx：

$$x = \hat{x} \oplus \delta x \tag{III.11.30}$$

此处的符号 $\oplus$ 仅表示流形上的元素和切空间元素的运算，其结果仍为流形上的元素；不存在交换律。对于旋转来说，$\oplus$ 意味着左扰动近似或右扰动近似 (是两种不同的 $\oplus$，不可混用)，例如

$$
\begin{aligned}
R \oplus \Delta\phi &= \exp((\Delta\phi)_\times) R \quad \text{(left disturbance)} \\
R \oplus \Delta\phi &= R \exp((\Delta\phi)_\times) \quad \text{(right disturbance)} \\
q \oplus \Delta\theta &= \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} \otimes q \quad \text{(left disturbance)} \\
q \oplus \Delta\theta &= q \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} \quad \text{(right disturbance)}
\end{aligned} \tag{III.11.31}
$$

进而，我们可以定义函数 f, h 对误差状态的导数

$$
\begin{aligned}
\frac{\partial f}{\partial \delta x} &:= \lim_{\delta x \to 0} \frac{f(x \oplus \delta x) - f(x)}{\delta x} \\
\frac{\partial h}{\partial \delta x} &:= \lim_{\delta x \to 0} \frac{h(x \oplus \delta x) - h(x)}{\delta x}
\end{aligned} \tag{III.11.32}
$$

这样，假设 δx 满足高斯分布。我们在卡尔曼滤波中用新息计算出的修正步更新量，其实就是 δx_k

$$
\delta x_k = K_k(z_k - h(\hat{x}_k^-, u_k)) \tag{III.11.33}
$$

最终，我们可以将 δx_k 更新到标称状态 $\hat{x}$ 中。

$$
\hat{x}_{k+1} \leftarrow \hat{x}_k^- \oplus \delta x_k \tag{III.11.34}
$$

使用上述方法分别替代 EKF 中的 $\frac{\partial f}{\partial x}, \frac{\partial h}{\partial x}$，我们总结 ESKF 算法如下。

算法	**误差状态卡尔曼滤波**
问题类型	最优滤波问题 [离散]
已知	离散系统 $x_{k+1} = f(x_k, u_k, w_k)$ 观测方程 $z_k = h(x_k, u_k, v_k)$ 系统输入输出 u_k, z_k 系统噪声分布 $w_k \sim \mathcal{N}(0, Q_k)$ 输出噪声分布 $v_k \sim \mathcal{N}(0, R_k)$
求	滤波器 $\hat{x} = o(u, z, \hat{x})$ 使得 $\mathrm{Var}[\hat{x}]$ 最小
算法性质	滤波器，递推

Algorithm 27: 误差状态卡尔曼滤波

Input: 可微系统和输出方程 f, h

Input: 对称正定矩阵列 Q_k, R_k，对称正定矩阵 P_0

Input: 系统输入序列 u_k，系统输出序列 z_k

Input: 系统初值 $\hat{x}_0$

Output: 估计状态 $\hat{\mathbf{x}}$

for $k \in 0, 1, \ldots, K$ **do**

$\hat{x}_k^- \leftarrow f(\hat{x}_k, u_k)$

$F_k \leftarrow \frac{\partial f}{\partial \delta x}(\hat{x}_k)$

$P_{k+1}^- \leftarrow F_k P_k F_k^T + Q_k$

$H_k \leftarrow \frac{\partial h}{\partial \delta x}(\hat{x}_k)$

$K_k \leftarrow P_{k+1}^- H_k^T (H_k P_{k+1}^- H_k^T + R_k)^{-1}$

$\delta x_k \leftarrow K_k(z_k - h(\hat{x}_k^-))$

$\hat{x}_{k+1} \leftarrow \hat{x}_k^- \oplus \delta x_k$

$P_{k+1} \leftarrow K_k R_k K_k^T + (I - K_k H_k) P_{k+1}^- (I - K_k H_k)^T$

对应的 python 代码如下所示

```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
```

事实上，对绝大部分以三轴姿态为状态的滤波问题 (如 VIO 滤波，详见第 VII 部分) 来说，实际使用的都是 ESKF 方法。

11.4 无迹卡尔曼滤波 UKF

上节中，我们已经介绍了针对非线性系统的 EKF 和 ESKF 方法。这些方法都是非线性系统的不同形式的一阶泰勒展开，无法保证高阶统计量的准确性，且在强非线性情况下性能会明显下降。

事实上，非线性系统滤波困难这一问题的本质是：在 f 和 h 高度非线性的情况下，随机变量 $f(x)$ 服从的近似高斯分布与 $N(f(\bar{x}), F^T P F + Q)$ 相差比较大，导致预测步误差较大 ($h(x)$ 与修正步同理)。而如果将一阶

泰勒展开变为高阶泰勒展开，计算量又将大大增加。那么，能否在不进行复杂的高阶泰勒展开计算的情况下，仍然保持估计状态和真实状态的高阶统计量无偏估计呢？

11.4.1 无迹变换

事实上，我们可以通过采样的方法进行随机变量函数期望/方差的估计。具体来说，假设 $\mathbf{x} \sim \mathcal{N}(\bar{\mathbf{x}}, P) \in \mathbb{R}^n$，我们如下定义一组**采样点**

$$\begin{aligned}
\chi_0 &= \bar{\mathbf{x}} \\
\chi_i &= \bar{\mathbf{x}} + \mathbf{p}_i, \quad i = 1, ..., n \\
\chi_{i+n} &= \bar{\mathbf{x}} - \mathbf{p}_i, \quad i = 1, ..., n
\end{aligned}$$

假设 $\sqrt{A}$ 表示对矩阵 A 求平方根，有

$$\sqrt{(n+\lambda)P} = \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \dots & \mathbf{p}_n \end{bmatrix}$$

可以看到，这组 $2n+1$ 个采样点是在 $\mathbf{x}$ 所在的空间内，在 $\bar{\mathbf{x}}$ 附近进行的采样。

定义采样点权重

$$\begin{aligned}
w_0^m &= \frac{\lambda}{n+\lambda} \\
w_0^c &= \frac{\lambda}{n+\lambda} + (1 - \alpha^2 + \beta) \\
w_i^c &= w_i^m = \frac{1}{2(n+\lambda)}, \quad i = 1, ..., 2n
\end{aligned}$$

则这些采样点满足

$$\begin{aligned}
\bar{\mathbf{x}} &= \sum_{i=0}^{2n} w_i^m \chi_i \\
P &= \sum_{i=0}^{2n} w_i^c (\chi_i - \bar{\mathbf{x}})^T (\chi_i - \bar{\mathbf{x}})
\end{aligned}$$

简单进行一下推导。对期望结论，有

$$\begin{aligned}
&\sum_{i=0}^{2n} w_i^m \chi_i \\
&= \frac{\lambda}{n+\lambda}\bar{\mathbf{x}} + \sum_{i=1}^{n} \frac{1}{2(n+\lambda)}(\bar{\mathbf{x}} + \mathbf{p}_i) + \sum_{i=1}^{n} \frac{1}{2(n+\lambda)}(\bar{\mathbf{x}} - \mathbf{p}_i) \\
&= \bar{\mathbf{x}}
\end{aligned}$$

对方差结论，有

$$
\begin{aligned}
&\sum_{i=0}^{2n} w_i^c(\chi_i-\bar{\mathbf{x}})^T(\chi_i-\bar{\mathbf{x}}) \\
&=\sum_{i=0}^{n} w_i^c(\chi_i-\bar{\mathbf{x}})^T(\chi_i-\bar{\mathbf{x}})+\sum_{i=n+1}^{2n} w_i^c(\chi_i-\bar{\mathbf{x}})^T(\chi_i-\bar{\mathbf{x}}) \\
&=\frac{1}{2(n+\lambda)}\sum_{i=0}^{n} p_i p_i^T+\frac{1}{2(n+\lambda)}\sum_{i=0}^{n} p_i p_i^T \\
&=\frac{1}{(n+\lambda)}\begin{bmatrix}\mathbf{p}_1 & \mathbf{p}_2 & \cdots & \mathbf{p}_n\end{bmatrix}\begin{bmatrix}\mathbf{p}_1 & \mathbf{p}_2 & \cdots & \mathbf{p}_n\end{bmatrix}^T \\
&=\frac{1}{(n+\lambda)}\sqrt{(n+\lambda)P}\sqrt{(n+\lambda)P} \\
&=P
\end{aligned}
$$

也就是说，向量组 $\chi_{0:2n}$ 是 $\bar{\mathbf{x}}$ 附近的一组点，每个点都意味着对 $\bar{\mathbf{x}}$ 在 $\sqrt{P}$ 的某一特征向量方向 (双向) 上施加的扰动。由于这 $2n+1$ 个点覆盖了 $\bar{\mathbf{x}}$ 周围的局部空间，我们认为，可以用这组点的统计量代表 $\bar{\mathbf{x}}$ 附近的性质。即：对对非线性函数 f，则随机变量 $f(\mathbf{x})$ 的期望和方差估计值分别为

$$
\begin{aligned}
E[f(\mathbf{x})] &\approx \sum_{i=0}^{2n} w_i^m f(\chi_i) \\
Var[f(\mathbf{x})] &\approx \sum_{i=0}^{2n} w_i^c(f(\chi_i)-E[f(\mathbf{x})])(f(\chi_i)-E[f(\mathbf{x})])^T
\end{aligned}
$$

上述表达式中，$\lambda, \alpha, \beta, \kappa$ 为参数，其关系为

$$
\lambda=\alpha^2(n+\kappa)-n
$$

11.4.2 滤波算法

通过如上的近似计算方法，对于非线性系统滤波问题，我们就可以在不求导的情况下，近似计算预测步 (x_k^-, P_{k+1}^-) 与修正步 (x_{k+1}, P_{k+1})。

具体来说，对于预测步，假设

$$
\begin{aligned}
\chi_0 &= x_k \\
\chi_i &= x_k+\mathbf{p}_i, \quad i=1,...,n \\
\chi_{i+n} &= x_k-\mathbf{p}_i, \quad i=1,...,n \\
\sqrt{(n+\lambda)P_k} &= \begin{bmatrix}\mathbf{p}_1 & \mathbf{p}_2 & \cdots & \mathbf{p}_n\end{bmatrix}
\end{aligned}
\tag{III.11.35}
$$

且

$$
\begin{aligned}
w_0^m &= \frac{\lambda}{n+\lambda} \\
w_0^c &= \frac{\lambda}{n+\lambda}+(1-\alpha^2+\beta) \\
w_i^c &= w_i^m=\frac{1}{2(n+\lambda)}, \quad i=1,...,2n
\end{aligned}
\tag{III.11.36}
$$

我们有对 $f(x_k)$ 的均值和方差的估计

$$
\begin{aligned}
x_k^- &= \sum_{i=0}^{2n} w_i^m f(\chi_i) \\
P_{k+1}^- &= \sum_{i=0}^{2n} w_i^c (f(\chi_i) - x_k^-)(f(\chi_i) - x_k^-)^T + Q_k
\end{aligned}
\tag{III.11.37}
$$

对于修正步，首先我们要计算每个采样点的预测观测

$$
\gamma_i = h(\chi_i), \quad i = 0, ..., 2n \tag{III.11.38}
$$

因此

$$
\hat{z}_k = \sum_{i=0}^{2n} w_i^m \gamma_i \tag{III.11.39}
$$

接下来，我们需要计算卡尔曼增益。在 EKF 中，卡尔曼增益和 H_k, P_{k+1}^-, R_k 有关，其中 H_k 是线性化后的 h，也就是估计观测和估计状态之间的线性关系。在 UKF 中，这个矩阵被表述为随机变量 z 和随机变量 x 之间的协方差矩阵。该矩阵也可以由采样点估计

$$
H_k = \sum_{i=0}^{2n} w_i^c (\gamma_i - \hat{z}_k)(f(\chi_i) - x_k^-)^T \tag{III.11.40}
$$

综合上述推导，我们总结无迹卡尔曼滤波算法如下

算法	**无迹卡尔曼滤波**
问题类型	最优滤波问题 [离散]
已知	离散系统 $x_{k+1} = f(x_k, u_k, w_k)$ 观测方程 $z_k = h(x_k, u_k, v_k)$ 系统输入输出 u_k, z_k 系统噪声分布 $w_k \sim \mathcal{N}(0, Q_k)$ 输出噪声分布 $v_k \sim \mathcal{N}(0, R_k)$
求	滤波器 $\hat{x} = o(u, z, \hat{x})$ 使得 $\mathrm{Var}[\hat{x}]$ 最小
算法性质	滤波器，递推

Algorithm 28: 无迹卡尔曼滤波

Input: 系统和输出方程 f, h

Input: 对称正定矩阵列 Q_k, R_k，对称正定矩阵 P_0

Input: 系统输入序列 u_k，系统输出序列 z_k

Input: 系统初值 $\hat{x}_0$

Parameter: α, κ, β

Output: 估计状态 $\hat{\mathbf{x}}$

$\lambda \leftarrow \alpha^2(n+\kappa) - n$

$w_0^m \leftarrow \lambda/(n+\lambda)$

$w_0^c \leftarrow \lambda/(n+\lambda) + (1-\alpha^2+\beta)$

for $i \in 1, \dots, n$ **do**

  $w_i^c \leftarrow = \frac{1}{2(n+\lambda)}$

  $w_i^m \leftarrow = \frac{1}{2(n+\lambda)}$

for $k \in 0, 1, \dots, K$ **do**

  $\chi_0 \leftarrow x_k$

  $\begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \dots & \mathbf{p}_n \end{bmatrix} \leftarrow \sqrt{(n+\lambda)P_k}$

  for $i \in 1, \dots, n$ **do**

    $\chi_i \leftarrow x_k + \mathbf{p}_i, \quad i = 1, \dots, n$

    $\chi_{i+n} \leftarrow x_k - \mathbf{p}_i, \quad i = 1, \dots, n$

  $x_k^- \leftarrow \sum_{i=0}^{2n} w_i^m f(\chi_i)$

  $P_{k+1}^- \leftarrow \sum_{i=0}^{2n} w_i^c (f(\chi_i) - x_k^-)(f(\chi_i) - x_k^-)^T + Q_k$

  $\hat{z}_k \leftarrow \sum_{i=0}^{2n} w_i^m h(\chi_i)$

  $H_k \leftarrow \sum_{i=0}^{2n} w_i^c (h(\chi_i) - \hat{z}_k)(f(\chi_i) - x_k^-)^T$

  $K_k \leftarrow P_{k+1}^- H_k^T (H_k P_{k+1}^- H_k^T + R_k)^{-1}$

  $\hat{x}_{k+1} \leftarrow \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, u_k))$

  $P_{k+1} \leftarrow K_k R_k K_k^T + (I - K_k H_k) P_{k+1}^- (I - K_k H_k)^T$

对应的 python 代码如下所示

```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
```

与 EKF/ESKF 相比，UKF 最大的优势是无需对系统函数或输出函数求导，但代价是需要多次进行 $f(\cdot)$ 和 $h(\cdot)$ 的计算。

12 最优控制基础

12.1 最优控制问题

在机器人中，我们往往会面临一类问题：在已知系统模型的条件下，希望设计一条系统的轨迹，使这条轨迹尽量满足一些量化目标。例如：希望耗费的能量最小、时间最短、跟踪误差最小、满足某些等式或不等式约束等等。这种问题，统称为**最优控制**问题。

在最优控制问题中，我们希望计算出使某个量达到最值的控制。这个我们关注的量，就称为**优化目标**，我们将其写为 $J(\mathbf{x},\mathbf{u})$。对于最小化问题，优化目标也称为**代价**。不同的需求带来不同的优化目标形式。稍后我们会总结常见的优化目标形式。

最优控制中的对象包括连续对象和离散对象。在机器人中，我们一般会将连续对象离散化，因此本章侧重于离散对象的介绍，但也会介绍相应的连续结论。

约束条件在最优控制中非常重要。例如，在一些优化控制问题中，控制输入不能超出某个范围的限制；某些问题中需要在一定的时间内解决问题，等等。约束条件的有无对于最优控制的设计会产生关键性的影响。本章中，我们主要讨论无约束和等式约束的最优控制问题。对于有不等式约束的最优控制问题，我们一般考虑使用下一章介绍的模型预测控制以及 QP 求解算法解决。

如上一章所述，在控制理论中，我们将期望系统稳定在固定值的问题称为**调节问题**。相应的，希望系统跟踪一个变化轨迹称为**跟踪问题**。对于最优控制问题，从调节问题开始分析将很大程度上简化分析过程。在本章中，如无特殊说明，取调节问题的期望状态 $x_d = 0$。

我们将离散环境下的 **(无约束) 最优控制问题**以数学语言描述，分别总结如下。注意：本章中的 $f(\cdot), g(\cdot), h(\cdot)$ 函数和第3章" 最优化算法" 中定义不同。

问题	(无约束) 一般最优调节控制 [离散]
问题简述	已知离散系统、代价函数，求最优的调节控制输入
已知	连续系统 $x_{k+1} = f(x_k, u_k)$ 代价函数 $J(\mathbf{x},\mathbf{u}) = h(x_N) + \sum_{k=0}^{N-1} g(x_k, u_k)$ 初始状态 x_0
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x},\mathbf{u})$

问题	(无约束) 一般最优跟踪控制 [离散]
问题简述	已知离散系统、代价函数、参考轨迹，求最优的跟踪控制输入
已知	连续系统 $x_{k+1} = f(x_k, u_k)$ 代价函数 $J(\mathbf{x},\mathbf{u}) = h(x_N) + \sum_{k=0}^{N-1} g(x_k, u_k)$ 初始状态 x_0，参考轨迹 $\mathbf{x}_d$
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x},\mathbf{u},\mathbf{x}_d)$

在这里我们需要注意，状态量 x_k 往往是向量形式，即具有多个维度的状态。在表达中，我们以不加粗的 x_k 表示其在某时刻的取值，而以加粗的 $\mathbf{x}$ 表示从开始时刻至结束时刻所有的状态。我们默认仅处理有限时间情况，用 N 代表总步数。

在最优控制问题中，系统函数 f 是已知函数。它可能是线性函数，也可能是非线性函数；但当前时刻的状态仅依赖上一个时刻的状态和控制输入。对于这里的代价函数，我们将其写成了比较通用的形式。它包含全过程的状态和控制输入，也包括我们单独写出的末端状态。代价函数 J 也已知，即 h 和 g 都是已知函数。

还要注意的一点是：我们求出的最优控制 u 往往是同时刻状态量的函数，即

$$u_k = u_k(x_k) \tag{III.12.1}$$

接下来，我们考察代价函数 $\mathbf{J}$ 的具体形式。在实际问题中，我们往往关注以下几种代价函数形式。注意：这些代价函数一般不会单独出现，而是根据实际问题情况组合出现。

首先是**时间最短代价**。它的代价函数就是总的时间步 N。

$$J(\mathbf{x}, \mathbf{u}) = N \tag{III.12.2}$$

其次是调节问题的**末端控制代价**，我们希望末状态和参考状态 x_d 尽可能接近。

$$J(\mathbf{x}, \mathbf{u}) = ||x_N - x_d||_S \tag{III.12.3}$$

这里我们使用记号 $||\cdot||_S$ 表示向量二次型

$$||a||_S = a^T S a \tag{III.12.4}$$

对于最优控制中的末端状态，我们可能只关心其中的几个维度，或者对不同维度的关心程度不同。此时调整 S 矩阵的系数就可以表示这种关注的分配。一般我们会选取对角阵。

对于跟踪问题，我们有**跟踪代价**，即希望全过程的状态 $\mathbf{x}$(即**轨迹**) 和某条参考轨迹 $\mathbf{x}_d$ 的差距尽量小

$$J(\mathbf{x}, \mathbf{u}) = \sum_{k=0}^{N} ||x_k - x_{d,k}||_{Q_k} \tag{III.12.5}$$

很多时候，我们也希望施加的总控制越小越好，这往往意味着节省能量、避免震荡等。因此，我们有**最小控制量代价**。

$$J(\mathbf{x}, \mathbf{u}) = \sum_{k=0}^{N-1} ||u_k||_{R_k} \tag{III.12.6}$$

综合以上各类最优控制目标，我们有 **(无约束) 最优调节控制问题**和 **(无约束) 最优跟踪控制问题**。

问题	**(无约束) 二次型最优调节控制 [离散]**
问题简述	已知离散系统，求使综合代价最小的调节控制输入
已知	离散系统 $x_{k+1} = f(x_k, u_k)$ 初始状态 x_0 半正定矩阵列 R_k, Q_k，半正定矩阵 S
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} ||x_N||_S + \sum_{k=0}^{N-1}(||x_k||_{Q_k} + ||u_k||_{R_k})$

问题	**(无约束) 二次型最优跟踪控制 [离散]**
问题简述	已知离散系统，求使综合代价最小的跟踪控制输入
已知	离散系统 $x_{k+1} = f(x_k, u_k)$ 初始状态 x_0，参考轨迹 $\mathbf{x}_d$ 半正定矩阵列 R_k, Q_k，半正定矩阵 S
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} ||x_N - x_{d,N}||_S$ $+ \sum_{k=0}^{N-1}(||x_k - x_{d,k}||_{Q_k} + ||u_k||_{R_k})$

12.2 贝尔曼最优与动态规划

1957 年，贝尔曼 (Richard Bellman) 针对最优控制问题表述了**最优性原理**。这一原理是后续所有序贯决策方法的基石，包括最优控制和强化学习方法。对于这一原理，我们可以用数学语言表达如下。

考虑初始条件 $k_0 = 0, x_0 = \xi_0$ 的最优控制问题 $\mathcal{P}_0$，对应代价函数记为 $J_{0:N}(\mathbf{x}, \mathbf{u})$。求解该问题，可得到最优控制 $\mathbf{u}[\mathbf{0}]^* = \nu_{0:N}(x)$ 以及最优轨迹 $\mathbf{x}[\mathbf{0}]^* = \xi_{0:N}$。

现在考虑 $\mathcal{P}_0$ 的子问题 $\mathcal{P}_m$，即取初始条件 $k_0 = m, x_0 = \xi_m$，其代价函数为 $J_{0:N}$ 中 $k = m$ 之后的部分，记为 $J_{m:N}(\mathbf{x}, \mathbf{u})$。对于问题 $\mathcal{P}_m$，求解其得到的最优控制记为 $\mathbf{u}[\mathbf{m}]^* = \tau_{m:N}(x)$，最优轨迹记为 $\mathbf{x}[\mathbf{m}]^* = \chi_{m:N}$

贝尔曼最优性原理认为：子问题 $\mathcal{P}_m$ 单独求解会得到 τ 和 χ，它们和原问题最优解 ν, ξ 的相应部分是相同的，即

$$\tau_{0:N-m}(x) = \nu_{m:N}(x)$$

$$\xi_{0:N-m} = \chi_{m:N}$$

如果我们将最优控制下的代价函数记为 $\mathcal{J}$，即

$$\mathcal{J}_{m:N}(\mathbf{x}) = J_{m:N}(\mathbf{x}, \nu_{m:N}(\mathbf{x}))$$

那我们就有

$$\min_{\mathbf{u}_{0:N}} J_{0:N}(\mathbf{x}, \mathbf{u}) = \min_{\mathbf{u}_{0:m}} (J_{0:m}(\mathbf{x}_{0:m}, \mathbf{u}) + \mathcal{J}_{m:N}(\mathbf{x}_{m:N})) \tag{III.12.7}$$

并且，取到最小值时，有

$$\mathbf{x}_{m:N} = \mathbf{x}[\mathbf{m}]^* = \chi_{m:N} \tag{III.12.8}$$

贝尔曼最优性原理揭示了，对于一大类序贯最优化问题 (满足最优控制问题定义)，整个问题的求解可以拆分为不同的步骤，先求解规模更小的后半段，再以后半段为基础求解前半段。这样的拆分可以递归地应用下去，比如对于后半段，还可以先找” 后半段的后半段”……最终，我们可以从最末尾的一小段开始，由后向前推导整个最优控制问题的最优解。这种问题拆分的思路，我们称为**动态规划**。

具体来说，让我们考虑这样的一个具体的例子：**滑块减速**。一维轨道上的滑块，在 0 时刻位于位置 $p_0 = 2$，速度 $v_0 = -1$。现希望通过在离散时间施力的方式，使其在 $k = N$ 时停止于位置 $p_N = 0$，且速度 $v_N = 0$。作用力分别作用在 $k = 0 : N - 1$ 的时刻，为其提供大小为 μ_k 的速度增量。求一种使得控制输入比较小的控制方案。

将上述例子形式化为最优控制问题，我们有：

离散系统状态 $x_k = [p_k, v_k]^T$，控制输入 $u_k = [0, \mu_k]^T$，满足如下系统方程

$$x_{k+1} = f(x_k, u_k) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_k$$

代价函数

$$J(\mathbf{x}, \mathbf{u}) = ||x_N||_S + \sum_{k=0}^{N-1} ||u_k||_R$$

其中

$$S=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, R=\begin{bmatrix}0 & 0\\ 0 & 1\end{bmatrix}$$

已知初始条件 $x_0=[2,-1]^T$，求使代价函数最小的控制输入。

为简化叙述和计算，我们取 $N=2$。将上面的式子更简单的写法，我们有

$$\begin{aligned}\mathcal{P}_0: \min_{\mathbf{u}} \quad & J_{0:2}(\mathbf{x},\mathbf{u})=||x_2||^2+\sum_{k=0}^{1}\mu_k^2\\ \text{s.t.} \quad & x_{k+1}=\begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix}x_k+\begin{bmatrix}0 & 1\end{bmatrix}u_k\end{aligned}$$

针对这一问题，让我们使用上面的动态规划思路进行分析。上面描述的问题就是问题 $\mathcal{P}_0$。为了求出它的最优控制 $\nu_{0:1}$ 和最优轨迹 $\xi_{0:2}$，我们需要先分析其尾部子问题 $\mathcal{P}_1$。根据上面的分析，我们去除代价函数中 $k=1$ 之前的部分。

$$\begin{aligned}\mathcal{P}_1: \min_{u_1} \quad & J_{1:2}(\mathbf{x}_{1:2},\mathbf{u}_2)=||x_2||^2+\mu_1^2\\ \text{s.t.} \quad & p_2=p_1+v_1\\ & v_2=v_1+\mu_1\end{aligned}$$

注意，我们此时的优化目标不是 x_k 而是 u_k。因此我们只求解 u **和** x **的关系**，而不求解具体的数值。所有的 x_k 都受到系统状态方程的约束，并且还有初始条件 x_0 的约束。我们只在逆推到 $k=0$ 时才能代入 x_0，得出 u 和 x 的数值解。

现在，我们的 $J_{1:2}$ 的表达式中出现了 x_2 和 μ_1，但 x_2 是 μ_1 的函数。利用系统状态方程，我们可以写为

$$x_2=\begin{bmatrix}p_1+v_1\\ v_1+\mu_1\end{bmatrix}$$

对于问题 $\mathcal{P}_1$ 来说，我们假定 $x_1=[p_1,v_1]^T$ 已知。因此，我们得到了 x_2 和 μ_1 的关系。我们于是有

$$||x_2||^2=(p_1+v_1)^2+(v_1+\mu_1)^2$$

因此，$J_{1:2}$ 就可以写成 x_1 和 μ_1 的函数

$$J_{1:2}(x_1,\mu_1)=(p_1+v_1)^2+(v_1+\mu_1)^2+\mu_1^2$$

根据贝尔曼最优性原理，问题 $\mathcal{P}_0$ 对应的最优控制 $\nu_{0:2}(x)$ 中，$\nu_2(x)$ 就是使 $J_{1:2}(x_1,u_1)$ 最小的 $u_1(x)$(忽略第一个维度，实际就是 $\mu_1(x)$)。因此，我们进行如下求解

$$\begin{aligned}\arg\min_{\mu_1} J_{1:2}(x_1,\mu_1) &= \arg\min_{\mu_1}\left((p_1+v_1)^2+(v_1+\mu_1)^2+\mu_1^2\right)\\ &= \arg\min_{\mu_1}\left(2\mu_1^2+2v_1\mu_1+v_1^2\right)\\ &= -\frac{1}{2}v_1\end{aligned}$$

因此我们有

$$\nu_1(x)=\tau_1(x)=-\frac{1}{2}v$$

这里我们再次强调，最优控制的动态规划倒退求解的结果，既不是最优的控制量取值，也不是最优的状态取值，而是控制输入关于状态的函数。[11]

无论是母问题的 $\mathbf{u}[\mathbf{0}]^* = \nu_{0:N}(x)$ 还是子问题的 $\mathbf{u}[\mathbf{m}]^* = \tau_{m:N}(x)$，都是一组关于 x 的函数。时刻 k 对应的函数仅作用于时刻 k 的状态 x_k，而不会作用到 x_{k-1} 或 x_{k+1}。但是，对于一个问题，只要代价函数不不变，即使状态初值 x_0 改变，导致最优控制对应的轨迹改变，函数组 $\nu_{0:N}(x)$ 也不会改变。

接下来，我们继续推导下一个问题 $\mathcal{P}_0$ 的求解。原始的 $\mathcal{P}_0$ 问题可以写成

$$\begin{aligned}\mathcal{P}_0: \min_{u_{0:1}} \quad & J_{0:2}(\mathbf{x}_{0:2}, \mathbf{u}_{0:1}) = ||x_2||^2 + \mu_1^2 + \mu_0^2 \\ \text{s.t.} & p_2 = p_1 + v_1 \\ & v_2 = v_1 + \mu_1 \\ & p_1 = p_0 + v_0 \\ & v_1 = v_0 + \mu_0\end{aligned}$$

我们看到，$\mathcal{P}_0$ 问题的代价是完全包含 $\mathcal{P}_1$ 的代价 $J_{1:2}$ 的。根据贝尔曼最优原理，$\mathcal{P}_0$ 代价中的 $J_{1:2}$，可以替换成 $\mathcal{P}_1$ 取到最优控制时的代价函数 $\mathcal{J}_{1:2}(\mathbf{x})$，同时求解的变量由 $u_{0:1}$ 减少为只求解 u_1。即

$$\min_{\mathbf{u}_{0:2}} J_{0:2}(\mathbf{x}, \mathbf{u}) = \min_{\mathbf{u}_{0:2}} (J_{0:1}(\mathbf{x}_{0:1}, \mathbf{u}) + \mathcal{J}_{1:2}(\mathbf{x}_{1:2}))$$

因此，让我们首先尝试写出 $\mathcal{J}_{1:2}(\mathbf{x}_{1:2})$ 的表达式。根据定义

$$\begin{aligned}\mathcal{J}_{1:2}(\mathbf{x}_{1:2}) &= J_{1:2}(\mathbf{x}_{1:2}, \nu_1(x_1)) \\ &= (p_1 + v_1)^2 + (v_1 + \nu_1(x_1))^2 + \nu_1(x_1)^2 \\ &= (p_1 + v_1)^2 + (v_1 - \frac{1}{2}v_1)^2 + (-\frac{1}{2}v_1)^2 \\ &= (p_1 + v_1)^2 + \frac{1}{2}v_1^2\end{aligned}$$

因此，我们可以得到新的 $J_{0:2}(\mathbf{x}, \mathbf{u})$ 表达式

$$J_{0:2}(\mathbf{x}, \mathbf{u}) = (p_1 + v_1)^2 + \frac{1}{2}v_1^2 + \mu_0^2$$

让我们重新表述一下问题 $\mathcal{P}_0$ 为 $\mathcal{P}_0'$

$$\begin{aligned}\mathcal{P}_0': \min_{u_1} \quad & J_{0:2}(\mathbf{x}_{0:1}, \mathbf{u}_1) = (p_1 + v_1)^2 + \frac{1}{2}v_1^2 + \mu_0^2 \\ \text{s.t.} & p_1 = p_0 + v_0 \\ & v_1 = v_0 + \mu_0\end{aligned}$$

现在，我们的 $J_{0:2}$ 的表达式中出现了 x_1 和 μ_0，但 x_1 是 μ_0 的函数。利用系统状态方程，我们可以写为

$$\begin{bmatrix} p_1 \\ v_1 \end{bmatrix} = \begin{bmatrix} p_0 + v_0 \\ v_0 + \mu_0 \end{bmatrix}$$

对于问题 $\mathcal{P}_0'$ 来说，我们假定 $x_0 = [p_0, v_0]^T$ 已知。由于我们得到了 x_1 和 μ_0 的关系，于是有

[11] 从这个角度来看，其实我们的 $J_{m:N}$ 是一个关于函数 u 的泛函。我们目前是在离散域中求解这个泛函。

$$
\begin{aligned}
(p_1+v_1)^2+\frac{1}{2}v_1^2 &= (p_0+v_0+v_0+\mu_0)^2+\frac{1}{2}(v_0+\mu_0)^2 \\
&= (p_0+2v_0)^2+\frac{1}{2}v_0^2+(2p_0+5v_0)\mu_0+\frac{3}{2}\mu_0^2
\end{aligned}
$$

因此，$J_{0:2}$ 就可以写成 x_0 和 μ_0 的函数

$$
J_{0:2}(x_1,\mu_1)=(p_0+2v_0)^2+\frac{1}{2}v_0^2+(2p_0+5v_0)\mu_0+\frac{5}{2}\mu_0^2
$$

根据贝尔曼最优性原理，问题 $\mathcal{P}_0$ 对应的最优控制 $\nu_{0:1}(x)$ 中，$\nu_1(x)$ 就是使 $J_{0:2}$ 最小的 $\mu_0(x)$。因此，我们进行如下求解

$$
\begin{aligned}
\arg\min_{\mu_0} J_{0:2}(x_0,\mu_0) &= \arg\min_{\mu_0}\left((p_0+2v_0)^2+\frac{1}{2}v_0^2+(2p_0+5v_0)\mu_0+\frac{5}{2}\mu_0^2\right) \\
&= -v_0-\frac{2}{5}p_0
\end{aligned}
$$

因此我们有

$$
\nu_0(x)=\tau_0(x)=-v-\frac{2}{5}p
$$

至此，在这个最简单的 $N=2$ 的最优控制问题中，我们已经解出了所需的最优控制

$$
u_0^*(x_0)=\nu_0(x_0)=-v_0-\frac{2}{5}p_0
$$

$$
u_1^*(x_1)=\nu_1(x_1)=-\frac{1}{2}v_1
$$

如果我们需要求出最优轨迹和具体的最优控制量，我们可以将初始条件 $x_0=[2,-1]^T$ 代入，随后使用状态方程**正向求解**。求解的结果总结在下表中。

	k=0	k=1	k=2
p_k 表达式	2	p_0+v_0	p_1+v_1
v_k 表达式	-1	$v_0+\mu_0$	$v_1+\mu_1$
μ_k 表达式	$-v_0-\frac{2}{5}p_0$	$-\frac{1}{2}v_1$	-
$J_{k:N}$ 表达式	$\|\|x_2\|\|^2+\mu_1^2+\mu_0^2$	$\|\|x_2\|\|^2+\mu_1^2$	$\|\|x_2\|\|^2$
p_k 取值	2	1	1/5
v_k 取值	-1	-4/5	-2/5
μ_k 取值	1/5	2/5	-
$J_{k:N}$ 取值	2/5	9/25	1/5

我们可以看到，尽管我们求解的控制并没有将末状态严格限制在 $[0,0]^T$，但它在控制量大小和末状态之间取得了某种平衡。如果我们去除控制量相关的限制，或者通过 S 矩阵或 Q 矩阵改变其比例，得出的控制就会有所不同。

小结一下，我们首先利用贝尔曼最优性原理，从原问题的末状态附近中切分出最小的子问题。最优性原理保证子问题求解的最优解正是原问题最优解的一部分。接下来我们逐步向前推进，扩大求解的范围，每次只求解一步的最优控制，直至推到 $k=0$，也就是原问题。这样，我们通过**逆向求解**得到了每个时刻的最优控制 $\nu_k(x)$。在此基础上，我们再从原问题的初始条件 x_0 开始，根据系统状态方程和最优控制进行**正向求解**，从而获得特定初始条件下的轨迹和控制量。

虽然我们只是演示了一个非常简单的例子，但我们可以依此思路，总结动态规划法求解最优控制问题的一般算法。

算法	最优控制-动态规划求解
问题类型	(无约束) 一般最优调节控制 [离散]
已知	离散系统 $x_{k+1} = f(x_k, u_k)$ 代价函数 $J(\mathbf{x}, \mathbf{u}) = h(x_N) + \sum_{k=0}^{N-1} g(x_k, u_k)$ 初始状态 x_0
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$
算法性质	model-based, DP, 解析推导

Algorithm 29: 最优控制-动态规划求解

Input: 离散系统 $x_{k+1} = f(x_k, u_k)$
Input: 代价函数 $J(\mathbf{x}, \mathbf{u}) = h(x_N) + \sum_{k=0}^{N-1} g(x_k, u_k)$
Input: 初始状态 x_0
Output: 最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$
$\mathcal{J}_{N:N}(\mathbf{x}_{N:N}) \leftarrow h(x_N)$
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 连续最优性原理

最优性原理也可用于连续的情况。首先，我们引入连续状态方程下的 (无约束) 一般最优调节/跟踪控制问题

问题	(无约束) 一般最优调节控制 [连续]
问题简述	已知连续系统、代价函数，求最优的调节控制输入
已知	连续系统 $\dot{x} = f(x, u, t)$ 代价函数 $J(x, u) = h(x_{t_f}, t_f) + \int_0^{t_f} g(\mathbf{x}(t), \mathbf{u}(t), t)dt$ 初始状态 $x_{t_0} = x_0$
求	最优控制 $\mathbf{u}^*(t) = \arg\min_{\mathbf{u}} J(\mathbf{x}(t), \mathbf{u}(t))$

问题	(无约束) 一般最优跟踪控制 [连续]
问题简述	已知连续系统、代价函数、期望轨迹，求最优的跟踪控制输入
已知	连续系统 $\dot{x} = f(x, u, t)$ 代价函数 $J(x, u) = h(x_{t_f}, t_f) + \int_0^{t_f} g(\mathbf{x}(t), \mathbf{u}(t), t)dt$ 初始状态 $x(t_0) = x_0$，期望轨迹 $\mathbf{x}_d(t)$
求	最优控制 $\mathbf{u}^*(t) = \arg\min_{\mathbf{u}} J(\mathbf{x}(t), \mathbf{u}(t), \mathbf{x}_d(t))$

在这里，我们将代价函数中的 $g(\cdot)$ 写成了依赖时间的形式，从而使其可以表达“代价中的项随时间变化”的更广泛情况，类似于介绍离散问题时矩阵 Q_k, R_k 是依赖于时间步 k 的矩阵列。这种写法覆盖了不随时间变化的情况，是更广泛的表达形式。

这里仍然要注意控制输入 $u(t)$ 和状态量 $x(t)$ 的关系。我们的最终目标是求解 $u(t)$ 和 $x(t)$ 的关系，也就是 $u(x(t))$；但在分析时，我们有必要将其作为独立变量考虑，从而得出其最优解满足的性质。

我们仍然优先考虑调节问题。采用和上面离散形式类似的思路，首先我们尝试写出最优性原理的数学描述。连续问题的时间取值范围是 $[t_0, t_f]$。我们考虑一个离起始时刻较近的中间时刻 $t_0+\tau$。

$$\begin{aligned}&\min_u \left(h(x_{t_f}, t_f) + \int_{t_0}^{t_f} g(x(t), u(t), t)dt\right)\\ =&\min_u \int_{t_0}^{t_0+\tau} g(x(t), u(t), t)dt + \min_u \left(h(x_{t_f}, t_f) + \int_{t_0+\tau}^{t_f} g(x(t), u(t), t)dt\right)\end{aligned}$$

和离散情形类似，我们可以记 $\mathcal{J}_{t_0+\tau}^{t_f}(x(t), t)$ 为” 最小代价”[12]，即求出并代入最优解 $u^*(x(t))$ 后，子问题价值函数的表达式。

$$\mathcal{J}_{t_0}^{t_f}(x(t), t) = \min_u \int_{t_0}^{t_f} g(x(t), u(t))dt + h(x_{t_f}, t_f) \tag{III.12.9}$$

于是，我们有

$$\mathcal{J}_{t_0}^{t_f}(x(t), t) = \min_u \int_{t_0}^{t_0+\tau} g(x(t), u(t), t)dt + \mathcal{J}_{t_0+\tau}^{t_f}(x(t), t) \tag{III.12.10}$$

下面，我们将 $\mathcal{J}_{t_0+\tau}^{t_f}(x(t), t)$ 在 $x = x(t_0), t = t_0$ 附近进行泰勒展开。注意泰勒展开时我们将 x 和 t 视为独立变量。于是有

$$\mathcal{J}_{t_0+\tau}^{t_f}(x(t), t) = \mathcal{J}_{t_0}^{t_f}(x(t), t) + \frac{\partial \mathcal{J}}{\partial t}(x(t), t)\tau + \frac{\partial \mathcal{J}}{\partial x}(x(t), t)\left(x(t_0+\tau) - x(t_0)\right) + o(\tau)$$

注意到系统方程已知：$\dot{x} = f(x(t), u(t), t)$。$u(t)$ 取最优控制 $u^*(x(t))$ 时，有

$$x(t_0+\tau) - x(t_0) = f(x(t_0), u^*(t_0), t_0)\tau + o(\tau)$$

因此我们有

$$\mathcal{J}_{t_0+\tau}^{t_f}(x(t), t) - \mathcal{J}_{t_0}^{t_f}(x(t), t) = \left(\frac{\partial \mathcal{J}}{\partial t}(x(t), t) + \frac{\partial \mathcal{J}}{\partial x}(x(t), t) \cdot f(x(t_0), u^*(t_0), t_0)\right)\tau + o(\tau)$$

联系式III.12.10，我们有

$$-\int_{t_0}^{t_0+\tau} g(x(t), u^*(t), t)dt = \left(\frac{\partial \mathcal{J}}{\partial t}(x(t), t) + \frac{\partial \mathcal{J}}{\partial x}(x(t), t) \cdot f(x(t_0), u^*(t_0), t_0)\right)\tau + o(\tau)$$

取 $\lim_{\tau \to 0}$，我们有

$$0 = g(x(t_0), u^*(t_0), t_0) + \frac{\partial \mathcal{J}}{\partial t}(x(t), t_0) + \frac{\partial \mathcal{J}}{\partial x}(x(t_0), t_0) \cdot f(x(t_0), u^*(t_0), t_0)$$

整理成更常见的形式，我们有

[12] 注意花体 $\mathcal{J}$ 和 J 的区别

$$-\frac{\partial \mathcal{J}}{\partial t}(x(t),t) = g(x(t),u^*(t),t) + \frac{\partial \mathcal{J}}{\partial x}(x(t),t)\cdot f(x(t),u^*(t),t) \tag{III.12.11}$$

或者

$$-\frac{\partial \mathcal{J}}{\partial t}(x(t),t) = \min_{u(t)} \mathcal{H}\left(x(t),u(t),\frac{\partial \mathcal{J}}{\partial x},t\right) \tag{III.12.12}$$

式III.12.12就是著名的 **Hamilton-Jacobi-Bellman 方程**，简称 **HJB 方程**。这里面 Bellman 表示该方程有 Bellman 最优性原理导出，Jacobi 表示里面出现了求导。Hamilton 代表的是 **Hamiltonian**，或哈密尔顿量，记为 $\mathcal{H}$，其定义为

$$\mathcal{H}\left(x(t),u(t),\frac{\partial \mathcal{J}}{\partial x},t\right) = g(x(t),u(t),t) + \frac{\partial \mathcal{J}}{\partial x}(x(t),t)\cdot f(x(t),u(t),t) \tag{III.12.13}$$

HJB 方程的一个边界条件为

$$\mathcal{J}(x(t_f),t_f) = h(x(t_f),t_f) \tag{III.12.14}$$

HJB 方程是关于最优代价 $\mathcal{J}$ 的偏微分方程，描述的是它需要满足的性质，或者说必要条件。细心的读者可能已经注意到，最优代价的偏导和系统状态 f、瞬时代价 g 两个函数有关，它们都依赖最优控制 u^*。但同时，最优控制 u^* 的求取又依赖于最小化 Hamilton 量或最优代价。因此，这构成了一个循环依赖。

对于这个循环依赖，在连续域中，事实上我们没什么好办法处理。也就是说，即使能写出 HJB 方程，无约束最优控制问题 (连续形式) 也不存在一个通用的解析解法。在离散情况下，我们已经看到，可以借助动态规划的思想，我们交替求出最优的控制和最优的代价。

对于连续问题，一方面，在一些特殊情况下，例如 LQR 问题，我们可以尝试猜测 $\mathcal{J}$ 的形式，随后利用它求解其中的系数。另一方面，我们可以考虑其他求解方式，并把 HJB 方程当作求解最优控制后的验证性条件。

12.4 离散时间 LQR

12.4.1 公式推导

在 2.1 节中，我们介绍了最优控制问题的几种特殊形式，包括离散形式的无约束调节控制问题。该问题的定义没有限定系统状态方程的形式。在这里，我们研究该问题最简单的一种情况——线性系统。由于其指标写成二次型形式，该问题也称为**线性二次型调节问题**，或 **LQR 问题 [离散]**.

问题	**LQR 调节控制 [离散]**
问题简述	已知线性离散系统，求使二次型代价最小的调节控制输入
已知	离散线性系统 $x_{k+1} = Ax_k + Bu_k$ 初始状态 x_0 半正定矩阵列 R_k, Q_k，半正定矩阵 S
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N\|\|_S + \sum_{k=0}^{N-1}(\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$

事实上，我们在第 2 节中的例子就是 LQR 问题。因此，接下来我们采用一致的动态规划求解思路，推导 LQR 问题的通用解法。

首先，我们写出全过程的问题 $\mathcal{P}_0$：

$$\mathcal{P}_0 : \min_{\mathbf{u}} \quad J_{0:N}(\mathbf{x},\mathbf{u}) = ||x_N||_S + \sum_{k=0}^{N-1}(||x_k||_{Q_k} + ||u_k||_{R_k})$$
$$\text{s.t.} \quad x_{k+1} = Ax_k + Bu_k, k = 0, 1, ..., N-1$$

按照动态规划的思路，我们从 $k = N-1$ 开始倒推，依次求解最优代价函数 $\mathcal{J}_{k:N}$ 和最优轨迹 $\tau_{k:N}$

首先是 $k = N-1$ 时的子问题

$$\mathcal{P}_{N-1} : \min_{\mathbf{u}} \quad J_{N-1:N}(\mathbf{x},\mathbf{u}) = x_N^T S x_N + x_{N-1}^T Q_{N-1} x_{N-1} + u_{N-1}^T R_{N-1} u_{N-1}$$
$$\text{s.t.} \quad x_{k+1} = Ax_k + Bu_k, k = N-1$$

通过代入离散系统状态方程，我们可以消去 $J_{N-1:N}$ 中的 x_N 一项

$$J_{N-1:N}(x_{N-1}, u_{N-1}) = (Ax_{N-1} + Bu_{N-1})^T S(Ax_{N-1} + Bu_{N-1}) + x_{N-1}^T Q_{N-1} x_{N-1} + u_{N-1}^T R_{N-1} u_{N-1}$$

对该式的同类项进行合并，有

$$\begin{aligned} J_{N-1:N}(x_{N-1}, u_{N-1}) = {} & x_{N-1}^T (Q_{N-1} + A^T S A) x_{N-1} + x_{N-1}^T A S B u_{N-1} \\ & + u_{N-1}^T B^T S A x_{N-1} + u_{N-1}^T (R_{N-1} + B^T S B) u_{N-1} \end{aligned}$$

由于该问题是凸问题（这里不加证明），我们可以通过一阶导数为 0 直接求出全局最优解。因此我们对 u_{N-1} 求偏导。

$$\frac{\partial J_{N-1:N}}{\partial u_{N-1}}(x_{N-1}, u_{N-1}) = (2B^T S A x_{N-1} + 2(R_{N-1} + B^T S B) u_{N-1})^T$$

令该偏导等于 0，我们有子问题 $\mathcal{P}_{N-1}$ 的最优解 ν_{N-1}[13]

$$\nu_{N-1} = -(R_{N-1} + B^T S B)^{-1} B^T S A x_{N-1}$$

由于这里最优控制和状态的关系是线性的，我们可以将这个线性变换的矩阵记为 F_k。因此这里的 F_{N-1} 就是

$$F_{N-1} = -(R_{N-1} + B^T S B)^{-1} B^T S A$$

现在我们已经有 $\nu_{N-1} = F_{N-1} x_{N-1}$，按照动态规划的思路，接下来我们写出最优价值 $\mathcal{J}_{N-1:N}$

$$\begin{aligned} \mathcal{J}_{N-1:N}(x_{N-1}) = {} & (Ax_{N-1} + BF_{N-1}x_{N-1})^T S(Ax_{N-1} + BF_{N-1}x_{N-1}) \\ & + x_{N-1}^T Q_{N-1} x_{N-1} + (F_{N-1}x_{N-1})^T R_{N-1}(F_{N-1}x_{N-1}) \\ = {} & x_{N-1}^T((A + BF_{N-1})^T S(A + BF_{N-1}) + F_{N-1}^T R_{N-1} F_{N-1} + Q_{N-1}) x_{N-1} \end{aligned}$$

注意到这是一个关于 x_{N-1} 的二次型。我们将中间的部分记为 P_{N-1}，即

$$P_{N-1} = (A + BF_{N-1})^T S(A + BF_{N-1}) + F_{N-1}^T R_{N-1} F_{N-1} + Q_{N-1}$$

那么我们有

[13] 由于限定了 R_{N-1} 和 S 是半正定，这里可以直接求逆，下同

$$\mathcal{J}_{N-1:N}(x_{N-1}) = x_{N-1}^T P_{N-1} x_{N-1}$$

接下来，我们考虑 $k = N-2$ 的情况。根据贝尔曼最优性原理，$J_{N-1:N}$ 的部分可以被 $\mathcal{J}_{N-1:N}$ 代替，即

$$\begin{aligned}\mathcal{P}_{N-2}: \min_{\mathbf{u}} \quad & J_{N-2:N}(\mathbf{x}, \mathbf{u}) = x_{N-1}^T P_{N-1} x_{N-1} + x_{N-2}^T Q_{N-2} x_{N-2} + u_{N-2}^T R_{N-2} u_{N-2} \\ \text{s.t.} \quad & x_{k+1} = Ax_k + Bu_k, k = N-2\end{aligned}$$

细心的读者可能已经感觉到了熟悉。我们对比 $\mathcal{P}_{N-1}$ 和 $\mathcal{P}_{N-2}$，就能发现两个问题的区别非常小，仅仅在于所有的角标都往前推了一步，并且 S 换成了 P_{N-1}。因此我们跳过一些推导步骤，直接写出偏导形式

$$\frac{\partial J_{N-2:N}}{\partial u_{N-2}}(x_{N-2}, u_{N-2}) = (2B^T P_{N-1} A x_{N-2} + 2(R_{N-2} + B^T P_{k+1} B) u_{N-2})^T$$

因此我们有最优控制

$$\nu_{N-2} = -(R_{N-2} + B^T P_{k+1} B)^{-1} B^T P_{k+1} A x_{N-2}$$

即

$$F_{N-2} = -(R_{N-2} + B^T P_{k+1} B)^{-1} B^T P_{k+1} A$$

将最优控制代入 $J_{N-2:N}$，得到的最优价值 $\mathcal{J}_{N-2:N}$ 的形式应该也和 $\mathcal{J}_{N-1:N}$ 相同

$$\begin{aligned}\mathcal{J}_{N-2:N}(x_{N-2}) = x_{N-2}^T ((A + BF_{N-2})^T P_{N-1} (A + BF_{N-2}) \\ + F_{N-2}^T R_{N-2} F_{N-2} + Q_{N-2}) x_{N-2}\end{aligned}$$

于是我们有

$$P_{N-2} = (A + BF_{N-2})^T P_{N-1} (A + BF_{N-2}) + F_{N-2}^T R_{N-2} F_{N-2} + Q_{N-2}$$

写到这里，读者不难发现，我们已经得到了 P_{N-2} 的表达式。并且，这个表达式和 P_{N-1} 的唯一区别就在于 S 变成了 P_{N-1}。因此，我们可以大胆猜测，后续的 P_k 和 P_{k+1} 的关系也符合类似的递推形式。

这样，我们考虑的各个子问题 $\mathcal{P}_k$ 的形式也都是一致的，也就是说后续的问题和最优解都符合这几个公式的形式。这使得我们可以直接总结如下的递推关系式 $(k = 0, 1, ..., N-1)$。

$$\begin{aligned}F_k &= -(R_k + B^T P_{k+1} B)^{-1} B^T P_{k+1} A \\ P_k &= (A + BF_k)^T P_{k+1} (A + BF_k) + F_k^T R_k F_k + Q_k \\ \nu_k &= F_k x_k \\ \mathcal{J}_{k:N}(x_k) &= \min u_k J_{k:N}(x_k, u_k) = x_k^T P_k x_k\end{aligned} \tag{III.12.15}$$

作为起始，我们有

$$P_N = S \tag{III.12.16}$$

注意到 P_k 的递推式还能进一步化简为不出现 F_k 的形式，即

$$
\begin{aligned}
P_k &= A^T P_{k+1} A + F_k^T B^T P_{k+1} A + A^T P_{k+1} B F_k + F_k^T (B^T P_{k+1} B + R_k) F_k + Q_k \\
&= A^T P_{k+1} A + F_k^T B^T P_{k+1} A + A^T P_{k+1} B F_k - F_k^T B^T P_{k+1} A + Q_k \\
&= A^T P_{k+1} A + A^T P_{k+1} B F_k + Q_k \\
&= A^T P_{k+1} A - A^T P_{k+1} B (R_k + B^T P_{k+1} B)^{-1} B^T P_{k+1} A + Q_k
\end{aligned}
$$

因此我们也有

$$
\begin{aligned}
F_k &= -(R_k + B^T P_{k+1} B)^{-1} B^T P_{k+1} A \\
P_k &= A^T P_{k+1} A + A^T P_{k+1} B F_k + Q_k
\end{aligned}
\tag{III.12.17}
$$

式III.12.15就是离散时间无约束 LQR 问题最优解的通用表达形式。我们将这个问题的求解过程总结一下。

算法	**离散 LQR 迭代求解**
问题类型	LQR 调节控制 [离散]
已知	离散线性系统 $x_{k+1} = Ax_k + Bu_k$ 半正定矩阵列 R_k, Q_k，半正定矩阵 S 初始状态 x_0
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N\|\|_S + \sum_{k=0}^{N-1} (\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$
算法性质	model-based, DP, 解析解

Algorithm 30: 离散 LQR 迭代求解

Input: 离散线性系统 $x_{k+1} = Ax_k + Bu_k$

Input: 半正定矩阵列 R_k, Q_k，半正定矩阵 S, 初值 x_0

Output: 最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$

$P_N \leftarrow S$

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 $k \in 0, \ldots, N-1$ **do**

  $\nu_k \leftarrow F_k x_k$

  $x_{k+1} \leftarrow Ax_k + B\nu_k$

$\mathbf{u}^* \leftarrow \nu_{0:N}(x)$

对应的 python 代码如下所示 (该代码兼容了可变的 A 和 B 矩阵)

```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
```

特别地，我们可以考虑一类 Q_k 和 R_k 对所有 k 都相同的 LQR 问题[14]。这类问题一般对应的 N 很大，甚至可以认为是无穷大。此时终端代价不再重要。问题表述如下。

问题	无限时间 LQR 调节控制 [离散]
问题简述	已知线性离散系统，求使二次型代价最小的控制输入
已知	离散线性系统 $x_{k+1} = Ax_k + Bu_k$ 半正定矩阵列 R_k, Q_k
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \sum_{k=0}^{\inf}(\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$

这类问题中，我们的递归式有

$$F_k = -(R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A$$
$$P_k = A^T(P_{k+1} - P_{k+1}B(R + B^T P_{k+1} B)^{-1} B^T P_{k+1})A + Q$$

如果 N 足够大以至于无限大，那么上述递归方程中的 P_k 就可能收敛到 P。此时递推式变为一个关于 P 的方程，我们称之为**矩阵 Riccati 方程**。

$$P = Q + A^T PA - A^T PB(R + B^T PB)^{-1} B^T PA \tag{III.12.18}$$

对于无限时间 LQR 问题，我们只需求解该方程得到 P，随后使用下面的公式求出 F，就可以得到恒定的最优控制率。

$$F = -(R + B^T PB)^{-1} B^T PA \tag{III.12.19}$$

求解矩阵 Riccati 方程已有很多成熟的方法，封装成现成的工具包可以使用。

12.4.2 倒立摆求解

[本部分内容将在后续版本中更新，敬请期待]

[14]我们将其称之为平稳假设

12.5 连续时间 LQR

前面我们说过，对一般形式的连续最优控制问题，虽然可写出 HJB 方程，但由于循环依赖问题无法写出最优控制的通用形式。不过，对于线性二次型的情况，我们可以猜测写出。

首先，我们仿照离散形式定义连续的 LQR 问题

问题	**LQR 调节控制 [连续]**
问题简述	已知连续线性系统，求使综合代价最小的调节控制输入
已知	连续线性系统 $\dot{x}(t) = Ax(t) + Bu(t)$ 初始状态 x_0 半正定函数矩阵 $R(t), Q(t)$，半正定矩阵 S
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_{t_f}\|\|_S + \int_0^{t_f} (\|\|x(t)\|\|_{Q(t)} + \|\|u(t)\|\|_{R(t)})dt$

这里，我们的代价函数为

$$J(\mathbf{x}, \mathbf{u}) = ||x_{t_f}||_S + \int_0^{t_f} (||x(t)||_{Q(t)} + ||u(t)||_{R(t)})dt \tag{III.12.20}$$

在上面的代价下，我们写出 Hamiltonian 量，它是多个函数的泛函（包括目前形式未知的最优代价对 x 偏导 $\frac{\partial \mathcal{J}}{\partial x}(x(t), t)$）

$$\mathcal{H}(x(t), u(t), \frac{\partial \mathcal{J}}{\partial x}, t) = \frac{1}{2}(||x(t)||_{Q(t)} + ||u(t)||_{R(t)}) + \frac{\partial \mathcal{J}}{\partial x}(x(t), t) \cdot (Ax(t) + Bu(t))$$

如前所述，最优控制就是满足 HJB 方程的 u^*，也就是使 Hamiltonian 最小的 $u(t)$。因此，我们将其对 $u(t)$ 求偏导，并使其为 0

$$\frac{\partial \mathcal{H}}{\partial u(t)}(x(t), u(t), \frac{\partial \mathcal{J}}{\partial x}, t) = R(t)u(t) + B^T \left(\frac{\partial \mathcal{J}}{\partial x}\right)^T (x(t), t) = 0$$

此处，我们可以验证二阶偏导 $\frac{\partial^2 \mathcal{H}}{\partial u^2(t)} = R(t)$ 为正定，因此满足一阶导为 0 的 $u(t)$ 一定就是使泛函取最小值的 $u^*(t)$。即

$$u^*(t) = -R^{-1}(t)B^T \left(\frac{\partial \mathcal{J}}{\partial x}\right)^T (x(t), t) \tag{III.12.21}$$

为表达方便，记

$$\mathcal{J}_x = B^T \left(\frac{\partial \mathcal{J}}{\partial x}\right)^T (x(t), t) \tag{III.12.22}$$

将该结论代入 Hamiltonian，有

$$\begin{aligned}
\mathcal{H}(x(t), u^*(t), \frac{\partial \mathcal{J}}{\partial x}, t) &= \frac{1}{2}(x^T Q x + \mathcal{J}_x^T B R^{-1} R R^{-1} B^T \mathcal{J}_x) + \mathcal{J}_x^T A x - \mathcal{J}_x^T B R^{-1}(t) B^T \mathcal{J}_x \\
&= \frac{1}{2}(x^T Q x + \mathcal{J}_x^T B R^{-1} B^T \mathcal{J}_x) + \mathcal{J}_x^T A x - \mathcal{J}_x^T B R^{-1}(t) B^T \mathcal{J}_x \\
&= \frac{1}{2} x^T Q x - \frac{1}{2} \mathcal{J}_x^T B R^{-1} B^T \mathcal{J}_x + \mathcal{J}_x^T A x
\end{aligned}$$

我们可以写出 LQR 情况下的连续 HJB 方程

$$-\frac{\partial \mathcal{J}}{\partial t}(x(t),t)=\frac{1}{2}x^TQx-\frac{1}{2}\mathcal{J}_x^TBR^{-1}B^T\mathcal{J}_x+\mathcal{J}_x^TAx \tag{III.12.23}$$

此处，我们仍不知道最优代价对 x 偏导 $\mathcal{J}_x(x(t),t)$ 的具体形式。不过，回顾上节，在离散形式的推导中，我们可以确定离散最优代价一定是二次型：$\mathcal{J}_{k:N}=x_k^TP_kx_k$。因此，类似地，我们**大胆猜测**连续形式的 $\mathcal{J}(x(t),t)$ 也是二次型形式

$$\mathcal{J}(x(t),t)=\frac{1}{2}x^T(t)P(t)x(t) \tag{III.12.24}$$

其中，P 为对称阵。这样，我们有

$$\begin{aligned}\frac{\partial \mathcal{J}}{\partial x}(x(t),t)&=P(t)x(t)\\ \frac{\partial \mathcal{J}}{\partial t}(x(t),t)&=\frac{1}{2}x(t)\dot{P}(t)x(t)\end{aligned} \tag{III.12.25}$$

代入 HJB 方程中，我们有

$$-\frac{1}{2}x(t)\dot{P}(t)x(t)=\frac{1}{2}x^T(t)Q(t)x(t)-\frac{1}{2}x^T(t)P^T(t)BR^{-1}(t)B^TP(t)x(t)+x^T(t)P^T(t)Ax(t) \tag{III.12.26}$$

整理，得

$$\dot{P}(t)=P^T(t)BR^{-1}(t)B^TP(t)-2P^T(t)A-Q(t) \tag{III.12.27}$$

由对称性，可证 $P^T(t)A=\frac{1}{2}(P(t)A+A^TP(t))$，因此有

$$\dot{P}(t)=P^T(t)BR^{-1}(t)B^TP(t)-P^T(t)A-A^TP(t)-Q(t) \tag{III.12.28}$$

式III.12.28即为连续 LQR 情况下的 HJB 方程，或最优控制需要满足的条件。这是一个关于 P 的微分方程，其边界条件为

$$P(t_f)=S$$

求解 $P(t)$ 后，可得最优控制为

$$u^*(t)=-R^{-1}(t)B^TP(t)x(t) \tag{III.12.29}$$

对于 R,Q 非时变的情况，或时变但 $t\to\infty$ 的情况，有 $\dot{P}(t)=0$，此时式III.12.28变为

$$P^TBR^{-1}B^TP-P^TA-A^TP-Q=0 \tag{III.12.30}$$

该式也称为**代数 Riccati 方程**，本质是一组代数方程，目前可用计算机很容易的求解。

小结一下，对于非时变的连续 LQR 问题，我们有如下的解析解法。

算法	连续 LQR 解析解
问题类型	LQR 调节控制 [连续]
已知	连续线性系统 $\dot{x}(t) = Ax(t) + Bu(t)$ 初始状态 x_0 半正定矩阵 R, Q, S
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_{t_f}\|\|_S + \int_0^{t_f} (\|\|x(t)\|\|_Q + \|\|u(t)\|\|_R)dt$
算法性质	model-based, DP, 解析解

Algorithm 31: 连续 LQR 解析解

Input: 连续线性系统 $\dot{x}(t) = Ax(t) + Bu(t)$

Input: 半正定矩阵 R, Q, S

Output: 最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$

$P \leftarrow \text{solve\_riccati\_algebra}(P^T BR^{-1}B^T P - P^T A - A^T P - Q = 0)$

$u^*(t) \leftarrow -R^{-1}B^T Px(t)$

(参考资料：《控制之美 • 卷 2》，张杰《最优控制》)

12.6 非线性近似 LQR 控制

上面的几节中，我们介绍了很多处理线性系统二次型代价的最优控制算法。然而，我们生活在一个非线性的世界，许多实际的系统都带有非线性的特性，代价也不总是二次型形式。LQR 有着简单的表达、优美的推导和解析解，但它的最大短板在于只能处理线性问题。对于非线性系统，我们可以尝试扩展 LQR 的适用范围。

12.6.1 公式推导

为简单起见，本节仅讨论离散情况。首先，让我们回顾一般的无约束 (非线性) 最优调节控制问题 (见问题12.1)。这里，我们的对象 $f(\cdot)$ 和代价 $g(\cdot)$ 都是已知的非线性函数。

之前的几节中，我们已经介绍过线性系统下的 LQR 处理方法。对于控制问题，将非线性问题线性化也是最直接的想法[15]，即将非线性的 $f(\cdot)$ 和 $g(\cdot)$ 在某个工作点附近展开，从而用某个工作点附近的 $Ax + Bu$ 和二次型代替它们。这里，我们的工作点不是状态空间中的一个点，而是**状态空间中的一组轨迹** $\mathbf{x}^{ref}$。

那么，就会产生两个问题。其一，这样求解的最优控制，是不是全局的最优？其二，如何获取这样的一组轨迹？

第一个问题很容易回答。$\mathbf{x}^{ref}$ 只是所有轨迹组成的庞大轨迹空间中的一个点。代价函数是定义在整个轨迹空间（注意不是状态空间）中的函数，这种局部线性化只是 $\mathbf{x}^{ref}$ 附近的线性化。在这种情况下，求取的最优也只是 $\mathbf{x}^{ref}$ 附近的最优。也就是说，这是一个局部的改进，并非全局的改进。

当我们意识到这是一种局部的改进，我们就自然想到，这种思路可以用于迭代求解。这就自然的回答了第二个问题：只需在初始给定一条可行的轨迹，随后迭代地在轨迹周围进行泰勒展开，每一次展开后将非线性问题转化为线性的 LQR 问题进行求解。每次展开求出的最优控制，都是当前轨迹周围的最优解，是一种局部改进。将最优控制作用于实际系统，就可以采样得到更好的一条轨迹，从而迭代地提升，最终同时搜索得到最优轨迹和最优控制。

在这里，读者需要注意区分两种不同的迭代。在 LQR 问题中，我们求解线性二次型最优控制，使用动态规划思想先进行后向迭代求解 $\mathbf{u}^*$，再根据系统状态方程进行前向迭代求解最优估计 $\mathbf{x}^*$。这里的迭代是基于时

[15] 类似于从 KF(卡尔曼滤波) 到 EKF(扩展卡尔曼滤波)，见上一章

间步的迭代。而对于上面的思路，我们将这样的一组" 后向迭代-前向迭代" 作为一个大步。每一大步都获得一条参考轨迹 $\mathbf{x}_r^{ref}$，从初始轨迹开始经过很多大步，最终才能得到最优轨迹 $\mathbf{x}^*$。可以说，LQR 的求解是“内层循环”，这里的轨迹迭代则是“外层循环”。

在上面的轨迹迭代思路下，我们具体地介绍两种求解方法：**iLQR**(iterative LQR, 迭代 LQR) 和 **DDP** (Differential Dynamic Processing, 微分动态规划)。它们的思路都是迭代优化轨迹，区别仅在于线性化的具体方式。

我们先来看**微分动态规划**方法。既然是动态规划，就毫无疑问要涉及贝尔曼方程。考虑离散情况下，单步子问题的贝尔曼方程

$$\min_{\mathbf{u}_{k:N}} J_{k:N}(\mathbf{x},\mathbf{u}) = \min_{\mathbf{u}_k}(g(x_k,u_k) + \mathcal{J}_{k+1:N}(x_{k+1})) \tag{III.12.31}$$

假设我们已知一条参考轨迹 $\mathbf{x}_{k:N}^{ref}$，我们考虑在它的邻域内对 x_k 和 u_k 进行优化。假设 x_k 上的扰动为 δx_k，u_k 上的扰动为 δu_k。为了简单起见，我们记

$$Q(x_k^{ref},u_k^{ref}) = g(x_k^{ref},u_k) + \mathcal{J}_{k+1:N}(f(x_k^{ref},u_k^{ref})) \tag{III.12.32}$$

在后续推导中，在不引起歧义的情况下，我们将 $\frac{\partial Q}{\partial x}$ 记为 Q_x，将 $\frac{\partial^2 Q}{\partial x \partial u}$ 记为 Q_{xu}。

我们对 $Q(x_k,u_k)$ 进行二阶泰勒展开

$$Q(x_k^{ref}+\delta x_k, u_k^{ref}+\delta u_k) = Q(x_k^{ref},u_k^{ref}) + \begin{bmatrix} Q_x & Q_u \end{bmatrix} \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix}^T \begin{bmatrix} Q_{xx} & Q_{ux} \\ Q_{xu} & Q_{uu} \end{bmatrix} \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix} + o(\delta x_k, \delta u_k)$$

也就是说，在轨迹 $\mathbf{x}^{ref}$ 附近，优化代价函数就相当于优化 $Q(x_k,u_k)$，而优化 Q 就相当于找出最优的 $\delta x_k, \delta u_k$。在二阶展开的情况下，问题在局部是凸的。我们通过求导可知最优控制需满足这样的条件：

$$\begin{bmatrix} Q_{xx} & Q_{ux} \\ Q_{xu} & Q_{uu} \end{bmatrix} \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix} + \begin{bmatrix} Q_x \\ Q_u \end{bmatrix} = 0$$

也就是

$$\nabla^2 Q(x_k,u_k) \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix} = -\nabla Q(x_k,u_k)$$

回顾离散时间 LQR 推导，我们要求的最优控制是作为状态量的函数 $u^*(x(k))$。现在，我们希望找到最好的改进 δu_k 也应该是依赖于 δx_k 的。因此我们有

$$\delta u_k^* = -Q_{uu}^{-1}(Q_u + Q_{xu}\delta x_k) \tag{III.12.33}$$

下面我们需要计算 $\nabla Q(x_k,u_k)$ 和 $\nabla^2 Q(x_k,u_k)$。我们注意到 Q 的表达式含有 $\mathcal{J}_{k+1:N}(x_{k+1})$，而 x_{k+1} 通过状态方程依赖于 x_k, u_k。因此，我们先把状态方程进行一阶展开，有

$$\delta x_{k+1} = \begin{bmatrix} \nabla_x^T f & \nabla_u^T f \end{bmatrix} \begin{bmatrix} \delta x_k \\ \delta u_k \end{bmatrix}$$

记

$$\mathcal{J}_x = \frac{\partial \mathcal{J}_{k+1:N}}{\partial x_{k+1}}$$

于是我们有

$$\begin{bmatrix} Q_x \\ Q_u \end{bmatrix} = \begin{bmatrix} g_x + \nabla_x^T f \mathcal{J}_x \\ g_u + \nabla_u^T f \mathcal{J}_x \end{bmatrix} (x_k, u_k) \tag{III.12.34}$$

对一阶导继续求导，我们可以得到二阶的求导结果。注意此处需要对系统状态方程 $f(x,u)$ 求二阶导。

$$\begin{aligned} Q_{xx} &= g_{xx} + f_{xx}\mathcal{J}_x + \mathcal{J}_x^T \nabla_x^T f \mathcal{J}_x \\ Q_{xu} &= g_{xu} + f_{xu}\mathcal{J}_x + \mathcal{J}_u^T \nabla_x^T f \mathcal{J}_x \\ Q_{ux} &= g_{ux} + f_{ux}\mathcal{J}_x + \mathcal{J}_x^T \nabla_x^T f \mathcal{J}_u \\ Q_{uu} &= g_{uu} + f_{uu}\mathcal{J}_u + \mathcal{J}_u^T \nabla_u^T f \mathcal{J}_u \end{aligned} \tag{III.12.35}$$

注意这里的 $\nabla Q(x_k, u_k)$ 和 $\nabla^2 Q(x_k, u_k)$ 都是关于 x_k, u_k 的函数矩阵，并且还需要对 $\mathcal{J}$ 进行求导，也就是我们需要得到其解析表示。这就意味着，δu_k^* 表达式中的系数矩阵都需要在后向迭代中计算。

算法	**非线性最优控制-DDP**
问题类型	一般最优调节控制 [离散]
已知	离散非线性系统 $x_{k+1} = f(x_k, u_k)$ 单步代价函数 $g(x_k, u_k)$，终端代价函数 $h(x_{t_f})$ 初始状态 x_0
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} h(x_{t_f}) + \sum_{k=0}^{N-1} g(x_k, u_k)$
算法性质	model-based, DP, 迭代解

让我们总结一下，写出 **DDP 算法**的全貌。注意，如果严格按照理论推导，每一步的 $\mathcal{J}_{k:N}$ 都依赖于上一步的优化结果，从而写出 $\mathcal{J}_x, \mathcal{J}_u$ 解析表达式可能会非常困难。因此，实际使用中，我们往往用末端代价对 x_N, u_N 的导数代替迭代求解的 $\mathcal{J}_x, \mathcal{J}_u$。

Algorithm 32: 非线性最优控制-DDP

Input: 离散非线性系统 $x_{k+1} = f(x_k, u_k)$

Input: 单步代价函数 $g(x_k, u_k)$，终端代价函数 $h(x_{t_f})$

Output: 最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}}(h(x_{t_f}) + \sum_{k=0}^{N-1} g(x_k, u_k))$

$\mathbf{x}^0, \mathbf{u}^0 \leftarrow feasible\_trajectory(f, g, h)$

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 是一种迭代算法。它的外循环迭代次数 N_R 取决于轨迹 $\mathbf{x}^r$ 是否收敛。每一次外循环中，都需要借助真实环境进行前向迭代生成新轨迹。相比于线性版本的 LQR，它无法给出解析解，也无法给出不同 x_0 下最优控制的通用表达式，只能得到对应某个 x_0 的最优轨迹相关的控制率。

DDP 算法可以用于求解各类非线性优化问题，但它的缺陷是求解比较复杂。尤其是在计算过程中它需要求解系统状态方程的二阶导数，也就是 Hessian 矩阵。不过，二阶展开能保证它的收敛性，因此它常用于实时性要求不高的场合。

算法	**非线性最优控制-iLQR**
问题类型	一般最优调节控制 [离散]
已知	离散非线性系统 $x_{k+1} = f(x_k, u_k)$ 单步代价函数 $g(x_k, u_k)$，终端代价函数 $h(x_{t_f})$ 初始状态 x_0
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} h(x_{t_f}) + \sum_{k=0}^{N-1} g(x_k, u_k)$
算法性质	model-based, DP, 迭代解

对于实时性要求更高的场合，一阶的算法更加具有优势，这就是 **iLQR 算法**。iLQR 和 DDP 的唯一区别就在于：计算 Q 的 Hessian 矩阵时，去掉了 $f(x, u)$ 的二阶导数项。我们将 iLQR 算法总结如下。

Algorithm 33: 非线性最优控制-iLQR

Input: 离散非线性系统 $x_{k+1} = f(x_k, u_k)$

Input: 单步代价函数 $g(x_k, u_k)$，终端代价函数 $h(x_{t_f})$

Output: 最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}}(h(x_{t_f}) + \sum_{k=0}^{N-1} g(x_k, u_k))$

$\mathbf{x}^0, \mathbf{u}^0 \leftarrow feasible\_trajectory(f, g, h)$

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**

    $u_k^r \leftarrow u_k^{r-1} - (Q_{uu}^r)^{-1}(Q_u^r + Q_{xu}^r(x_k^r - x_k^{r-1}))$

    $x_{k+1}^r \leftarrow f(x_k^r, u_k^r)$

  $\mathbf{x}^r \leftarrow x_{0:N}^r$

iLQR 虽然速度更快，但实际使用时其更新可能不稳定，迭代后的轨迹可能比迭代前的代价还高。究其原因，iLQR 对系统状态方程做的近似只有一阶，而对于高度非线性的系统，一阶展开的拟合能力是很有限的。这类似于求解数值优化问题时，一阶的最速下降法很可能没有牛顿法收敛速度快。

(参考资料：https://zhuanlan.zhihu.com/p/690023196)

12.6.2 倒立摆求解

[本部分内容将在后续版本中更新，敬请期待]

12.7 最优跟踪控制

12.7.1 LQR 跟踪控制

在前面的几节中，我们讨论了对于在无约束的条件下进行最优调节控制的算法。实际上，LQR 类方法不仅用于调节控制问题，也可用于跟踪控制问题。在前面我们已经定义了一般 (非线性) 系统下的最优跟踪控制问题，现在我们将其特化到线性二次型情况。

问题	**LQR 跟踪控制 [离散]**
问题简述	已知线性离散系统，求使二次型代价最小的跟踪控制输入
已知	离散线性系统 $x_{k+1} = Ax_k + Bu_k$ 初始状态 x_0，参考轨迹 $\mathbf{x}_d$ 半正定矩阵列 R_k, Q_k，半正定矩阵 S
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N - x_{d,N}\|\|_S$ $+ \sum_{k=0}^{N-1}(\|\|x_k - x_{d,k}\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$

在跟踪问题中，代价函数是关于 $\mathbf{x}, \mathbf{x}_d, \mathbf{u}$ 的函数

$$J_{0:N}(\mathbf{x}, \mathbf{x}_d, \mathbf{u}) = ||x_N - x_{d,N}||_S + \sum_{k=0}^{N-1}(||x_k - x_{d,k}||_{Q_k} + ||u_k||_{R_k})$$

针对该问题，我们可以采用扩张状态变量的思路，将其化为一个类似 LQR 调节问题的形式。具体来说，假设 x_d 满足

$$x_{d,k+1} = A_{d,k}x_{d,k} \tag{III.12.36}$$

如果轨迹 $x_{d,:}$ 并非来自于模型，可使用如下方式获取一个矩阵 A_d, k:

$$A_{d,k} = \frac{1}{||x_{d,k}||^2}x_{d,k+1}x_{d,k}^T + \lambda\left(I - \frac{1}{||x_{d,k}||^2}x_{d,k}x_{d,k}^T\right) \tag{III.12.37}$$

特别地，若 x_d 为恒值，则 $A_d = I_n$。因此，我们可以记扩张状态为

$$x_{e,k} = \begin{bmatrix} x_k \\ x_{d,k} \end{bmatrix} \tag{III.12.38}$$

则扩张状态满足如下系统方程

$$x_{e,k+1} = A_{e,k}x_{e,k} + B_e u_k \tag{III.12.39}$$

其中

$$A_{e,k} = \begin{bmatrix} A & 0 \\ 0 & A_{d,k} \end{bmatrix}, B_e = \begin{bmatrix} B \\ 0 \end{bmatrix} \tag{III.12.40}$$

因此，关于 $\mathbf{x}, \mathbf{x}_e, \mathbf{u}$ 的代价函数就可以写为关于 $\mathbf{x}_e, \mathbf{u}$ 的代价函数

$$J_{0:N}(\mathbf{x}_e, \mathbf{u}) = ||x_{e,N}||_{S_e} + \sum_{k=0}^{N-1}(||x_{e,k}||_{Q_{e,k}} + ||u_k||_{R_k})$$

其中

$$\begin{aligned} S_e &= \begin{bmatrix} S & -S \\ -S & S \end{bmatrix} \\ Q_{e,k} &= \begin{bmatrix} Q_k & -Q_k \\ -Q_k & Q_k \end{bmatrix} \end{aligned} \tag{III.12.41}$$

因此，我们可以直接使用 LQR 推导的结论。我们总结离散情况下的 LQR 跟踪控制算法如下

算法	LQR 跟踪控制
问题类型	LQR 跟踪控制 [离散]
已知	离散线性系统 $x_{k+1} = Ax_k + Bu_k$ 参考轨迹 $x_{d,1:N}$, 初始状态 x_0 半正定矩阵列 R_k, Q_k，半正定矩阵 S
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N\|\|_S$ $+ \sum_{k=0}^{N-1}(\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$
算法性质	model-based, DP, 迭代解

Algorithm 34: LQR 跟踪控制

Input: 离散线性系统 $x_{k+1} = Ax_k + Bu_k$

Input: 参考轨迹 $x_{d,1:N}$, 初始状态 x_0

Input: 半正定矩阵列 R_k, Q_k，半正定矩阵 S

Output: 最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{x}_d, \mathbf{u})$

$$P_N \leftarrow \begin{bmatrix} S & -S \\ -S & S \end{bmatrix}, B_e \leftarrow \begin{bmatrix} B & 0 \end{bmatrix}$$

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**

$$\nu_k \leftarrow F_k x_{e,k}$$

$$x_{e,k+1} \leftarrow A_{e,k} x_{e,k} + B_e \nu_k$$

$$\mathbf{u}^* \leftarrow \nu_{0:N}(x)$$

对应的 python 代码如下所示

```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 输入增量控制

上述方法可以有效地生成跟踪轨迹的控制量。不过，有时候我们希望跟踪时控制量尽量平滑，尽量不要出现突变 (如实际执行器限制)。此时，我们会重新定义代价函数为关于 $\mathbf{x}, \mathbf{x}_d, \Delta\mathbf{u}$ 的函数

问题	**LQR 平滑跟踪控制 [离散]**
问题简述	已知线性离散系统，求使二次型代价最小的平滑跟踪控制输入
已知	离散线性系统 $x_{k+1} = Ax_k + Bu_k$ 初始状态 x_0，参考轨迹 $\mathbf{x}_d$ 半正定矩阵列 R_k, Q_k，半正定矩阵 S
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N - x_{d,N}\|\|_S$ $+\sum_{k=0}^{N-1}(\|\|x_k - x_{d,k}\|\|_{Q_k} + \|\|\Delta u_k\|\|_{R_k})$
算法性质	model-based, DP, 迭代解

其中

$$\Delta u_k = u_k - u_{k-1} \tag{III.12.42}$$

此时，我们仍然考虑进行状态扩张，扩展状态为

$$x_{e,k} = \begin{bmatrix} x_k \\ x_{d,k} \\ u_{k-1} \end{bmatrix} \in \mathbb{R}^{2n+m} \tag{III.12.43}$$

则扩张状态满足如下系统方程

$$x_{e,k+1} = A_{e,k}x_{e,k} + B_e\Delta u_k$$

其中

$$A_{e,k} = \begin{bmatrix} A & 0 & 0 \\ 0 & A_{d,k} & 0 \\ 0 & 0 & I \end{bmatrix}, B_e = \begin{bmatrix} B \\ 0 \\ I \end{bmatrix} \tag{III.12.44}$$

因此，代价函数

$$J_{0:N}(\mathbf{x}_e, \Delta\mathbf{u}) = ||x_{e,N}||_{S_e} + \sum_{k=0}^{N-1}(||x_{e,k}||_{Q_{e,k}} + ||\Delta u_k||_{R_k})$$

其中

$$
\begin{aligned}
S_e &= \begin{bmatrix} S & -S & 0 \\ -S & S & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
Q_{e,k} &= \begin{bmatrix} Q_k & -Q_k & 0 \\ -Q_k & Q_k & 0 \\ 0 & 0 & 0 \end{bmatrix}
\end{aligned} \tag{III.12.45}
$$

和跟踪控制类似，我们总结 **LQR 输入增量控制**算法如下

算法	**LQR 输入增量控制**
问题类型	LQR 平滑跟踪控制 [离散]
已知	离散线性系统 $x_{k+1} = Ax_k + Bu_k$ 参考线性系统矩阵列 $A_{d,k}$ 半正定矩阵列 R_k, Q_k，半正定矩阵 S 初始状态 $x_0, x_{d,0}$
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N - x_{d,N}\|\|_S$ $+ \sum_{k=0}^{N-1} (\|\|x_k - x_{d,k}\|\|_{Q_k} + \|\|\Delta u_k\|\|_{R_k})$
算法性质	model-based, DP, 解析解

Algorithm 35: LQR 输入增量控制

Input: 离散线性系统 $x_{k+1} = Ax_k + Bu_k$

Input: 线性系统 $x_{k+1} = Ax_k + Bu_k$

Input: 参考线性系统矩阵列 $A_{d,k}$

Input: 半正定矩阵列 R_k, Q_k，半正定矩阵 S

Output: 最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{x}_d, \mathbf{u})$

$P_N \leftarrow \begin{bmatrix} S & -S & 0 \\ -S & S & 0 \\ 0 & 0 & 0 \end{bmatrix}, B_e \begin{bmatrix} B \\ 0 \\ I \end{bmatrix}$

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**

$\Delta\nu_k \leftarrow F_k x_{e,k}$

$x_{e,k+1} \leftarrow A_{e,k} x_{e,k} + B_e \nu_k$

$\nu_k \leftarrow n u_{k-1} + \Delta\nu_k$

$\mathbf{u}^* \leftarrow \nu_{0:N}(x)$

(参考资料：《控制之美 • 卷 2》)

12.8 约束最优控制问题

前面讨论的问题都是无约束下的最优控制问题。例如第 3 节中，给出了无约束连续最优控制问题。实际问题中，针对连续系统，可能出现各类形式的约束，如等式约束、微分约束、积分约束、不等式约束等。

下面，我们分别定义几类不同约束下的连续最优控制问题。实际问题也可能是这几类问题的组合。首先，我们看等式和微分约束的情况。

问题	等式约束最优调节控制 [连续]
问题简述	已知连续系统、代价函数，求等式约束下最优的调节控制输入
已知	连续系统 $\dot{x}=f(x,u,t)$ 代价函数 $J(x,u)=h(x_{t_f},t_f)+\int_0^{t_f}g(\mathbf{x}(t),\mathbf{u}(t),t)dt$ 初始状态 $x_{t_0}=x_0$
求	最优控制 $\mathbf{u}^*(t)=\arg\min_{\mathbf{u}}J(x(t),u(t))$ 使得 $F(x(t),t)=0$

问题	微分约束最优调节控制 [连续]
问题简述	已知连续系统、代价函数，求微分约束下最优的调节控制输入
已知	连续系统 $\dot{x}=f(x,u,t)$ 代价函数 $J(x,u)=h(x_{t_f},t_f)+\int_0^{t_f}g(x(t),u(t),t)dt$ 初始状态 $x_{t_0}=x_0$
求	最优控制 $u^*(t)=\arg\min_{u}J(x(t),u(t))$ 使得 $F(\dot{x}(t),x(t),t)=0$

事实上，对于这两种问题，我们只需使用拉格朗日乘子法。我们定义拉格朗日乘子 λ，将代价函数扩充为

$$\bar{J}(x,\lambda,u)=h(x_{t_f},t_f)+\int_0^{t_f}g(x(t),u(t),t)+\lambda F(\dot{x}(t),x(t),t)dt$$

注意到 $\dot{x}(t)=f(x(t),u(t),t)$ 已知。因此，我们可以记扩充后的过程代价为

$$\bar{g}(x(t),\lambda,u(t),t)=g(x(t),u(t),t)+\lambda F(f(x(t),u(t),t),x(t),t) \tag{III.12.46}$$

即

$$\bar{J}(x,\lambda,u)=h(x_{t_f},t_f)+\int_0^{t_f}\bar{g}(x(t),\lambda,u(t),t)dt \tag{III.12.47}$$

随后，我们将 $x(t)$ 和 λ 同时作为新的状态，即可写出新的 HJB 方程

$$-\frac{\partial\bar{\mathcal{J}}}{\partial t}(x(t),\lambda,t)=\min_{u(t)}\bar{\mathcal{H}}\left(x(t),\lambda,u(t),\frac{\partial\mathcal{J}}{\partial x},t\right) \tag{III.12.48}$$

其中，扩充的 Hamilton 量为

$$\bar{\mathcal{H}}\left(x,\lambda,u,\frac{\partial\mathcal{J}}{\partial x},t\right)=g(x,u,t)+\lambda F(f(x,u,t),x,t)+\frac{\partial\mathcal{J}}{\partial x}(x,\lambda,t)\cdot f(x,u,t) \tag{III.12.49}$$

注意，由于引入的拉格朗日乘子非时变，扩充后的 Hamilton 量中并没有 $\frac{\partial \mathcal{J}}{\partial \lambda}^T \dot{\lambda}$ 的一项。

接下来，我们考虑 $\int_0^{t_f} b(\dot{x}(t), x(t), t)dt = B$ 形式的积分约束

问题	积分约束最优调节控制 [连续]
问题简述	已知连续系统、代价函数，求积分约束下最优的调节控制输入
已知	连续系统 $\dot{x} = f(x, u, t)$ 代价函数 $J(x, u) = h(x_{t_f}, t_f) + \int_0^{t_f} g(x(t), u(t), t)dt$ 初始状态 $x_{t_0} = x_0$
求	最优控制 $u^*(t) = \arg\min_{\mathbf{u}} J(x(t), u(t))$ 使得 $\int_0^{t_f} b(\dot{x}(t), x(t), t)dt = B$

引入新的状态变量 $z(t)$，其定义为

$$z(t) = \int_0^t b(\dot{x}(\tau), x(\tau), \tau)d\tau \tag{III.12.50}$$

注意到 $\dot{x}(t) = f(x(t), u(t), t)$ 已知，因此状态变量 $z(t)$ 的系统轨线为

$$\dot{z}(t) = b(f(x(t), u(t), t), x(t), t)$$

且其初末状态为 “$z(0) = 0$，$z(t_f) = B$。我们可以记扩充后的终端代价为

$$\bar{h}(x_{t_f}, z_{t_f}, t_f) = h(x_{t_f}, t_f)(z(t_f) - B) \tag{III.12.51}$$

即

$$\bar{J}(x, \lambda, z, u) = \bar{h}(x_{t_f}, z_{t_f}, t_f) + \int_0^{t_f} g(x(t), u(t), t)dt$$

随后，我们将 $x(t)$ 和 $z(t)$ 同时作为新的状态，即可写出新的 HJB 方程

$$-\frac{\partial \bar{\mathcal{J}}}{\partial t}(x(t), z(t), t) = \min_{u(t)} \bar{\mathcal{H}}\left(x(t), z(t), u(t), \frac{\partial \mathcal{J}}{\partial x}, \frac{\partial \mathcal{J}}{\partial z}, t\right) \tag{III.12.52}$$

其中，扩充的 Hamilton 量为

$$\bar{\mathcal{H}}\left(x, z, u, \frac{\partial \mathcal{J}}{\partial x}, \frac{\partial \mathcal{J}}{\partial z}, t\right) = g(x, u, t) + \frac{\partial \mathcal{J}}{\partial x} \cdot f(x, u, t) + \frac{\partial \mathcal{J}}{\partial z} \cdot b(f(x, u, t), x, t) \tag{III.12.53}$$

满足终值条件

$$\bar{h}(x_{t_f}, z_{t_f}, t_f) = 0 \tag{III.12.54}$$

注意，和无约束状态类似，此处的 $\frac{\partial \mathcal{J}}{\partial x}, \frac{\partial \mathcal{J}}{\partial z}$ 都是未知的关于 $x(t), z(t), t$ 的泛函，最小代价泛函 $\mathcal{J}$ 需要满足的必要条件，就是对 $x(t)$ 和 $z(t)$ 分别求导后，导数要等于这两个泛函。

类似地，约束连续最优控制问题没有通用的解析解，在面对具体问题时需要具体推导。

(参考资料：张杰《最优控制》)

13 模型预测控制基础

13.1 基本概念

在上一章中，我们介绍了不少求解最优 (调节/跟踪) 控制问题的方法。这些方法基于动态规划思想，通过后向迭代和前向迭代求出最小化某种优化目标的最优控制序列，统称为最优控制方法。

最优控制方法清晰、简洁，但也存在一些问题。首先，此类方法往往只能处理无约束的优化问题，最多再处理等式约束问题，而**不等式约束**比较难处理。其次，此类方法要求对象有精确的数学模型，最好还是线性模型。如果模型估计有**误差**，或实际系统中有无法建模的**干扰**，则此类算法会受一定影响。

回顾第3章中，我们介绍了大量优化问题及其求解方法。其中一些方法可以求解带有不等式约束的优化问题，并通过计算机进行迭代求解。如果我们可以将含不等式约束的最优控制问题转化为含不等式约束的优化问题，就可以利用这些算法，来进行求解。

具体来说，假设我们知道离散的系统方程 $x_{k+1} = f(x_k, u_k)$。假设 x_0 已知，则任意时刻的系统状态 x_k 就是历史输入序列 $u_0, ..., u_k$ 的函数。既然如此，而最优控制问题的代价函数 $J_{1:k}(\mathbf{x}, \mathbf{u})$ 又是系统状态序列 $x_0, ..., x_k$ 的函数，那么就可以将代价函数写成输入序列 $u_0, ..., u_k$ 的函数，从而构造只以输入序列 $u_0, ..., u_k$ 为优化变量的优化问题。

在上述过程中，我们反复地使用系统方程 $x_{k+1} = f(x_k, u_k)$，根据历史状态和输入迭代地预测后续的状态。这个过程就是利用系统的模型进行预测。因此，采用上述思路求解最优控制问题的方法就称为**模型预测控制** (Model Predictive Control, MPC)。

以上的思路虽然可以以 $u_0, ..., u_k$ 为变量，求解出一条最优控制序列 $u_0^*, ..., u_k^*$，但是如果将这样的的控制序列直接应用于系统，则无法应对前述的模型误差和干扰问题。无论是模型误差还是干扰误差，都会在预测的过程中积累，从而使预测的最优控制偏离实际的最优解。

这个问题的本质是，上述求解过程是**开环**的，并没有使用系统真实状态，也就是反馈。事实上，如果求解的速度足够快，我们可以在**每个离散时间步** t，都使用系统反馈的状态作为预测的初值 $x_{t|t}$，并对一定长度 N_p 的区间进行最优化问题求解。求解得到的轨迹 $u_{t|t}^*, ..., u_{t|N_p}^*$ 会逐渐偏离真实的最优轨迹，所以我们仅取其第一项 $u_{t|t}^*$ 作为当前步的控制量去执行，并直接舍弃其余预测量 $u_{t|t+1}^*, ..., u_{t|N_p}^*$。在下一个时间步 $t+1$，我们重复这个过程，再预测 N_p 步而只取 1 步。这样，每一次执行的控制量都采用了系统反馈的最新信息，可以有效应对模型误差与干扰。

像这样的控制方案，我们称为**滚动优化控制**，或称为**闭环 MPC**。在本章及后续章节中，我们仅讨论闭环下的 MPC。

滚动优化的思想确实兼顾了模型预测、优化问题求解、状态反馈以及干扰/模型误差应对等方面。但其代价是：要在每个控制周期内，完成整个优化问题的求解。注意：若控制量 u_k 为 N_u 维的向量，而预测区间长度为 N_p，则整个优化问题的优化变量为 $N_u N_P$ 维。好在，随着计算机技术的发展，大规模优化问题的实时求解已不再困难。目前实际工程中的许多求解器 (软件 + 硬件)，可以对数十维变量的优化问题在 0.001s 内完成求解，从而使 MPC 达到 1000Hz 左右的控制频率。

根据模型是否线性以及是否存在约束，MPC 可以分为不同的种类。最简单的 MPC 算法是对于线性系统、二次型优化目标 (即 LQR 问题) 的算法。下面，我们将针对无约束和有约束的情况，具体讨论滚动优化 MPC 的算法。

13.2 无约束线性 MPC

13.2.1 调节控制

首先，我们仍考虑最简单的优化控制问题：无约束的 LQR 调节问题。假设预测区间长度为 N_p，则每一时刻需要求解的优化问题的优化目标为

$$\min_{u} J_{k:k+N_p}(\mathbf{x},\mathbf{u}) = \frac{1}{2}\left(||x_{k+N_p}||_S + \sum_{i=k}^{k+N_p-1}\left(||x_{i+1}||_{Q_i} + ||u_i||_{R_i}\right)\right) \tag{III.13.1}$$
$$s.t. \quad x_{t+1} = Ax_t + Bu_t$$

假设状态 x_t 的维数为 N_x，输入 u_t 的维数为 N_u。对于时刻 k，我们取当前系统状态 (假设状态可观，无观测误差)x_k 为预测状态初值

$$x_{k|k} = x_k \tag{III.13.2}$$

则有

$$x_{k|t+1} = Ax_{k|t} + Bu_{k|t}$$

其中 $u_{k|t}t = k, k+1, ..., k+N_p$ 为我们的优化变量。为简单起见，我们将其记为

$$U_k = \begin{bmatrix} u_{k|k} \\ u_{k|k+1} \\ \cdots \\ u_{k|k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p N_u} \tag{III.13.3}$$

类似的，对于状态，记

$$X_k = \begin{bmatrix} x_{k|k} \\ x_{k|k+1} \\ \cdots \\ x_{k|k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p N_x} \tag{III.13.4}$$

注意上述 X_k 中不包括 x_k。由上滚动优化思想可知，对于最终需要求解的优化问题，X_k 的所有变量均为中间变量，即初值 x_k 和 U_k 的函数。我们来具体的表达一下这个函数关系

$$X_k = \Phi x_k + \Gamma U_k \tag{III.13.5}$$

其中

$$\Phi = \begin{bmatrix} I \\ A \\ A^2 \\ \cdots \\ A^{N_p} \end{bmatrix} \in \mathbb{R}^{N_p N_x \times N_x} \tag{III.13.6}$$

$$\Gamma = \begin{bmatrix} 0 & & & \\ B & & & \\ AB & B & & \\ \cdots & \cdots & \cdots & \\ A^{N_p-1}B & \cdots & AB & B \end{bmatrix} \in \mathbb{R}^{N_p N_x \times N_p N_u} \tag{III.13.7}$$

接下来，让我们表达时间步 k 时的优化目标 J_k。记

$$\mathcal{Q} = \begin{bmatrix} Q_k & & & \\ & Q_{k+1} & & \\ & & \cdots & \\ & & & S \end{bmatrix} \in \mathbb{R}^{N_p N_x \times N_p N_x} \tag{III.13.8}$$

取 $R_{k+N_p} = 0_{N_u \times N_u}$，有

$$\mathcal{R} = \begin{bmatrix} R_k & & & \\ & R_{k+1} & & \\ & & \cdots & \\ & & & R_{k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p N_x \times N_p N_x} \tag{III.13.9}$$

假设 $\mathcal{Q} = \mathcal{Q}^T, \mathcal{R} = \mathcal{R}^T$，则 J_k 是关于优化变量 U_k 的二次型

$$\begin{aligned} J_k(U_k) &= \frac{1}{2}(X_k^T \mathcal{Q} X_k + U_k^T \mathcal{R} U_k) \\ &= \frac{1}{2}((\Phi x_k + \Gamma U_k)^T \mathcal{Q}(\Phi x_k + \Gamma U_k) + U_k^T \mathcal{R} U_k) \\ &= \frac{1}{2}(U_k^T(\mathcal{R} + \Gamma^T \mathcal{Q}\Gamma)U_k + 2x_k^T \Phi^T \mathcal{Q}\Gamma U_k + x_k^T \Phi^T \mathcal{Q}\Phi x_k) \end{aligned}$$

进一步，记

$$\begin{aligned} G_k &= \Phi^T \mathcal{Q}\Gamma \\ H_k &= \mathcal{R} + \Gamma^T \mathcal{Q}\Gamma \end{aligned} \tag{III.13.10}$$

则有

$$J_k(U_k) = \frac{1}{2}U_k^T H_k U_k + x_k^T G_k U_k \tag{III.13.11}$$

上述表达式中，省去了和优化变量无关的 $x_k^T \Phi^T \mathcal{Q}\Phi x_k$ 一项，不影响优化问题求解。

至此，我们将 (无约束)LQR 最优控制问题通过滚动更新的方法，转变为了一个无约束的 QP 问题。根据3.5节的结论，该问题具有解析解

$$U_k^* = -H_k^{-1} G_k^T x_k \tag{III.13.12}$$

总结上述算法，我们有**无约束线性 MPC**。注意，当系统非时变时，该算法的大部分计算可离线完成，在线计算实际上只是线性反馈。若系统为时变系统，即 A, B 依赖于 x_k，则所有离线计算需改为在线计算。

算法	无约束线性 MPC
问题类型	LQR 调节控制 [离散]
已知	离散线性系统矩阵 A, B 半正定矩阵列 R_k, Q_k，半正定矩阵 S 系统实时状态 $x_{1:N}$
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N\|\|_S + \sum_{k=0}^{N-1}(\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$
算法性质	model-based, MPC, 解析解

Algorithm 36: 无约束线性 MPC

Input: 离散线性系统矩阵 A, B

Input: 半正定矩阵列 $R_{1:N}, Q_{1:N}$，半正定矩阵 S

Input: 系统实时状态 $x_{1:N}$

Output: 最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$

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 跟踪控制

[主要内容：无约束最优跟踪问题、无约束线性 MPC 跟踪算法、对比 MPC 和 OC]

[本部分内容将在后续版本中更新，敬请期待]

(参考资料：《控制之美 • 卷 2》)

13.3 有约束线性 MPC

下面，我们来考虑有约束的情况。首先，让我们定义有 (线性) 约束情况下的 LQR 控制问题。

问题	**有约束 LQR 调节控制 [离散]**
问题简述	已知线性离散系统和约束，求使二次代价最小的调节控制输入
已知	离散线性系统矩阵 A, B 系统实时状态 $x_{1:N}$ 半正定矩阵列 $R_{1:N}, Q_{1:N}$，半正定矩阵 S 矩阵列 $M_{x,1:N}, M_{u,1:N}$，向量列 $b_{1:N}$
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N\|\|_S + \sum_{k=0}^{N-1}(\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$ 使得 $M_{x,k}x_k + M_{u,k}u_k \leq b_k, k = 1, 2, ..., N$

这里，我们仅考虑线性形式的约束。对每个时刻 k，约束形如

$$M_{x,k}x_k + M_{u,k}u_k \leq b_k \tag{III.13.13}$$

其中，$M_{x,k} \in \mathbb{R}^{m\times N_x}$, $M_{u,k} \in \mathbb{R}^{m\times N_u}$, $b_k \in \mathbb{R}^m$，m 为同一时刻的约束维度数。
类似前面的处理，我们也将其写为预测区间 N_p 内的大矩阵的形式

$$\mathcal{M}_x = \begin{bmatrix} M_{x,k} & & & \\ & M_{x,k+1} & & \\ & & \cdots & \\ & & & M_{x,k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p m\times N_p N_x} \tag{III.13.14}$$

$$\mathcal{M}_u = \begin{bmatrix} M_{u,k} & & & \\ & M_{u,k+1} & & \\ & & \cdots & \\ & & & M_{u,k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p m\times N_p N_u} \tag{III.13.15}$$

$$\beta_k = \begin{bmatrix} b_k \\ b_{k+1} \\ \cdots \\ b_{k+N_p} \end{bmatrix} \in \mathbb{R}^{N_p m} \tag{III.13.16}$$

则预测区间内的不等式约束可以统一表示为

$$\mathcal{M}_x X_k + \mathcal{M}_u U_k \leq \beta_k \tag{III.13.17}$$

代入 X_k 的表达式，有

$$\mathcal{M}_x(\Phi x_k + \Gamma U_k) + \mathcal{M}_u U_k \leq \beta_k$$

即

$$(\mathcal{M}_x\Gamma + \mathcal{M}_u)U_k \leq \beta_k - \mathcal{M}_x\Phi x_k$$

记

$$\begin{aligned} M_k &= \mathcal{M}_x\Gamma + \mathcal{M}_u \in \mathbb{R}^{N_p m\times N_p N_u} \\ \mathbf{b}_k &= \beta_k - \mathcal{M}_x\Phi x_k \in \mathbb{R}^{N_p m} \end{aligned} \tag{III.13.18}$$

此时，预测区间内的优化问题变为

$$\begin{aligned} \min_{U_k} J_k(U_k) &= \frac{1}{2}U_k^T H_k U_k + x_k^T G_k U_k \\ s.t. \quad & M_k U_k \leq \mathbf{b}_k \end{aligned} \tag{III.13.19}$$

这是一个含不等式约束的 QP 问题。我们可以使用3.5节介绍的算法，对其进行求解。

总结上述算法，我们有**不等式约束线性 MPC**。注意：由于约束条件中的 $\mathbf{b}_k$ 和 x_k 有关，**优化问题必须在线求解**。这与无约束算法中，反馈矩阵 F_k 可以离线求解有本质区别。

Algorithm 37: 不等式约束线性 MPC

Input: 离散线性系统矩阵 A, B

Input: 半正定矩阵列 $R_{1:N}, Q_{1:N}$，半正定矩阵 S

Input: 矩阵列 $M_{x,1:N}, M_{u,1:N}$，向量列 $b_{1:N}$

Input: 系统实时状态 $x_{1:N}$

Output: 最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} J(\mathbf{x}, \mathbf{u})$

Offline:

$\Phi \leftarrow [I, A, A^2, ..., A^{N_p}]^T$

$\Gamma \leftarrow \Gamma(A, B)$

Online:

for $k \in 1, 2, ..., N$ **do**

$\mathcal{Q}_k \leftarrow \text{diag}(Q_k, Q_{k+2}, ..., S)$

$\mathcal{R}_k \leftarrow \text{diag}(R_k, R_{k+1}, ..., R_{k+N_p})$

$G_k \leftarrow \Phi^T \mathcal{Q} \Gamma$

$H_k \leftarrow \mathcal{R} + \Gamma^T \mathcal{Q} \Gamma$

$\mathcal{M}_x \leftarrow \text{diag}(M_{x,k}, M_{x,k+1}, ...M_{x,k+N_p})$

$\mathcal{M}_u \leftarrow \text{diag}(M_{u,k}, M_{u,k+1}, ...M_{u,k+N_p})$

$\beta_k \leftarrow [b_k^T, b_{k+1}^T, ..., b_{k+N_p}^T]^T$

$M_k \leftarrow \mathcal{M}_x \Gamma + \mathcal{M}_u$

$\mathbf{b}_k \leftarrow \beta_k - \mathcal{M}_x \Phi x_k$

$U_k^* \leftarrow \text{solve\_QP\_barrier}(H_k, G_k^T x_k, none, none, M_k, \mathbf{b}_k)$

$u_k^* \leftarrow (U_k^*)_{1:N_u}$

$\mathbf{u}^* \leftarrow u_{1:N}^*$

算法	**不等式约束线性 MPC**
问题类型	有约束 LQR 调节控制 [离散]
已知	离散线性系统矩阵 A, B 半正定矩阵列 R_k, Q_k，半正定矩阵 S 系统实时状态 $x_{1:N}$ 矩阵列 $M_{x,1:N}, M_{u,1:N}$，向量列 $b_{1:N}$
求	最优控制 $\mathbf{u}^* = \arg\min_{\mathbf{u}} \|\|x_N\|\|_S + \sum_{k=0}^{N-1}(\|\|x_k\|\|_{Q_k} + \|\|u_k\|\|_{R_k})$ 使得 $M_{x,k}x_k + M_{u,k}u_k \leq b_k, k = 1, 2, ..., N$
算法性质	model-based, MPC, 迭代解

(参考资料：《控制之美 • 卷 2》)
(参考资料：https://zhuanlan.zhihu.com/p/368705959)

Part IV

机器人控制基础

14 机器人控制概述

机器人的控制，是指我们希望一台机器人按照我们希望的方式运动，包括到达指定的位置、跟踪指定的轨迹、产生一定的力或力矩，等等。根据第 II 部分介绍的运动学和动力学模型，我们已经可以实现**开环控制**。然而，在实际系统中，仅仅开环的控制是远远不够的。

这是因为，在一个机器人系统中，电机可能是不理想的 (电流-角速度建模误差, 执行误差)，环境可能是不理想的 (内部/外部阻力等)，观测可能也是不理想的 (位置/速度/力矩)。在这样的条件下，我们仍然希望机器人能精准的地按照我们的要求运动。这就是机器人控制的核心目标。

如第 III 部分所述，**控制的本质是通过闭环反馈，将开环系统的非理想特性变为闭环系统的理想特性**。事实上，机器人 (机械臂) 的控制是控制理论的一个非常典型的应用场景。在机器人这个控制对象中，**关节力矩是我们可以自由配置的物理控制量**。基于第 II 部分所介绍的运动学和动力学方程，我们就可以提出各种机器人控制方法，达到我们的目的。

在整个机器人系统中，控制可能只是全系统的一环。根据系统设计的不同需求，机器人可能会接收到**关节空间指令**或**操作空间指令**。由此，我们产生了如下 **3 种控制思路**中的一种：独立关节控制、关节空间控制、操作空间控制。

在给出关节指令的情况下，最直接的控制思路方法就是直接对每一关节的电机单独设计控制器。这种控制思路称为**独立关节控制**。不过，在机器人系统中，不同关节的动力学存在复杂的非线性耦合关系。因此，要想进行独立关节控制，需要首先将关节动力学进行解耦，并将不同关节间的非线性耦合项视为扰动进行抑制。

独立关节控制对不同关节单独进行配置，在系统层面可能会产生耦合问题。因此，更好的办法是将机器人在关节空间中统一建模，由一个控制器同时给出所有关节空间的指令。在这个过程中，经过实时计算的非线性反馈，仍可以实现不同控制器的解耦。这种控制思路也称为**关节空间控制**。

机器人用户往往更关心机器人末端的情况，其直接给出的指令来自操作空间。我们可以在操作空间直接设计控制器，再将其转换为关节空间的直接控制量 (关节力矩)。这样的控制思路，称为**操作空间控制**。

在系统设计时，除了会给出不同的指令形式，还会对控制子系统提出一定的控制目标。这些控制目标可以归纳为 3 类：**运动控制**、**阻抗/导纳控制**、**约束力控制**。三种控制思路基本都可以处理这三类控制目标，它们进行组合就产生了各种控制问题。下面，我们按照不同目标，详细介绍这些控制问题。

14.1 运动控制

运动控制就是希望机械臂实现某种给定的运动。在第 III 部分中，我们定义了**调节**和**跟随**两类运动控制的需求，其区别为参考值是否存在一阶和高阶导数。因此，对于机器人 (机械臂)，就存在**调节控制** (即**点对点控制**) 和**跟随控制**两类控制需求。其区别在于：调节控制仅给出单点控制指令，或目标位置为常值；跟随控制则要求在某一空间中跟随指定轨迹，且一阶/高阶导数也需要进行跟随。

无论是调节还是跟随问题，无论指令来自操作空间还是关节空间，我们一般都假设机器人关节角 (及各阶导数) 已知。在实际系统中，它们来自于关节传感器或滤波器的观测。我们将不同指令的不同运动控制问题总结如下

问题	关节空间调节控制
问题简述	已知机器人参数，设计控制器使机器人关节角稳定在参考值
已知	机器人参数 (D-H 参数, 连杆参数) 关节向量及微分 $q, \dot{q}$ 参考关节角 q_d
求	控制量 (驱动力矩) $u(q, \dot{q}, q_d)$

问题	关节空间跟踪控制
问题简述	已知机器人参数，设计控制器使机器人关节角跟随参考值变化
已知	机器人参数 (D-H 参数, 连杆参数) 关节向量及微分 $q, \dot{q}$ 参考关节轨迹 $q_d(t)$ 及各阶导数
求	控制量 (驱动力矩) $u(q, \dot{q}, q_d, \dot{q}_d, \ddot{q}_d)$

问题	操作空间调节控制
问题简述	已知机器人参数，设计控制器使机器人末端稳定在参考值
已知	机器人参数 (D-H 参数, 连杆参数) 关节向量及微分 $q, \dot{q}$ 参考轨迹 x_d
求	控制量 (驱动力矩) $u(q, \dot{q}, x_d)$

问题	操作空间跟踪控制
问题简述	已知机器人参数，设计控制器使机器人末端位姿跟随参考值变化
已知	机器人参数 (D-H 参数, 连杆参数) 关节向量及微分 $q, \dot{q}$ 参考轨迹 $x_d(t)$ 及各阶导数
求	控制量 (驱动力矩) $u(q, \dot{q}, x_d, \dot{x}_d, \ddot{x}_d)$

14.2 阻抗/导纳控制

在机器人运动控制任务中，我们需要机器人的末端或关节精确地沿指定的轨迹进行运动。这样的机器人表现为一种刚性，即对于任何的外力，机器人都会尽力保持预设的轨迹。在面对外界突然施加的力/力矩时，机器人的电机可能会因为扭矩过载导致损坏。也就是说，为了避免机器人因为刚性损坏，我们希望机器人表现出某种柔性。

除此之外，我们还有一些现实需求需要机器人表现柔性。例如人与机器人处于同一空间时，如果机器人表现为完全刚性，则可能对人造成伤害。为了保护人类，需要机器人对外力表现出一定的柔性，例如弹性。

上述” 柔性” 就构成了一种特别的控制目标——希望机器人对外力表现出某种**力和速度的动态关系**。更具体的来说，我们希望机器人的某种**速度** v_r 和机器人的某种**力** F 满足

$$F(t) = m\dot{v}_r(t) + cv_r(t) + k\int v_r(t)dt \tag{IV.14.1}$$

这正是式III.10.10的二阶系统动态。只不过，此处 v 对应式III.10.10中的 $\dot{y}$。式中的 m, c, k 分别对应二阶系统的质量系数、阻尼系数、劲度系数。将上式写为频域关系，有

$$Z_m(s) = \frac{F(s)}{V_r(s)} = c + ms + \frac{k}{s} \tag{IV.14.2}$$

该式的形式与串联 RLC 电路的总电压 $V(s)$ 和电流 $I(s)$ 关系非常类似

$$Z(s) = \frac{V(s)}{I(s)} = R + Ls + \frac{C}{s}$$

由于在电学中，我们将 $Z(s)$ 称为阻抗，因此我们将式IV.14.2中的 $Z_m(s)$ 也称为阻抗。如上所述，**机器人的阻抗就是力和速度间的动态关系**。我们希望机器人具有阻抗特性，这样的控制目标就称为**阻抗/导纳控制**。

式IV.14.2中的 F 和 v_r 并未指定具体是机器人的什么力或速度。事实上，在关节空间和操作空间，都可以定义相应的阻抗控制。对于关节空间，F, v_r 就对应关节扭矩和 $\dot{q}$。对于操作空间，F, v_r 就对应 F_e, v_e。

严格来说，由于机器人阻抗的定义是 $F(s)/V_r(s)$，因此阻抗控制器是以机器人的某种位姿速为输入、关节力矩为输出的控制。针对不同的指令，我们将阻抗控制问题总结如下

问题	**关节空间阻抗控制**
问题简述	已知机器人参数和状态，设计控制器使机器人关节呈现阻抗特性
已知	机器人参数 (D-H 参数, 连杆参数) 关节向量及微分 $q, \dot{q}$ 参考关节轨迹 $q_d(t)$ 及各阶导数
求	控制器 (驱动力矩) $u(q, \dot{q}, q_d, \dot{q}_d, \ddot{q}_d)$

问题	**操作空间阻抗控制**
问题简述	已知机器人参数和状态，设计控制器使机器人末端呈现阻抗特性
已知	机器人参数 (D-H 参数, 连杆参数) 关节向量及微分 $q, \dot{q}$ 参考轨迹 $x_d(t)$ 及各阶导数
求	控制器 (驱动力矩) $u(q, \dot{q}, x_d, \dot{x}_d, \ddot{x}_d)$

特别地，在操作空间阻抗控制中，如果控制器可以获取末端力 F_e 的反馈，则可以实现关节阻抗解耦配置。

在阻抗控制中，我们还有一类特殊的需求。在对机器人进行示教时，我们希望拖动机器人的末端或全机进行运动，此时机器人应当抵消重力等，对外表现出二阶系统的特性，但在位置上没有参考位置需要跟踪。这样的需求我们称为**零力控制**。

问题	**操作空间零力控制**
问题简述	已知机器人参数和状态，设计控制器使机器人末端呈现零力特性
已知	机器人参数 (D-H 参数, 连杆参数) 关节向量及微分 $q, \dot{q}$
求	控制器 (驱动力矩) $u(q, \dot{q})$

除阻抗控制外，我们还可以基于运动控制，设计一种**导纳控制器**。我们定义运动学导纳如下

$$S_m(s) = \frac{V(s)}{F(s)} = \frac{1}{c + ms + \frac{k}{s}} \tag{IV.14.3}$$

导纳控制器是一种基于运动控制器的控制思路。具体来说，为了使系统对外表现出阻抗控制特性，导纳控制器主动观测外力的大小，并设计机器人位置误差轨迹 (即第二参考轨迹)。当机器人跟踪该轨迹时，对外力即表现出阻抗状态。

我们也可以定义关节空间和操作空间的导纳控制问题如下

问题	关节空间导纳控制
问题简述	已知机器人参数和外力矩，设计轨迹使机器人关节呈现阻抗特性
已知	机器人参数 (D-H 参数, 连杆参数) 参考关节轨迹 $q_d(t)$ 及各阶导数 关节向量及微分 $q, \dot{q}$ 关节外力矩 τ_o
求	导纳参考轨迹 $q_a(\tau_o, q_d, \dot{q}_d, \ddot{q}_d)$

问题	操作空间导纳控制
问题简述	已知机器人参数和外力，设计轨迹使机器人末端呈现阻抗特性
已知	机器人参数 (D-H 参数, 连杆参数) 参考轨迹 $x_d(t)$ 及各阶导数 关节向量及微分 $q, \dot{q}$ 外力 F_o
求	导纳参考轨迹 $x_a(F_o, x_d, \dot{x}_d, \ddot{x}_d)$

14.3 约束力控制

除了柔性控制，我们有时也希望机器人能在受约束的环境中执行任务。例如，粉刷、写字、擦玻璃等任务需要机器人使用工具对表面进行处理。这样的任务有如下两个特点：**运动必须沿给定表面进行**，且我们希望**控制接触力处于一定范围**。

以用硬笔写字为例：为了让字写在纸上，一方面，笔尖位置必须处于纸张的平面内；另一方面，纸币接触的力必须适中，过大的接触力会导致笔尖和纸张损坏，而过小的接触力会使墨水无法附着在纸上。我们希望能够控制笔和纸的接触力。

像这样的环境对机器人末端的力和运动有要求的控制问题，称为**约束力控制**问题。其中环境施加的约束称为**环境约束**。环境约束针对末端力 F_e 或末端速度 v_e。在每个末端自由度上必有一个环境约束，要么是力约束，要么是环境约束。

仍以用硬笔写字为例，假设纸张所在平面为环境坐标系的 XY 平面。对于用笔写字的任务，假设末端有 6 自由度，则我们有如下环境约束条件：末端在 z 方向速度需为 0，在 x, y 方向受力等于阻力；环境在三个轴向均不向末端施加力矩。列为表达式形式，即

$$v_z = 0$$
$$F_{ex} = f_{ex}$$
$$F_{ey} = f_{ey}$$
$$M_{ex} = 0$$
$$M_{ey} = 0$$
$$M_{ez} = 0$$

对于末端有 k_f 个自由度的机器人，其末端力 F_e 和末端速度 v_e 共有 $2k_f$ 个维度。k_f 个环境约束会带来 k_f 个方程，因此我们有 k_f 维的可控子空间。我们可以在这一子空间内设计轨迹，并设计控制器，使机器人在满足上述约束的条件下，对子空间内的轨迹进行精确跟踪控制。

事实上，上一节描述的”自由”运动控制也是约束控制的一种特殊情况：环境对末端没有接触，因此 v_e 的所有分量均自由，且在所有方向上环境力约束 $F_e = 0$。

对于约束力控制问题，我们一般假设末端力已知，这要求机器人配备末端力传感器。我们将约束力控制问题总结如下

问题	**约束力控制**
问题简述	已知机器人参数，设计控制器使机器满足约束且跟踪给定力和速度轨迹
已知	机器人参数 (D-H 参数, 连杆参数) 关节向量及微分 $q, \dot{q}$ 环境约束 $E(v_e, F_e) = 0$ 人为约束 $F_d(t), x_d(t)$
求	控制量 (驱动力矩) $u(q, \dot{q}, F_d(t))$

(参考资料：《机器人学：建模、规划与控制》)

15 独立关节控制

机器人由多个刚体相连的电机组成。对于关节空间控制，最直接的方法就是直接对每一关节的电机单独设计控制器，并将不同关节间的耦合项看作扰动进行抑制。

15.1 基本概念

本节中，我们首先简单介绍实际机器人使用的电机，随后对电机进行建模，最后介绍机器人中电机动力学的解耦方法，从而引出独立关节的抗扰调节/跟踪控制问题。

15.1.1 机器人电机简介

在机器人中，最常用的关节驱动器件就是电机。机器人中使用的电机要满足伺服电机的需求，即实现精确的位置控制。因此，在众多电机种类中，我们一般使用**永磁同步电机**作为关节电机。

永磁同步电机属于**直流电机**，分为**有刷电机**和**无刷电机**两种。有刷电机以永磁体为定子，线圈 (电枢) 为转子，依赖换向器改变电流方向。无刷电机以多相线圈为定子产生旋转磁场，以永磁体为转子。

无论使用有刷电机还是无刷电机，都需要**电机驱动器件**为其提供输入。电机驱动器件和主电源相连，以指令信号为输入，以一定电压/电流的驱动信号为输出 (无刷电机的多相驱动信号是交流的)，一般由单片机和功率半导体器件实现。

对于现实中的关节电机产品，一般其内部除集成电机本体、电机驱动器件外，还会集成**减速机构**、**传感器**和**控制器**。

减速机构用于减小电机输出的角速度，增大电机输出的扭矩，也可增加电机的精度。目前最常见的减速机构包括**行星减速器**和**谐波减速器**。

电机上的**传感器**一般包括**角度/角速度传感器**，部分电机还会配备**力矩传感器**。这些传感器的输出可以为机器人控制算法提供相应的反馈值。

电机一般还会集成**控制器**。控制器一般可以接收电机上传感器的信号和外部指令，实现闭环控制。实际控制器一般是预先编程的单片机，内置了电流环等内层控制算法 (详见后续小节)。

15.1.2 永磁同步电机建模

无论是有刷电机还是无刷电机，其电气驱动过程都可以用相同的**永磁同步电机模型**来描述。具体来说，根据电路公式，我们有连续复频域内的公式

$$V_a(s) = (sl_a + r_a)I_a(s) + V_g(s)$$

其中，V_a, I_a 表示绕组 (电枢) 电压和电流，r_a, l_a 表示电枢的电阻和电感，V_g 表示反电动势，它和电机角速度 Ω_m 成正比

$$V_g(s) = k_v\Omega_m(s)$$

此外，根据安培力原理，电机的驱动力矩 C_m 与上式中的 I_a 成正比

$$C_m(s) = k_tI_a(s)$$

上述两式中的 k_v, k_t 分别称为电压常数与转矩常数，它们是直流电机中电学量和力学量之间的桥梁。根据转动定律，电机的力学方程为

$$C_m(s) - C_l(s) = (si_m + f_m)\Omega_m(s) \tag{IV.15.1}$$

其中，C_l 表示负载力矩，i_m 表示转子的惯量，f_m 表示粘滞摩擦系数。综合上式我们有

$$\begin{aligned} I_a(s) &= \frac{V_a(s) - k_v\Omega_m(s)}{sl_a + r_a} \\ \Omega_m(s) &= \frac{k_t I_a(s) - C_l(s)}{si_m + f_m} \end{aligned} \tag{IV.15.2}$$

关于电枢电压 V_a，它和外部控制电压信号 V_c 相关。在**无电流反馈** (无电流环) 的情况下，其关系是简单的比例关系

$$V_a(s) = g_v V_c(s)$$

在**有电流反馈** (有电流环) 的情况下，其关系为

$$V_a(s) = Kg_v(V_c(s) - k_i I_a(s))$$

其中 k_i 表示电流反馈系数，它有电阻相同的量纲；K 表示电流反馈控制器增益，一般取一个较大的值。

总结上述公式，我们有两种永磁同步电机模型：有电流环以及无电流环。**有电流环的永磁同步电机模型**为

$$\begin{aligned} I_a(s) &= \frac{V_a(s) - k_v\Omega_m(s)}{sl_a + r_a} \\ \Omega_m(s) &= \frac{k_t I_a(s) - C_l(s)}{si_m + f_m} \\ V_a(s) &= Kg_v(V_c(s) - k_i I_a(s)) \end{aligned}$$

式中，$i_m, f_m, l_a, r_a, k_v, g_v, k_i, k_t, K$ 均为常数。

另一方面，**无电流环的永磁同步电机模型**为

$$\begin{aligned} I_a(s) &= \frac{g_v V_c(s) - k_v\Omega_m(s)}{sl_a + r_a} \\ \Omega_m(s) &= \frac{k_t I_a(s) - C_l(s)}{si_m + f_m} \end{aligned}$$

式中，$i_m, f_m, l_a, r_a, k_v, g_v, k_t$ 均为常数。

可以看到，由于反电动势的存在，直流电机自然构成了一个调速系统，在电枢电压一定时可以稳定转速 ω_m，并且形成对于外部负载力矩的抗扰作用。

15.1.3 永磁同步电机一阶模型

下面，我们分别写出两种电机的一阶近似输入/输出传递函数。首先，对于**无电流环**的永磁同步电机，化简上述公式，我们有

$$\begin{aligned} \frac{\Omega_m(s)}{V_c(s)} &= \frac{\frac{g_v}{k_v}}{1 + \frac{(si_m + f_m)(sl_a + r_a)}{k_t k_v}} \\ \frac{\Omega_m(s)}{C_l(s)} &= \frac{\frac{sl_a + r_a}{k_t k_v}}{1 + \frac{(si_m + f_m)(sl_a + r_a)}{k_t k_v}} \end{aligned}$$

由于一般有

$$\frac{i_m}{f_m} >> \frac{l_a}{r_a}$$

因此我们可以**近似忽略电枢电感和摩擦**，认为 $l_a \approx 0, f_m \approx 0$，从而有一阶近似传递函数

$$\Omega_m(s) \approx \frac{\frac{g_v}{k_v}}{1+s\frac{i_m r_a}{k_t k_v}} V_c(s) + \frac{\frac{r_a}{k_t k_v}}{1+s\frac{i_m r_a}{k_t k_v}} C_l(s)$$

另一方面，对于**有电流环**的永磁同步电机，我们假设反电动势可以忽略不计，因此有

$$\Omega_m(s) \approx \frac{k_t}{si_m + f_m} \frac{Kg_v}{Kg_v k_i + sl_a + r_a} V_c(s) - \frac{1}{si_m + f_m} C_l(s)$$

进一步地，由于一般有 $Kg_v k_i >> r_a, l_a$，因此一阶近似传递函数为

$$\Omega_m(s) \approx \frac{\frac{k_t}{k_i f_m}}{1+s\frac{i_m}{f_m}} V_c(s) - \frac{\frac{1}{f_m}}{1+s\frac{i_m}{f_m}} C_l(s)$$

以上两类电机的输入/输出和扰动/输出传递函数可以写为统一的形式

$$\begin{aligned} \frac{\Omega_m(s)}{V_c(s)} &\approx \frac{k_m}{1+sT_m} \\ \frac{\Omega_m(s)}{C_l(s)} &\approx \frac{k_d}{1+sT_d} \end{aligned} \tag{IV.15.3}$$

对于**无电流环电机**，有

$$\begin{aligned} k_m &= \frac{g_v}{k_v} \\ T_d = T_m &= \frac{i_m r_a}{k_t k_v} \\ k_d &= \frac{r_a}{k_t k_v} \end{aligned} \tag{IV.15.4}$$

对于**有电流环电机**，有

$$\begin{aligned} k_m &= \frac{k_t}{k_i f_m} \\ T_d = T_m &= \frac{i_m}{f_m} \\ k_d &= -\frac{1}{f_m} \end{aligned} \tag{IV.15.5}$$

15.1.4 电机系统动力学解耦

在机器人中，电机并不是相互独立的对象，而是由刚体相连的一个整体。对于机器人中的每个关节电机，上述电机模型的负载力矩 d 实际上与其他关节电机的关节位置/关节速度相关。也就是说，**机器人中电机力矩的动力学存在耦合关系**。

事实上，上述耦合关系正是机器人动力学，即式II.8.13所描述的规律。该式中的关节角和关节力矩是实际关节的位置和力矩。而电机的直接输出 q_m 和关节角 q 之间往往存在减速机构。我们使用减速比矩阵 K_r 描述这一关系。在理想情况下 (忽略减速机构的误差)，有

$$q_m = K_r q$$
$$\tau_m = K_r^{-1} \tau$$

结合式II.8.13，忽略整个机器人的外力，并假设 F_f 为常数 (即不依赖 $q, \dot{q}$)，我们可以写出机器人系统的电机动力学方程

$$K_r^{-1} B(q) K_r^{-1} \ddot{q}_m + K_r^{-1} C(q, \dot{q}) K_r^{-1} \dot{q}_m + K_r^{-1} F_f K_r^{-1} \dot{q}_m + K_r^{-1} g(q) = \tau_m \tag{IV.15.6}$$

如上所述，该方程是一个耦合方程。下面我们做一些假设，将其分为线性解耦和非线性耦合两个部分。首先，我们假设 $B(q)$ 可分解为依赖 q 和不依赖 q 的两部分

$$B(q) = \bar{B} + \Delta B(q)$$

随后我们记

$$f_m = K_r^{-1} F_f K_r^{-1}$$

则上式可进一步写为

$$K_r^{-1} \bar{B} K_r^{-1} \ddot{q}_m + f_m \dot{q}_m + d = \tau_m \tag{IV.15.7}$$

其中 d 表示非线性耦合项，其表达式为

$$d = K_r^{-1} (\Delta B(q) K_r^{-1} \ddot{q}_m + C(q, \dot{q}) K_r^{-1} \dot{q}_m + g(q)) \tag{IV.15.8}$$

可以看到，耦合项 d 包括动力学方程中的重力项和非重力项。非重力项在机器人运动缓慢的情况下可近似为 0。此外，还注意到式IV.15.7中的 $K_r, \bar{B}, f_m$ 均为对角阵，有

$$K_r = \text{diag}(k_{r,1}, k_{r,2}, ..., k_{r,N})$$
$$\bar{B} = \text{diag}(\bar{b}_1, \bar{b}_2, ..., \bar{b}_N)$$
$$f_m = \text{diag}(f_{m,1}, f_{m,2}, ..., f_{m,N})$$

因此，对于每个关节 n，我们都有

$$k_{r,n}^{-2} \bar{b}_n \ddot{q}_{m,n} + f_{m,n} \dot{q}_{m,n} + d_n = \tau_{m,n}$$

即

$$\tau_{m,n} - d_n = k_{r,n}^{-2} \bar{b}_n \dot{\omega}_{m,n} + f_{m,n} \omega_{m,n} \tag{IV.15.9}$$

事实上，这正是式IV.15.1(电机动力学方程) 的时域形式。两式的对应关系为

$$i_{m,n} = k_{r,n}^{-2} \bar{b}_n$$
$$c_{m,n} = \tau_{m,n}$$
$$c_{l,n} = d_n$$

这样，由式IV.15.8，机器人动力学的非线性耦合项，就合并为单关节电机的负载力矩 $c_{l,n}$。也就是说，机器人中的每个关节，都满足前面小节介绍的电机力学方程 (即式IV.15.1)。因此，机器人中的每个关节，都可以直接使用上一小节中的一阶动力学 (式IV.15.3) 进行建模。

基于这一电机模型，我们就将整个机器人的关节空间调节/跟踪控制问题，转化为了**各关节电机独立的抗扰调节/跟踪控制问题**。

问题	关节抗扰调节控制
问题简述	已知电机参数，设计控制器使关节角稳定在参考值，且抑制干扰
已知	电机参数 $k_r, i_m, k_t, f_m, r_a, k_v, g_v$ 关节角 θ 参考关节角 θ_d
求	控制量 (控制电压) $u(\theta, \theta_d)$

问题	关节抗扰跟踪控制
问题简述	已知电机参数，设计控制器使关节角跟随参考值变化，且抑制干扰
已知	电机参数 $k_r, i_m, k_t, f_m, r_a, k_v, g_v$ 关节角及微分 $\theta, \dot{\theta}$ 参考关节角及微分 $\theta_d, \dot{\theta}_d, \ddot{\theta}_d$
求	控制量 (控制电压) $u(\theta, \dot{\theta}, \theta_d, \dot{\theta}_d, \ddot{\theta}_d)$

下面，我们介绍一些独立关节控制方法。

15.2 独立关节运动控制

15.2.1 调节控制

首先我们来看简单的调节问题。根据上节的介绍，机器人中的每个关节电机都可以用下列方程进行描述

$$s\Theta_m(s) = \frac{k_m}{1+sT_m}V_c(s) + \frac{k_d}{1+sT_d}D(s) \tag{IV.15.10}$$

其中 $D(s) = C_l(s)$ 为干扰力矩 (负载力矩)，对应机器人动力学中的关节耦合项。现希望设计控制器，使关节角 θ 跟踪参考值 θ_d。注意关节角和电机的角度间存在比例关系

$$\theta_m = k_r\theta$$
$$\theta_{md} = k_r\theta_d$$

由于上述关节电机简化方程是角速度的方程，为消除静差，我们可以使用基于误差反馈的 PI 控制策略

$$\begin{aligned} E(s) &= \Theta_{md}(s) - \Theta_m(s) \\ V_c(s) &= (k_P + \frac{k_I}{s})E(s) \end{aligned} \tag{IV.15.11}$$

因此，我们的反馈闭环系统为

$$sk_r\Theta(s) = \frac{k_m}{1+sT_m}V_c(s) + \frac{k_d}{1+sT_d}D(s)$$
$$E(s) = k_r(\Theta_d(s) - \Theta(s))$$
$$V_c(s) = \left(k_P + \frac{k_I}{s}\right)E(s)$$

闭环系统中，从 $\Theta_d(s)$ 到 $\Theta(s)$ 构成单位负反馈，其开环前向传递函数为

$$H_{o,\theta}(s) = \frac{k_m k_I(1+s\frac{k_P}{k_I})}{s^2(1+sT_m)}$$

闭环传递函数为

$$H_{c,\theta}(s) = \frac{1+s\frac{k_P}{k_I}}{1+s\frac{k_P}{k_I}+s^2\frac{1}{k_m k_I}+s^3\frac{T_m}{k_m k_I}}$$

而从 $D(s)$ 到 $\Theta(s)$ 的闭环传递函数为

$$H_{c,d}(s) = \frac{\frac{k_d}{k_m k_I}(1+sT_m)}{(1+sT_d)(1+s\frac{k_I+k_m k_P}{k_m k_I}+s^2\frac{T_m k_r}{k_m k_I})}$$

根据频域稳定性要求，需要将两个闭环传递函数的极点均配置在左半平面；为抑制扰动，还应使 k_I 尽量大。

我们整理**独立关节 PI 控制**如下

算法	**独立关节 PI 控制**
问题类型	关节抗扰调节控制
已知	电机参数 $k_r, i_m, k_t, f_m, r_a, k_v, g_v$ 关节角 θ 参考关节角 θ_d
求	控制量 (控制电压) $u(\theta, \theta_d)$
算法性质	model-based, 解析解

Algorithm 38: 独立关节 PI 控制

Input: 关节角 θ, 参考关节角 θ_d

Parameter: 参数 k_P, k_I

Output: 电压指令 u

$e \leftarrow k_r(\theta_d - \theta)$

$u \leftarrow k_P e + k_I \int e dt$

15.2.2 跟踪控制

下面我们来看跟踪问题。相比调节问题，跟踪问题在关节角度之外，还需要关节角速度 ω 的反馈。假设参考轨迹的一阶导数也已知且连续，有

$$\omega_m = k_r\omega$$
$$\omega_{md} = k_r\omega_d$$

这样我们就有了跟踪误差微分的观测，可以使用 PID 策略

$$\begin{aligned} E(s) &= \Theta_{md}(s) - \Theta_m(s) \\ sE(s) &= \Omega_{md}(s) - \Omega_m(s) \\ V_c(s) &= \left(k_P + \frac{k_I}{s}\right) E(s) + K_D s E(s) \end{aligned} \tag{IV.15.12}$$

我们整理**独立关节 PID 控制**如下

算法	**独立关节 PID 控制**
问题类型	关节抗扰跟踪控制
已知	电机参数 $k_r, i_m, k_t, f_m, r_a, k_v, g_v$ 关节角及角速度 θ, ω 参考关节角及角速度 θ_d, ω_d
求	控制量 (控制电压) $u(\theta, \omega, \theta_d, \omega_d)$
算法性质	model-based, 解析解

Algorithm 39: 独立关节 PID 控制

Input: 关节角及角速度 θ, ω

Input: 参考关节角及角速度 θ_d, ω_d

Parameter: 参数 k_P, k_I

Output: 电压指令 u

$u \leftarrow k_r(k_P(\theta_d - \theta) + k_I \int(\theta_d - \theta)dt + k_D(\omega_d - \omega))$

(参考资料：《机器人学：建模、规划与控制》)

16 关节空间控制

关节空间的控制既包括基础的调节和跟踪控制 (统称为运动控制)，也包括阻抗控制方法。

16.1 关节空间运动控制

16.1.1 关节空间重力补偿 PD 控制

首先，我们考虑在给定参考关节角 q_d 的情况下，使机器人稳定在某一姿态，即关节空间的调节问题。

由于我们考虑的是调节问题，稳定时机器人处于静止，动力学方程中不会有惯量和科里奥利力项。因此，我们考虑最简单的控制形式，即在反馈重力后使用 PID 控制。

如前文所述，PID 控制需要使用系统的误差状态。由于我们是在关节空间进行控制，机器人关节向量 $q(t)$ 即为被控状态。误差状态定义为

$$\tilde{q}(t) = q_d - q(t) \tag{IV.16.1}$$

我们需要使用比例和微分项，也就是需要误差状态及其一阶导数 $\dot{\tilde{q}}(t)$。由于我们考虑的是调节问题，q_d 对时间导数为 0，因此我们仅需要知道 $\dot{\tilde{q}}(t) = -\dot{q}(t)$。在实际机器人系统中，$q(t)$ 和 $\dot{q}(t)$ 由各类传感器和观测器 (一般是某种滤波器) 求得。

写出重力补偿下的 PD 控制为

$$u(t) = K_P\tilde{q}(t) - K_D\dot{q}(t) + g(q) \tag{IV.16.2}$$

注意，此处的 $u(t)$ 物理意义为关节驱动力矩向量，和 $q(t)$ 一样均为 N 维向量。在这种情况下，K_P 和 K_D 均为方阵。事实上，这两个矩阵应当选取正定矩阵。

结合含有滑动摩擦阻力项的机器人动力学方程 (即开环系统动态)

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) = u \tag{IV.16.3}$$

闭环系统的结构如下所列

$$\begin{aligned} B(q(t))\ddot{q}(t) + C(q(t),\dot{q}(t))\dot{q}(t) + F_f\dot{q}(t) + g(q(t)) &= u(t) \\ \tilde{q}(t) &= q_d - q(t) \\ u(t) &= K_P\tilde{q}(t) - K_D\dot{q}(t) + g(q(t)) \end{aligned}$$

总结关节空间重力补偿 PD 控制算法如下

Algorithm 40: JS 重力补偿 PD 控制

Input: 连杆参数 $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$

Input: D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$

Input: 关节向量 q 及其微分 $\dot{q}$

Input: 参考关节角 q_d

Parameter: 矩阵参数 K_P, K_D

Output: 力矩指令 u

$g \leftarrow \text{robot\_dyn}(q, m_n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$

$u \leftarrow K_P(q_d - q) - K_D\dot{q}(t) + g$

算法	**JS 重力补偿 PD 控制**
问题类型	关节空间调节控制
已知	D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$ 连杆参数 $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ 关节向量及其微分 $q, \dot{q}$ 参考关节角 q_d
求	控制量 (驱动力矩) $u(q, \dot{q}, q_d)$
算法性质	model-based, 解析解

下面，我们进行系统稳定性证明。

证明. 为证明上述闭环系统稳定性，选择以下正定二次型作为 Lyapunov Candidate 函数

$$V(\dot{q}, \tilde{q}) = \frac{1}{2}\dot{q}^T B(q)\dot{q} + \frac{1}{2}\tilde{q}^T K_P \tilde{q} \quad > 0, \quad \forall \dot{q}, \tilde{q} \neq 0 \tag{IV.16.4}$$

求一阶导，有

$$\dot{V} = \frac{1}{2}\dot{q}^T \dot{B}(q)\dot{q} + \dot{q}^T B(q)\ddot{q} - \dot{q}^T K_p \tilde{q} \tag{IV.16.5}$$

代入动力学方程 (式IV.16.3)，有

$$\begin{aligned}\dot{V} &= \frac{1}{2}\dot{q}^T \dot{B}(q)\dot{q} + \dot{q}^T(u - C(q,\dot{q})\dot{q} - F_f\dot{q} - g(q)) - \dot{q}^T K_p\tilde{q} \\ &= \frac{1}{2}\dot{q}^T(\dot{B}(q) - 2C(q,\dot{q}))\dot{q} - \dot{q}^T F_f\dot{q} + \dot{q}^T(u - g(q) - K_p\tilde{q})\end{aligned}$$

使用第 II 部分第 4 章介绍的 N 矩阵性质，有

$$\dot{V} = -\dot{q}^T F_f\dot{q} + \dot{q}^T(u - g(q) - K_p\tilde{q})$$

代入式IV.16.2的控制律，有

$$\begin{aligned}\dot{V} &= -\dot{q}^T F_f\dot{q} + \dot{q}^T(K_P\tilde{q} - K_D\dot{q}(t) + g(q) - g(q) - K_p\tilde{q}) \\ &= -\dot{q}^T(F_f + K_D)\dot{q}\end{aligned}$$

摩擦系数矩阵 F_f 为半正定阵。取 K_D 为对称正定阵，有

$$\dot{V} < 0, \quad \forall \dot{q} \neq 0$$

因此，若 K_p, K_D 均为对称正定，根据 Lyapunov 稳定性判据，闭环系统是渐近稳定的。

□

16.1.2 关节空间逆动力学控制

上述的关节空间重力补偿 PD 控制可用于静止或机器人缓慢运动的场合。但如果机器人的目标轨迹是较为快速的运动，上述控制算法的收敛性就会变差。这是因为，当参考轨迹是较快速运动时，$B(q)\ddot{q}$ 和 $C(q,\dot{q})\dot{q}$ 就不再可以忽略。

为了应对参考轨迹为 $q_d(t)$ 的关节空间跟踪控制，我们可以在控制回路中重新加入这两项。具体来说，此时的误差状态为

$$\tilde{q}(t) = q_d(t) - q(t) \tag{IV.16.6}$$

注意，作为参考轨迹，其各阶导数 $\dot{q}_d(t), \ddot{q}_d(t)...$ 均为已知。

在机器人系统中，关节驱动力矩 τ 是可以自由控制的量，我们称为**直接控制量**。按照控制理论的一般记法，我们将其记为 u。

按照精确线性化思路，我们使用动力学公式，引入非线性反馈项，从而将自由控制 τ 转变为自由控制 $\ddot{q}$。

$$u = B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) \tag{IV.16.7}$$

此时，系统中的 $\ddot{q}$ 是可以自由配置的控制量，称为**间接控制量**，记为 y。

$$y = \ddot{q} \tag{IV.16.8}$$

通过这样的设计，我们事实上实现了控制器的解耦。控制量 y 的第 n 个分量，仅影响第 n 个关节，而与其他关节无关。

具体控制器设计，我们仍然使用 PD 控制器

$$y = K_P\tilde{q} + K_D\dot{\tilde{q}} + \ddot{q}_d \tag{IV.16.9}$$

此时，闭环系统的结构如下所示

$$\begin{aligned} B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) &= u \\ u = B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) \\ \tilde{q}(t) = q_d(t) - q(t) \\ y = K_P\tilde{q} + K_D\dot{\tilde{q}} + \ddot{q}_d \end{aligned}$$

闭环系统的误差状态满足如下二阶微分方程

$$\ddot{\tilde{q}} + K_D\dot{\tilde{q}} + K_P\tilde{q} = 0 \tag{IV.16.10}$$

可以看到，闭环系统的误差动态类似于式III.10.10描述的典型二阶系统。只不过这个系统没有输入量。可以理解为一个没有外力、在一定初始条件下会自动回到 0 点的典型二阶系统。K_D, K_P 类似于力学二阶系统中的阻尼系数与劲度系数。

总结关节空间逆动力学控制算法如下

Algorithm 41: JS 逆动力学 PD 控制

Input: 连杆参数 $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$

Input: D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$

Input: 关节向量 q 及其微分 $\dot{q}$

Input: 参考关节轨迹及其微分 $q_d, \dot{q}_d, \ddot{q}_d$

Parameter: 矩阵参数 K_P, K_D, 摩擦矩阵 F_f

Output: 力矩指令 u

$B, C, g \leftarrow \text{robot\_dyn}(q, \dot{q}, m_n, \mathbf{I}_n^n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$

$y \leftarrow K_P(q_d - q) + K_D(\dot{q}_d - \dot{q}) + \ddot{q}_d$

$u \leftarrow By + C\dot{q} + F_f\dot{q} + g$

算法	**JS 逆动力学 PD 控制**
问题类型	关节空间跟踪控制
已知	D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$ 连杆参数 $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ 关节向量及其微分 $q, \dot{q}$ 参考关节轨迹及其微分 $q_d, \dot{q}_d, \ddot{q}_d$
求	控制量 (驱动力矩) $u(q, \dot{q}, q_d, \dot{q}_d, \ddot{q}_d)$
算法性质	model-based, 解析解

下面，我们进行系统稳定性证明。

证明. 假设 K_P 正定，取 Lyapunov Candidate 函数为

$$V(\tilde{q}, \dot{\tilde{q}}) = \frac{1}{2}\tilde{q}^T K_P \tilde{q} + \frac{1}{2}\dot{\tilde{q}}^T \tilde{q} \quad > 0, \quad \forall \dot{q}, \tilde{q} \neq 0$$

对时间进行求导，有

$$\begin{aligned} \dot{V} &= \tilde{q}^T K_P \dot{\tilde{q}} + \dot{\tilde{q}}^T \ddot{\tilde{q}} \\ &= \tilde{q}^T K_P \dot{\tilde{q}} + \dot{\tilde{q}}^T (-K_D \dot{\tilde{q}} - K_P \tilde{q}) \\ &= -\dot{\tilde{q}}^T K_D \dot{\tilde{q}} \end{aligned}$$

假设 K_D 对称正定，有

$$\dot{V} < 0, \quad \forall \dot{q} \neq 0$$

因此，若 K_p, K_D 均为对称正定，根据 Lyapunov 稳定性判据，闭环系统是渐近稳定的。

□

实际使用时，可取 K_P, K_D 为对角阵，从而将每个关节的动力学均解耦为二阶系统。式中 w_n 代表每个系统的自然频率，ξ_n 代表每个系统的阻尼比。

$$\begin{aligned} K_P &= \text{diag}\{w_1^2, ..., w_N^2\} \\ K_D &= \text{diag}\{2\xi_1 w_1, ..., 2\xi_N w_N\} \end{aligned} \tag{IV.16.11}$$

此外，还要注意：无论是关节空间重力补偿 PD 控制，还是关节空间逆动力学控制，都需要在控制律中实时计算部分或全部的动力学方程项。

如第 III 部分的第 4 章所述，动力学方程的计算是比较复杂的。以较大的频率实时计算这些项曾经是一个工程难点，不过随着计算机技术的发展，现在这样的算力需求已经不再是问题。

(参考资料：《机器人学：建模、规划与控制》)

16.2 关节空间阻抗控制

[本部分内容将在后续版本中更新，敬请期待]

17 操作空间控制

上一章中，我们介绍了关节空间的控制。与之不同，在操作空间控制中，参考信号是来自操作空间。因此，我们需要灵活地使用雅可比矩阵和逆运动学进行信号变换。

17.1 操作空间运动控制

17.1.1 操作空间重力补偿 PD 控制

和关节空间类似，我们首先仍考虑调节问题。设末端参考轨迹点为 x_d，则操作空间误差状态定义为

$$\tilde{x}(t) = x_d - x_e(t) \tag{IV.17.1}$$

和操作空间控制的重要区别是：我们假设机器人状态 $x_e(t)$ 和 $\dot{x}_e(t)$ 不可直接测得，而是需要从关节空间状态 $q(t)$ 和 $\dot{q}(t)$ 中求取[16]，即

$$\begin{aligned} x_e &= k(q) \\ \dot{x}_e &= J_a(q)\dot{q} \end{aligned} \tag{IV.17.2}$$

注意到求取速度时使用了分析雅可比，这是因为我们提供的轨迹往往是以欧拉角或四元数的形式呈现的。为实现操作空间误差状态的减法，也需要使用分析雅可比而非几何雅可比进行观测。

使用类似操作空间的思路，我们以 PD 形式写出控制律。注意此处操作空间 PD 直接得到的是末端的执行力，需要使用转置雅可比将其转换为关节力矩。

$$\begin{aligned} u(t) &= J_a^T(K_P\tilde{x}(t) - K_D\dot{x}_e(t)) + g(q) \\ \tilde{x}(t) &= x_d - k(q(t)) \\ \dot{x}_e &= J_a(q)\dot{q} \end{aligned}$$

代入两个操作空间状态的观测过程，有

$$u(t) = J_a^T(K_P\tilde{x} - K_D J_a(q)\dot{q}) + g(q) \tag{IV.17.3}$$

整理整个闭环系统结构，有

$$\begin{aligned} B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) &= u(t) \\ \tilde{x}(t) &= x_d - k(q(t)) \\ u(t) &= J_a^T(K_P\tilde{x} - K_D J_a(q)\dot{q}) + g(q) \end{aligned}$$

总结操作空间重力补偿 PD 控制算法如下

[16] 实际使用时，也可以使用其他的传感器/观测器观测系统状态。但要注意，无论系统状态实际如何取得，它们都满足这样的关系

Algorithm 42: OS 重力补偿 PD 控制

Input: 连杆参数 $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$

Input: D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$

Input: 关节向量 q 及其微分 $\dot{q}$

Input: 参考轨迹 x_d

Parameter: 矩阵参数 K_P, K_D

Output: 力矩指令 u

$x_e \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q)$

$J_a \leftarrow \text{robot\_jacobian\_a}(q, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$

$g \leftarrow \text{robot\_dyn}(q, m_n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$

$u \leftarrow J_a^T(K_P(x_d - x_e) - K_D J_a \dot{q}) + g$

算法	**OS 重力补偿 PD 控制**
问题类型	操作空间调节控制
已知	D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$ 连杆参数 $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ 关节向量及其微分 $q, \dot{q}$ 参考轨迹 x_d
求	控制量 (驱动力矩) $u(q, \dot{q}, x_d)$
算法性质	model-based, 解析解

下面，我们进行系统稳定性证明。

证明. 取操作空间误差 $\tilde{x}$ 和关节空间速度 $\dot{q}$ 为闭环系统状态。假设 K_P 对称正定，构造 Lyapunov Candidate 函数为

$$V(\tilde{x}, \dot{q}) = \frac{1}{2}\dot{q}^T B(q)\dot{q} + \frac{1}{2}\tilde{x}^T K_P \tilde{x} \quad > 0, \quad \forall \dot{q}, \tilde{q} \neq 0$$

对时间进行求导，有

$$\begin{aligned}
\dot{V} &= \frac{1}{2}\dot{q}^T \dot{B}(q)\dot{q} + \dot{q}^T B(q)\ddot{q} - \dot{\tilde{x}}^T K_p \tilde{x} \\
&= \frac{1}{2}\dot{q}^T \dot{B}(q)\dot{q} + \dot{q}^T (u - C(q, \dot{q})\dot{q} - F_f \dot{q} - g(q)) - \dot{\tilde{x}}^T K_p \tilde{x} \\
&= \frac{1}{2}\dot{q}^T (\dot{B}(q) - 2C(q, \dot{q}))\dot{q} - \dot{q}^T F_f \dot{q} + \dot{q}^T (u - g(q)) - \dot{\tilde{x}}^T K_p \tilde{x} \\
&= -\dot{q}^T F_f \dot{q} + \dot{q}^T (u - g(q)) - \dot{\tilde{x}}^T K_p \tilde{x}
\end{aligned}$$

注意到

$$\dot{\tilde{x}} = -\dot{x}_e = -J_a(q)\dot{q} \tag{IV.17.4}$$

有

$$\begin{aligned}\dot{V} &= -\dot{q}^T F_f \dot{q} + \dot{q}^T (u - g(q)) + \dot{q}^T J_a^T(q) K_p \tilde{x} \\ &= -\dot{q}^T F_f \dot{q} + \dot{q}^T (u - g(q) + J_a^T(q) K_p \tilde{x})\end{aligned}$$

代入控制律 (式IV.17.3)，有

$$\begin{aligned}\dot{V} &= -\dot{q}^T F_f \dot{q} + \dot{q}^T (J_a^T (K_P \tilde{x} - K_D J_a(q)\dot{q}) + g(q) - g(q) + J_a^T(q) K_p \tilde{x}) \\ &= -\dot{q}^T F_f \dot{q} - \dot{q}^T J_a^T K_D J_a(q) \dot{q} \\ &= -\dot{q}^T (F + J_a^T(q) K_D J_a(q)) \dot{q}\end{aligned}$$

摩擦系数矩阵 F 为半正定阵。取 K_D 为对称正定阵，假设分析雅可比矩阵满秩，有

$$\dot{V} < 0, \quad \forall \dot{q} \neq 0$$

因此，若 K_p, K_D 均为对称正定，根据 Lyapunov 稳定性判据，闭环系统是渐近稳定的。

□

17.1.2 操作空间逆动力学控制

类似关节空间逆动力学控制的情况，在操作空间中，针对时变参考轨迹 $x_d(t)$ 及其导数的跟踪问题，我们也需要使用更多的补偿控制项。

假设 $x_d(t)$ 的各阶导数已知，定义操作空间误差状态如下

$$\tilde{x}(t) = x_d(t) - x_{ea}(t) \tag{IV.17.5}$$

仿照关节空间的精确线性化思路，我们仍进行精确线性化，使用 $\ddot{q}$ 作为间接控制量 y。

$$u = B(q) y + C(q, \dot{q})\dot{q} + F_f \dot{q} + g(q) \tag{IV.17.6}$$

我们有

$$y = \ddot{q}$$

接下来，我们来设计间接控制量。仿照关节空间的情况，我们希望将操作空间中闭环系统的误差动态设计为二阶线性形式

$$K_P \tilde{x} + K_D \dot{\tilde{x}} + \ddot{\tilde{x}} = 0 \tag{IV.17.7}$$

和关节空间不同的是：此处，操作空间的状态及导数 $x_{ea}(t), \dot{x}_{ea}(t), \ddot{x}_{ea}(t)$ 无法直接获得。回顾系统动力学以及一阶和二阶雅可比方程 (式II.7.1和II.7.5)，我们有

$$\begin{aligned}\tilde{x}(t) &= x_d(t) - k(q(t)) \\ \dot{\tilde{x}}(t) &= \dot{x}_d(t) - J_a(q)\dot{q}(t) \\ \ddot{\tilde{x}}(t) &= \ddot{x}_d(t) - J_a(q)\ddot{q}(t) - \dot{J}_a(q)\dot{q}(t)\end{aligned} \tag{IV.17.8}$$

式IV.17.7左侧可以写为

$$K_P \tilde{x} + K_D \dot{\tilde{x}} + \ddot{\tilde{x}} = K_P \tilde{x} + K_D \dot{\tilde{x}} + \ddot{x}_d(t) - J_a(q)\ddot{q}(t) - \dot{J}_a(q)\dot{q}(t)$$

考虑 $y = \ddot{q}(t)$ 可自由配置，我们只需使该式等号右侧为 0，求出 $y(t)$，即可达到我们的目的

$$K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - J_a(q)y(t) - \dot{J}_a(q)\dot{q}(t) = 0$$

由此，设计出的间接控制量为

$$y(t) = J_a^{-1}(q)(K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - \dot{J}_a(q)\dot{q}(t)) \tag{IV.17.9}$$

这里用到了分析雅可比的逆矩阵。该矩阵可逆当且仅当**机械臂非冗余**，这也是操作空间逆动力学控制的必要条件。

整理整个闭环系统结构，我们有

$$\begin{aligned}
B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) &= u(t) \\
u &= B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) \\
y &= J_a^{-1}(q)(K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d - \dot{J}_a(q,\dot{q})\dot{q}) \\
\tilde{x}(t) &= x_d(t) - k(q(t)) \\
\dot{\tilde{x}}(t) &= \dot{x}_d(t) - J_a(q)\dot{q}(t)
\end{aligned}$$

整理操作空间逆动力学控制如下。注意：该算法中，我们使用了自动微分求解分析雅可比矩阵对时间的导数；以同时必须确保分析雅可比可逆

算法	**OS 逆动力学 PD 控制**
问题类型	操作空间跟踪控制
已知	D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$ 连杆参数 $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ 摩擦矩阵 F_f 关节向量及其微分 $q, \dot{q}$ 参考轨迹及各阶导数 $x_d, \dot{x}_d, \ddot{x}_d$
求	控制量 (驱动力矩) $u(q, \dot{q}, x_d, \dot{x}_d, \ddot{x}_d)$
算法性质	model-based, 解析解

Algorithm 43: OS 逆动力学 PD 控制

Input: 连杆参数 $m_{1:N}, \mathbf{I}_{:}^{:}, p_{l_:}^{:}$, D-H 参数 $d_{1:N}, a_{1:N}, \alpha_{1:N}$

Input: 摩擦矩阵 F_f

Input: 关节向量 q 及其微分 $\dot{q}$

Input: 参考轨迹及各阶导数 $x_d, \dot{x}_d, \ddot{x}_d$

Parameter: 矩阵参数 K_P, K_D, 摩擦矩阵 F_f

Output: 力矩指令 u

$x_e \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q)$

$J_a \leftarrow \text{robot\_jacobian\_a}(q, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$

$\dot{J}_a \leftarrow \text{auto\_diff}(J_a, t)$

$B, C, g \leftarrow \text{robot\_dyn}(q, \dot{q}, m_n, \mathbf{I}_n^n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$

$y \leftarrow J_a^{-1}(K_P(x_d - x_e) + K_D(\dot{x}_d - J_a\dot{q}) + \ddot{x}_d - \dot{J}_a\dot{q})$

$u \leftarrow By + C\dot{q} + F_f\dot{q} + g$

对于稳定性证明，由于闭环系统已被反馈线性化为线性二阶形式 (式IV.17.7)，当 K_p, K_D 均为对称正定时，该二阶系统的镇定证明与关节空间逆动力学控制完全相同，不再赘述。

(参考资料：清华《智能机器人》课件, 《机器人学：建模、规划与控制》)

17.2 操作空间阻抗控制

下面，我们来考虑操作空间的阻抗/导纳控制问题。

17.2.1 无反馈阻抗控制

首先，假设末端力不可测，考虑含有末端外力项的系统动力学

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) = u(t) - J_a^T F_e \tag{IV.17.10}$$

仍采用操作空间逆动力学控制的方法，此时闭环系统的误差应满足这样的形式

$$K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{\tilde{x}} = B_f F_e \tag{IV.17.11}$$

其中 B_f 为待定的正定矩阵。由于我们的精确线性化方法不包含外力项

$$u = B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q)$$

外力项会和间接控制量一起反映在操作空间加速度中

$$\ddot{q} = y - B^{-1}(q)J_a^T(q)F_e \tag{IV.17.12}$$

将二阶雅可比和间接控制量代入误差动态，我们有

$$K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - J_a(q)(y - B^{-1}(q)J_a^T(q)F_e) - \dot{J}_a(q)\dot{q}(t) = B_f F_e$$

即

$$y = J_a(q)^{-1}(K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - \dot{J}_a(q)\dot{q}(t) + (J_a(q)B^{-1}(q)J_a^T(q) - B_f)F_e)$$

由于控制量中不带有末端力，因此最后一项为 0。即：系统的误差和外力之间的关系，即为系统在操作空间的的惯量矩阵之逆

$$B_f(q) = (J_a^{-1}(q)B(q)J_a^{-T}(q))^{-1} \tag{IV.17.13}$$

因此我们可以得出结论：在操作空间逆动力学控制律下，系统自然呈现式IV.14.2描述的阻抗特性。然而，该特性的 B_f 是非解耦且系统固有的。

17.2.2 有反馈阻抗控制

在上述系统中，我们已经可以通过 K_P 和 K_D 控制系统的即阻尼特性和劲度特性。然而，由于 $B_f(q)$ 存在且不可配置，上述系统不是解耦的。如果想单独调节某个关节的特性，就需要进行复杂的计算。

为了进行**解耦控制**，我们需要对末端力 F_e 进行测量。假设可得到 F_e 的精确测量值，我们就可以在控制量中加入反馈线性化项

$$u = B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) + J_a^T(q)F_e \tag{IV.17.14}$$

此时，我们再引入新的等效力 $F_a = B_iF_e$，作为系统闭环误差动态的输入。此处的矩阵 B_i 即对应被动阻抗控制中的 B_f，但区别在于 B_i 可以按需选取配置

$$K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{\tilde{x}} = B_iF_e = F_a \tag{IV.17.15}$$

此时，系统间接控制量为

$$y = J_a(q)^{-1}(K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - \dot{J}_a(q)\dot{q}(t) - B_iF_e) \tag{IV.17.16}$$

整理整个闭环系统结构，我们有

$$\begin{aligned}
B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) &= u(t) - J_a^TF_e \\
u &= B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) + J_a^T(q)F_e \\
y &= J_a(q)^{-1}(K_P\tilde{x} + K_D\dot{\tilde{x}} + \ddot{x}_d(t) - \dot{J}_a(q)\dot{q}(t) - B_iF_e) \\
\tilde{x}(t) &= x_d(t) - k(q(t)) \\
\dot{\tilde{x}}(t) &= \dot{x}_d(t) - J_a(q)\dot{q}(t)
\end{aligned}$$

综合前述推导，我们整理机器人阻抗控制算法如下

Algorithm 44: OS 阻抗控制

Input: 连杆参数 $m_{1:N}, \mathbf{I}_{:}^{:}, p_{l_{:}}^{:}$, D-H 参数 $d_{1:N}, a_{1:N}, \alpha_{1:N}$
Input: 摩擦矩阵 F_f
Input: 关节向量 q 及其微分 $\dot{q}$, 末端力 F_e
Input: 参考轨迹及各阶导数 $x_d, \dot{x}_d, \ddot{x}_d$
Parameter: 矩阵参数 K_P, K_D, 系数矩阵 B_i
Output: 力矩指令 u

$x_e \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q)$
$J_a \leftarrow \text{robot\_jacobian\_a}(q, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$\dot{J}_a \leftarrow \text{auto\_diff}(J_a, t)$
$B, C, g \leftarrow \text{robot\_dyn}(q, \dot{q}, m_n, \mathbf{I}_n^n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$
$y \leftarrow J_a^{-1}(K_P(x_d - x_e) + K_D(\dot{x}_d - J_a\dot{q}) + \ddot{x}_d - \dot{J}_a\dot{q} - F_a)$
$u \leftarrow By + C\dot{q} + F_f\dot{q} + g + J_a^TF_e$

算法	OS 阻抗控制
问题类型	操作空间阻抗控制问题
已知	D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$ 连杆参数 $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ 摩擦矩阵 F_f 关节向量及其微分 $q, \dot{q}$, 末端力 F_e 参考轨迹及各阶导数 $x_d, \dot{x}_d, \ddot{x}_d$
求	控制量 (驱动力矩) $u(q, \dot{q}, x_d, \dot{x}_d, \ddot{x}_d, F_e)$
算法性质	model-based, 解析解

注意：上述算法的使用条件是：需要使用自动微分求解分析雅可比矩阵对时间的导数，且分析雅可比可逆。

实际使用中，阻抗控制能实现的自由度和能测量的末端力维数有关。如果末端力为 6 维，就可以实现 6 自由度阻抗控制。如果末端力为 3 维，就只能实现 3 自由度阻抗控制。

17.2.3 零力控制

操作空间的零力控制可以很容易地在阻抗控制的基础上修改得到。具体来说，零力控制没有给定的轨迹，机器人需要在外力的作用下表现出二阶系统特性，而外力消失后应当静止。因此，我们在上述阻抗控制的间接控制量中，去除位置跟踪项，并使 $\dot{x}_d, \ddot{x}_d$ 为 0

$$y = J_a(q)^{-1}(K_D\dot{x} - \dot{J}_a(q)\dot{q}(t) - B_i F_e) \tag{IV.17.17}$$

此时的闭环系统结构为

$$\begin{aligned} B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) &= u(t) - J_a^T F_e \\ u &= B(q)y + C(q,\dot{q})\dot{q} + F_f\dot{q} + g(q) + J_a^T(q)F_e \\ y &= J_a(q)^{-1}(K_D\dot{x} - \dot{J}_a(q)\dot{q}(t) - B_i F_e) \\ \dot{x}(t) &= -J_a(q)\dot{q}(t) \end{aligned}$$

从而，我们有机器人**操作空间零力控制**算法

Algorithm 45: OS 零力控制

Input: 连杆参数 $m_{1:N}, \mathbf{I}_{:}^{:}, p_{l_:}^{:}$, D-H 参数 $d_{1:N}, a_{1:N}, \alpha_{1:N}$

Input: 摩擦矩阵 F_f

Input: 关节向量 q 及其微分 $\dot{q}$, 末端力 F_e

Parameter: 矩阵参数 K_D, 系数矩阵 B_i

Output: 力矩指令 u

$x_e \leftarrow \text{robot\_fk}(d_{1:N}, a_{1:N}, \alpha_{1:N}, q)$

$J_a \leftarrow \text{robot\_jacobian\_a}(q, x_e, d_{1:N}, a_{1:N}, \alpha_{1:N})$

$\dot{J}_a \leftarrow \text{auto\_diff}(J_a, t)$

$B, C, g \leftarrow \text{robot\_dyn}(q, \dot{q}, m_n, \mathbf{I}_n^n, p_{l_n}^n, d_{1:N}, a_{1:N}, \alpha_{1:N})$

$y \leftarrow J_a(q)^{-1}(K_D\dot{x} - \dot{J}_a(q)\dot{q}(t) - B_i F_e)$

$u \leftarrow By + C\dot{q} + F_f\dot{q} + g + J_a^T F_e$

算法	OS 零力控制
问题类型	操作空间零力控制
已知	D-H 参数 $d_n, a_n, \alpha_n, n = 1, ..., N$ 连杆参数 $m_n, \mathbf{I}_n^n, p_{l_n}^n, n = 1, ..., N$ 摩擦矩阵 F_f 关节向量及其微分 $q, \dot{q}$, 末端力 F_e
求	控制量 (驱动力矩) $u(q, \dot{q}, x_d, \dot{x}_d, \ddot{x}_d, F_e)$
算法性质	model-based, 解析解

17.2.4 导纳控制

[本部分内容将在后续版本中更新，敬请期待]

17.3 操作空间约束力控制

[本部分内容将在后续版本中更新，敬请期待]

Part V

深度学习方法

在此前的几个部分中，我们介绍了机器人领域的数学/物理基础知识、机器人的通用建模方法及结论 (即运动学/动力学)、控制理论以及机器人的简单控制方法。通过这些介绍，读者已经可以对机器人中用到的数学有基础掌握。

然而，上述理论早在上世纪 70 年代左右便已基本成熟。事实上，通过上述理论搭建的机器人，虽然具有严格执行人类指令的能力，但距离“自主感知”“自主判断”等更高级的人类智能尚有距离。这样的机器人，例如工厂流水线中的机械臂等，构成的是目前大众普遍认知中“机械、僵硬、刻板”的机器人形象，而非自然、仿生、灵动的现代机器人形象。

从第 V 部分开始，我们将介绍现代机器人领域的强大工具——深度学习；并以此为基础，介绍使机器人具备初步**自主感知、自主决策、自主学习**能力的现代机器人技术——强化学习、视觉 SLAM、足式步态等。它们是未来具有真正仿生智能的类人机器人的技术的基石。

18 深度学习基础

我们——人类——是这个星球上独一无二的物种。构成我们独特性的原因有很多，但其中最核心的是我们的智能——自适应的、灵动感知的、善于学习的、通用泛化的、具有语言和思维的、拥有逻辑和情感的——人类智能。

自现代科学诞生以来，我们从未停下探索我们自身智能的脚步——从生理学、神经科学到脑科学，从语言学、心理学到社会学，从控制科学、计算机科学到机器人学——我们希望探索我们智慧的根源，**我们希望制造出和我们一样智慧的机器人，从而更好地服务人类社会**。

在这一探索的历程中，**深度学习** (Deep Learning, DL) 是为我们带来了最多惊喜的技术领域。机器学习 (Machine Learning, ML) 是信息科学中数据科学的一个分支，研究如何从庞大的数据中寻找规律、获取知识。深度学习是机器学习领域的一个分支，聚焦于机器学习中带有”深层”性质的机器学习模型。

自 2012 年以来，一类名为**深度神经网络** (Deep Neural Network, DNN) 的深度学习模型从一众机器学习方法中脱颖而出，在数据科学领域取得了重大突破。具体来说，DNN 是一类以人类神经元为灵感的多层非线性机器学习模型。其结构多样、参数量庞大，展现出强大的复杂函数拟合能力。自 2012 年起，DNN 在图像、文字、语音和时序信息、半结构化信息、多模态信息等以往认为”困难复杂”的信息处理领域取得了跨越式的突破，解决了长久以来对这些信息特征提取能力弱、任务指标差等问题，成为了机器学习领域最重要的方法。在本部分的后续章节中，我们将具体介绍这些任务、数据类型，以及相应的 DNN 方法。

深度学习就是以 DNN 等多层非线性机器学习模型为研究对象的机器学习子领域。目前，深度学习最重要的应用领域分为三部分：**计算机视觉** (Computer Vision, CV)、**强化学习** (Reinforcement Learning, RL)、**自然语言处理** (Natural Language Processing, NLP)。这三个领域都是广泛的人工智能领域的重要组成部分，对构建真正智能的类人机器人至关重要。

在本章中，我们将首先介绍机器学习领域的重要基本概念，随后介绍 DNN 模型的概念，并以最简单的 DNN 模型 (多层感知机 MLP) 为例，介绍深度学习的基本范式。

18.1 机器学习基本概念

机器学习是研究如何从数据中学习规律的科学。机器学习的基本研究对象之一是一组相同格式的数据，称为**数据集**。数据集中隐藏着未知的规律，机器学习希望通过某种方式**参数化建模**这种规律，这个模型称为**机器学习模型**，或模型；模型中的参数称为**模型参数**。对于每个特点的数据集，会有一个特定的任务，称为**机器学习任务**。我们针对不同任务/数据集的特点，设计不同的机器学习模型，并设计方法来优化求解其参数，这一过程中的计算方法也称为**机器学习算法**。模型参数优化的过程也称为**模型学习**或**模型训练**。

18.1.1 机器学习基本问题

具体来说，在机器学习中，我们有四大基本问题，它们构成了绝大多数的机器学习任务

问题	回归问题
问题简述	已知样本集、标签集，求最优变换以拟合样本到标签映射
已知	样本集 $X = [\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(\mathbf{N})}] \in \mathbb{R}^{n \times N}$ 标签集 $Y = [y^{(1)}, y^{(2)}, .., y^{(n)}]^T$
求	变换 $f: \mathbb{R}^n \to \mathbb{R}$，使得 $f(\mathbf{x}^{(\mathbf{i})})$ 尽量好的拟合 $y^{(i)}$

问题	分类问题
问题简述	已知样本集、标签集，求最优变换以拟合样本到分类标签映射
已知	样本集 $X = [\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(\mathbf{N})}] \in \mathbb{R}^{n \times N}$ 标签集 $Y = [y^{(1)}, y^{(2)}, .., y^{(n)}]^T$，其中 $\mathbf{y}^{(\mathbf{i})} \in L_c = \{c_i \mid i = 1, ..., c\}$
求	求变换 $f: R^n \to L_c$，使得 $f(\mathbf{x}^{(\mathbf{i})}) = y^{(i)}$

问题	降维问题
问题简述	已知数据集，求编解码变换，最小化重构误差[17]
已知	数据集 $X = [\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(\mathbf{N})}] \in \mathbb{R}^{n \times N}$
求	编码和解码变换 $f: \mathbb{R}^n \to \mathbb{R}^m, g: R^m \to R^n, m < n$ 使得重构样本 $g(f(\mathbf{x}^{(\mathbf{i})}))$ 尽量好的拟合原样本 $x^{(i)}$

问题	聚类问题
问题简述	已知数据集，求类内差异小、类间差异大的聚类划分方法
已知	数据集 $X = [\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(\mathbf{N})}] \in \mathbb{R}^{n \times N}$
求	变换 $f: R^n \to L_c, L_c = \{c_i \mid i = 1, ..., c\}$ 使得相似数据尽量被分到同一个类别标签中

在这四大基本问题中，前两个基本问题要求数据集分为一一对应的样本集和标签集，且需要模型表示从样本集到标签集之间映射的规律。这两个问题对应的问题也称为**监督学习**问题。相应的，后两个问题不涉及映射，表示的是数据集自身内部的规律，称为**非监督学习**问题。

上述任务的基本前提都是具有一定量的数据。事实上，**机器学习就是一门通过数据本身寻找规律的科学**。自人类走入信息化以来，各类传感器、计算机在人类社会中的普遍应用，为人类带来了大量的数据，以及处理这些数据的需求。机器学习正是在这样的背景下诞生的科学。

在机器人技术涉及的各大领域中，有非常多的问题，都可以预先或即时收集大量对应的数据。这就为使用机器学习/深度学习方法解决这些领域问题提供了重要的基本条件。

18.1.2 机器学习任务指标

任务指标是评价 ML 模型完成任务好坏的量化评价标准。对于不同的具体任务，不同的指标也有不同的意义，有些指标越大越好，有些指标越小越好，不可一概而论。下面，我们介绍一些**分类和回归任务**中的常见指标。

对于**分类任务**，我们首先要介绍**混淆矩阵**的概念。混淆矩阵是一种展示 ML 模型的分类结果和数据标签异同情况的结果表示方法，它由两行两列组成。对于每个样本，它的分类结果必定出现且仅出现在 1 个单元格中。

	实际为正类	实际为负类
预测为正类	TP	FP
预测为负类	FN	TN

具体来说，无论是二分类还是多分类问题，每个样本都有一个真实类别标签。对于这一样本，类别标签对应的类别称为阳，类别标签以外的称为阴；ML 模型的预测和真实标签重合称为真，不重合称为假。这样，每个样本的预测结果就必定属于 4 类之一。我们将 4 个格子统计出的样本总数分别称为真阳性数 (TP)、假阳性数 (FP)、真阴性数 (TN)、假阴性数 (FN)。有

$$TP + FP + TN + FN = N \tag{V.18.1}$$

基于混淆矩阵，我们首先定义**准确率** (Accuracy) 指标，它表示模型预测正确的样本数占总样本数的比例：

$$\text{Accuracy} = \frac{TP + TN}{N} \tag{V.18.2}$$

准确率是分类问题中最符合直觉的指标，然而当数据存在类别不平衡时，准确率可能会产生误导。例如，在一个 90% 为负类、10% 为正类的数据集中，即使模型将所有样本均判定为负类，其准确率也可以达到 90%。准确率在这种情况下并不符合主观判断。

为此，我们可以定义两个更细致的指标：**精确率** (Precision) **召回率** (Recall)。精确率衡量的是：预测为正类的样本中，有多少是真正的正类：

$$\text{Precision} = \frac{TP}{TP + FP} \tag{V.18.3}$$

相应的，召回率衡量的是：所有实际为正类的样本中，有多少被模型正确预测为正类：

$$\text{Recall} = \frac{TP}{TP + FN} \tag{V.18.4}$$

精确率强调预测的准确性，而召回率强调对正类的覆盖能力。对于不同的实际任务需求，我们困难会更看重二者中的某一指标。例如，对于垃圾邮件识别等假阳性成本很高的场景，精确率更为重要；而对于疾病检测等假阴性成本较高的场景，召回率更加重要。

对于一些二者都比较重要的场景，为了同时体现精确率和召回率的作用，我们引入 **F1 分数**指标，其定义为：

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{V.18.5}$$

F1 分数是精确率和召回率的调和平均，尤其适用于类别不平衡的情况。

对于**回归任务**，常用的评价指标是**均方根误差（Root Mean Squared Error, RMSE）**，它衡量的是预测值与真实值之间的平均偏差程度：

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2} \tag{V.18.6}$$

其中，$y^{(i)}$ 是真实值，$\hat{y}^{(i)}$ 是 DNN 的预测值。RMSE 对较大的误差更加敏感，因此特别适用于对预测精度要求较高的场景。

对于无监督学习问题、生成问题，也有相应的指标。我们不在这里过多介绍，读者若有兴趣，可以参考其他深度学习相关教材。

18.1.3 机器学习损失函数

在深度学习中，损失函数是 ML 模型和任务目标之间的桥梁，是可学习参数的优化目标。具体来说，在监督学习中，损失函数是模型输出和真实数据标签的函数，用于衡量模型预测值与真实值之间的差异，必须对模型输出可导。针对不同的任务，我们会使用不同的损失函数。

对分类任务来说，最常用的损失是**交叉熵损失** (Cross-Entropy Loss)

$$\mathcal{L}(p_{:,:}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_c^{(i)} \ln(p_c^{(i)}) \tag{V.18.7}$$

其中，N 为样本数量，C 为类别数，$y_c^{(i)}$ 表示第 i 个样本是否属于类别 c，通常使用 one-hot 编码；$p_c^{(i)}$ 是神经网络的输出，即对样本 i 属于类别 c 概率的预测。在分类任务中，网络最终层的激活函数会选取 softmax 函数，确保 $\sum_{c=1}^{C} p_c^{(i)} = 1$。

在传统 ML 中，我们有以下结论：对于高斯先验分布的 Logistic 回归模型，其参数的最大似然估计即表现为交叉熵损失[18]。对于多层的 DNN(如 MLP，详见下节)，我们可以认为其最后一层是一个 Softmax 分类器，而前 $L-1$ 层是一个特征提取器。在训练的过程中，同时学习了数据特征提取方法以及分类权重。这也是一种**特征学习**思想的体现。

交叉熵损失的缺点是：对于错误分类的样本惩罚较大，可能导致梯度爆炸；对标签噪声敏感，尤其在标签不准确时容易过拟合。在工程实现中，我们一般使用正则化方法应对此类问题，在下一章中具体介绍。

KL 散度 (Kullback-Leibler Divergence) 是另一种常用的分类任务损失，其定义为

$$\mathcal{L}_{KL}(P||Q) = \sum_i P(i) \ln \frac{P(i)}{Q(i)} \tag{V.18.8}$$

KL 散度是两个概率分布 P 和 Q 之间差异的度量，是一种非对称的度量。在分类任务中，我们一般取 P 为真实标签，Q 为 DNN 输出的分布。和交叉熵损失类似，使用 KL 散度时 DNN 的最后一次也一般使用 Softmax 函数激活。

KL 散度的非对称特点使其在某些任务中可能不稳定。此外，KL 散度要求模型输出非 0：当 $Q(i) = 0$ 而 $P(i) > 0$ 时，KL 散度趋于无穷大，会导致数值不稳定和梯度爆炸。实际使用中，KL 散度一般用于建模为条件分类的文本生成任务。使用 Transformer 结构 (即自回归自注意力 DNN) 的大语言模型 (LLM) 一般就使用 KL 散度作为损失函数。

[18] 虽然推导仅限于二分类，但对于多酚类的 Softmax 回归也成立

对于回归任务，我们一般使用 **L2 损失**作为损失函数。L2 损失和作为回归任务指标的 RMSE 很像，在计算上仅差了一个开平方：

$$\mathcal{L}(\hat{y}^{(:)}; y^{(:)}) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2 \tag{V.18.9}$$

其中，$y^{(i)}$ 是样本真实标签，$\hat{y}^{(i)}$ 是模型预测。

L2 损失非常简单直观，它对于对大误差更敏感，有助于模型更快纠正偏差；导数计算也比较简便。它的缺点是：对异常值（outliers）非常敏感，可能导致模型过度关注极端样本；并且在误差分布非高斯时效果不佳。

在实际的机器学习任务中，我们会根据具体任务需求具体设计损失的形式，但大部分通用情形的损失都以上述几种损失为基础。

18.2 深度学习与 DNN

18.2.1 DNN 基本范式

在上节中，我们介绍了机器学习领域的基本问题。我们注意到这些基本问题本质都是求解某种数据集中的规律，或者**映射** f。

DNN 是人工神经元组成的多层人工神经网络，是一个带有大量可学习参数的非线性映射：$\text{DNN}(\cdot;\Theta) = f(\cdot)$。**使用 DNN 解决进行机器学习的基本范式是：在预先收集数据集的条件下，构建一定网络结构的、包含可学习参数的 DNN；通过一定优化算法优化可学习参数，以降低数据集上的损失函数值，从而求解出最优的参数；最优的参数和网络结构就构成了我们所需的映射** f。

这样，我们将各类机器学习基本问题转化为了求解函数的问题，继而将函数求解问题转化为了求解最优参数的问题。事实上，这一**映射参数化**的思想，正是深度学习领域的核心思想。

在上述基本范式中，我们可以提取出四要素：**网络结构**、**可学习参数**、**损失函数**、**优化算法**。

如上所述，DNN 是人工神经元组成的多层人工神经网络。神经元之间互相连接，一个神经元接受一定的输入、产生一定的输出。每个神经元输出一个值，称为**神经元激活值**。对于一个神经元，不同的输入激活值会使其按照一定的函数关系产生不同的输出激活值，这其中的线性系数称为 **(连接) 权重**。权重就是 DNN 中最主要的可学习参数。

神经元的连接方式又称**网络结构**，十分重要。针对不同的数据、针对不同的任务，我们会设计不同的网络结构。例如，本章后续节介绍的多层感知机 MLP 就是一种网络结构。后续章节中，我们还会介绍针对特殊的任务和数据的、更专用更强大的其他网络结构。

损失函数是针对任务定义的一个标量值函数，越大说明网络在数据集上完成任务的能力越差。在上一节中我们已经介绍了针对监督学习任务的基本损失函数。在深度学习中，损失函数以 DNN 的预测输出为输入，而 DNN 的预测输出和网络参数相关。因此，损失函数结合网络结构，就是网络参数的函数。我们根据这一关系，设计网络参数的更新算法。

上述网络参数的更新算法，或**优化算法**，主要依赖于损失函数对网络参数的梯度 (或偏导数)。几乎所有 DNN 都使用**梯度下降**类方法更新网络参数。在本章后续节中，我们将以 MLP 为例，介绍神经网络的反向传播和梯度下降算法。

18.2.2 DNN 的特点

相比于传统 ML 模型，DNN 具有一些重要的特点，使其与传统 ML 模型既有很强的联系又有所区分，从而形成深度学习这一较为独立的子领域。

算法即模型 机器学习领域中的模型一般指对某个分布建模的方式。在深度学习中，**模型指从输入 (及可学习参数) 到输出的 DNN 非线性函数**，即

$$\mathbf{y} = DNN(\mathbf{x}; \mathbf{W}) \tag{V.18.10}$$

在深度学习中，这个函数也称为**前向传播算法**、**模型结构**或**网络结构**[19]。DNN 的结构是非常多样和自由的，事实上，如何针对不同的任务和数据，设计适当的 f(以及相应的损失函数、优化算法)，正是深度学习的核心技术。

此外，DNN 的输入和输出的形式也非常多样，可以是向量、标量、矩阵、张量或其组合，由具体任务决定。DNN 的可学习参数量是预先确定的，一般是许多个矩阵或者张量，可学习参数的值也称为**模型权重**。

设计-训练-推理 DNN 的核心应用流程分为三个阶段：模型设计、模型训练、模型推理。所谓**模型设计**，就是在一个 Well-defined 问题中，明确输入和输出的形式，并按照函数可微等设计原则，设计选取上述网络结构 f。所谓**模型训练**，就是通过某种**优化算法**，在一定的**数据集**上，努力降低某个损失函数的值，直到得到一组较好的模型权重。所谓**模型推理**，就是按照设定好的 DNN 函数 f，根据输入和训练好的模型权重进行前向传播、求出输出。

大模型 由于 DNN 的结构可以是多个非线性函数的嵌套，其网络结构可以非常庞大和复杂，包含**巨量的可学习参数**。当参数量达到千万乃至十亿级别，我们就可以称之为 **DNN 大模型**。我们对大模型本质的理解尚浅，但结合当前的研究，我们可以认为：大模型中的参数可以隐式地" 编码" 某种具有通用性的知识——关于图像、文字、数据的知识——其编码能力远远超出" 小模型" 的能力，表现出**涌现**的特性，使得某种意义上的" 智能" 成为可能。

大数据 DNN 大量的参数需要与之匹配的大量数据进行训练，否则模型会发生**过拟合**的问题 (详见第 14 章)。事实上，近 10 年来深度学习爆发式发展的一个重要原因，就是互联网和信息技术的发展，使得百万至亿级规模的数据生产、收集、处理、整理成为可能。尽管一些研究使用的数据量庞大，深度学习社区在发展的过程中，还是逐渐形成了" 数据集开源 + 公开对比结果" 的研究范式，不断激励着研究者提出更好的方法。

并行计算 DNN 需要大量的数据处理、大量的参数学习，使得并行处理和计算的需求极为旺盛。深度学习时代到来的另一大原因，就是现代大规模并行计算硬件 (如 GPU) 的发展，使得千万级别 DNN 的训练成为可能。当前 (尤其是 2023 年以来)，深度学习已经成为高性能计算领域发展最主要的推动力。主流的并行计算硬件厂商如 NVIDIA/AMD 都在努力推出更高性能的计算卡，以支持深度学习领域的需求。

虽然实际应用中，DNN 的任务、结构丰富多样，但其基础原理十分一致。在本章接下来的节中，我们将以 MLP 为例，介绍**反向传播**与**梯度下降**这两个深度学习最重要的基础算法。在接下来的两章中，我们将介绍更多实用的 DNN 训练技巧和网络结构。

[19] 后续使用时，我们将不再区分这三个词

18.3　多层感知机 MLP

在所有 DNN 结构中，多层感知机是最简单的网络结构。在本节中，我们将首先介绍多层感知机的构成，随后以多层感知机为例，介绍反向传播和梯度下降算法。

18.3.1　MLP 结构

多层感知机 (Multi-Layer Perceptron, MLP) 是结构最简单的 DNN。对于简单任务，即输入和输出均非常简单、映射也不复杂的分类或回归任务，MLP 也是最常用的 DNN。在各类实用 DNN 中，它还是许多基础网络结构模块的原型。

MLP 由许多**人工神经元**构成。人工神经元是一种模仿生物神经元进行运算的多元函数，它接收一个向量为输入，对向量进行仿射变换，随后对结果进行非线性变换，最后输出一个标量。即

$$y(\mathbf{x}) = \phi(z) = \phi(\mathbf{w}^\top \mathbf{x} + b) \tag{V.18.11}$$

在人工神经元中，w 称为线性权重，b 称为偏置，w 和 b 合称为**权重**。仿射变换的结果 z 称为**净输入**，非线性变换 ϕ 称为**激活函数**，整个神经元的输出 y 也称为活性值。

在 MLP 中，多个神经元同时接受一组向量输入，称为一个**神经网络层**。每一层多个神经元的输出组成一个新的向量，称为**网络层输出** a。单层神经网络称为**感知机**；多个网络层堆叠串联，每层以前一层的输出作为输入，即为多层感知机。

对于一个 L 层的 MLP，记第 l 层的输入为 $a^{(l-1)}$，输出为 $a^{(l)}$，层内的净输入为 $z^{(l)}$。记该层的线性权重为 $W^{(l)}$，偏置为 $b^{(l)}$，有

$$\begin{aligned} z^{(l)} &= W^{(l)} a^{(l-1)} + b^{(l)} \\ a^{(l)} &= \phi(z^{(l)}) \end{aligned} \tag{V.18.12}$$

其中，假设 $a^{(l-1)} \in \mathbb{R}^{d_{l-1}}$，$a^{(l)} \in \mathbb{R}^{d_l}$，则 $z^{(l)} \in \mathbb{R}^{d_l}$，$W^{(l)} \in \mathbb{R}^{d_{l-1} \times d_l}$，$b^{(l)} \in \mathbb{R}^{d_l}$。我们定义 d_0 为输入的大小，d_L 为输出的大小。在 MLP 中，每一层的神经元数量 d_l 也称为该层的宽度，总层数 L 也称为 MLP 的深度。MLP 的深度和宽度是重要的结构超参数。

对于一个 MLP,激活函数的选择也非常重要。一般一个 MLP 的各层均使用同一激活函数。历史上,Sigmoid 函数和 Tanh 函数都曾被广泛应用，但目前最常见的激活函数是修正线性单元 (ReLU) 函数 (及其变种)。这几种激活函数的定义如下所示

Sigmoid 函数

$$\phi(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{V.18.13}$$

双曲正切 (Tanh) 函数

$$\phi(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{V.18.14}$$

修正线性单元 (ReLU)

$$\phi(x) = \max(0, x) \tag{V.18.15}$$

在以上三种激活函数中，Logistic 函数的输出范围为 $(0, 1)$，在 x 较大或较小时梯度趋近于 0。Tanh 函数输出范围为 $(-1, 1)$，相比 Sigmoid 存在均值为 0 的优点，但仍存在梯度趋近于 0 的问题。这种问题也被称为**饱和**。激活函数的梯度非常重要，涉及**梯度消失问题**。我们将在下一章详细说明该问题及解决方法。

相比之下，ReLU 函数的前向运算和梯度运算都非常简单，并且其正半部分有效缓解了饱和的问题。然而，在其净输入为负时，仍会出现 0 梯度，此时会出现**死亡神经元**问题。我们也在下一章详细说明该问题及解决方法。

以 ReLU 作为激活函数，我们总结 MLP 的前向传播算法如下

Algorithm 46: MLP 前向传播 (MLP)

Input: 网络输入 x

Input: 各层参数 $W^{(1:L)}, b^{(1:L)}$

Parameter: 网络层数 L, 各层宽度 $d^{1:L}$

Output: 网络输出 $\hat{y}$

$a^{(0)} \leftarrow x$

for $l \in 1, ..., L$ **do**

  $z^{(l)} \leftarrow W^{(l)} a^{(l-1)} + b^{(l)}$

  $a^{(l)} \leftarrow \max(0, z^{(l)})$

$\hat{y} \leftarrow a^{(L)}$

对应的 python 代码如下所示

```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 学习问题

深度学习的核心问题就是 NN 中大量可学习参数的求解。这一过程一般称为” 参数的学习”。和 Logistic 回归、RBM、SBN 等传统 ML 模型类似，DNN 也是基于梯度进行参数学习的。不同的是，深度学习中往往定义的是损失函数，我们希望其越小越好而非越大越好，因此 DNN 的学习是基于**梯度下降**而非梯度上升。

要想进行梯度下降，就需要先求解梯度。以 MLP 为例，我们定义**梯度求解问题**和**参数学习问题**。

问题	**MLP 梯度求解**
问题简述	已知 MLP 损失函数、输入、参数，求解各参数梯度
已知	损失函数 $\mathcal{L}(\hat{y}; y)$ 网络输入 x 各层参数 $W^{(1:L)}, b^{(1:L)}$
求	各层参数梯度 $\frac{\partial L}{\partial W^{(1:L)}}, \frac{\partial L}{\partial b^{(1:L)}}$

问题	**MLP 参数学习**
问题简述	已知数据集、MLP 损失函数，求解最佳的参数
已知	损失函数 $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$
求	最优参数 $W^{(1:L)}, b^{(1:L)}$

这里需要注意：一般意义上的损失函数，是定义在整个数据集上的损失。损失函数对某一参数的梯度涉及数据集中所有的样本。然而，对于绝大部分损失函数，总损失都是单个样本损失的均值；因此求解整个数据集上的梯度，就相当于分别对每个样本求解梯度再取平均。基于此，在 MLP 梯度求解问题中，我们仅考虑单个样本的梯度求解。

梯度求解是参数学习的基础。接下来，我们首先介绍 MLP 的梯度求解方法，也就是**反向传播算法**。随后以此为基础，介绍最简单的参数学习方法：梯度下降法。在下一章中，我们将以这些方法为基础，介绍大量的改进学习方法。

18.3.3 MLP 参数学习

首先我们来考虑 MLP 梯度求解问题。根据链式法则，损失函数 $\mathcal{L}$ 对第 l 层参数的 (偏) 导数为

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{\partial \mathcal{L}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial W^{(l)}}$$
$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \frac{\partial \mathcal{L}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial b^{(l)}}$$

因此，我们其实就将问题分解为：如何求损失函数对 $a^{(l)}$ 的导数，以及如何求 $a^{(l)}$ 对 $W^{(l)}$ 和 $b^{(l)}$ 的导数。对于前者，根据式V.18.12，结合链式法则，我们有

$$\frac{\partial \mathcal{L}}{\partial a^{(l)}} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial \hat{y}} \Pi_{m=L}^{l+1} \frac{\partial a^{(m)}}{\partial a^{(m-1)}}$$

进一步地，我们有

$$\frac{\partial a^{(l)}}{\partial a^{(l-1)}} = \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial a^{(l-1)}}$$

而对于后者，我们有

$$\frac{\partial a^{(l)}}{\partial W^{(l)}} = \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial W^{(l)}}$$
$$\frac{\partial a^{(l)}}{\partial b^{(l)}} = \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial b^{(l)}}$$

因此，MLP 梯度求解问题可以进一步分解为五个子问题：求解损失函数对网络输出的偏导 $\frac{\partial \mathcal{L}}{\partial \hat{y}}$、求解每层输出对净输入的偏导 $\frac{\partial a^{(l)}}{\partial z^{(l)}}$、求解净输入对本层输入的偏导 $\frac{\partial z^{(l)}}{\partial a^{(l-1)}}$、求解净输入对网络参数的偏导 $\frac{\partial z^{(l)}}{\partial W^{(l)}}, \frac{\partial z^{(l)}}{\partial b^{(l)}}$。

对于 $\frac{\partial \mathcal{L}}{\partial \hat{y}}$，以回归问题为例，我们计算一下 L2 损失的偏导，它的形式非常简单 (注意：本章中，我们统一使用导数的分子布局)

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = (\hat{y} - y)^T \tag{V.18.16}$$

对于 $\frac{\partial a^{(l)}}{\partial z^{(l)}}$，实际上这是求解激活函数的导数。假设 MLP 使用 ReLU 激活函数，我们有

$$\frac{\partial a^{(l)}}{\partial z^{(l)}} = \text{diag}\left(\mathbf{1}[z_1^{(l)} > 0], \mathbf{1}[z_2^{(l)} > 0], ..., \mathbf{1}[z_{d_l}^{(l)} > 0]\right) \tag{V.18.17}$$

其中，$\mathbf{1}[\cdot]$ 表示 $0-1$ 函数，当方括号内条件为真时取 1，否则取 0.

对于 $\frac{\partial z^{(l)}}{\partial a^{(l-1)}}$，这实际上就是线性权重

$$\frac{\partial z^{(l)}}{\partial a^{(l-1)}} = W^{(l)} \tag{V.18.18}$$

对于 $\frac{\partial z^{(l)}}{\partial b^{(l)}}$，这一偏导就是单位阵

$$\frac{\partial z^{(l)}}{\partial b^{(l)}} = I_{d_l} \tag{V.18.19}$$

最后，对于 $\frac{\partial z^{(l)}}{\partial W^{(l)}}$，我们将其分拆为对矩阵不同行向量的导数。假设矩阵 $W^{(l)}$ 可写为行向量形式

$$W^{(l)} = \begin{bmatrix} (w_1^{(l)})^T \\ (w_2^{(l)})^T \\ ... \\ (w_{d_l}^{(l)})^T \end{bmatrix}$$

则偏导 $\frac{\partial z^{(l)}}{\partial w_i^{(l)}}$ 为 $d_{l-1} \times d_l$ 的矩阵，且仅有第 i 行非 0

$$\frac{\partial z^{(l)}}{\partial w_i^{(l)}} = \begin{bmatrix} 0^T \\ ... \\ (a^{(l-1)})^T \\ ... \\ 0^T \end{bmatrix} \tag{V.18.20}$$

综合上述推导，以 ReLU 为激活函数，我们总结 **MLP 反向传播算法**如下

算法	**MLP 反向传播**
问题类型	MLP 梯度求解
已知	网络输入 x, 标签 y 各层参数 $W^{(1:L)}, b^{(1:L)}$ 损失函数 $\mathcal{L}(\hat{y}; y)$
求	梯度 $\frac{\partial L}{\partial W^{(1:L)}}, \frac{\partial L}{\partial b^{(1:L)}}$
算法性质	解析解

Algorithm 47: MLP 反向传播 (MLP_BP)

Input: 网络输入 x, 标签 y

Input: 各层参数 $W^{(1:L)}, b^{(1:L)}$

Input: 损失函数 $\mathcal{L}(\hat{y}; y)$

Output: 梯度 $\frac{\partial L}{\partial W^{(1:L)}}, \frac{\partial L}{\partial b^{(1:L)}}$

$\mathcal{L}, \hat{y}, a^{(1:L)}, z^{(1:L)} \leftarrow \text{MLP}(x; W^{(1:L)}, b^{(1:L)})$

$G^T_{a^{(L)}} \leftarrow \frac{\partial \mathcal{L}}{\partial \hat{y}}$

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**

$\frac{\partial z^{(l)}}{\partial w_i^{(l)}} = \begin{bmatrix} 0 & ... & a^{(l)} & ... & 0 \end{bmatrix}^T$

$\frac{\partial \mathcal{L}}{\partial w_i^{(l)}} \leftarrow G^T_{a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial w_i^{(l)}}$

对应的 python 代码如下所示

```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
```

在第3章中，我们介绍了许多非线性优化的算法。事实上，深度学习中的参数学习问题，正是一个典型的非线性优化问题。由于我们已经求出了梯度，我们很容易想到使用**梯度下降法**迭代优化 MLP 每层的网络参数

$$
\begin{aligned}
W^{(l)} &\leftarrow W^{(l)} - \alpha \nabla_{W^{(l)}} \mathcal{L} \\
W^{(l)} &\leftarrow b^{(l)} - \alpha \nabla_{b^{(l)}} \mathcal{L} \\
\nabla_{W^{(l)}} \mathcal{L} &= \frac{\partial \mathcal{L}}{\partial W^{(l)}} \Big|_{1:N}^{T} \\
\nabla_{b^{(l)}} \mathcal{L} &= \frac{\partial \mathcal{L}}{\partial b^{(l)}} \Big|_{1:N}^{T}
\end{aligned}
$$

其中，α 是**学习率**，是 DNN 中重要的超参数。需要注意：此处的梯度是整个数据集上的梯度，也就是需要先在每个样本上求解上述梯度，再在整个数据集上计算平均。为简单起见，我们使用 Θ 代表网络中的所有参数 (即 $W^{(1:L)}, b^{(1:L)}$)，记

$$
G_{\Theta,i} = \frac{\partial \mathcal{L}}{\partial \Theta} \Big|_{i}^{T} \tag{V.18.21}
$$

则梯度下降可以写为

$$
\Theta \leftarrow \Theta - \alpha G_{\Theta,1:N} \tag{V.18.22}
$$

参数 Θ 需要初值。我们将在下一章详细介绍 MLP 的参数初始化方法，此处我们可以先按照标准差 σ_W, σ_b，进行 0 均值的随机高斯噪声初始化。

综上，我们总结 MLP 的**梯度下降优化算法**如下

Algorithm 48: GD 优化

Input: 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$
Input: 损失函数 $\mathcal{L}(\hat{y}; y)$
Parameter: 随机初始化参数 σ_W, σ_b
Parameter: 学习率 α, 目标损失 L_{tar}
Output: 最优参数 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
```

算法	GD 优化
问题类型	MLP 参数学习
已知	损失函数 $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$
求	最优参数 $W^{(1:L)}, b^{(1:L)}$
算法性质	迭代解

对应的 python 代码如下所示

```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 深度学习理论

18.4.1 通用近似定理

神经网络的通用近似定理是深度学习的基石，它描述了一个 DNN 拥有拟合任意 (满足一定条件) 的函数的能力，从理论上确保了 DNN 的可用性。

我们假设 X 是欧几里得空间 $\mathbb{R}^n$ 的子集，以该集合为定义域、$\mathbb{R}^m$ 为值域的**连续**函数集合表示为 $C(X, \mathbb{R}^m)$。激活函数 $\sigma \in C(\mathbb{R}, \mathbb{R})$，且对于向量 $x \in \mathbb{R}^n$, $(\sigma \circ x)_i = \sigma(x_i)$。

通用逼近定理的内容为：则当且仅当 σ 不是一个多项式函数时，$\forall n \in \mathbb{N}, m \in \mathbb{N}$，对于任意紧集 $K \subseteq \mathbb{R}^n$。对于任意目标函数 $f \in C(K, \mathbb{R}^m)$，以及任意 $\varepsilon > 0$，存在 $k \in \mathbb{N}$、矩阵 $A \in \mathbb{R}^{k \times n}$、向量 $b \in \mathbb{R}^k$、矩阵 $C \in \mathbb{R}^{m \times k}$，使得如下不等式成立：

$$\sup_{x\in K}\|f(x)-g(x)\|<\varepsilon \tag{V.18.23}$$

其中，

$$g(x)=C\cdot(\sigma\circ(A\cdot x+b)) \tag{V.18.24}$$

该定理由 George Cybenko 和 Kurt Hornik 证明，发表于 1989-1991 年的一系列文章中。注意到，该定理描述了一个单层神经网络，当该神经网络的宽度 k 可以任意大时，它可以以任意精度逼近任意 $C(X,\mathbb{R}^m)$ 中的连续函数。

18.4.2 没有免费午餐

没有免费午餐定理 (No Free Lunch, NFL) 是机器学习领域中的经典定理，是上面介绍的通用近似定理的重要补充。即：尽管 DNN 有强大的拟合能力，但每一个 DNN 仅能拟合一个特定问题的解，而不能同时拟合所有问题的解。这个定理的实质是：不可能存在一个能针对所有不同问题都起效的神经网络。如果两个问题有本质区别，那么它们就不可能用同一个 DNN 拟合。

在我们进行问题定义时，没有免费午餐定理是一个重要参考原则。我们需要深入思考，我们所定义的 DNN 是否尝试解决不同本质的问题。

18.4.3 奥卡姆剃刀

奥卡姆剃刀原则可以用一句话概括：**如无必要，勿增实体**。

深度学习的算法是黑箱算法，算法的效果受到训练过程中许多条件的影响。这里的条件主要指各类超参数，包括我们上面介绍的模型超参数，以及下一章将介绍的正则化相关超参数。这些超参数的设定和调节，很多时候是经验性的。我们往往仅能以验证集的指标作为依据调整这些参数，也就是**面向结果调参**。在这一过程中，根据奥卡姆剃刀原则，我们应该遵守如下的规则：如果调节参数的结果和不调节区别不大，那么就维持超参数的原始值。

同样的，在下一章和后续章节中，我们将介绍各类针对基础算法的改进提升方法与技巧 (trick)。这些方法与技巧很多时候都是模块化的，即可以针对某个问题的某个 DNN 模型选择性加入或不加入。这种时候，我们进行决策的依据，基本也仅来自实验结论（包括我们自己的实验和前人做过的实验）。我们仍然按照奥卡姆剃刀原则进行决策：如果增加一个 trick 的效果和不增加没什么区别，那就不采用它。

19 优化与正则化

19.1 DNN 的困难

在上一章中，我们已经介绍了最基础的 MLP 反向传播/梯度下降算法，实现了 DNN 参数的迭代优化。然而，这样训练的 DNN 会面临两个关键问题：**优化困难**和**泛化困难**。这两种困难普遍存在于使用 DNN 的各类任务中，极大影响 DNN 的性能。本节中我们将介绍这两种困难的原理，后续几节将介绍各类优化技巧。

19.1.1 优化困难

如上一章所述，深度学习的参数学习问题是一个非线性优化问题。事实上，由于深度学习使用的 DNN 网络层数一般较多，相对网络参数的损失函数具有高度非凸性、维数极高，优化存在很大困难。

首先是**梯度消失问题**。注意到，在反向传播算法中，有

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \theta^{(l)}} &= G_{a^{(L)}}^T \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial \theta^{(l)}} \\ G_{a^{(l-1)}}^T &= G_{a^{(l)}}^T \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial a^{(l-1)}}\end{aligned} \tag{V.19.1}$$

式中的 $\frac{\partial a^{(l)}}{\partial z^{(l)}}$ 是神经元激活值对净输入的偏导，也就是激活函数自身的导数。注意到 Sigmoid 函数的导数为

$$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2} \tag{V.19.2}$$

而 tanh 函数的导数为

$$\tanh'(x) = \frac{4}{(e^x + e^{-x})^2} \tag{V.19.3}$$

这两个激活函数的导数在 $x \to +\infty$ 时会很快趋于 0。因此，当神经网络中某一神经元的输出过大 (称为**输出饱和**)，则该层的参数对损失函数的梯度将变为极小。不仅如此，层编号小于该层的网络层的梯度也将极小。即使单个神经元的梯度并不明显趋于 0，但如果多个网络层的梯度都小于 1，由于反向传播算法中各层梯度的累乘关系，靠前的网络层的梯度也将变得极小，从而使梯度下降存在很大困难。这一问题，我们称之为**梯度消失问题**。

与梯度消失相反，tanh 函数的梯度可能大于 1。在累乘的情况下，多个大于 1 的梯度相乘将导致靠前的网络层的梯度变得极大，从而优化过程中数值不稳定甚至溢出。这样的问题称为**梯度爆炸问题**。

自神经网络问世以来的很长时间，梯度消失和梯度爆炸问题限制了神经网络的大规模化和深度化。ReLU 函数较好的解决了这个问题，因此成为许多多层神经网络的首选损失函数。

除了梯度爆炸和梯度消失，DNN 的优化还存在**鞍点与平坦区**问题。鞍点是高维函数一阶导数为 0 但并非局部极值的点。在高维空间中，鞍点比局部极小点更常见。由于鞍点附近的梯度接近于零，当优化变量进入这一区域，优化过程就容易陷入停滞。平坦区域与鞍点类似，也是一片梯度接近 0 的区域。它们会导致优化速度非常缓慢，损失难以下降。

在接下来的几节中，我们将介绍缓解优化困难的优化算法。

19.1.2 泛化困难

除了优化困难，DNN 和深度学习面临的另一大难题是**泛化困难**。所谓泛化就是指：机器学习方法在某一特定数据集上进行训练，而良好地应用于其他相同任务的不同数据上的能力。

DNN 拥有大量参数。在使用优化算法训练 DNN 时，随着训练轮数的增加，测试集上的损失函数会呈现 U 型曲线。左侧下降部分称为模型**欠拟合**，即 DNN 尚未完全获取数据集中的规律信息。右侧上升部分称为**过拟合**，即 DNN 通过大量参数" 记忆" 了数据集中的特化数据，超出了对共性规律的关注。所谓泛化问题，主要就是指 DNN 的过拟合问题。

过拟合问题和模型/数据集大小有较大关系。模型的大小，或模型中可学习参数的多少，决定了模型的" 潜在表示能力"，或模型可以表达的非线性函数的" 复杂程度"。如果任务简单而模型复杂，就会出现过拟合问题。如果任务复杂但数据量小，模型也会出现过拟合问题。

为了更好地描述泛化困难，我们可以将模型在数据集外的其他数据上的误差分解为**偏差、方差与噪声**三部分

$$\mathbb{E}_{\mathcal{D}}[(f(x;\mathcal{D})-y)^2] = \text{Bias}^2 + \text{Variance} + \text{Noise} \tag{V.19.4}$$

式中，**偏差 (Bias)** 表示 DNN 学习的映射与理想的非线性函数之间的误差；**方差 (Variance)** 表示更换不同训练集时，模型输出的变化程度；**噪声 (Noise)** 表示额外数据自身的随机性。

针对上述泛化困难，我们希望通过各种设计方法，尽量**降低模型的方差**，提升模型的泛化能力。这一设计理念也称为**正则化**。在接下来的几节中，我们也将介绍模型正则化方法。

19.2 DNN 优化算法

在本节中，我们首先介绍缓解优化困难的一些算法。下一节中我们将介绍应对泛化困难的方法。

19.2.1 SGD 与批量 SGD

上一章中，我们介绍了使用梯度下降 (GD) 进行 MLP 参数优化的方法。GD 方法中，每次优化参数使用的梯度都来自整个数据集的梯度平均，在数据集规模较大时计算较为麻烦。

事实上，我们也可以在每次迭代时只使用一个样本的梯度。虽然单样本的梯度和全数据集的梯度可能相差较大，但在期望意义下，该方法的梯度仍然是符合 GD 的。此外，随机采集样本带来的随机性可能有助于网络逃离鞍点或平坦区域。这样的优化策略也称为**随机梯度下降** (Stochastic Gradient Descent, SGD)。

我们将 MLP 的 **SGD 优化算法**总结如下：

Algorithm 49: SGD 优化

Input: 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$, 损失函数 $\mathcal{L}(\hat{y};y)$
Parameter: 学习率 α, 目标损失 L_{tar}
Parameter: 随机初始化参数 σ_W, σ_b
Output: 最优参数 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
  $\Theta \leftarrow \Theta - \alpha G_{\Theta,i}$

对应的 python 代码如下所示

算法	**SGD 优化**
问题类型	MLP 参数学习
已知	损失函数 $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$
求	最优参数 $W^{(1:L)}, b^{(1:L)}$
算法性质	迭代解

对应的 python 代码如下所示

```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
```

相比于 GD，SGD 每次梯度更新方向相差较大，优化路径的震荡比较明显。在实际使用时，我们往往使用二者的折衷，即**小批量随机梯度下降** (Batch Stochastic Gradient Descent, BSGD) 方法。

具体来说，BSGD 算法每次更新参数时，使用的梯度既不是全数据集的平均，也不是单个随机样本，而是数据集中的一个小组 (称为一个 **batch**) 的数据。我们记一个 batch 的索引集合为 $\mathcal{B}$，其大小为 b，有

$$\Theta \leftarrow \Theta - \alpha \frac{1}{b} \sum_{i \in \mathcal{B}} G_{\Theta,i} \tag{V.19.5}$$

事实上，常数 $\frac{1}{b}$ 可包含于超参数 α 中。我们总结 MLP 的 **BSGD 优化算法**如下：

算法	BSGD 优化
问题类型	MLP 参数学习
已知	损失函数 $\mathcal{L}(\hat{y}^{(:)};y^{(:)})$ 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$
求	最优参数 $W^{(1:L)},b^{(1:L)}$
算法性质	迭代解

Algorithm 50: MLP-BSGD 优化

Input: 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$, 损失函数 $\mathcal{L}(\hat{y};y)$

Parameter: 目标损失 L_{tar}, 学习率 α, 批大小 b

Parameter: 随机初始化参数 σ_W,σ_b

Output: 最优参数 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

  $\Theta \leftarrow \Theta - \alpha G_\Theta$

对应的 python 代码如下所示

```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 学习率衰减

在上述 GD 及其变种算法中，学习率 α 都是一个非常关键的超参数。如果学习率过大，可能导致训练震荡甚至发散；如果学习率过小，可能导致参数收敛缓慢。然而，固定学习率可能并不是最好的策略。

经过大量实验，研究人员开发出**学习率衰减**方法。学习率衰减是指：使学习率以某种方式，随数据集迭代轮数增大而逐渐减小；即在 DNN 训练的前期使用较大学习率，后期使用较小学习率。这样，DNN 在前期迅速使 loss 下降，后期精细寻找一定范围内的极小值。

具体来说，最常用的学习率衰减策略是**指数衰减**和**余弦衰减**。对于指数衰减，第 k 轮的学习率 α_k 为

$$\alpha_k = \alpha_0 \gamma^k \tag{V.19.6}$$

其中 $0 < \gamma < 1$ 为超参数。对于余弦衰减，有

$$\alpha_k = \alpha_0 \cdot \frac{1}{2}\left(1 + \cos\left(\frac{k\pi}{N_I}\right)\right) \tag{V.19.7}$$

我们以指数衰减为例，总结 MLP 的**学习率衰减 BSGD 优化算法**如下：

算法	**学习率衰减 BSGD 优化**
问题类型	MLP 参数学习
已知	损失函数 $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$
求	最优参数 $W^{(1:L)}, b^{(1:L)}$
算法性质	迭代解

Algorithm 51: 学习率衰减 BSGD 优化

Input: 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$, 损失函数 $\mathcal{L}(\hat{y}; y)$

Parameter: 目标损失 L_{tar}, 批大小 b

Parameter: 学习率初值 α_0, 学习率衰减率 γ

Parameter: 随机初始化参数 σ_W, σ_b

Output: 最优参数 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

上面介绍的学习率衰减方法可以从大到小调整学习率。然而，这样的调整是针对所有参数的。事实上，在一次更新中，每个参数的梯度大小是不一样的，如果采用相同学习率，可能有些参数更新较大，有些更新较小。

我们希望：在比较理想的情况下，所有参数的更新步长都比较类似。也就是说，梯度大的参数其学习率小一些，梯度小的参数学习率大一些，也就是学习率能对于不同的参数自适应调整。具体来说，对于每个 $\theta \in \Theta$，我们可以使用如下的策略

$$\theta \leftarrow \theta - \frac{\alpha}{\sqrt{G_\theta^2 + \epsilon}} G_\Theta \tag{V.19.8}$$

不过，由于 SGD/BSGD 中，样本的选取具有随机性，同一个参数每轮的梯度可能也变化较大。我们希望参数更新的步长在一定的窗口内较为平滑。对此，我们将 $||G_\theta||^2$ 替换为一种历史滑动平均。我们有

$$\begin{aligned} v_{\theta,0} &= 0 \\ v_{\theta,k} &= \rho v_{\theta,k-1} + (1-\rho) G_\theta^2 \end{aligned} \tag{V.19.9}$$

其中 $\rho \in [0, 1]$ 是动量系数，也是一个重要的超参数。ϵ 是防止除零的小常数。即

$$\theta \leftarrow \theta - \frac{\alpha}{\sqrt{v_{\theta,k} + \epsilon}} G_\Theta \tag{V.19.10}$$

上述梯度自适应学习率的优化方法称为 **RMSProp**。我们总结该算法如下

Algorithm 52: RMSProp 优化

Input: 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$, 损失函数 $\mathcal{L}(\hat{y}; y)$

Parameter: 目标损失 L_{tar}, 批大小 b

Parameter: 学习率初值 α_0, 学习率衰减率 γ, 动量系数 ρ, 小值 ϵ

Parameter: 随机初始化参数 σ_W, σ_b

Output: 最优参数 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**

    $v_{\theta,k} \leftarrow \rho v_{\theta,k-1} + (1-\rho) G_\theta^2$

    $\theta \leftarrow \theta - (\alpha_k / \sqrt{v_{\theta,k} + \epsilon}) G_\theta$

算法	RMSProp 优化
问题类型	MLP 参数学习
已知	损失函数 $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$
求	最优参数 $W^{(1:L)}, b^{(1:L)}$
算法性质	迭代解

对应的 python 代码如下所示

```python
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 动量法

上述历史加权平均技巧也可直接用于 (Loss 对参数的) 梯度，这种方法称为**动量法**。具体来说，我们有滑动平均估计

$$\begin{aligned} m_{\theta,0} &= 0 \\ m_{\theta,k} &= \beta m_{\theta,k-1} + (1-\beta) G_\theta \end{aligned} \tag{V.19.11}$$

而参数更新直接使用当前滑动平均值

$$\theta \leftarrow \theta - \alpha m_{\theta,k} \tag{V.19.12}$$

这样，我们总结**动量 BSGD 优化算法**如下：

算法	动量 BSGD 优化
问题类型	MLP 参数学习
已知	损失函数 $\mathcal{L}(\hat{y}^{(:)};y^{(:)})$ 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$
求	最优参数 $W^{(1:L)},b^{(1:L)}$
算法性质	迭代解

Algorithm 53: 动量 BSGD 优化

Input: 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$, 损失函数 $\mathcal{L}(\hat{y};y)$

Parameter: 目标损失 L_{tar}, 批大小 b

Parameter: 学习率初值 α_0, 学习率衰减率 γ, 动量系数 β

Parameter: 随机初始化参数 σ_W,σ_b

Output: 最优参数 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**

    $m_{\theta,k}\leftarrow\beta v_{\theta,k-1}+(1-\beta)G_\theta$

    $\theta\leftarrow\theta-\alpha_k m_{\theta,k}$

对应的 python 代码如下所示

```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 优化

将上述两种历史加权平滑方法 (动量与 RMSProp) 结合，我们就得到了 Adam 优化算法。我们将其中的 ρ, β 重命名为 β_2, β_1。

$$
\begin{aligned}
m_{\theta,0} &= 0 \\
v_{\theta,0} &= 0 \\
m_{\theta,k} &= \beta_1 m_{\theta,k-1} + (1-\beta_1) G_\theta \\
v_{\theta,k} &= \beta_2 v_{\theta,k-1} + (1-\beta_2) G_\theta^2
\end{aligned}
\tag{V.19.13}
$$

其参数更新为

$$
\theta \leftarrow \theta - \alpha \frac{m_{\theta,k}}{v_{\theta,k} + \epsilon} \tag{V.19.14}
$$

这样，我们总结 **Adam 优化算法**如下：

算法	**Adam 优化**
问题类型	MLP 参数学习
已知	损失函数 $\mathcal{L}(\hat{y}^{(:)}; y^{(:)})$ 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$
求	最优参数 $W^{(1:L)}, b^{(1:L)}$
算法性质	迭代解

Algorithm 54: Adam 优化

Input: 样本集 $x^{(1:N)}$, 标签集 $y^{(1:N)}$, 损失函数 $\mathcal{L}(\hat{y};y)$

Parameter: 目标损失 L_{tar}, 批大小 b

Parameter: 学习率初值 α_0, 学习率衰减率 γ, 动量系数 β

Parameter: 随机初始化参数 σ_W, σ_b

Output: 最优参数 Θ

$m_{\Theta,0}, v_{\Theta,0} \leftarrow 0_\Theta, 0_\Theta$

$\Theta \leftarrow \text{random\_init}(\sigma_W, \sigma_b)$

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**

    $m_{\theta,k} \leftarrow \beta_1 m_{\theta,k-1} + (1-\beta_1) G_\theta$

    $v_{\theta,k} \leftarrow \beta_2 v_{\theta,k-1} + (1-\beta_2) G_\theta^2$

    $\theta \leftarrow \theta - \alpha \frac{m_{\theta,k}}{v_{\theta,k}+\epsilon}$

对应的 python 代码如下所示 (此处我们给出完整代码)

```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 正则化技巧

上一节中我们介绍了针对优化困难的各种优化方法。针对泛化困难，我们也有各种正则化技巧。

19.3.1 正则化损失

[主要内容：L1 损失、L2 损失]

[本部分内容将在后续版本中更新，敬请期待]

19.3.2 批归一化 BN

批归一化 (Batch Normalization, BN) 是一种基于 BSGD 类方法，在层中进行归一化的方法。

具体来说，对于每个批量 $\mathcal{B}$，我们统计每层的输入 $a^{(l-1)}$ 的均值 $\mu^{(l)}$ 和方差 $(\sigma^2)^{(l)}$(假设层中各神经元独立，不考虑协方差)

$$\begin{aligned}\mu^{(l)} &= \frac{1}{b}\sum_{i=1}^{b} a_i^{(l-1)} \\ (\sigma^2)^{(l)} &= \frac{1}{b}\sum_{i=1}^{b}(a_i^{(l-1)} - \mu^{(l)}) \odot (a_i^{(l-1)} - \mu^{(l)})\end{aligned} \tag{V.19.15}$$

因此归一化的输入为

$$z_i^{(l)} = \frac{a_i^{(l)} - \mu^{(l)}}{\sqrt{(\sigma^2)^{(l)} + \epsilon}} \tag{V.19.16}$$

此时，由于我们通过归一化去掉了偏置，因此网络层中的 $b^{(l)}$ 相当于无效，也就是我们的网络在前向传播时可以省略偏置参数 b，即 (以 ReLU 为激活函数)

$$\begin{aligned}z^{(l)} &= (a^{(l-1)} - \mu^{(l)})/(\sqrt{(\sigma^2)^{(l)} + \epsilon}) \\ \hat{z}^{(l)} &= W^{(l)} z^{(l)} \\ a^{(l)} &= \max(0, \hat{z}^{(l)})\end{aligned} \tag{V.19.17}$$

此时，我们可以总结**带 BN 的 MLP 前向传播算法**如下

Algorithm 55: MLP-BN 前向传播 (MLP_BN)

Input: 批量输入 $x^{(1:b)}$, 各层参数 $W^{(1:L)}$

Parameter: 网络层数 L, 各层宽度 $d^{(1:L)}$

Output: 批量网络输出 $\hat{y}^{(1:b)}$

Output: 各层归一化信息 $\mu^{(1:L)}, \sigma^{(1:L)}$

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**

  $\hat{y}^{(i)} \leftarrow a_i^{(L)}$

对应的 python 代码如下所示

```python
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]
```

由于在神经元中增加了额外的线性变换，因此反向传播算法也发生了变化。我们有

$$\frac{d\hat{z}_i}{dz_j} = \frac{\delta_{ij} - 1/b}{\sqrt{\sigma^2+\epsilon}} I - \frac{(z_i-\mu)(z_j-\mu)^T}{b\sqrt{\sigma^2+\epsilon}^3} \tag{V.19.18}$$

注意：此时的 MLP 中不再需要线性偏置，也无需计算导数，因为线性偏置会被 μ 完全替代。其导数已被包含于 $\frac{d\hat{z}}{dz}$ 中。

算法	MLP-BN 反向传播
问题类型	MLP 梯度求解
已知	批量输入 $x^{(1:b)}$, 标签 $y^{(1:b)}$ 各层参数 $W^{(1:L)}, b^{(1:L)}$ 损失函数 $\mathcal{L}(\hat{y}; y)$
求	梯度 $\frac{\partial L}{\partial W^{(1:L)}}$
算法性质	解析解

Algorithm 56: MLP-BN 反向传播 (MLP_BP_BN)

Input: 批量输入 $x^{(1:b)}$, 标签 $y^{(1:b)}$

Input: 各层参数 $W^{(1:L)}$, 损失函数 $\mathcal{L}(\hat{y}; y)$

Output: 梯度 $\frac{\partial L}{\partial W^{(1:L)}}$

$\hat{y}^{(1:b)}, a^{(:)}_{1:b}, z^{(:)}_{1:b}, \mu^{(:)}, \sigma^{(:)} \leftarrow \text{MLP\_BN}(x^{(1:b)}; W^{(:)})$

$G^T_{a^{(L)}} \leftarrow \frac{\partial \mathcal{L}}{\partial \hat{y}}(\hat{y}^{(i)}, y^{(i)})$

for $l \in L, L-1, ..., 1$ **do**

  for $i \in 1, ..., b$ **do**

    $\left.\frac{\partial a^{(l)}}{\partial \hat{z}^{(l)}}\right|_i \leftarrow \text{diag}\left(\mathbf{1}[z^{(l)}_i > 0]\right)$

    $\frac{\partial z^{(l)}_i}{\partial a^{(l-1)}_i} \leftarrow W^{(l)}$

    for $j \in 1, ..., b$ **do**

      $\frac{d\hat{z}_i}{dz_j} = \frac{\delta_{ij} - 1/b}{\sqrt{\sigma^2+\epsilon}} I - \frac{(z_i-\mu)(z_j-\mu)^T}{b\sqrt{\sigma^2+\epsilon}^3}$

      $G^T_{a^{(l-1)}_i} \leftarrow \sum_j G^T_{a^{(l)}_j} \frac{\partial a^{(l)}_j}{\partial \hat{z}^{(l)}_j} \frac{\partial \hat{z}^{(l)}_j}{\partial z^{(l)}_i} \frac{\partial z^{(l)}_i}{\partial a^{(l-1)}_i}$

  $\frac{\partial \mathcal{L}}{\partial W^{(l)}} \leftarrow \sum_i \sum_j G^T_{a^{(l)}_j} \frac{\partial a^{(l)}_j}{\partial \hat{z}^{(l)}_j} \frac{\partial \hat{z}^{(l)}_j}{\partial z^{(l)}_i} a^{(l-1)}_i$

对应的 python 代码如下所示

```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 梯度, (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
```

可以看到，引入 BN 后，MLP 的前向传播和反向传播均变得复杂，然而该方法可以有效避免梯度爆炸和梯度消失问题。事实上，在实际使用中，我们一般将 BN 和基于 batch 的优化方法结合使用，如 BSGD，RMSProp，动量法和 Adam 等，这里不再赘述。

19.3.3 Dropout 方法

Dropout、带 Dropout 的 MLP 推理、带 Dropout 的 BSGD 优化算法

19.3.4 数据正则化

label smooth
数据归一化

19.3.5 其他正则化技巧

参数初始化

Part VI

强化学习方法

20　强化学习基础

在上一部分中，我们介绍了基础的深度学习方法。深度学习是一套强大的工具，可以借助大规模数据，解决机器人中的许多智能问题。不过，机器人领域中的大部分问题涉及控制或规划，是一种和环境交互过程中的决策。深度学习本身并不包括和环境交互。

强化学习 (Reinforcement Learning, RL) 是机器学习中的另一个分支，它研究的正是如何从与环境交互的数据中学习，从而解决一定的控制问题。自 2012 年以来，各类深度强化学习 (Deep Reinforcement Learning, DRL) 方法也在包括机器人在内的许多领域取得了突破性进展。在本章中，我们介绍强化学习的基础概念；后续章节将介绍强化学习领域中的关键算法。

20.1　状态空间

状态是变化系统的切面。在前面的章节中，我们已经介绍了优化/系统控制/滤波等领域中的状态。在强化学习中，我们也用状态描述系统，并关注状态的转换方式。如果状态转换符合这样的规则：当前时刻的状态仅依赖前一时刻的状态，而不依赖此前时刻的状态，我们就称其具有**马尔可夫性**。这样的随机过程，称为马尔可夫过程。

奖励是对状态转换的“目的”的建模。在强化学习中，具体的任务目标被量化为或正或负的数字，称为奖励。具有奖励的马尔可夫过程，称为**马尔可夫回报过程** (MRP, Markov Reward Process)。

在我们聚焦于马尔可夫回报过程之前，我们需要建立**状态空间**的概念。状态空间是由所有马尔可夫过程中可能出现的状态构成的空间，记为 $\mathcal{S}$。状态空间可以是离散空间，也可以是连续空间；可以是有限空间 (离散情况)，可以是无限空间。状态空间越大，马尔科夫回报过程越复杂，对其进行处理也就越困难。

后续我们讨论的一部分问题，都是在**离散有限状态空间**进行的。这是因为，在离散有限状态空间下，状态总数是有限的；对于我们关心的 (状态空间上定义的) 函数，我们总可以通过列表的方法，显式地表达出每种状态取值的情况，并据此进行后续分析和决策等。这类算法也被称为表格型算法，在第 3 章讨论。

然而，现实中的许多情况下，我们面临的都是无限状态空间，尤其是**连续状态空间**。尽管可以将其离散化后，按照离散状态空间进行处理，但一方面，这会带来误差，另一方面，多维状态空间离散化会面临严重的**指数爆炸**问题。假设连续状态有 d 维，每维度离散化为 N 个区间，那么状态空间的总维数就达到了 N^d。随着 N 和 d 的增长，计算资源将很快不够用。这也被称为**维数爆炸问题**。此时，我们非常需要直接处理连续状态的方法，也就是第 4-5 章介绍的**策略梯度方法**。

20.2　MRP 与价值

在介绍这些方法之前，让我们先从**离散有限状态空间**开始，聚焦于一个具体的马尔可夫回报过程。

首先，我们需要一个初始状态。一般我们会有一个初始状态的分布 $P(S_0)$ ，我们从中采样，得到初始状态 S_0。接着，我们从状态转移分布 $P(S_{t+1}|S_t)$ 中采样，得到下一个状态。根据该状态的回报分布 $P_R(R_{t+1}|S_t)$，我们获得或多或少、或正或负的回报 R_{t+1} 。接着回到第二步，持续获取状态和奖励，直到某种预设的停止条件。停止条件可以是某种停止状态，或设定一定的步数限制，或设定最高奖励的限制，或其他。

在这里，我们用 $P_R(R_{t+1}|S_t)$ 表示回报的分布是依赖于状态的。不过在后面，我们更常用的是在某个状态下回报的期望，我们记为 $\bar{R}(s)$

$$\bar{R}(s)=\sum_{r\in\mathcal{R}} rP_R(r|s) \tag{VI.20.1}$$

注意，在上面描述的 MRP 中，全过程只有一个”角色”在行动，或称之为”上帝”，或称之为”**环境**”。只有这个”角色”的”move”决定了接下来的状态或奖励。因此，为了达到最大化奖励的目的，我们就需要研究其中的规律。

让我们先暂且放下复杂的无限状态空间，而仅仅聚焦于有限状态空间。奖励是随机变量，不好研究。一个朴素的想法是：研究空间中每种状态下得到奖励的期望。

计算奖励的期望固然是很不错的想法，在大数定律的帮助下，只要收集足够多的数据，便可以用平均值逼近期望。然而，这里潜藏着一个容易被忽视的事实——有时候，一个当前不产生很多奖励的状态，未必不是”好”状态；同样，一个当前不产生很多负奖励的状态，也未必不是”坏”状态。

生活中，有很多这样的例子。很多时候我们做一件事情，并不能得到立即直接的正向反馈，但我们还是选择做，因为我们知道持续做下去，很可能在未来得到不错的反馈或奖励。这种”延迟满足”的思维是人类具有高层次智力的一种体现，它让我们摆脱最基础欲求的控制，得以追求更高级的价值。

在 MRP 中，我们对状态的评价，同样不只是这个状态本身能得到的奖励，也包括这个状态倾向于转移到的状态能得到的奖励；或者说，在 MRP 的未来能得到的奖励。同时考虑当下和未来，这样的评价方式，我们称为”**价值**”。价值是状态当前和未来能得到的期望奖励的评价。

为了量化地计算价值，我们引入**折扣因子**，将未来可能得到的奖励”折算”到当下。折扣因子是一个取值在 $[0,1]$ 的量 γ，它表示未来每多一步，其奖励折算到当下的等效比例。将未来每一步的回报都折算到当下并相加，就是我们对当前状态的一种全局量化评价，我们称之为**折扣回报** (也称为累积回报)。其计算公式如下：

$$G_t=R_t+\gamma R_{t+1}+\gamma^2 R_{t+2}+\dots \tag{VI.20.2}$$

折扣回报是一个随机变量，它在我们走完未来所有时间步之前无法计算。对这个随机变量求期望，就是我们定义的 **MRP 状态价值 (函数)**。

$$V(s)=E[G_t|S_t=s] \tag{VI.20.3}$$

如前所述，状态价值不仅描述当前状态在当前能够得到奖励的多少，也反映了它容易到达的那些状态能够得到奖励的多少。状态之间通过一定的概率互相连接。我们称这种状态的转换为”状态转移”。我们可以用一个转移分布矩阵 P 描述每个状态转移到其他状态的概率分布，这个矩阵就称为**状态转移矩阵**。并且，由概率的性质可知，构成这个矩阵每个行向量的二范数都是 1。

这里我们还需强调一个重要的细节：**状态价值函数中的期望，是在对哪个随机变量求期望**？如果不理解这个问题，就无法顺畅理解后续所有关于状态价值函数展开的推导。准确的答案是：**对后续所有的状态和奖励序列求期望**。这是因为，状态价值函数是折扣回报 G_t 的期望，而 G_t 由当前步后所有的回报 R_{t+i} 组成，而回报又由未来的状态 R_{t+i} 决定。它们各自有各自的分布，即 $P(s'|s)$ 和 $P_R(r|s)$。因此，这个期望，实际上是未来所有的状态和奖励，在马尔可夫链意义上求的期望。

20.3 MRP 的价值估计

那么，在一定的环境中，我们如何求得状态价值呢？这就是 **MRP 的价值估计问题**。我们将这个问题以更加清晰的数学语言进行描述，总结在下表中。

问题	**MRP 价值估计 [依赖模型]**
问题简述	MRP 已知环境模型，求各状态的状态价值
已知	状态转移矩阵 $P(s'\|s)$ 以及回报函数 $\bar{R}(s)$
求	状态价值 $V(s), s \in \mathcal{S}$

这里面，" **模型**" 指的就是 MRP 的状态转移矩阵 P 以及回报函数 $\bar{R}(s)$。在后续的问题中，我们会看到，问题中**模型是否已知**是一个非常关键的因素。目前大多数主流强化学习算法都不需要/不依赖模型。

根据 MRP 状态价值定义式VI.20.3，我们可以将状态价值以一步状态转移的形式展开。

$$\begin{aligned}
V(s) &= E[G_t|S_t = s] \\
&= E[R_t + \gamma R_(t+1) + \gamma^2 R_(t+2) + \ldots |S_t = s] \\
&= E[R_t|S_t = s] + \gamma E[R_(t+1) + \gamma R_(t+2) + \gamma^2 R_(t+3) + \ldots |S_t = s] \\
&= \bar{R}(s) + \gamma E[G_{t+1}|S_t = s] \\
&= \bar{R}(s) + \gamma \mathbb{E}_{p(s'|s)}[E[G_{t+1}|S_{t+1} = s']] \\
&= \bar{R}(s) + \gamma \mathbb{E}_{p(s'|s)}[V(s')] \\
&= \bar{R}(s) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s) V(s')
\end{aligned}$$

即有

$$V(s) = \bar{R}(s) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s) V(s') \tag{VI.20.4}$$

该方程也被称为 **MRP 的贝尔曼方程**。

在模型 (状态转移矩阵/回报函数) 已知的条件下，通过 MRP 的贝尔曼方程求解状态价值函数比较简单。为方便表示，我们首先将状态价值写为向量形式 $\mathbf{V}$ 。V 的维数对应状态空间中所有状态的数量。其次将奖励期望函数也写为向量形式 $\mathbf{R}$ 。结合先前介绍的状态转移矩阵 $\mathbf{P}$ ，我们可以写出：

$$\mathbf{V} = \mathbf{R} + \gamma \mathbf{P} \mathbf{V} \tag{VI.20.5}$$

因此，模型已知条件下，经过简单推导，我们就能得到 $\mathbf{V}$ 的解析解

$$\mathbf{V} = (I - \gamma \mathbf{P})^{-1} \mathbf{R} \tag{VI.20.6}$$

解析解算法总结如下。

算法	**MRP-价值解析解**
问题类型	MRP 价值估计 [依赖模型]
已知	状态转移矩阵 $P(s'\|s)$ 以及回报函数 $\bar{R}(s)$
求	状态价值 $V(s), s \in \mathcal{S}$
算法性质	model-based, 表格型

Algorithm 57: MRP-价值解析解

Input: 状态转移矩阵 $P(s'|s)$ 以及回报函数 $\bar{R}(s)$

Output: 状态价值 $V(s), s \in \mathcal{S}$

$\mathbf{V} \leftarrow (I - \gamma \mathbf{P})^{-1}\mathbf{R}$

解析解固然非常方便，然而在很多情况下，解析解并不好取得。在上一节中我们讨论过维度爆炸问题，状态空间维度越大，**P** 阵的规模就越大，求解析解的逆矩阵就越复杂。在这种情况下，我们或许可以不追求一次求出所有状态价值，而是利用贝尔曼方程迭代求解。

动态规划 (Dynamic Processing, DP) 就是这样的一种方法。作为一大类算法构造思想的名称，动态规划是指：通过问题本身的特质，将一个大问题递归分解为规模更小的子问题，进而通过逐步求解简单的子问题而最终求出原问题的解。所有动态规划方法的根本原理都是贝尔曼最优性原理。

在本书第 III 部分，我们介绍了最优控制问题的动态规划方法。通过贝尔曼最优性原理，我们把最优控制求解问题分解成更小的子问题，从而对最优控制率进行迭代的求解。

动态规划思想的核心是子问题的分解。对于 MDP 价值估计，贝尔曼方程 (式VI.20.4) 实际上就给出了一种子问题的分解方式。然而，这种分解方式和其他动态规划求解的一大区别在于，分解出的子问题在规模上并没有变小，分解出的子问题也通过 P 矩阵的互联关系互相依赖。

因此，在 MDP 的价值估计问题中，我们也无法通过有限的分解解决问题。我们的解决方法是迭代估计——迭代价值函数表。先随机初始化，在迭代步数足够多后，价值函数估计会收敛于真值。这一收敛性有论文进行保证。

关于动态规划更广泛的应用场景，读者可参见本书 I.7 章的拓展内容。

算法	**MRP-DP 价值估计**
问题类型	MRP 价值估计 [依赖模型]
已知	状态转移矩阵 $P(s'|s)$ 以及回报函数 $\bar{R}(s)$
求	状态价值 $V(s), s \in \mathcal{S}$
算法性质	model-based, 表格型, 自举

Algorithm 58: MRP-DP 价值估计

Input: 状态转移矩阵 $P(s'|s)$ 以及回报函数 $\bar{R}(s)$

Output: 状态价值 $V(s), s \in \mathcal{S}$

$\hat{V}(s) \leftarrow random([V_{min}, V_{max}]), \forall s \in \mathcal{S}$

for $t \in 1, \ldots, T$ **do**

  for $s \in \mathcal{S}$ **do**

    $\hat{V}(s) \leftarrow \bar{R}(s) + \sum_{s' \in \mathcal{S}} p(s'|s)\hat{V}(s')$

实际使用动态规划估计 MRP 价值时，常用的外循环停止条件不是设定一个时间步 T，而是计算每两轮之间价值函数的差值 $||\mathbf{V}_{t+1} - \mathbf{V}_t||_2$，当小于给定的误差界 ϵ 时，认为迭代收敛。

动态规划方法避免了解析解法大矩阵求逆的弊端，通过迭代实现了价值估计。然而，这种方法还是需要依赖模型，也就是状态转移矩阵和回报函数。很多时候我们对环境没有什么信息，这种情况下，我们能否从多次实验中总结出价值函数表呢？

沿此思路，我们得到了蒙特卡洛价值估计方法。不过在此之前，我们需要明确目前我们在解决的问题：不依赖于模型的 MRP 价值估计问题。

问题	**MRP 价值估计 [无模型]**
问题简述	MRP 未知环境模型，求各状态的状态价值
已知	MRP 环境 $\mathcal{E}_R$
求	状态价值 $V(s), s \in \mathcal{S}$

在这里我们引入了一个概念：MRP 环境 $\mathcal{E}_R$。这个概念用来表示一个特定的马尔可夫奖励过程的环境，它可以按照一定的规律依次给出状态和奖励序列 $s_1, r_1, s_2, r_2, \ldots$，它是随机变量组成的序列。本书中，当给出这样的环境，意味着**我们可以从环境中获取这样的随机变量序列，但无法获取环境的模型**。

蒙特卡洛方法 (Monte-Carlo, MC) 来自一个朴素的想法：针对一个环境进行多次实验，根据大数定理，频率会逐渐逼近概率。在 MRP 的价值估计问题中，我们需要估计的是价值函数，也就是定义在状态空间上的期望。只要我们进行足够多的实验，随后根据价值函数的定义，计算折扣回报并计算统计期望，就可以逼近真正的价值函数了。

具体来说，假设每一轮 (每个 episode) 和环境的交互中，我们获取到这样一个有限的状态奖励序列 $SRS(t)$：

$$SRS(n) := s_1, r_1, s_2, r_2, \ldots, s_{L_e}, r_{L_e}$$

其中 L_e 表示一个 episode 的长度。对于有终止状态的 MRP，状态转移到终止状态的轮数就是这一个 episode 的 L_e。对于无终止状态，或正终止状态不好达到的 MRP，可以人为设定 L_e 来截断和环境的交互。

对于在第 n 个 episode 从环境采样得到的 $SRS(t)$，我们可以通过倒推的方式，得到其中第 t 步的折扣回报：

$$G_t(n) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{L_e - t} r_{L_e}, t = 1, 2, ..., L_e$$

$G_t(n)$ 对应于 s_t，它是 $V(s_t)$ 的一个采样。当我们进行实验的轮数足够多，采样到同一个状态的次数也足够多，我们就可以将所有特定状态的 $G_t(n)$ 取平均，从而作为 $V(s_t)$ 的估计。

在这样的思路下，我们将 MRP 问题的 MC 价值估计算法总结如下。

算法	**MRP-MC 价值估计**
问题类型	MRP 价值估计 [无模型]
已知	MRP 环境 $\mathcal{E}_R$
求	状态价值 $V(s), s \in \mathcal{S}$
算法性质	model-free, 表格型, MC

Algorithm 59: MRP-MC 价值估计

Input: MRP 环境 $\mathcal{E}_R$

Output: 状态价值 $V(s), s \in \mathcal{S}$

$\hat{V}(s) \leftarrow 0, \forall s \in \mathcal{S}$

$N(s) \leftarrow 0, \forall s \in \mathcal{S}$

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**

    $N(s_t) \leftarrow N(s_t) + 1$

    $\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \frac{1}{N(s_t)}(G_t - \hat{V}(s_t))$

上面的算法中，我们使用了一个结论，称为**序贯更新公式**。假设我们有依次得到的一列数 $x_i, i = 1, 2, 3, \ldots$，我们想在得到下一个数后立即更新这列数的平均值。如果按照平均值的定义式，我们就需要保留当前时刻 t 前所有的 x：

$$\bar{x}_t = \frac{x_1 + x_2 + x_3 + \cdots + x_t}{t}$$

然而，如果我们将 $\bar{x}_t$ 与 $\bar{x}_t$ 相减，再经过一些变换，我们就可以得到如下公式

$$\bar{x}_t = \bar{x}_{t-1} + \frac{1}{t}(x_t - \bar{x}_{t-1})$$

采用序贯更新方式，可以有效节省存储空间。与此同时，它还能揭示出新值对于平均值的” 修正” 作用会随着数据的积累而慢慢变少——符合我们大数定理中，” 均值收敛于期望” 的直觉。读者可自行验证：我们给出的序贯更新是否等价于记录所有状态对应的 G_t，再求平均。

20.4 MDP 与动作价值

在上一节中，我们讨论了 MRP 相关的价值以及价值估计问题。让我们再回顾引入 MRP 时的一段描述——(MRP 中) 全过程只有一个” 角色” 在行动，或称之为” 上帝”，或称之为” **环境**”；只有这个” 角色” 的”move” 决定了接下来的状态或奖励。

在现实中，我们往往更关心的是如何使用一个” 可操控” 角色主动做出选择，获取更大的收益；而非被动的等待环境不断给出状态转移和价值。也就是说，需要有一个属于我们自己的” 角色”，能够和环境进行有效的” 交互”。相比于” 上帝” 或” 环境”，我们的角色就是” 玩家”。

由此，我们引出**马尔可夫决策过程** (Markov Decision Process, MDP) 的定义：Agent 和环境序贯进行交互，环境向 Agent 反馈当前的状态以及奖励，Agent 向环境提交一个动作。环境反馈的当前状态和奖励仅依赖于上一个状态 (马尔可夫性)。

在这里我们引入了**动作** (Action) 的概念。我们将智能体在环境中做出的选择称为动作。所有可能的动作构成**动作空间**，记为 $\mathcal{A}$。和状态空间类似，动作空间也分为有限和无限、离散和连续。在第 3 章及以前，我们主要讨论有限离散动作空间。

除此之外，我们还引入了 Agent(有时译为" 代理" 或" 智能体") 的概念。这一概念用于指代能和环境交互、主动决策下一个动作的" 角色"。在强化学习中，我们通过不断采集数据，从数据中学习和环境相关的知识，从而让 Agent 能不断做出比先前更好的决策。这也是" 强化学习" 中" 强化" 概念的关键。

和 MRP 类似，MDP 环境也有其数学描述。在 MRP 中，环境是状态转移矩阵 $P(s'|s)$ 和回报函数 $\bar{R}(s)$。在 MDP 中，环境不仅基于上一个状态决定下一个状态，而且基于 Agent 给出的动作。因此，这两个函数都加入动作 a 作为条件。也就是说，MDP 环境的数学描述 (模型)，由**动作-状态转移矩阵** $P(s'|s,a)$ 和回报函数 $\bar{R}(s,a)$ 组成。类似的，如果不知道模型，但环境可以和智能体交互，在接下来的表述中，我们会将其称为 MDP 环境 $\mathcal{E}_D$。

让我们关注 MDP 的一个完整过程。首先，环境根据初始正态分布给出初始状态 s_0(有时我们也附带一个初始回报 r_0)。接下来，Agent 根据上一个状态 s_t 决定一个动作 a_{t+1}。环境根据动作 a_{t+1} 和 s_t 共同决定下一个状态 S_{t+1} 以及相应的奖励 r_{t+1}。这样，我们就构成了一个**状态奖励动作序列** $SRAS(t)$:

$$SRAS(t) := s_0, r_0, a_1, s_1, r_1, a_2, s_2, r_2, \ldots, a_{L_e}, s_{L_e}, r_{L_e}$$

和 MDP 的状态奖励序列类似，一个状态奖励动作序列也是一次 MDP 的采样。Agent 在和环境交互的过程中获取了奖励序列 $r_1, r_2, \ldots, r_{L_e}$，我们的目的自然是最大化 (当前和未来) 奖励的期望。我们仍然沿用 MDP 的折扣回报公式 (式VI.20.4)，将未来的奖励" 折算" 到现在。

折扣回报是一个随机变量，我们在 MDP 中使用" 价值" 来表示它的期望。在 MDP 中，我们也类似地定义**状态价值函数** $V_\pi(s)$。

$$V_\pi(s) = \mathbb{E}_{s' \sim P(s'|s'',a), r \sim p_r(r|s',a), a \sim \pi(a|s')}[G_t|S_t = s] \tag{VI.20.7}$$

这里，我们显式地 (冗长但不够严谨地) 写出了价值函数中的" 期望" 是在对谁求期望。可以看到，这里面事实上有三组随机变量的期望。[20] 一方面是未来的状态 s' 要依赖于其上一个状态 s'' 和动作 a，同时未来的奖励依赖于奖励的分布 $p_r(r|s',a)$；另一方面是，未来的动作 a 依赖于 **Agent 相关的、某种动作的分布** $\pi(a|s')$。[21]

由此，我们引入**策略**的概念。所谓策略就是 Agent 根据上一个状态决定这一步动作的方式。如果是下棋游戏就是下棋的策略，如果是电子游戏就是进行下一个操作的逻辑。如果在同样的一个状态下，Agent 可以选择不同的动作，我们就将策略写成动作的条件概率分布的形式 $\pi(a|s)$，这类策略被称为**随机性策略**。相应的，如果确定在某个状态下就选择唯一的一种状态，我们将其称为**确定性策略**。

由此我们可以看到，MDP 中状态价值函数和 MRP 中最大的区别就在于：**MDP 状态价值函数依赖于某个特定策略** π。因为 MDP 中存在两个" 角色"：环境和 Agent。未来由环境和 Agent 共同决定。环境保持不变，意味着未来状态和奖励的分布不变，但 Agent 的改变意味着未来动作分布的改变。因此，**对于 MDP，在没有策略的语境下讨论状态价值是没有意义的。**

在 MRP 中，我们定义状态价值函数是为了了解哪些状态对我们" 有利"。在 MDP 中，由于动作/策略的加入，我们不仅应该单独评估状态，而且应该评估**状态和动作的组合对我们的有利程度**。这样，我们引入 MDP 的 **(状态-) 动作价值函数** $Q_\pi(s)$

$$Q_\pi(s,a) = \mathbb{E}_{a' \sim \pi(a'|s')}[G_t|S_t = s, A_{t+1} = a] \tag{VI.20.8}$$

在此后的表述中，为简洁起见，我们一般将其称为" 动作价值函数" 或"Q 函数"。如果状态/动作空间都是离散有限的，我们将其称为"Q 表"。相应的，状态价值函数会称为"V 函数"。

[20] 为了区分状态价值函数的自变量和未来的状态，我们将未来状态写成 s' 和 s'' 的形式

[21] s',a 代表的不是未来的某一个时刻，而是未来所有时刻状态和动作的统称，不严谨表达

需要注意的是，和 MDP 的状态价值函数类似，**动作价值函数也依赖特定的策略** π。上面的表达式中，我们为简洁起见省略了未来的状态和奖励依赖的分布，但读者仍需注意它们也是求期望的一部分。显式写出的 $a' \sim \pi(a'|s')$ 也是为了强调 Q 函数一定依赖于策略 π。

在上面，我们分别定义了状态价值函数和动作价值函数。它们都是某种对未来随机变量的分布求取的期望。那么，这两种价值是否能互相转换呢？

根据 Q 值定义式，我们很容易将 Q 值转化为 V 值

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_{a\sim\pi(a|s'),o\sim p_o(o)}[G_t|S_t=s] \\ &= \sum_{a\in\mathcal{A}} \pi(a|s)\mathbb{E}_{o\sim p_o(o)}[G_t|S_t=s,A_t=a] \\ &= \sum_{a\in\mathcal{A}} \pi(a|s)Q_\pi(s,a) \end{aligned}$$

即

$$V_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)Q_\pi(s,a) \tag{VI.20.9}$$

注意，在上面的推导式中，我们使用 o 代表了：状态价值函数求期望时，除了 s 的下一个动作 a 之外的其他随机变量。写成这种形式仍然是要强调 V 和 Q 定义中，是在对很多的未来随机变量求期望。

类似地，我们也可以给出由状态价值 V 求动作价值 Q 的公式。

$$Q_\pi(s,a) = \bar{R}(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a)V_\pi(s') \tag{VI.20.10}$$

读者需要注意：在式VI.20.9中，仅需要知道当前的策略 π，就可以从 Q 求 V。而在式VI.20.10中，我们需要知道期望奖励 $\bar{R}(s,a)$ 和转移矩阵 $P(s'|s,a)$，也就是需要已知模型。这点区别十分关键。

如果我们继续将 V 求 Q 的公式代入 Q 求 V 的公式，那么我们就将得到状态价值和下一轮状态价值之间的关系式。

$$V_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)(\bar{R}(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a)V_\pi(s')) \tag{VI.20.11}$$

也可以通过将 Q 求 V 的公式代入 V 求 Q 的公式，得到动作价值和下一轮动作价值之间的关系式。

$$Q_\pi(s,a) = \bar{R}(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a)(\sum_{a'\in\mathcal{A}} \pi(a'|s')Q_\pi(s',a')) \tag{VI.20.12}$$

式VI.20.12和式VI.20.11也称为 MDP 的贝尔曼期望方程。它揭示了动作价值函数和状态价值函数的递归性质。需要注意的是，使用这两个公式时，除了需要策略外，也需要已知模型。

20.5 MDP 预测问题

在 MDP 中，状态价值函数 (以及动作价值函数) 依赖于策略。因此，一个很自然的问题就是：如何在已知策略的情况下，求解状态价值函数？

这个问题的提法，和此前讨论的" 如何在 MRP 中求解状态价值函数" 很相似。事实确实如此，不过还请读者注意这两个问题的关键区别：MRP 中不存在" 玩家"，一种环境就对应一个价值函数；MDP 中有了" 玩家"，有了策略 π，价值函数和策略是绑定的，策略改变但模型不变，价值函数就会变化。

因此，在 MDP 中，求取价值函数的问题称为**策略评估**，它评价的是在同样的环境下，一种策略是好还是坏。这里的" 好/坏" 我们稍后就将用一种偏序定义加以规范化。有时我们也将其称为 **MDP 预测问题**，以和后面的控制问题区分。

问题	MDP 预测问题 [依赖模型]
问题简述	MDP 已知环境模型和策略，求各状态的状态价值
已知	状态转移矩阵 $P(s'\|s,a)$，回报函数 $\bar{R}(s,a)$，策略 $\pi(s\|a)$
求	状态价值 $V_\pi(s), s \in \mathcal{S}$

既然问题和 MRP 价值估计类似，我们就可以把解决 MRP 的思路迁移到 MDP 预测问题。

考虑迁移动态规划算法 (算法58) 的思路用到预测问题中。算法58基于公式VI.20.4进行迭代，在 MDP 中应改为基于公式VI.20.11进行迭代。

算法	MDP-DP 策略评估
问题类型	MDP 预测问题 [依赖模型]
已知	状态转移矩阵 $P(s'\|s,a)$，回报函数 $\bar{R}(s,a)$，策略 $\pi(s\|a)$
求	状态价值 $V_\pi(s), s \in \mathcal{S}$
算法性质	model-based, 表格型, 自举

Algorithm 60: MDP-DP 策略评估

Input: 状态转移矩阵 $P(s'|s,a)$，回报函数 $\bar{R}(s,a)$，策略 $\pi(s|a)$

Parameter: 折算因子 γ

Output: 状态价值 $V_\pi(s), s \in \mathcal{S}$

$\hat{V}(s) \leftarrow random([V_{min}, V_{max}]), \forall s \in \mathcal{S}$

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**

      $\hat{V}_{next}(s,a) \leftarrow \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s')$

      $\hat{V}_\pi(s) \leftarrow \hat{V}_\pi(s) + \pi(a|s)(\bar{R}(s,a) + \hat{V}_{next}(s,a))$

对应的 python 代码如下所示

```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):
```

```python
        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
```

可以看到，DP 方法是完全基于模型的方法，没有模型就无法使用。此外，在每一步更新中，都需要遍历 (a, s') 的组合，也就是 $A \times S$ 空间。在状态空间和动作空间较大的时候，这样的操作无疑比较低效。

针对无模型的情况，我们也可以提出 MDP 预测问题.

问题	**MDP 预测问题 [无模型]**
问题简述	MDP 未知环境模型，求某策略下各状态的状态价值
已知	MDP 环境 $\mathcal{E}_D$，策略 $\pi(s\|a)$
求	状态价值 $V_\pi(s), s \in \mathcal{S}$

类似地，我们也可以沿用 MRP 无模型价值估计中的蒙特卡洛思路求解。

算法	**MDP-MC 策略评估**
问题类型	MDP 预测问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$，策略 $\pi(s\|a)$
求	状态价值 $V_\pi(s), s \in \mathcal{S}$
算法性质	model-free, 表格型, MC

我们仍然使用增量学习的思路。和 MRP 的区别在于，环境不是一股脑的给我们产生一整个 episode 的数据，而是我们要实际根据策略 π 和环境进行交互。交互后，根据增量学习公式更新价值函数表格。

增量学习还有一个好处。虽然 $\hat{V}_\pi(s_t) \leftarrow \hat{V}_\pi(s_t) + \frac{1}{N(s_t)}(G_t - \hat{V}_\pi(s_t))$ 满足均值的更新，但实际上增量前面的系数并不必须是采样次数的倒数。只需要是在一定条件下逐渐变小的一个参数，就可以让值函数收敛。因此，我们有时候也把增量学习写成这样的形式：

$$\hat{V}_\pi(s_t) \leftarrow \hat{V}_\pi(s_t) + \alpha(G_t - \hat{V}_\pi(s_t)) \tag{VI.20.13}$$

在上面的算法框中，实际展示的是 $\alpha = \frac{1}{N(s_t)}$ 的情况。但实际应用时，将其设为一个超参数 α，可以让我们根据实际学习情况进行调参，从而找到最好的学习速度。这类似于深度学习中对学习率的调参。在后续的增量学习中，我们都会统一使用 α 表示学习率参数。

蒙特卡洛方法通过采样弥补了未知模型的不足。不过，在 MC 策略评估算法中，我们必须采样一个 episode 后才能更新价值表格。然而，式VI.20.9、VI.20.10已经揭示：价值函数间存在相互依赖关系，可以迭代更新。先前，在已知模型的情况下，我们利用这种依赖关系设计了动态规划方法，来自举估计价值表格。那么，在没有模型的情况下，我们能否采用类似的思路，在每步采样后迭代地更新价值表格呢？

Algorithm 61: MDP-MC 策略评估

Input: MDP 环境 $\mathcal{E}_D$, 策略 $\pi(s|a)$

Parameter: 折算因子 γ

Output: 状态价值 $V_\pi(s), s \in \mathcal{S}$

$\hat{V}_\pi(s), N(s) \leftarrow 0, 0, \forall s \in \mathcal{S}$

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**

    $N(s_t) \leftarrow N(s_t) + 1$

    $\hat{V}_\pi(s_t) \leftarrow \hat{V}_\pi(s_t) + (G_t - \hat{V}_\pi(s_t))/N(s_t)$

对应的 python 代码如下所示

```python
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):
```

```python
    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
```

事实上，这样的一类方法称之为**时序差分** (Time Difference, TD) 方法。具体来说，蒙特卡洛方法之所以要等待一个 episode 完成，是因为价值学习需要依赖累计回报 G_t，而 G_t 的推算又需要逆时间方向，从后向前计算。那么，能否找到一个更好的方法估计 G_t 呢？

我们还是从 G_t 的定义式 (式VI.20.2) 入手。注意到 G_t 可以一步展开成递归形式

$$G_t = r_t + \gamma G_{t+1}$$

虽然在不获取整个 episode 的情况下，我们既无法知道 G_t 也无法知道 G_{t+1}，但我们由价值函数定义式 (VI.20.7) 知道，$V(S_{t+1})$ 是 G_{t+1} 的期望。也就是说，如果知道 r_t 和 $V(S_{t+1})$，我们就可以得到 G_t 的估计值。

$$G_t \approx r_t + \gamma V(s_{t+1})$$

这里，我们将这个估计值称为**时序差分目标**，记作 T_t，也就是策略评估算法" 学习" 的目标。对于最简单的情况，我们将 G_t 展开一个时间步，那么这个目标仅需要我们知道下一时刻的状态就可以计算，这比需要整个 episode 才能计算的 G_t 容易许多。

于是我们有增量学习的通用形式

$$\hat{V}_\pi(s_t) \leftarrow \hat{V}_\pi(s_t) + \alpha(T_t - \hat{V}_\pi(s_t)) \tag{VI.20.14}$$

以及最简单的一步估计方法

$$T_t = r_t + \gamma V(s_{t+1}) \tag{VI.20.15}$$

整理一下，我们就得到了被称为 **0 步时序差分** (TD(0)) 的时序差分策略评估算法。其中 TD 代表 Time-Difference。

算法	MDP-TD(0) 策略评估
问题类型	MDP 预测问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$，策略 $\pi(s\|a)$
求	状态价值 $V_\pi(s), s \in \mathcal{S}$
算法性质	model-free, 表格型, TD

Algorithm 62: MDP-TD(0) 策略评估

Input: MDP 环境 $\mathcal{E}_D$, 策略 $\pi(s|a)$

Parameter: 折算因子 γ, 更新参数 α

Output: 状态价值 $V_\pi(s), s \in \mathcal{S}$

$\hat{V}_\pi(s) \leftarrow 0, \forall s \in \mathcal{S}$

$N(s) \leftarrow 0, \forall s \in \mathcal{S}$

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**

    $a_t \sim \pi(a|s = s_{t-1})$

    $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$

    $T_{t-1} = r_{t-1} + \gamma \hat{V}_\pi(s_t)$

    $\hat{V}_\pi(s_{t-1}) \leftarrow \hat{V}_\pi(s_{t-1}) + \alpha(T_{t-1} - \hat{V}_\pi(s_{t-1}))$

对应的 python 代码如下所示

```python
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
```

TD(0) 中的时序差分目标只包含一个真实采样的奖励值。事实上，我们可以进一步融合 MC 方法和 TD(0)，让时序差分目标包含此后多步的采样值。这就是 TD(n) 方法。

相比 TD(0)，TD(n) 只需要改变时序差分目标：

$$T_t = r_t + \gamma r_{t+1} + ... + \gamma^n r_{t+n} + V(s_{t+n+1}) \tag{VI.20.16}$$

我们可以总结 $TD(n)$ 策略评估算法如下

算法	MDP-TD(n) 策略评估
问题类型	MDP 预测问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$，策略 $\pi(s\|a)$
求	状态价值 $V_\pi(s), s \in \mathcal{S}$
算法性质	model-free, 表格型, TD

Algorithm 63: MDP-TD(n) 策略评估

Input: MDP 环境 $\mathcal{E}_D$, 策略 $\pi(s|a)$

Parameter: 折算因子 γ, 迭代轮数 n, 更新参数 α

Output: 状态价值 $V_\pi(s), s \in \mathcal{S}$

$\hat{V}_\pi(s) \leftarrow 0, \forall s \in \mathcal{S}$

$N(s) \leftarrow 0, \forall s \in \mathcal{S}$

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**

      $T_{t-n-1} = r_{t-n-1} + \gamma r_{t-n} + ... + \gamma^n r_{t-1} + \gamma^{n+1} V(s_t)$

      $\hat{V}_\pi(s_{t-n-1}) \leftarrow \hat{V}_\pi(s_{t-n-1}) + \alpha(T_{t-n-1} - \hat{V}_\pi(s_{t-n-1}))$

对应的 python 代码如下所示

```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 控制问题

到目前为止，我们已经介绍了很多价值估计相关的方法。MRP 中，在一定策略下的价值估计固然很重要，但我们更关心的，其实是如何寻找更好的策略。

这里面就涉及我们之前没有深入的问题了：如何定义一个策略比另一个策略” 好”？什么才是一个好策略？借助状态价值函数，我们可以非常直接地给出定义：对每一个状态，如果状态价值函数都更好，就说这个策略更好。我们记为 $\pi \geq \pi'$

$$\pi \geq \pi' := V_\pi(s) \geq V_{\pi'}(s) \tag{VI.20.17}$$

既然有” 好”，我们自然要考虑” 最好”。在这样的定义下，我们可以给出**最优策略**的定义

$$\pi^* := \arg\max_\pi V_\pi(s), \forall s \in \mathcal{S} \tag{VI.20.18}$$

或者

$$V_{\pi^*}(s) \geq V_\pi(s), \forall s \in \mathcal{S}, \forall \pi$$

相应的价值函数，也称为**最优价值函数** $V^*(s)$

$$V^*(s) := \max_\pi V_\pi(s) = V_{\pi^*}(s) \tag{VI.20.19}$$

对于最优策略，有几点需要注意。首先，最优策略针对的是特定的环境。如果环境改变（状态转移矩阵、期望价值函数改变），最优策略就会改变。其次，最优策略是一个理论上的策略。对于模型未知的环境，只能通过各类方法逼近；对于模型已知的环境，计算出其真值往往十分困难。第三，最优策略不一定是唯一的，如果环境存在某种对称性，最优策略可能就有多个甚至无穷多个。

此外，式VI.20.18还隐藏着一个容易忽视的问题：**假设我们已知最优价值函数，我们应该如何求最优策略？**

最优策略是一个概率分布 $\pi(a|s)$，不过也可以表示确定性策略（即将所有概率值赋予某个 a 的取值）。想要取得最优策略，我们只要知道在某个状态下最优的动作 $a*$(或等价最优的一些动作)，再将概率全部赋予它就好了。问题就在于：状态价值函数 $V^*(s)$ 中，根本就没涉及动作。

动作-状态价值函数倒是满足我们的需求。我们如果有最优的 $V^*(s)$ 对应的最优的 $Q^*(s,a)$，我们就可以比较同一个 s 下不同 a 的 Q 值，取最高的一个或多个作为策略就可以。问题是，**如何用 V 函数求对应的 Q 函数呢？**

回顾第20.4节中 V 函数和 Q 函数关系的公式 (式VI.20.9和VI.20.10)。值得注意的是，这些公式也可以写成最优策略下的形式。

$$V^*(s) = \sum_{a \in \mathcal{A}} \pi^*(a|s) Q^*(s,a) \tag{VI.20.20}$$

$$Q^*(s,a) = \bar{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*(s') \tag{VI.20.21}$$

$$V^*(s) = \sum_{a \in \mathcal{A}} \pi^*(a|s) (\bar{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*(s')) \tag{VI.20.22}$$

$$Q^*(s,a) = \bar{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) (\sum_{a' \in \mathcal{A}} \pi^*(a'|s') Q^*(s',a')) \tag{VI.20.23}$$

这些公式中，VI.20.22和VI.20.23也被称为**贝尔曼最优公式**。

然而，我们所需要的是 V 函数到 Q 函数的公式，也就是式VI.20.21，而该式显然依赖于环境模型（状态转移矩阵）。也就是说，**在缺乏模型的情况下，我们无法从最优状态价值 $V^*(s)$ 求出最优策略 $\pi^*(a|s)$**。

这一关键的结论揭示了 **Q 函数和 V 函数的本质不同**：虽然同为价值函数，但 Q 函数能给我们提供如何进行动作选择，也就是策略设计的关键信息，而 V 函数只能告诉我们如何评价策略和状态。也就是说，尽管两个函数有转化关系，但在实际条件下，**如果我们想追求最优策略，求出 Q 函数比求出 V 函数更为关键**。

由此，我们可以正式地引入 **MDP 控制问题**：在给定环境的 MDP 中，在已知模型/未知模型的情况下，求出最优的策略 $\pi^*(a|s)$。

在进入未知模型的情况之前，让我们先从有模型的简单情形开始。

问题	MDP 控制问题 [依赖模型]
问题简述	MDP 已知环境模型，求最优策略
已知	状态转移矩阵 $P(s'\|s,a)$，回报函数 $\bar{R}(s,a)$
求	最优策略 $\pi^*(a\|s)$

一种朴素的想法是：利用我们之前策略评估的方法。策略评估可以告诉我们一个策略对应的价值函数。在这个价值函数的基础上，如果我们能有一种把策略变得更好的方法，我们就可以获得一个新的策略。随后，我们又可以通过策略评估得到更好的策略函数。如此循环，最终策略会收敛到最优策略。

这样的思路已被证明是可行的，我们称之为**策略迭代**，它由迭代循环的两步组成：**策略评估**和**策略改进**。在上节中，我们已经介绍不少策略评估的方法。那么，如何进行策略改进呢？

显然，我们需要用到定义”更优策略”的公式VI.20.17。根据该公式，我们在策略改进时，只需保证第 k 轮迭代后的策略 π_k，其对应的价值函数 $V_k(s)$ 比第 $k-1$ 轮迭代策略 π_{k-1} 对应的 $V_{k-1}(s)$ 更好 (对于状态空间中的每个状态)，就可以了。如何做到这一点呢？

事实上，我们只需要运用一种称为**贪心**的思路就可以了。

$$\pi_{k+1}(a|s) = \begin{cases} 1, a = \arg\max_a Q_k(s,a) \\ 0, others \end{cases} \tag{VI.20.24}$$

贪心，即选取当前情景下的最优 (局部最优)。尽管在迭代收敛前，每一步的价值函数 $V_k(s)$ 不是最优价值函数 $V^*(s)$，但我们可以把每一轮迭代后的策略 $\pi_k(a|s)$ 设计为：对应于局部 $V_k(s)$ 的最优策略。也就是说，我们找出让 $Q_k(s,a)$ 取最大的 a，然后作为当前的固定策略 $\pi_k(a|s)$。

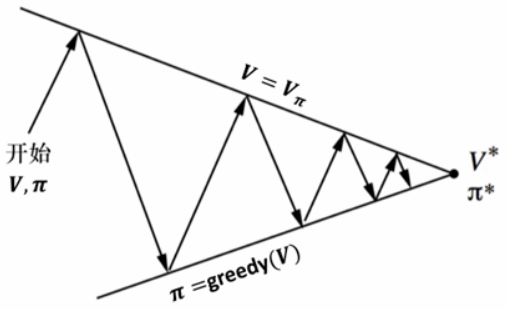

图 VI.20.1: 策略迭代过程

注意到，这个贪心的过程我们用到了 Q 值。前面已经讨论过，从 V 求 Q 需要已知模型。也就是说，该方法解决的是依赖模型的 MDP 控制问题。这样，基于上节介绍的 DP 策略评估方法，我们整理求解 MDP 控制问题的 **DP 策略迭代**方法。

算法	MDP-DP 策略迭代 (贪心)
问题类型	MDP 控制问题 [依赖模型]
已知	状态转移矩阵 $P(s'\|s,a)$，回报函数 $\bar{R}(s,a)$
求	最优策略 $\pi^*(a\|s)$
算法性质	model-based, 表格型, 自举

Algorithm 64: MDP-DP 策略迭代 (贪心)

Input: 状态转移矩阵 $P(s'|s,a)$，回报函数 $\bar{R}(s,a)$

Output: 最优策略 $\pi^*(a|s)$

$\pi_0(a|s) \leftarrow random(A \times S)$

for $k \in 1, \ldots, N_K$ **do**

  $V_k(s) \leftarrow \text{MDP\_DP\_policy\_est}(\pi_k)$

  for $s \in \mathcal{S}$ **do**

    for $a \in \mathcal{A}$ **do**

      $Q_k(s,a) \leftarrow \bar{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s')$

    $\pi_{k+1}(a|s) \leftarrow \begin{cases} 1, a = \arg\max_a Q_k(s,a) \\ 0, others \end{cases}$

对应的 python 代码如下所示

```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
```

在基于 DP 的策略迭代中，对于每个大循环，我们都要交替进行策略评估和策略改进。策略评估和策略改进都要遍历每一组可能的状态-动作。显然，这种算法比较繁琐，能不能让其稍微简洁一些呢？

我们注意到，大循环实际上是 $V_k \to Q_k \to \pi_{k+1} \to V_{k+1}$ 的循环，在策略评估和策略改进中分别使用了贝尔曼公式 (式VI.20.12和式VI.20.11)。在大循环中，最优策略只是一个中间变量。然而，根据式VI.20.9，我们其实可以直接从 Q 函数求 V 函数。我们事实上可以用很简单的贪心策略完成这个过程

$$V_{k+1}(s) = \arg\max_{a} Q_{\pi}(s,a) \tag{VI.20.25}$$

这样一来，我们就可以在不使用贝尔曼公式的情况下，直接进行 $V_k \to Q_k \to \pi_{k+1} \to V_{k+1}$ 的迭代。只要 $V_k(s)$ 可以收敛到 $V^*(s)$，就不影响最终的策略是最优策略 $\pi^*(s)$。这种不求取中间最优策略，直接利用 Q 和 V 表迭代求取最优价值的算法，我们称为**价值迭代**。接下来，我们总结确定性价值迭代算法。

算法	**MDP-确定性价值迭代**
问题类型	MDP 控制问题 [依赖模型]
已知	状态转移矩阵 $P(s'\|s,a)$，回报函数 $\bar{R}(s,a)$
求	最优策略 $\pi^*(a\|s)$
算法性质	model-based, 表格型, 自举

Algorithm 65: MDP-确定性价值迭代

Input: 状态转移矩阵 $P(s'|s,a)$，回报函数 $\bar{R}(s,a)$

Output: 最优策略 $\pi^*(a|s)$

$V_0(s) \leftarrow random([V_{min}, V_{max}]), \forall s \in \mathcal{S}$

for $k \in 1,\ldots,N_K$ **do**

  for $s \in \mathcal{S}$ **do**

    for $a \in \mathcal{A}$ **do**

      $Q_k(s,a) = \bar{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s')$

    $V_{k+1}(s) = \arg\max_a Q_{\pi}(s,a)$

$$\pi^*(a|s) \leftarrow \begin{cases} 1, a = \arg\max\limits_{a} Q_K(s,a) \\ 0, others \end{cases}$$

对应的 python 代码如下所示

```python
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)
```

```python
        V = np.max(Q_mat, axis=1)
    return pi_func_greedy
```

策略迭代和价值迭代是在已知模型情况下，求解 MDP 控制问题的两大思路。这两大思路的核心区别就是：是否产生完整的中间策略 π_k。策略迭代每一个外循环都产生一个策略，在外循环中对其不断提升。价值迭代只希望求出最好的价值函数，最后才会给出策略。

20.7 强化学习问题

回顾我们到目前为止的讨论，我们首先从带奖励的马尔可夫过程开始，讨论什么是 MRP 以及如何估计 MRP 的价值。其中我们引入了模型的概念，强调了在有模型和无模型的情况下求解问题的思路完全不同。其次我们讨论了带有主动交互过程 (策略) 的 MDP，介绍了什么是动作，区分了状态价值和动作价值，强调了 MDP 中价值依赖于策略。我们讨论了预测问题和控制问题，介绍了 MDP 中如何对策略进行评估，以及如何在有模型的情况下计算最优的策略。

毫无疑问，我们目前为止的旅程是充实的，然而我们还未涉足一个真实世界的 MDP 面临的问题：**在没有模型的情况下，我们应该如何求解 MDP 的控制问题？**可以说，我们经过了一章的漫长旅程，实际都是在为这个问题做铺垫。

问题	**MDP 控制问题 [无模型]**
问题简述	MDP 未知环境模型，求最优策略
已知	MDP 环境 $\mathcal{E}_D$
求	最优策略 $\pi^*(a\|s)$

没有模型，价值函数要靠实验来近似。没有模型，状态价值 V 无法计算动作价值 Q。我们面临的挑战增大了，但不必担心，本章中的一些算法中，解决之道已经初见端倪。

所谓强化学习，就是在与环境的交互中不断强化 Agent 自身的策略，根据已有的数据不断学习，提升 Agent 解决问题的能力。无模型的 MDP 控制问题，就是一类典型的 MDP 问题。

然而，强化学习不止于此。在 MDP 控制问题中，我们假设环境给我们的状态 s 包含的信息足以让智能体做出下一步决策。但是，在实际问题中，我们能观察到的环境信息可能是不足的。这就是**部分可观测性**。在部分可观测性的条件下的 MDP，我们称为**部分可观测马尔可夫决策过程** (Partially Observable Markov Decision Process)。

如果熟悉控制理论中的状态空间方法，我们就会知道，可观测性对于控制器的设计具有多么大的影响。

在部分可观测性的条件下，求解无模型 MDP 控制问题，这就是在马尔可夫框架下，形式上最困难的强化学习问题。

问题	**POMDP 控制问题 [无模型]**
问题简述	POMDP，部分可观测，未知环境模型，求最优策略
已知	POMDP 环境 $\mathcal{E}_{PO}$
求	最优策略 $\pi^*(a\|s)$

在本书后续章节介绍的深度强化学习方法，将成为 POMDP 问题重要的解决思路。

20.8 强化学习观点

Value-based/policy-based/actor-critic

世界模型、Model-based/model-free

K 臂赌博机问题、探索-利用困境、采样、on-policy/off-policy

21 表格型方法

在上一章的结尾，我们介绍了两类重要的强化学习问题：无模型下的 MDP 控制问题以及 POMDP 控制问题。在此之前，我们还针对这些问题的简化版本介绍了一些方法。这些方法的核心都是估计价值函数 V 或 Q。由于我们将讨论限定在有限离散状态空间/动作空间中，V 和 Q 函数实际上可以用表格的方法将所有取值一一列举。拥有这类特征的方法，称为**表格型方法**。

21.1 蒙特卡洛策略迭代

对于无模型的 MDP 控制问题，我们依然使用表格型方法的思路。相比于第一章结尾介绍的有模型控制问题，无模型的区别就是无法获取 $P(s'|s,a)$ 以及 $\bar{R}(s,a)$。

在此之前，我们也处理过类似的问题，使用的是蒙特卡洛采样的方法 (算法61)。我们可以将这一思路融合到策略迭代方法 (算法64)，用蒙特卡洛采样估计的价值代替模型计算出的价值。

算法	**MDP-MC 策略迭代 (ϵ-贪心)**
问题类型	MDP 控制问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$
求	最优策略 $\pi^*(a\|s)$
算法性质	value-based, on-policy?, 表格型

不过，这里会出现我们上一章讨论过的**探索-利用问题**。蒙特卡洛是一种采样方法，需要基于一定的策略进行采样。采样的结果用来更新 Q 表，变成新的策略。如果前期没有充分探索整个环境，可能就会陷入局部最优。

因此，我们会采用两个技巧。第一个叫做**探索性开始**，第二个叫做 **ϵ-贪心**。

探索性开始是指，在策略迭代的初期阶段，我们不使用 Q 表总结的策略，而是尽可能采用随机策略，对不同的 (a,s) 对进行采样。

ϵ-贪心是指，为了实现探索性开始的效果，我们在学习过程中通过参数 ϵ 描述随机策略的比例。开始时 ϵ 接近 1，越往后随机策略的占比越小。也就是说，我们采用“先探索再利用”的思路。

为了达到这个目的，我们可以设计一个依赖于 episode 采样次数 k 的 ϵ 更新策略，例如

$$\epsilon(k) = \frac{1}{k} \tag{VI.21.1}$$

因此 ϵ-贪心策略可以记作

$$\pi_\epsilon(a;s,Q_k) = \begin{cases} 1-\epsilon+\dfrac{\epsilon}{||\mathcal{S}||}, & a = \arg\max\limits_a Q_k(s,a) \\ \dfrac{\epsilon}{||\mathcal{S}||}, & others \end{cases} \tag{VI.21.2}$$

我们总结 MC 策略迭代 (ϵ-贪心) 算法如下。

Algorithm 66: MDP-MC 策略迭代 (ϵ-贪心)

Input: MDP 环境 $\mathcal{E}_D$

Output: 最优策略 $\pi^*(a|s)$

$Q_0(s,a) \leftarrow random(A \times S)$

$N(s,a) \leftarrow 0, \forall (a,s) \in \mathcal{S} \times \mathcal{A}$

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**

    $a_t \sim \pi_\epsilon(a; s, Q_k)$

    $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$

  $G_{L_e+1} \leftarrow 0$

  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**

    $N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$

    $Q_k(s_t, a_t) \leftarrow Q_k(s_t, a_t) + \frac{1}{N(s_t, a_t)}(G_t - Q_k(s_t, a_t))$

$\pi^*(a|s) \leftarrow \pi_\epsilon(s, a; Q_K)$

对应的 python 代码如下所示

```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
```

和蒙特卡洛策略评估 (算法61) 相比，蒙特卡洛价值迭代中，我们学习的是 Q 表而不是 V 表。这是因为 Q 表才能帮我们选择动作，从而直接生成策略。

21.2 SARSA 方法

在上一章中，我们介绍过三种针对价值评估问题的大思路，分别是动态规划 DP、蒙特卡洛 MC 和时序差分 TD。相比于动态规划的依赖模型、蒙特卡洛的更新周期缓慢，时序差分 TD 很好地结合了二者的优点，兼顾了采样和学习。本节，我们尝试将 TD 方法引入控制问题。

我们首先尝试 TD(0) 版本的策略迭代。主要思路是将算法62融入算法66的框架中。

算法	**MDP-SARSA 策略迭代**
问题类型	MDP 控制问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$
求	最优策略 $\pi^*(a\|s)$
算法性质	value-based, on-policy, 表格型

这里仍然需要注意的就是，根据目标的不同，我们学习的对象从 V 表换成了 Q 表。因此，时序差分目标也从未来的 V 值，变成了未来的 Q 值。因此，我们除了需要知道当前步的 (状态 s、动作 a、奖励 r) 外，还需要知道下一时间步的 (状态 s、动作 a)。把这五个字母连起来就是 **SARSA**，也是这个算法的另一个名字。

Algorithm 67: MDP-SARSA 策略迭代

Input: MDP 环境 $\mathcal{E}_D$

Output: 最优策略 $\pi^*(a|s)$

$Q_0(s,a) \leftarrow random(A \times S)$

$N(s,a) \leftarrow 0, \forall (a,s) \in \mathcal{S} \times \mathcal{A}$

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**

    $a_t \sim \pi_\epsilon(a; s, Q_k)$

    $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$

    $T_{t-1} = r_{t-1} + \gamma Q_k(s_t, a_t)$

    $Q_k(s_{t-1}, a_{t-1}) \leftarrow Q_k(s_{t-1}, a_{t-1}) + \alpha(T_{t-1} - Q_k(s_{t-1}, a_{t-1}))$

$\pi^*(a|s) \leftarrow \pi_\epsilon(s, a; Q_K)$

对应的 python 代码如下所示

```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
```

类似地，我们可以写出采样 n 步的 SARSA 方法，或者 **n 步 SARSA** 方法。

算法	MDP-n 步 SARSA 策略迭代
问题类型	MDP 控制问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$
求	最优策略 $\pi^*(a\|s)$
算法性质	value-based, on-policy, 表格型

Algorithm 68: MDP-n 步 SARSA 策略迭代

Input: MDP 环境 $\mathcal{E}_D$

Output: 最优策略 $\pi^*(a|s)$

$Q_0(s,a) \leftarrow random(A \times S)$

$N(s,a) \leftarrow 0, \forall (a,s) \in \mathcal{S} \times \mathcal{A}$

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**

    $a_t \sim \pi_\epsilon(a; s, Q_k)$

    $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$

    $T_{t-n} = r_{t-1} + \sum_{i=1}^{n} \gamma^i Q_k(s_{t-n+i}, a_{t-n+i})$

    $Q_k(s_{t-n}, a_{t-n}) \leftarrow Q_k(s_{t-n}, a_{t-n}) + \alpha(T_{t-1} - Q_k(s_{t-n}, a_{t-n}))$

$\pi^*(a|s) \leftarrow \pi_\epsilon(s, a; Q_K)$

对应的 python 代码如下所示

```
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
```

n 步 SARSA 方法需要选择合适的步数 n 作为时序差分目标 T_{t-n}。如果 n 过大或过小都会造成采样不充分的问题，造成算法难以稳定。那么，我们是否可以同时使用不同的 n 对应的 Q 值估计，作为时序差分目标呢？

SARSA(λ) 就是这样的一种方法。它同时计算多个 n 值对应的 Q 值估计，并通过 $[0,1]$ 间的参数 λ 控制不同估计的加权系数。具体来说，SARSA(λ) 的时序差分目标是

$$T_{t+N}^{\lambda} = (1-\lambda)\sum_{n=1}^{N}\lambda^{n-1}Q(s_{t+n}, a_{t+n}) \tag{VI.21.3}$$

由此，我们可以总结 SARSA(λ) 算法如下 (TODO)。

21.3 Q 学习方法

SARSA 方法的一个主要问题是偏向保守。即使使用了 ϵ-贪心策略，在后期 Agent 也会偏向于使用学习到的 Q 表做决策，从而容易陷入局部最优。

造成这个问题的根本原因，在于 SARSA 的学习策略。在 SARSA 中，时序差分目标中的 Q 值是未来采样到的 (s,a) 对应的 Q 值。然而，未来的动作 a 未必是对应情况下最优的动作。这一差异可能会使 SARSA 错过潜在的“高价值路径”，导致其最终的学习不够好。

Q 学习就是这样的一个技巧。它仍需要采样下一时刻的状态，但学习的目标是该状态对应所有 Q 值中最大的那个。Q 学习和 SARSA 的区别在于：我们只需把时序差分目标改成

$$T_{t-1} = r_{t-1} + \gamma \max_{a} Q_k(s_t, a) \tag{VI.21.4}$$

因此，我们将 Q 学习方法总结如下：

算法	**MDP-Q 学习方法**
问题类型	MDP 控制问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$
求	最优策略 $\pi^*(a\|s)$
算法性质	value-based, off-policy, 表格型

Algorithm 69: MDP-Q 学习方法

Input: MDP 环境 $\mathcal{E}_D$

Output: 最优策略 $\pi^*(a|s)$

$Q_0(s,a) \leftarrow random(A \times S)$

$\pi_0(a|s) \leftarrow random(A \times S)$

$N(s,a) \leftarrow 0, \forall (a,s) \in \mathcal{S} \times \mathcal{A}$

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**

    $a_t \sim \pi_\epsilon(a; s, Q_k)$

    $s_t, r_t \leftarrow \mathcal{E}_D(a_t)$

    $T_{t-1} \leftarrow r_{t-1} + \gamma \max_a Q_k(s_t, a)$

    $Q_k(s_{t-1}, a_{t-1}) \leftarrow Q_k(s_{t-1}, a_{t-1}) + \alpha(T_{t-1} - Q_k(s_{t-1}, a_{t-1}))$

$\pi^*(a|s) \leftarrow \pi_\epsilon(s, a; Q_K)$

对应的 python 代码如下所示

```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
```

值得注意的是，Q 学习被分类为一种 off-policy 算法。这是为了强调产生采样数据的策略不同于学习目标。在时序差分类算法中，学习目标就是前面定义的时序差分目标，是算法更新的方向。

本章介绍的各类表格型方法思路直观，易于理解，但实际应用中受到很多限制。尤其是在连续空间下，会造成维度爆炸问题。虽然也有 DQN 类的融合深度学习的表格类方法，但对于机器人的学习而言，策略梯度方法有更广的适用范围。在下面的章节中，我们将介绍策略梯度方法。

21.4 倒立摆求解

在上面的几节中，我们介绍了不同的表格型强化学习方法。我们仍在经典的一阶/二阶平面倒立摆问题中验证这些方法的有效性。

□

[本部分内容将在后续版本中更新，敬请期待]

22 策略梯度方法

22.1 策略参数化

在上一章中，我们介绍了不少基于价值 (value-based) 的深度学习方法。这类方法都使用价值定义式或贝尔曼公式的近似来估计价值，最终产生某种基于价值 (V 或 Q) 的贪心策略。

事实上，回顾 MDP 控制问题以及 POMDP 控制问题，我们要求解的其实是最优策略，最优策略并不一定要依赖于价值估计而存在。**基于策略的强化学习方法** (policy-based RL method) 就是这样的一类强化学习方法——我们尝试直接拟合一个策略函数 $\pi(a|s)$。

如上一章的许多算法所示，$\pi(a|s)$ 是一个以状态空间为输入，以动作空间上的分布为输出的函数，可以认为是 $\mathcal{S} \to \mathcal{P}(\mathcal{A})$。一方面它要求输出的概率分布是归一化的，另一方面它的输入输出维度一般都不小，并且 $\mathcal{S}, \mathcal{A}$ 可能是连续空间。此外 MDP 问题本身一般也是比较复杂的，这都使得 $\pi(a|s)$ 是一个形式复杂的函数。

在第 V 部分中，我们介绍了深度学习这一强大的工具，让我们能借助 DNN 拟合高维、复杂、非线性的函数，并提供了基于梯度下降和反向传播的 DNN 训练方法。这样，我们就可以使用 DNN 表达策略函数 $\pi(a|s)$。一个 DNN 有一组参数 θ，当我们用含 θ 的 DNN 拟合策略函数 $\pi(a|s)$，就相当于把函数空间的求解问题转化为了参数的优化问题。这种思想我们称之为**策略参数化**。我们一般使用 $\pi_\theta(a|s)$ 或 $\pi(a|s;\theta)$ 表示参数化的策略。

使用 DNN 建模策略还有一个额外的好处：对于连续动作/状态空间的难题，DNN 可以直接输入和输出连续量。如果我们想拟合一个**确定性策略** (即某个状态对应唯一一个动作，其概率为 1)，我们可以让 DNN 直接在连续动作空间输出。如果我们需要一个**随机性策略**，我们可以拟合参数的分布，然后使用重参数化技巧进行采样（详见第22.5.2小节）。这样，策略参数化无需像大部分表格型方法一样对连续空间做离散化，**避免了维数爆炸问题**。

在上述策略参数化的假设下，寻找最优策略 π^* 的问题也就变成了寻找最优 DNN 参数 θ^* 的问题。这样，我们就可以重新表述 MDP 控制问题。

问题	**MDP 控制问题 [无模型参数化]**
问题简述	MDP 未知环境模型，求最优参数化策略
已知	MDP 环境 $\mathcal{E}_D$
求	策略 $\pi_\theta(a|s)$ 的最优参数 θ^*

在深度学习中，我们除了需要设计神经网络的结构，还需要设计神经网络的目标函数 (DL 中称为损失函数)，作为优化目标。在强化学习中，这一优化目标是和价值函数有关的。

在第20章中，我们给出过最优策略的定义 (式VI.20.18)，即**最优策略是其对应的价值函数在任何状态都取最大的策略**。因此，我们的优化目标应当是使价值函数最大化。不过，状态空间包含很多状态，我们并不能把每个状态空间的价值同时作为优化目标。因此，我们考虑引入一个**权重**，对每个策略的价值进行加权

$$J(\theta) = \sum_{s\in\mathcal{S}} \eta(s) V_{\pi(\theta)}(s) \tag{VI.22.1}$$

这里的权重 $\eta(s)$ 表示我们对状态 s 的重视程度，其值越大，说明我们越关注这一状态。什么样的状态值得我们关注呢？显然是在整个空间中越容易出现的状态，越值得我们关注。因此，$\eta(s)$ 可以定义为：**任何步中状态 s 出现的概率/概率密度**。这样，权重 $\eta(s)$ 需要满足归一化条件，即在状态空间求和 (连续空间为积分) 应该等于 1

$$\sum_{s\in\mathcal{S}}\eta(s)=1 \tag{VI.22.2}$$

式VI.22.1解决了策略 DNN 优化目标定义的问题，然而这一定义有个重大缺陷：**状态的分布 $\eta(s)$ 是依赖于策略 π 的**，即 $\eta_\theta(s)$。不同的策略会有不同的价值函数，不同的价值函数下，一个状态被选择的概率是不同的，总的出现频率也不相同。

为了避免这个问题，我们必须给定一个不随策略变化的状态分布[22]。在这里我们选用**初始状态的分布** $p(s_0)$，它是由环境 $\mathcal{E}_D$ 给定的。这样，我们的优化目标实际是

$$J(\theta)=\mathbb{E}_{s_0\sim p(s_0)}[V_{\pi(\theta)}(s_0)] \tag{VI.22.3}$$

那么，求解无模型参数化 MDP 控制问题，实际上就是求解这样的最优参数

$$\theta^*=\arg\max_\theta \mathbb{E}_{s_0\sim p(s_0)}[V_{\pi(\theta)}(s_0)] \tag{VI.22.4}$$

注意，和深度学习最小化损失函数不同，策略参数化的优化问题是**最大化问题**。当然，我们也可以通过加入负号将其简单地变为最小化问题。

22.2 策略梯度定理

在深度学习中，DNN 参数的优化依赖于参数对损失函数梯度的求解。为求解这个梯度，我们仅需知道 DNN 输出 $\hat{y}$ 对损失函数的梯度 $\frac{\partial L}{\partial \hat{y}}$，以及用反向传播算法求解 $\frac{\partial \hat{y}}{\partial \Theta}$。这里，损失函数和 DNN 参数的关系是非常直观的。

而在强化学习中，DNN 的参数 θ 用于产生动作空间上的条件概率分布 $\pi_\theta(a|s)$，但优化目标是价值函数在初始状态分布上的期望 $V_{\pi(\theta)}(s_0)$。事实上，策略 $\pi_\theta(a|s)$ 影响每一个时间步，而每一个时间步的奖励经过折算才能转化为状态价值。也就是说，在 RL 中，目标函数和 DNN 参数的关系比较复杂。我们需要将这种复杂的依赖关系表示为可以求解的数学公式，表达出 $\nabla_\theta J(\theta)$，即**策略梯度**。

为了将初始状态的价值函数和策略 $\pi_\theta(a|s)$ 关联起来，我们可以借助价值函数间的依赖关系。根据式VI.20.9，对于某个给定的 s_0，我们有

$$\nabla_\theta V_{\pi(\theta)}(s_0)=\nabla_\theta\sum_{a_1\in\mathcal{A}}\pi_\theta(a_1|s_0)Q_{\pi(\theta)}(s_0,a_1)$$

上式中已经出现了策略 $\pi_\theta(a_1|s_0)$，这部分是可求的。然而，后面的 $Q_{\pi(\theta)}(s_0,a_1)$ 仍然依赖 θ。根据梯度的性质，我们有

$$\begin{aligned}\nabla_\theta V_{\pi(\theta)}(s_0)&=\nabla_\theta\sum_{a_1\in\mathcal{A}}\pi_\theta(a_1|s_0)Q_\theta(s_0,a_1)\\&=\sum_{a_1\in\mathcal{A}}(\nabla_\theta\pi_\theta(a_1|s_0))Q_\theta(s_0,a_1)+\pi_\theta(a_1|s_0)(\nabla_\theta Q_\theta(s_0,a_1))\end{aligned}$$

这里为了表达方便，我们将 $V_{\pi(\theta)}$ 简写为 V_θ，$Q_{\pi(\theta)}$ 简写为 Q_θ。读者需注意，在 MDP 中这两种价值都依赖特定的策略而存在，也就是依赖 θ 而存在。

上式中，我们将求解状态价值梯度转化为了求解动作价值梯度。根据式VI.20.10，我们有

[22]这其实是一种妥协

$$\begin{aligned}\nabla_\theta V_{\pi(\theta)}(s_0) &= \sum_{a_1\in\mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0,a_1) + \pi_\theta(a_1|s_0)\nabla_\theta Q_\theta(s_0,a_1) \\ &= \sum_{a_1\in\mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0,a_1) + \sum_{a_1\in\mathcal{A}} \pi_\theta(a_1|s_0)\nabla_\theta (R_1 + \sum_{s'\in\mathcal{S}} p(s_1|s_0,a_1)\gamma V_\theta(s_1))\end{aligned}$$

注意，在策略梯度方法中，我们假设折算因子 $\gamma = 1$。上式中，R 和 $p(s'|s,a)$ 均由环境决定，和 θ 无关，因此

$$\nabla_\theta V_{\pi(\theta)}(s_0) = \sum_{a_1\in\mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0,a_1) + \sum_{a_1\in\mathcal{A}} \pi_\theta(a_1|s_0) \sum_{s_1\in\mathcal{S}} p(s_1|s_0,a_1)\nabla_\theta V_\theta(s_1) \tag{VI.22.5}$$

注意到，这里的 $\sum_{a_1\in\mathcal{A}} \pi_\theta(a_1|s_0)p(s_1|s_0,a_1)$，就是在策略 θ 下，由状态 s_0 转换为状态 s_1 的概率。因此，我们可以引入一个**状态转移记法**

$$\Pr_\theta(S_{t'} = s_{t'}|S_t = s_t)$$

该记法表示：当前策略下，若第 t 个状态取值为 s_t，第 t' 个状态取值为 $s_{t'}$ 的概率[23]。这样，我们就有

$$\Pr_\theta(S_1 = s_1|S_0 = s_0) = \sum_{a_1\in\mathcal{A}} \pi_\theta(a_1|s_0)p(s_1|s_0,a_1) \tag{VI.22.6}$$

这一记法可以进行级联。例如，对于 $t = 0, t' = 2$，我们有

$$\Pr_\theta(S_2 = s_2|S_0 = s_0) = \sum_{s_1\in\mathcal{S}} \Pr_\theta(S_1 = s_1|S_0 = s_0)\Pr_\theta(S_2 = s_2|S_1 = s_1) \tag{VI.22.7}$$

使用这样的记法，我们可以将式VI.22.5简化为：

$$\nabla_\theta V_\theta(s_0) = \sum_{a_1\in\mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0,a_1) + \sum_{s_1\in\mathcal{S}} \Pr_\theta(S_1 = s_1|S_0 = s_0)\nabla_\theta V_\theta(s_1) \tag{VI.22.8}$$

可以看到，我们实际上写出了关于 $\nabla_\theta V_\theta(s_t)$ 的递推关系式。我们可以简单地将角标 $+1$，写出下一步的递推式

$$\nabla_\theta V_\theta(s_1) = \sum_{a_2\in\mathcal{A}} \nabla_\theta \pi_\theta(a_2|s_1) Q_\theta(s_1,a_2) + \sum_{s_2\in\mathcal{S}} \Pr_\theta(S_1 = s_1, S_2 = s_2)\nabla_\theta V_\theta(s_2)$$

我们可以总结这个通式，即

$$\nabla_\theta V_\theta(s_t) = \sum_{a_{t+1}\in\mathcal{A}} \nabla_\theta \pi_\theta(a_{t+1}|s_t) Q_\theta(s_t,a_{t+1}) + \sum_{s_t\in\mathcal{S}} \Pr_\theta(S_{t+1} = s_{t+1}|S_t = s_t)\nabla_\theta V_\theta(s_{t+1})$$

将递推式代入式VI.22.8，注意到此处可以使用状态转移记号的级联关系进行化简

[23]这里使用了大写和小写来区分随机变量和其取值

$$
\begin{aligned}
\nabla_\theta V_\theta(s_0) &= \sum_{a_1\in\mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0,a_1) + \sum_{s_1\in\mathcal{S}} \Pr_\theta(S_1=s_1|S_0=s_0)\nabla_\theta V_\theta(s_1) \\
&= \sum_{a_1\in\mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0,a_1) \\
&+ \sum_{s_1\in\mathcal{S}} \Pr_\theta(S_1=s_1|S_0=s_0) \sum_{a_2\in\mathcal{A}} \nabla_\theta \pi_\theta(a_2|s_1) Q_\theta(s_1,a_2) \\
&+ \sum_{s_1\in\mathcal{S}} \Pr_\theta(S_1=s_1|S_0=s_0) \sum_{s_2\in\mathcal{S}} \Pr_\theta(S_1=s_1,S_2=s_2)\nabla_\theta V_\theta(s_2) \\
&= \sum_{a_1\in\mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0,a_1) \\
&+ \sum_{s_1\in\mathcal{S}} \Pr_\theta(S_1=s_1|S_0=s_0) \sum_{a_2\in\mathcal{A}} \nabla_\theta \pi_\theta(a_2|s_1) Q_\theta(s_1,a_2) \\
&+ \sum_{s_2\in\mathcal{S}} \Pr_\theta(S_2=s_2|S_0=s_0)\nabla_\theta V_\theta(s_2)
\end{aligned}
$$

继续展开一步，我们会发现展开式中故案件的三项是状态转移项、策略 DNN 的梯度，以及动作价值函数

$$
\begin{aligned}
\nabla_\theta V_\theta(s_0) &= \sum_{a_1\in\mathcal{A}} \nabla_\theta \pi_\theta(a_1|s_0) Q_\theta(s_0,a_1) \\
&+ \sum_{s_1\in\mathcal{S}} \Pr_\theta(S_1=s_1|S_0=s_0) \sum_{a_2\in\mathcal{A}} \nabla_\theta \pi_\theta(a_2|s_1) Q_\theta(s_1,a_2) \\
&+ \sum_{s_2\in\mathcal{S}} \Pr_\theta(S_2=s_2|S_0=s_0) \sum_{a_3\in\mathcal{A}} \nabla_\theta \pi_\theta(a_3|s_2) Q_\theta(s_2,a_3) \\
&+ \sum_{s_3\in\mathcal{S}} \Pr_\theta(S_3=s_3|S_0=s_0)\nabla_\theta V_\theta(s_3)
\end{aligned}
$$

将上述时间步展开推广到无限步，我们有

$$
\nabla_\theta V_\theta(s_0) = \sum_{k=0}^{\infty} \sum_{s_k\in\mathcal{S}} \Pr_\theta(S_k=s|S_0=s_k) \sum_{a_{k+1}\in\mathcal{A}} \nabla_\theta \pi_\theta(a_{k+1}|s_k) Q_\theta(s_k,a_{k+1})
$$

注意到，通过状态转移记法，此处的梯度项已经与时序 k 无关，因此我们可以将符号简化，无需强调具体状态和动作随机变量的时序，我们有

$$
\nabla_\theta V_\theta(s_0) = \sum_{k=0}^{\infty} \sum_{s\in\mathcal{S}} \Pr_\theta(S_k=s|S_0=s_0) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)
$$

注意公式中的 $\sum_{k=0}^{\infty} \Pr_\theta(S_k=s|S_0=s_0)$ 部分，它的意义是：在初始状态为 s_0 的条件下，在任何一个时间步出现状态 s 的概率之和。事实上，这正是状态 s **出现次数的 (条件) 期望**。我们将其记为 $\mu_\theta(s|s_0)$，有：

$$
\mu_\theta(s|s_0) = \sum_{k=0}^{\infty} \Pr_\theta(S_k=s|S_0=s_0) \tag{VI.22.9}
$$

因此，我们有

$$
\nabla_\theta V_\theta(s_0) = \sum_{s\in\mathcal{S}} \mu_\theta(s|s_0) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)
$$

状态空间中，所有状态的出现次数期望之和，就是平均的 episode 长度。对于特定的策略 θ 和初始状态 s_0，这个长度是一个常数 L_m

$$L_m = \sum_{s\in\mathcal{S}} \mu_\theta(s|s_0) \tag{VI.22.10}$$

注意到，$\mu_\theta(s|s_0)$ 和 L_m 之比，正是上一节中我们定义的" 状态出现概率"$\eta(s)$，只不过是在给定初始状态的条件下

$$\eta(s|s_0) = \frac{\mu_\theta(s|s_0)}{L_m} \tag{VI.22.11}$$

因此，给定 s_0 时，初始状态的价值梯度可以进一步写为

$$\nabla_\theta V_\theta(s_0) = L_m \sum_{s\in\mathcal{S}} \eta(s|s_0) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)$$

根据式VI.22.1，对上式求期望，就是策略梯度的表达式

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{s_0\sim p(s_0)}[L_m \sum_{s\in\mathcal{S}} \eta(s|s_0) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)] \\
&= L_m \mathbb{E}_{s_0\sim p(s_0)}[\sum_{s\in\mathcal{S}} \eta(s|s_0) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)] \\
&= L_m \mathbb{E}_{s_0,s}\left[\sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)\right]
\end{aligned}$$

这一步中，我们将 s 在状态空间中的的依概率求和也写入了期望。注意到

$$\begin{aligned}
\nabla_\theta J(\theta) &= L_m \mathbb{E}_{s_0,s}\left[\sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)\right] \\
&= L_m \mathbb{E}_{s_0,s}\left[\sum_{a\in\mathcal{A}} \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s) Q_\theta(s,a)}{\pi_\theta(a|s)}\right] \\
&= L_m \mathbb{E}_{s_0,s,a}\left[\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} Q_\theta(s,a)\right] \\
&= L_m \mathbb{E}_{s_0,s,a}\left[\nabla_\theta \ln \pi_\theta(a|s) Q_\theta(s,a)\right]
\end{aligned}$$

在这一步中，我们使用了乘一项除一项的技巧，凑出了一个对策略的对数求导的形式，从而将动作 a 也写入了期望。整理一下，我们有

$$\nabla_\theta J(\theta) = E\left[\nabla_\theta \ln \pi_\theta(a|s) Q_\theta(s,a)\right] \tag{VI.22.12}$$

式VI.22.12也称为**策略梯度定理**，它是本章和下一章将介绍的所有策略梯度类方法的基础。注意：在这个公式中我们省略了常数 L_m，因为对于每组策略 θ，策略梯度用于更新神经网络，根据第 V 部分介绍的神经网络梯度下降算法，L_m 会被可调节的学习率参数 α 吸收。

22.3 原始 REINFORCE 算法

到目前为止，我们已经得到了一个策略梯度的表达式 (式VI.22.12)。然而，这个表达式和我们的目标仍有距离，因为它包含动作价值 (即 Q 函数)，但我们无法直接获得 Q 值，而只能从环境不断获得奖励。那么，如何用奖励估计 Q 值呢？

回顾上一章介绍的表格型方法，在解决无模型 MDP 控制问题的蒙特卡洛策略迭代算法 (算法21.1) 中，我们使用了蒙特卡洛采样的思路。即：为了求出每一组 (s_t, a_t) 对应的 Q，我们参考 Q 函数的定义，使用累积回报 G_t 替代动作价值，并用采样轨迹的平均代替期望。

在这里，我们依然可以使用类似的思路，即

$$\nabla_\theta J(\theta) = \mathbb{E}_{a\sim\pi_\theta}\left[G_t \nabla_\theta \ln \pi_\theta(a_t|s_{t-1})\right] \tag{VI.22.13}$$

由此，我们彻底解决了策略参数化方法中，目标函数和策略 DNN 的参数之间梯度求解的问题。对比深度学习的反向传播，式VI.22.13就类似于 $\frac{\partial L}{\partial \hat{y}}$。深度学习中，$\frac{\partial L}{\partial \hat{y}}$ 是网络梯度 $\frac{\partial \hat{y}}{\partial \theta} = \nabla_\theta NN(\cdot;\theta)$ 和损失函数梯度 $\frac{\partial L}{\partial \theta}$ 之间的桥梁。而在强化学习中，式VI.22.13连接起了策略梯度 $\nabla_\theta J(\theta)$ 和网络对数梯度 $\nabla_\theta \ln \pi_\theta(a_t|s_{t-1})$。尽管该式的推导比 DL 的链式法则复杂的多，但结果还是非常简洁的。

在第 V 部分中，我们介绍 DL 求解梯度的 BP 算法时提到，实际使用中，大部分情况下我们不需要手动实现 BP 算法，而仅需要实现前向传播；pytorch 等 DL 框架可以通过自动微分功能，从前向传播中自动获取 BP 的计算方法。也就是说，只要我们实现了从 DNN 输入到输出再到损失函数的整个步骤，就无需手算梯度。

在 RL 中，我们也可以借助这样的 DL 框架。事实上，根据上述策略梯度定理，我们可以写出一个 $\bar{J}(\theta)$，从而利用 DL 框架的自动微分梯度求解特性

$$\bar{J}(\theta) = \sum_{t=0}^{L_e} G_t \ln \pi_\theta(a_t|s_{t-1}) \tag{VI.22.14}$$

之所以称其为" 伪" 价值函数，是因为它的取值并不等于价值函数的定义 (式VI.22.3)，但其梯度和真实价值函数梯度近似相等 (仅相差一个常数)。策略参数化的目的是，借助 DL 工具，将 RL 的 MDP 控制问题转化为一个 DL 的优化问题，并使用 DL 的梯度下降等优化方法进行求解。使用伪价值函数进行梯度下降优化，就等同于用真实价值进行梯度下降优化。

由此，我们可以总结基于 DNN 梯度下降、策略梯度定理和蒙特卡洛近似的 MDP 控制问题求解算法，我们将其称为**原始 REINFORCE 算法**

算法	**原始 REINFORCE 算法**
问题类型	MDP 控制问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$
求	最优参数化策略 $\pi^*_\theta(a\|s)$
算法性质	policy-based, on-policy, 策略梯度

Algorithm 70: 原始 REINFORCE 算法

Input: MDP 环境 $\mathcal{E}_D$

Output: 最优参数化策略 $\pi_\theta^*(a|s)$

$\theta \leftarrow random(\Theta)$

for $m \in 1, \ldots, M$ **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**

    $G_t \leftarrow \sum_{\tau=t}^{L_e} r_\tau$

  $\bar{J}(\theta) = \frac{1}{L_e} \sum_{t=0}^{L_e} G_t \ln \pi_\theta(a_t|s_{t-1})$

  $\theta \leftarrow \theta + \alpha \nabla_\theta \bar{J}(\theta)$

$\pi^*(a|s) \leftarrow \pi_\theta(a|s)$

下面，我们写出上述原始 REINFORCE 算法的 python 代码。我们延续上一章的假设，仍认为 $\mathcal{S}, \mathcal{A}$ 为离散状态空间，状态和动作都用整数作为代表。采用第18章介绍的 MLP 作为策略 DNN$\pi_\theta(a|s)$ 的结构，其输入为 $|\mathcal{S}|$ 维的 one-hot 向量。

结合 MLP 的 BSGD 优化算法 (算法50)，上述算法的 python 代码如下所示


```python
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
```

在 MC 策略迭代 (算法21.1) 中，我们还用到了 ϵ-贪心策略以平衡探索与利用。这是因为，表格型方法的贪心策略是确定性策略，不利于探索。而在使用策略梯度方法时，策略网络的输出就是动作的分布，因此采样时就保证了随机性，不需要再额外进行随机化。

22.4 REINFORCE 算法

原始 REINFORCE 算法已经可以用于 MDP 参数化策略的优化求解。但这个算法存在一个悖论。

考虑这样的情况：假设某环境有三个动作 A, B, C 可选，且所有状态所有动作的奖励均为正 (如 1-100)。某轮采样，只采样到了 A 和 B 两个动作，占比各一半。动作 A 的累计奖励值都比较低 (如不超过 5)，动作 B 的累计奖励值都比较高 (如不低于 80)。那么，在一轮更新后，三个动作能被采样到的概率如何变化呢？

如果按照上面的原始 REINFORCE 算法，动作 A 和 B 的选择概率都会上升，因为其对应的 G_t 都为正；由于 B 的奖励比 A 高，且占比相同，因此动作 B 概率上升的幅度也比 A 更高。由于概率总和为 1，动作 C 的选择概率就会下降。这意味着，模型认为，三个动作的效用应该是 $B > A > C$。

然而，选择动作 C 真的比选择 A 还差吗？事实上，C 的累计奖励大概率位于 A 和 B 之间 (即 5-80 之间)，即 $B > C > A$。仅仅是由于采样有偏，才让模型得出了 A 比 C 好的错误更新。

这个悖论的两个关键要素在于：**采样不足**和**奖励恒正**。如果采样更多，动作 C 有足够的样本，那么网络自然会将 C 的选择概率调大。不过，采样不足在算法开始的初期是无法避免的。相比之下，我们可以从奖励恒正这一点入手，将所有动作的平均奖励调整到 0 左右，这就是**基线技巧**。我们定义

$$A(a_t, s_t) = G_t - b \tag{VI.22.15}$$

其中 b 为基线，是当前所有 G_t 的平均。我们可以进行渐进式更新

$$\begin{aligned} b_k &= \frac{1}{L_e}\sum_{t=0}^{L_e} G_t \\ b &\leftarrow b + \frac{1}{m}(b_k - b) \end{aligned} \tag{VI.22.16}$$

此时，策略梯度和伪价值函数变为

$$\nabla_\theta J(\theta) = \mathbb{E}_{a\sim\pi_\theta}\left[A(a_t, s_t)\nabla_\theta \ln \pi_\theta(a_t|s_{t-1})\right] \tag{VI.22.17}$$

$$\bar{J}(\theta) = \sum_{t=0}^{L_e} A(a_t, s_t)\pi_\theta(a_t|s_t) \tag{VI.22.18}$$

这里我们引入的 $A(a_t, s_t)$ 称为**优势函数**。它表示：在给定策略下，对于状态 s_t，采取某个动作 a_t **相对于其他动作的优势**。在基线技巧下，由于 b 的引入，无论累积奖励 G_t 是恒正还是恒负，优势函数都能体现动作间的相对优势，从而避免原始 REINFOECE 算法的悖论。

这里仍需要注意：优势函数 A 和状态价值函数 V、动作价值函数 Q 一样，都依赖于某一策略而存在。不存在通用于任何策略的优势函数。此外，$A(a_t, s_t)$ 和参数 θ 无关，我们不会对其求梯度。

由此，我们可以总结 **REINFORCE 算法**

算法	**REINFORCE 算法**
问题类型	MDP 控制问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$
求	最优参数化策略 $\pi_\theta^*(a\|s)$
算法性质	policy-based, on-policy, 策略梯度

Algorithm 71: REINFORCE 算法

Input: MDP 环境 $\mathcal{E}_D$

Output: 最优参数化策略 $\pi_\theta^*(a|s)$

$\theta \leftarrow random(\Theta)$

$b \leftarrow 0$

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**

      $G_t \leftarrow \sum_{\tau=t}^{L_e} r_\tau$

      $A(a_t, s_t) \leftarrow G_t - b$

    $b_k \leftarrow \frac{1}{L_e}\sum_{t=0}^{L_e} G_t$

    $b \leftarrow b + \frac{1}{m}(b_k - b)$

  $\bar{J}(\theta) = \frac{1}{KL_e}\sum_{k=0}^{K}\sum_{t=0}^{L_e} A(a_t, s_t)\ln\pi_\theta(a_t|s_t)$

  $\theta \leftarrow \theta + \alpha\nabla_\theta\bar{J}(\theta)$

$\pi^*(a|s) \leftarrow \pi_\theta(a|s)$

对应的 python 代码如下所示 (省略部分同原始 REINFORCE 算法)

```python
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 倒立摆求解

类似于表格型方法，策略梯度方法也可以用于求解倒立摆问题 (详见10.6节)。

22.5.1 离散动作求解

[本部分内容将在后续版本中更新，敬请期待]

22.5.2 连续动作求解

与表格型方法不同，策略梯度方法不仅可以拟合离散动作的分布，还可以直接输出连续动作的概率分布。

具体来说，在倒立摆问题中，我们希望策略网络 $\pi_\theta(a|s)$ 拟合一个连续条件概率分布 $p(a|s)$。这里的状态是上面的倒立摆连续状态，对于一阶倒立摆是 X 维，对于二阶倒立摆是 Y 维；而动作则是连续的一维量，即横向力 F。事实上，我们仅需将概率分布建模为高斯分布

$$\pi_\theta(a|s) = \mathcal{N}(\mu_{a;\theta}, \sigma^2_{a;\theta}) \tag{VI.22.19}$$

式中，$\mu_{a;\theta,s}$ 和 $\sigma^2_{a;\theta,s}$ 分别表示连续动作 a 这个随机变量的均值和方差。这两个量均依赖于网络参数和状态。也就是说，我们仅需建立一个策略神经网络，其输入是系统状态，输出维数是动作维数的 2 倍，分别代表动作的均值和方差，即可完成策略参数化。

在这样的设定下，网络损失函数也发生了小小的改变。我们需要将损失函数中的 $\ln \pi_\theta(a|s)$ 写为网络直接输出 μ_a, σ^2_a 的表达式。

[本部分内容将在后续版本中更新，敬请期待]

23 PPO 算法

对于本章讨论，在无特殊说明时，默认 MDP 为离散动作空间。

23.1 重要性采样

前文所述的 REINFORCE 算法使用策略梯度定理（以及一些其他技巧），可以持续从环境中采样并更新策略网络，是一种不错的深度强化学习算法。然而，该算法需要每次采样一组轨迹后，只能对网络更新一次。即使在模拟环境中，采样的速率也一般慢于网络更新。因此，整个算法的效率会被拉低。

为什么 REINFORCE 不能多次使用同一批数据更新呢？这是因为，更新后的网络 (记为 π_θ) 不再是采样这组数据的网络 (记为 $\pi_{\theta'}$)，而是一个相似但不同的网络。强行使用这组数据更新，就会导致不符合策略梯度定理的假设，网络无法收敛。

23.1.1 重要性权重

具体来说，让我们考虑一个更一般的情景。有一组数据来源于分布 $q(x)$，可以认为是上一次批量采样时的网络 $\pi_{\theta'}$。有一个和 $q(x)$ 相似但不同的分布 $p(x)$，可以认为是更新后的网络 π_θ。有一个依赖样本的函数 $f(x)$，可以认为是上面的策略梯度 $\nabla_\theta \bar{J}(\theta)$。现在的问题是：想计算 $\mathbb{E}_{x\sim p(x)}[f(x)]$，但实际只能从 $q(x)$ 中采样。

事实上，我们可以引入一个重要性权重，调整采样到的每个样本的" 重要性"。如果对于某样本，$p(x)$ 大而 $q(x)$ 小，就应当提升其权重。反之，$p(x)$ 小而 $q(x)$ 小，就应当降低其权重。这就是**重要性采样**

$$\mathbb{E}_{x\sim p}[f(x)] = \mathbb{E}_{x\sim q}\left[\frac{p(x)}{q(x)}f(x)\right] \tag{VI.23.1}$$

需要注意的是，虽然对于任意两个 $p(x)$ 和 $q(x)$，都存在上述关系，但我们不可以用相差很大的两个分布相互近似。这是因为，当我们计算方差，我们会发现

$$\begin{aligned}
\mathrm{Var}_{x\sim p}[f(x)] &= \mathbb{E}_{x\sim p}\left[f(x)^2\right] - \left(\mathbb{E}_{x\sim p}[f(x)]\right)^2 \\
\mathrm{Var}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right] &= \mathbb{E}_{x\sim q}\left[\left(f(x)\frac{p(x)}{q(x)}\right)^2\right] - \left(\mathbb{E}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right]\right)^2 \\
&= \mathbb{E}_{x\sim p}\left[f(x)^2\frac{p(x)}{q(x)}\right] - \left(\mathbb{E}_{x\sim p}[f(x)]\right)^2
\end{aligned} \tag{VI.23.2}$$

也就是说，$\mathrm{Var}_{x\sim p}[f(x)]$ 和 $\mathrm{Var}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$ 存在差距。$p(x)$ 和 $q(x)$ 相差越大，方差差距越大。在采样不足时，近似的期望的误差也就越大。

23.1.2 重要性采样价值

回到 REINFORCE 网络更新问题。如上所述，$q(x)$ 相当于 $\pi_{\theta'}$，$p(x)$ 相当于更新后的网络 π_θ，$f(x)$ 相当于策略梯度 $\nabla_\theta \bar{J}(\theta)$。因此，重要性采样下的策略梯度就可以写为

$$\nabla_\theta J(\theta;\theta') = \mathbb{E}_{a\sim\pi_{\theta'}}\left[\frac{\Pr_\theta(a_t, s_{t-1})}{\Pr_{\theta'}(a_t, s_{t-1})}A(a_t, s_{t-1})\nabla_\theta \ln \pi_\theta(a_t|s_{t-1})\right]$$

进一步，我们有

$$\frac{\Pr_\theta(a_t, s_{t-1})}{\Pr_{\theta'}(a_t, s_{t-1})} = \frac{\pi_\theta(a_t|s_{t-1})\Pr_\theta(s_{t-1})}{\pi_{\theta'}(a_t|s_{t-1})\Pr_{\theta'}(s_{t-1})}$$
$$\approx \frac{\pi_\theta(a_t|s_{t-1})}{\pi_{\theta'}(a_t|s_{t-1})}$$

这一步中，我们忽略了 $Pr_\theta(s_{t-1})$ 和 $Pr_{\theta'}(s_{t-1})$ 之间的差异。我们近似认为，状态分布的微小差异不会对策略梯度产生很大影响。由此，我没有

$$\nabla_\theta J(\theta;\theta') \approx \mathbb{E}_{a\sim\pi_{\theta'}}\left[\frac{\pi_\theta(a_t|s_{t-1})}{\pi_{\theta'}(a_t|s_{t-1})} A(a_t, s_{t-1}) \nabla_\theta \ln \pi_\theta(a_t|s_{t-1})\right]$$

再次使用 log 导数的技巧，我们可以写出重要性采样下的伪价值函数

$$\bar{J}(\theta;\theta') = \mathbb{E}_{a\sim\pi_{\theta'}}\left[\frac{\pi_\theta(a_t|s_{t-1})}{\pi_{\theta'}(a_t|s_{t-1})} A(a_t, s_{t-1})\right] \tag{VI.23.3}$$

在上一节中我们说到，使用重要性采样需要保证两个分布尽量相似。这里，我们为价值函数增加一个惩罚项，约束旧分布 $\pi_{\theta'}(a_t|s_{t-1})$ 和新分布 $\pi_\theta(a_t|s_{t-1})$ 尽量相似。这个惩罚项我们选取 KL 散度。因此，我们有重要性采样下的价值函数

$$J_{IS}(\theta;\theta') = \mathbb{E}_{a\sim\pi_{\theta'}}\left[\frac{\pi_\theta(a_t|s_{t-1})}{\pi_{\theta'}(a_t|s_{t-1})} A(a_t, s_{t-1})\right] - \mathrm{KL}(\pi_\theta||\pi_{\theta'}) \tag{VI.23.4}$$

KL 散度越大，说明两个分布相差越大，此时总价值函数越小，抑制参数向该方向更新。

23.1.3 KL 散度近似计算

对于两个定义在同一空间的离散分布，KL 散度的定义为

$$\mathrm{KL}(q||p) = \sum_x q(x) \ln \frac{q(x)}{p(x)} \tag{VI.23.5}$$

可以看到，计算 KL 散度需要对空间内所有 x 进行采样，并使用两个分布计算概率值。

然而，实际中我们的采样是有限的，无法覆盖整个概率空间。此时，一般使用如下三种方法计算近似 KL 散度

$$\mathrm{KL}(q||p) = \mathbb{E}[k1] = \mathbb{E}_{x\sim q}\left[\ln \frac{q(x)}{p(x)}\right]$$

$k1$ 估计器是无偏估计，但方差较大

$$\mathrm{KL}(q||p) \approx \mathbb{E}[k2] = \mathbb{E}_{x\sim q}\left[\frac{1}{2}\left(\ln \frac{q(x)}{p(x)}\right)^2\right]$$

$k2$ 估计器方差较小，但它是有偏估计

$$\mathrm{KL}(q||p) = \mathbb{E}[k3] = \mathbb{E}_{x\sim q}\left[\frac{p(x)}{q(x)} - 1 + \ln \frac{q(x)}{p(x)}\right] \tag{VI.23.6}$$

$k3$ 估计器方差小而且是无偏估计。计算时要注意第一项分母为 $p(x)$，而其他地方的分母均为 $q(x)$。

在 PPO 中，我们一般使用 $k3$ 估计器近似计算 KL 散度。

23.2 Critic 网络

23.2.1 优势函数

如上一章所述，优势函数代表” 对于某一种策略，在当前状态下，采取某个动作相对于其他动作的优势”。在上一章中，我们使用奖励减去基线的方式构造优势函数。那么，有没有更好的优势函数构造方法呢？

回顾状态价值函数 $V_\pi(s)$ 和动作价值函数 $Q_\pi(s,a)$ 的定义，我们会发现，动作价值意味着在某一状态下，选择某一动作的优势；而状态价值函数意味着，选择这一状态而不是其他状态的优势。状态价值函数是动作价值函数的平均 (见式VI.20.9)。因此，$Q_\pi(s,a)-V_\pi(s)$，正是一个很好的体现” 某个动作相对于其他动作的优势” 的估计量。

$$A_\pi(s,a) := Q_\pi(s,a) - V_\pi(s) \tag{VI.23.7}$$

事实上，这才是优势函数的定义，之前的引入是一个对该函数的估计。

回顾上一节，原始 REINFORCE 算法中的 G_t 可以替换为优势函数。如果我们能得到 $V_\pi(s)$ 和 $Q_\pi(s,a)$ 的值，我们就可以获得一个更好的 REINFORCE 算法。

如何估计 $Q_\pi(s,a)-V_\pi(s)$ 呢？假设我们在策略 π 下采集了很多状态奖励动作序列 $SARS(t)$，由式VI.20.9，我们可以将 Q 函数转化为下一步的 V 函数

$$Q_\pi(s,a) = \mathbb{E}_{\pi,s,a}[r_{t+1} + \gamma V_\pi(s_{t+1})]$$

这里的角标 $\mathbb{E}_{\pi,s,a}$ 表示：每个序列 $SARS(t)$ 都是在策略 π 下采集的，并且 $s_t = s, a_{t+1} = a$

这样，优势函数的估计就变为了

$$A_\pi(s,a) = \mathbb{E}_{\pi,s,a}[r_{t+1} + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)]$$

这样，实际计算中，对于每一组 (s_t, a_{t+1})，就可以直接取 $V_\pi(s_t)$ 和 $V_\pi(s_{t+1})$ 来估计 $A_\pi(s,a)$。这样，每组数据的计算变得简单。当一批数据一起更新策略网络时，就产生了求期望的效果。

23.2.2 价值网络

有了优势函数，如何知道 $V_\pi(s)$ 呢？采用和策略参数化类似的方式，我们也将状态价值函数参数化，用神经网络拟合 $V_\pi(s)$，称为**价值网络**。

假设该神经网络参数为 ω，我们将神经网络记为 $V_\pi(s;\omega)$。如果该价值函数对应的策略是一个策略网络 π_θ，我们也可以将价值网络记为 $V_\theta(s;\omega)$。注意：该记法中右下方角标不代表价值网络本身的参数，而代表其对应的策略网络的参数。

如何训练价值网络呢？注意到这是一个 MDP 的预测问题 (问题20.5)，我们可以借鉴此前解决预测问题的方法。例如，我们可以使用类似 TD(0) 策略评估的方法，构造时序差分目标，让网络进行自举迭代。具体来说，我们可以用二范数定义如下损失函数

$$\mathcal{L}(\omega) = \frac{1}{2}(r + \gamma V_\pi(s_{t+1};\omega) - V_\pi(s_t;\omega))^2 \tag{VI.23.8}$$

注意到，该损失函数中两项都是策略网络的输出。在实际训练时，我们不会对 $V_\pi(s_{t+1};\omega)$ 的部分计算梯度，而只会对 $V_\pi(s_t;\omega)$ 计算梯度，即

$$\nabla_\omega \mathcal{L}(\omega) = (r_t + \gamma V_\pi(s_{t+1};\omega) - V_\pi(s_t;\omega))\nabla_\omega V_\pi(s_t;\omega)$$

注意到，此处的价值网络误差项，形式上和优势函数的估计完全一致。我们可以将其记为 $\delta_\omega(t)$

$$\delta_\omega(t) := r_t + \gamma V_\pi(s_{t+1};\omega) - V_\pi(s_t;\omega) \tag{VI.23.9}$$

这样我们就有

$$\nabla_\omega \mathcal{L}(\omega) = \delta_\omega(t)\nabla_\omega V_\pi(s_t;\omega) \tag{VI.23.10}$$

以及

$$\hat{A}_\pi(s_t, a_{t+1}) = \delta_\omega(t)$$

此外还要注意，我们对策略网络定义的是损失函数，而不是价值函数。损失函数是越小越好，因此训练时应使用梯度下降而不是梯度上升。

尽管策略网络和价值网络是不同意义的网络，但它们的输入都是状态 s。从网络结构的角度看，它们的区别在于：策略网络输出动作的分布，价值网络输出价值标量。因此，实际工程中，往往让策略网络和价值网络共享前几层，这几层的参数在两个网络更新时都会更新。这相当于是：在同一个状态特征提取网络上，添加策略头和价值头，轮流用不同的损失函数进行训练。不过，在后续的描述中，我们仍然分开描述策略网络参数 θ 和价值网络参数 ω。

23.2.3 Actor-Critic 算法

总结一下，为了参数化策略，我们定义了策略网络 $\pi_\theta(a|s)$。接下来，为了给策略网络提供优势函数，我们定义了价值网络 $V_\theta(s;\omega)$。为了得到策略网络，需要在每次策略网络冻结的时候，对价值网络进行训练。这样，我们就得到了两个神经网络交替训练的强化学习算法，称为 **Actor-Critic 算法**

这个名字中，Actor 对应的是策略网络。环境就像一个舞台，策略网络就是舞台上练习的演员。Critic 对应价值网络，评估该策略下的价值函数，就像评论家对演员的表演进行点评。演员根据评论家的反馈提高自身的能力，评论家的评论也随着演员的变化而变化。事实上，这正是第 3 章介绍的**策略迭代**思路。

接下来，我们详细地总结 Actor-Critic 算法。

算法	**Actor-Critic 算法**
问题类型	MDP 控制问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$
求	最优参数化策略 $\pi^*_\theta(a|s)$
算法性质	policy-based, on-policy, 策略梯度

Algorithm 72: Actor-Critic 算法

Input: MDP 环境 $\mathcal{E}_D$

Output: 最优参数化策略 $\pi_\theta^*(a|s)$

$\theta \leftarrow random(\Theta)$

$\omega \leftarrow random(\Omega)$

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 广义优势估计 GAE

在上述 Actor-Critic 算法中，我们简单地使用了 $\delta_\omega(t)$ 来估计优势函数 $A_\pi(s_t, a_{t+1})$。这是一个单步展开，是很粗糙的估计；在采样不足的情况下，偏差可能很大。

为了更好地估计优势函数，我们可以借鉴 $TD(0) \to TD(n)$ 过程中的思路，对其进行多步展开，即

$$\hat{A}_\pi^{(n)}(s_t, a_{t+1}) = -V_\pi(s_t; \omega) + \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n V_\pi(s_{t+1}; \omega)$$

广义优势估计 (GAE) 算法认为，展开阶数越大，估计的方差越大，但偏差越小。因此最好将不同阶数进行加权平均，减小方差的同时确保偏差不大

$$\hat{A}_\pi^{GAE}(s_t, a_{t+1}) = (1 - \lambda) \sum_{j=1}^{n} \hat{A}_\pi^{(j)}(s_t, a_{t+1})$$

事实上，如果用 $\delta_\omega(t)$ 来表达，它有更加简洁的形式（当 n 较大，可省略较小的最后一项 $\gamma^n V_\pi(s_{t+1}; \omega)$）

$$\hat{A}_\pi^{GAE}(s_t, a_{t+1}) = \sum_{t=0}^{n} (\gamma\lambda)^t \delta_\omega(t)$$

这样，我们可以写出 GAE 版本的 Actor-Critic 算法

算法	**Actor-Critic 算法-GAE**
问题类型	MDP 控制问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$
求	最优参数化策略 $\pi_\theta^*(a\|s)$
算法性质	policy-based, on-policy, 策略梯度

Algorithm 73: Actor-Critic 算法-GAE

Input: MDP 环境 $\mathcal{E}_D$

Output: 最优参数化策略 $\pi_\theta^*(a|s)$

$\theta \leftarrow random(\Theta)$

$\omega \leftarrow random(\Omega)$

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 算法

23.3.1 原始 PPO 算法

在上面的两节中，我们通过采样多轮更新问题，介绍了重要性采样下的 REINFORCE 算法，引入了 Critic 网络，并介绍了广义优势估计下的 Actor-Critic 算法。

将以上这些方法整合起来，实现策略神经网络的多轮梯度上升/下降更新，我们就可以得到原始的 PPO(Proximal Policy Optimization, 近端策略优化) 算法。

算法	**原始 PPO 算法**
问题类型	MDP 控制问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$
求	最优参数化策略 $\pi_\theta^*(a\|s)$
算法性质	policy-based, on-policy, 策略梯度

Algorithm 74: 原始 PPO 算法

Input: MDP 环境 $\mathcal{E}_D$

Output: 最优参数化策略 $\pi_\theta^*(a|s)$

$\theta \leftarrow random(\Theta)$

$\omega \leftarrow random(\Omega)$

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.

    $\nabla_\omega \mathcal{L}(\omega) \leftarrow \frac{1}{N_2 L_e} \sum_{i=0}^{N_2} \sum_{t=0}^{L_e} \nabla_\omega \mathcal{L}(a_{t+1}, s_t)$

    $\omega \leftarrow \omega - \alpha_\omega \nabla_\omega \mathcal{L}(\omega)$

  $\theta' \leftarrow \theta$

$\pi^*(a|s) \leftarrow \pi_\theta(a|s)$

这里我们引入了超参数 β，用于平衡策略网络价值函数中的伪价值和 KL 散度。我们目前可以手动指定这个参数，就如手动指定 $\gamma, \alpha_\omega, \alpha_\theta$ 等超参数一样。在下一节，我们会介绍更好的更新方式。

需要注意的是，虽然 PPO 算法中，采样网络和当前学习的网络已经是两个不同的网络，但由于通过 KL 散度，约束了两个网络输出的差异不能过大，因此我们仍将它视作是同策略，即 on-policy 方法。

同理，虽然价值网络 $V_{\theta'}(s; \omega)$ 在每次大循环中，都对应着不同的策略网络 $\pi_{\theta'}$，但由于我们限定了相邻两轮策略网络更新的大小，我们可以认为两个价值函数近似。因此，对每个大循环，无需从头开始训练价值网络，而是可以直接在上一轮价值网络的基础上继续训练。

23.3.2 PPO-Penalty 算法

如前所述，KL 散度代价是为了约束策略网络不进行过大的更新，使用超参数 β 平衡这一代价和学习目标伪价值函数。然而，这一参数的确定并不容易。如果 β 过大，网络将非常保守，不愿意进行任何更新；如

果 β 过小，网络又将很激进，造成采样问题。

为此，我们可以采用一种自适应调节 β 的方法。我们指定一个 KL 散度的区间 $[\mathrm{KL}_{\min}, \mathrm{KL}_{\max}]$。如果估计的 KL 散度超过上限，我们就增大 β；如果低于下限，我们就减小 β。这个思路本质上是一种滞环，类似于滞环比较电路和滞环控制。

这样，我们引入 **PPO-Penalty 算法**

Algorithm 75: PPO-Penalty 算法

Input: MDP 环境 $\mathcal{E}_D$

Output: 最优参数化策略 $\pi_\theta^*(a|s)$

$\theta \leftarrow random(\Theta)$

$\omega \leftarrow random(\Omega)$

$\beta \leftarrow \beta_0$

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.

    $\nabla_\omega \mathcal{L}(\omega) \leftarrow \frac{1}{N_2 L_e} \sum_{i=0}^{N_2} \sum_{t=0}^{L_e} \nabla_\omega \mathcal{L}(a_{t+1}, s_t)$

    $\omega \leftarrow \omega - \alpha_\omega \nabla_\omega \mathcal{L}(\omega)$

$\theta' \leftarrow \theta$

$\pi^*(a|s) \leftarrow \pi_\theta(a|s)$

算法	**PPO-Penalty 算法**
问题类型	MDP 控制问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$
求	最优参数化策略 $\pi_\theta^*(a\|s)$
算法性质	policy-based, on-policy, 策略梯度

23.3.3 PPO-Clip 算法

前述方法确实可以实现重要性采样 + 多次更新策略网络，然而，其代价是 KL 散度的计算比较耗费资源。对每一步采样，都需要计算其 KL 散度，并在反向传播中更新网络。

使用 KL 散度的目的最终是为了防止策略网络一次进行很大的更新。为此，我们可以采用一种替代性的思路，即直接限制重要性权重的大小。这种方法称为**重要性裁剪**。

[本部分内容将在后续版本中更新，敬请期待]

这样，我们可以得到 **PPO-Clip 算法**

Algorithm 76: PPO-Clip 算法 (Part 1)

Input: MDP 环境 $\mathcal{E}_D$

Output: 最优参数化策略 $\pi_\theta^*(a|s)$

$\theta \leftarrow random(\Theta)$

$\omega \leftarrow random(\Omega)$

$\beta \leftarrow \beta_0$

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**

      $\delta_\omega(t) \leftarrow r_t + \gamma V_{\theta'}(s_{t+1}; \omega) - V_{\theta'}(s_t; \omega)$

      $\hat{A}_{\theta'}^{GAE}(s_t, a_{t+1}) \leftarrow \delta_\omega(t) + \gamma\lambda \hat{A}_{\theta'}^{GAE}(s_{t+1}, a_{t+2})$

      $\nabla_\omega \mathcal{L}(a_t, s_t) \leftarrow \delta_\omega(t) \nabla_\omega V_{\theta'}(s_t; \omega)$

      $r_{IS}(a_{t+1}, s_t) \leftarrow \frac{\pi_\theta(a_{t+1}|s_t)}{\pi_{\theta'}(a_{t+1}|s_t)}$

      if $\hat{A}_{\theta'}^{GAE}(s_t, a_{t+1}) > 0$ **then**

        $\bar{r}_{IS}(a_{t+1}, s_t) \leftarrow \min(r_{IS}(a_{t+1}, s_t), 1 + \epsilon)$

      else

        $\bar{r}_{IS}(a_{t+1}, s_t) \leftarrow \min(r_{IS}(a_{t+1}, s_t), 1 - \epsilon)$

    ...（续下页）

Algorithm 77: PPO-Clip 算法 (Part 2)

for $m \in 1, \ldots, M$ **do**

  ...（接上页）

  for $b \in 1, 2, \ldots, B_\theta$ **do**

    Randomly sample N_1 Trails from K Trails.

    $J_{PPO2}(\theta; \theta') \leftarrow \frac{1}{N_1 L_e} \sum_{i=0}^{N_1} \sum_{t=0}^{L_e} \bar{r}_{IS}(a_{t+1}, s_t) \hat{A}_{\theta'}^{GAE}(s_t, a_{t+1})$

    $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J_{PPO2}(\theta; \theta')$

  for $b \in 1, 2, \ldots, B_\omega$ **do**

    Randomly sample N_2 Trails from K Trails.

    $\nabla_\omega \mathcal{L}(\omega) \leftarrow \frac{1}{N_2 L_e} \sum_{i=0}^{N_2} \sum_{t=0}^{L_e} \nabla_\omega \mathcal{L}(a_{t+1}, s_t)$

    $\omega \leftarrow \omega - \alpha_\omega \nabla_\omega \mathcal{L}(\omega)$

$\theta' \leftarrow \theta$

$\pi^*(a|s) \leftarrow \pi_\theta(a|s)$

算法	**PPO-Clip 算法**
问题类型	MDP 控制问题 [无模型]
已知	MDP 环境 $\mathcal{E}_D$
求	最优参数化策略 $\pi_\theta^*(a\|s)$
算法性质	policy-based, on-policy, 策略梯度

(参考资料：蘑菇书)

23.3.4 连续动作空间

正如第22.5.2小节所介绍的，PPO 算法也可以通过高斯分布建模的方式兼容连续动作空间。

[本部分内容将在后续版本中更新，敬请期待]

23.4 倒立摆求解

我们仍然以平面倒立摆问题为例 (详见10.6节)，测试上节介绍的连续动作 PPO 算法。

[本部分内容将在后续版本中更新，敬请期待]

Part VII

视觉导航方法

24 视觉里程计基础

视觉是人类最重要的感觉器官。我们每天约 80% 的信息通过视觉输入。对于自主机器人，充分利用视觉传感器的各种信息非常重要。在本部分中，我们主要关注视觉导航方法。

在本章中，我们将介绍被称为视觉里程计的序贯视觉测量问题，以及重要的基础概念、基础技术。在接下来的几章中，我们将介绍图像特征提取、点对位姿求解等关键技术，随后具体介绍各类视觉里程计方法。

24.1 基本概念

24.1.1 视觉里程计

对于现代机器人而言，无论是固定基座机械臂还是移动机器人，估计末端或自身位姿状态都是非常基础的需求。

视觉运动估计为以上问题提供了一个重要的手段。所谓视觉运动估计，就是通过视觉传感器 (即相机) 的数据，估计相机 (及和相机固联的载体) 自身的位置、姿态、速度等运动信息。这样的任务和方法均称为**视觉里程计** (Visual Odometer, VO)。事实上，对于固定基座机器人的末端状态估计问题，可以将多种传感器的信息通过卡尔曼滤波等方法进行融合，从而获得更精确的状态估计；其中视觉传感器是一种重要的末端传感器。

具体来说，让我们定义视觉里程计问题。

问题	单目视觉里程计
问题简述	已知图像序列，求机器人位姿序列
已知	图像 $I_0, I_1, ..., I_n$ 内参矩阵 K，畸变参数
求	位姿序列 $T_1, ..., T_n$

上述定义中，视觉里程计的输出是位姿序列。所谓位姿就是位置与姿态，也就是两个坐标系之间的旋转 R 与平移 t。我们一般将其写成齐次矩阵 T 的形式。在本部分中，”位姿”和”齐次矩阵”基本等价，此后我们不再区分。内参矩阵和畸变参数是相机成像相关的参数，一般可通过标定获取。一些视觉里程计方法也可以在未知内参和畸变参数的情况下进行位姿求取。

需要注意，在视觉里程计问题中，我们的位姿往往不是相当于绝对坐标系 (如东北天) 的位姿，而是不同状态间的相对位姿。一般我们定义第一张图像 I_0 的相机坐标系 (下一节将介绍) 为**世界坐标系**，符号为 w。其余的位姿序列均为当时的相机坐标系相对世界坐标系的位姿。

视觉里程计可以使用不同类型的视觉传感器。上面我们介绍的只使用了最基础的配置——单目相机——因此被称为**单目视觉里程计**。事实上，单目视觉里程计存在理论上的根本缺陷——无尺度性[24]——我们将在对极几何一节详细介绍。其累计误差问题也比较显著。为此，我们发展了各类综合使用各类不同传感器的视觉里程计，并设计了各类不同的方法、算法。

在接下来几个小节中，我们将从传感器、图像利用方法、位姿优化方法三个维度，初步介绍各种视觉里程计相关概念。

[24] 其实，如果相机拍摄的是常见场景中的常见物体，也可以借助视觉常识进行绝对尺度的估计。端到端视觉里程计就是用这个原理恢复尺度的。

24.1.2 传感器

上一小节中，我们提到了视觉导航中的重要概念：单目无尺度性。相机成像的过程是 3 维空间到 2 维的投影 (详见下节" 相机模型")。在这个过程中，真实尺度和深度信息丢失了。尽管我们可以通过多个不同视角的成像恢复相对深度，但仍然无法恢复真实尺度。想象在一个计算机仿真的 3D 环境中，我们可以将所有物体等比例任意缩放，本质上并不影响成像。

为解决这一问题，我们可以引入已知的长度信息。人类和许多动物都有 2 只具有重叠视场范围的眼睛。当我们通过两眼同时观察一个物体，由于两眼间距 (称为**基线**) 的存在，我们左右眼接收到的图像存在微小差距 (称为**视差**)。借助视差，我们的大脑可以直观地判断物体的实际距离。仿照这样的生物机制，我们可以使用两个固定间距、视野重叠的相机同时获取图像，从而解决带绝对尺度的位姿估计问题，这就是**双目视觉里程计**。

问题	双目视觉里程计
问题简述	已知双目基线、双目图像序列，求机器人位姿序列
已知	左目图像 $I_{0,l}, I_{1,l}, ..., I_{n,l}$ 右目图像 $I_{0,r}, I_{1,r}, ..., I_{n,r}$ 双目基线 T_l^r 双目内参矩阵 K_1, K_2，双目畸变参数
求	位姿序列 $T_1, ..., T_n$

双目相机通过重叠视野对同一物体进行成像，随后可以通过三角测量方法 (详见第26.2节) 恢复物体深度。不过，这需要事先对双目图像进行匹配，并且对视差等有一定要求。为了直接从单幅图像中获取深度，人类发明了各类**深度相机**。深度相机主要有两大类实现原理：基于结构光和基于飞行时间。无论何种原理，深度相机在每一帧都会同时输出光度图 I 和同一视角下的深度图 D。基于深度相机的视觉里程计，我们也称为 **RGBD 视觉里程计**。

问题	RGBD 视觉里程计
问题简述	已知光度和深度图像序列，求机器人位姿序列
已知	光度图像 $I_0, I_1, ..., I_n$ 深度图像 $D_0, D_1, ..., D_n$ 内参矩阵 K，畸变参数
求	位姿序列 $T_1, ..., T_n$

除了相机，还有一种传感器常用于本体运动估计，那就是**惯性测量单元** (IMU, Inertial Mesurement Unit)。IMU 由加速度计和陀螺仪组成，常用均为 MEMS(微机电系统)，体积小，且不依赖任何外界信息，也无需主动发出信息，是非常便捷且低成本的传感器。

IMU 的量测量分为两部分：**比力量测**和**角速度量测**。比力的定义是：载体所受除重力以外的合外力产生的总加速度，记为 f，即

$$f = \frac{F - mg}{m} = a - g \tag{VII.24.1}$$

IMU 一般有 $x-y-z$ 三个敏感轴，一般呈右手系分布。一般我们认为，**载体上 IMU 定义的坐标系即为载体系。IMU 的直接测量量是载体系下的比力和角速度**，即 f^b 和 ω_{nb}^b。比力和角速度都是 3 维向量。

IMU 的数据更新频率一般为 100-200Hz，远超过普通相机的更新频率 (20-50Hz)。因此，每两个图像帧之间的 IMU 测量构成一个向量列，我们将其记为 $\mathbf{f}^b$ 和 $\mathbf{w}_{nb}^b$，表示两帧之间的所有比力和角速度。

通过 IMU，可以给视觉里程计提供重力方向，在方法层面减少了视觉里程计的不可观测性。使用 IMU 的视觉里程计称为**视觉惯性里程计** (VIO, Visual Inertial Odometry)。

问题	(单目) 视觉惯性里程计
问题简述	已知图像序列、IMU 测量序列，求机器人位姿序列
已知	图像 $I_0, I_1, ..., I_n$ 比力 $\mathbf{f}_1^b, ..., \mathbf{f}_n^b$，角速度 $\mathbf{w}_1^b, ..., \mathbf{w}_n^b$ 内参矩阵 K，外参矩阵 T_b^c，畸变参数
求	位姿序列 $T_1, ..., T_n$

本部分后续章节介绍的方法，多为单目/双目视觉里程计，以及视觉惯性里程计，我们也将其统称为 VO。

24.1.3 图像利用

相机的光度图像 I 一般是一个 2 维或 3 维矩阵，其横纵方向上的维数称为分辨率，每个元素称为一个像素。大部分相机可以获取彩色图像，即每个像素以向量形式存储红绿蓝三种颜色对应的强度，此时矩阵 $I \in \mathbf{R}^{H\times W\times 3}$。不过，在 VO 中的一些非稠密建图方法并不在意图像的颜色，因此会将三个颜色通道合成为灰色通道，此时图像为 2 维，即 $I \in \mathbf{R}^{H\times W}$

无论哪种具体的 VO 方法，都至少要以相机的光度图像 I 作为输入。相机的分辨率 H, W 一般都是数百甚至上千，因此 I 是一个非常大的矩阵，一般包含数十万至数百万个元素。这里面既包含位姿估计的关键信息，也包含大量冗余信息。如何高效利用视觉图像信息估计位姿，是视觉里程计的核心问题。目前，从利用图像信息的方式来看，主流的 VO 可以分为四大类：**间接法 VO**、**直接法 VO**、**端到端 VO**、**基于渲染的 VO**。

间接法 VO 将图像信息转化为某种中间表示，再从中间表示中求取运动。这种中间信息一般有助于快速将不同图像中的同一物体进行匹配，最常见的是**特征点法**与**光流法**。借助它们，间接法 VO 用某种方式求解运动的最优估计，以**特征点重投影误差**作为优化目标。关于间接法 VO 的具体介绍，详见第27章。

特征点就是图像中较容易与其他像素区分并匹配的点，它们一般对应空间中物体的角点或边缘点。特征点法先从图像中寻找角点或边缘点，获取其像素坐标 (2D)；再通过不同位姿图像中的匹配点，恢复特征点的空间坐标 (3D)。这些 3D 坐标就可以成为相机位姿估计的重要依据。关于特征点的具体介绍，详见下一章。

光流是物体相对相机运动在成像平面的投影构成的平面场。目前已有不少成熟方法可以求解相邻两帧间的光流，继而估计两帧的相对运动。相比于特征点法，光流法可以减少一部分匹配的计算量。关于光流的具体介绍，详见下一章。

不同于间接法，**直接法 VO** 直接使用图像中的光度信息及其梯度，进行相机位姿估计。直接法 VO 一般也求解优化问题，优化目标包括**光度误差**和深度误差等。有一些方法同时结合直接法和间接法的特点，称为**半直接法 VO**。关于直接法和半直接法的具体介绍，详见第28章。

端到端 VO 是指基于深度神经网络的视觉里程计。它将视觉里程计问题直接建模为一个拟合问题，神经网络以光度图像、IMU 等 VO 问题的输入为输入，以 VO 的输出即位姿 T 为输出，基于大量数据进行训练。除此之外，还有一类方法整体上仍然采用间接法或直接法的框架，但使用神经网络优化重要中间表示估计，如使用神经网络估计特征点、光流、深度图等。

基于渲染的 VO 是近年来新兴的一类 VO 方法。这种方法由于其场景表示十分便于新视角渲染，且可以较低成本表达复杂而精细的场景，从而获得了广泛的关注，尤其是 AR/VR/CG 等领域的关注。这类方法的理论基础是**可微分渲染**技术，主要分为 **NeRF**(Neural Radiance Field) 和 **3DGS**(3D Gaussian Splatting) 两个分支。目前，基于 3DGS 的 SLAM 方法由于其表达显式、渲染资源消耗小而尤其受到关注。关于基于渲染的 VO 的具体方法介绍，详见第29章。

相对于端到端 VO 和基于渲染的 VO，直接法 VO 和间接法 VO 也称为传统方法 VO。目前在移动机器人中，间接法 VO 的应用最为成熟和广泛。在本章及后续章节中，我们将主要介绍间接法 VO、直接法 VO 和基于渲染的 VO。

24.1.4 滑窗位姿优化

对 VO 来说，无论使用何种方法从图像中获取位姿信息，其初步估计的位姿都存在一定的**误差**。这种误差可能有各种来源，如不准确的相机参数、错误的特征点匹配、非凸优化中的局部极小值、IMU 等各类传感器的噪声等等。

由于视觉里程计的位姿估计是序贯的，此前轨迹中的误差会传递到后续估计，即**误差累积效应**。此外，对于非 RGBD 的 VO，环境中的 3D 信息 (深度信息) 是从 2D 信息中恢复的，深度恢复的误差也会传递到位姿估计中。如果任由误差积累，对位姿的估计就会很快发散。

针对这一问题，一方面我们可以从误差产生的原因出发，使用更准确的传感器校准、外点剔除等方法 (后续章节将具体介绍)。另一方面，我们可以基于 3D 视觉的基本原理，建立**联合优化问题**。即：将多个历史位姿及其同时观察 (即**共视**) 的 (3D) 视觉特征同时作为优化变量，通过相机模型等关系建立优化目标。通过求解这一联合优化问题，我们就可以获取更精准的历史位姿。

这样的思路带来了一个问题：算法在实时更新当前位姿的过程中，必须也保留历史位姿和 (3D) 视觉特征；随着轨迹的延长，历史位姿和视觉特征会越来越多，优化变量将变得越来越臃肿复杂。

为解决这一问题，我们引入**滑动窗口**思想。滑动窗口是一种队列，新的位姿和特征不断进入窗口；当窗口到达上限，就以一定的策略去除一些老旧的位姿和特征，从而保持优化问题的规模不随着轨迹的延长而无限增加。此外，为避免长时间静止带来的观测冗余，滑动窗口仅包含相互差异较为明显的图像帧，称为**关键帧**。

滑窗优化思想广泛被各类具体的 VO 算法采用。不过，这也带来另一个问题，即**精度-速度的 trade-off**。一方面，为了得到更精确的历史位姿，滑窗包括的信息就越多，求解的问题规模就越大，消耗的资源就越多；另一方面，VO 需要对每一新图像帧进行位姿预测 (称为**跟踪**)，相机采集频率越高，跟踪需要的资源也越多。

为了从一定程度上解决这一问题，主流 VO 方法均采用分离原则，将**新位姿跟踪**和**历史位姿优化**两部分分离，分别称为**前端**和**后端**。在工程实现上，一般也运行于两个不同的线程，称为前端线程和后端线程。利用现代 CPU 的多核并行处理特性，前后端分离可以一定程度上同时兼顾精度和速度。

不过，" 前端" 与" 后端" 的界限并没有一个公认的标准。一些间接法算法中的" 前端" 可能仅包含图像数据处理 (即到特征点匹配为止)，不包括位姿跟踪；另一些算法的" 前端" 可能会输出跟踪位姿。在介绍各类具体 VO/vSLAM 算法时，我们会以每个算法各自的划分为准。

如前面所介绍的，VO 的前端包括直接法、间接法等不同实现方法。其实，VO 的后端也包含不同的实现方法，体现了对 VO 问题的两种不同理解。

一种理解认为，VO 本质上就是一个优化问题，历史位姿就是优化变量。后端应当通过各种观测构造联合优化目标，随后使用优化算法求解优化问题。基于这种后端的 VO 方法，我们称为**基于优化的 VO**。

另一种理解认为，VO 是一个多传感器融合的去噪状态最优估计问题。应当对这个序贯过程建立滤波器，通过各类传感器的观测对状态估计进行修正。因此，我们可以使用第11章介绍的最优滤波方法 (一般是各类卡尔曼滤波) 进行处理，我们称为**基于滤波的 VO**。

无论基于滤波还是基于优化，VO 本质上都是在进行某种意义上的最优状态估计。在本部分的后续章节中，我们介绍的 VO 方法既有基于滤波的方法，也有基于优化的方法。

24.1.5 同时定位与建图

上一小节中我们提到，为缓解 VO 的累计误差问题，我们需要以滑窗形式记录历史位姿及对环境的 3D 观测，使用 VO 后端进行联合优化。这个过程中，历史位姿对环境的 3D 观测信息，实际上就构成了一种对 3 维环境的描述，或一种**地图**。

从 VO 问题来看，3D 地图是求解问题过程中的副产物。然而，获取地图本身也存在一些价值。一方面，我们有时候就是想通过一系列 2 维的图片，得到实际物体/环境的 3 维模型。这一问题称为**三维重建**。另一方面，VO 的跟踪依赖于当前图像和历史信息 (滑窗信息) 的匹配；如果由于某些原因 (如突然的遮挡、大幅度运动) 使某帧图像和历史信息无法匹配，跟踪算法就无法继续工作。我们将这一问题称为**跟踪丢失问题**。

VO 中，一旦跟踪丢失，恢复后的图像就可能和滑窗图像存在较大区别。如果区别较大，无法和滑窗进行匹配，那么整个算法就需要重新初始化。然而，如果我们可以将滑窗外的环境信息也存储记录下来，在丢失跟踪后不是与滑窗而是与全地图的信息进行匹配，就可能在不中断的情况下，恢复当前相机的跟踪。这一过程称为**重定位**。

在三维重建和重定位等需求下，建立地图变得和位姿估计同样重要。事实上，既通过传感器信息进行位姿估计，又在这一过程中建立环境的地图，这一过程、方法或问题就称为**同时定位与建图** (Simultaneous Localization and Mapping, SLAM)。通过视觉传感器进行 SLAM，称为**视觉 SLAM**(visual SLAM, 简称 vSLAM)。

惯例上，vSLAM 系统也区分为前端和后端。一般前端就是指当前相机位姿的跟踪模块；后端一般分为以下部分：**全局优化、重定位和回环检测**。前端对实时性的要求很高，后端可以以较长的时间代价运行一些复杂度较高的计算。

在后端中，联合优化、重定位的含义上文已经描述过，不再赘述。**回环检测**是指：当相机运动轨迹形成闭环，可能会重新观测到此前已观测到的环境特征。此时通过一定的方法，对前后两次观测进行匹配，就可以修正环路中的所有位姿，从而大大减少累积误差。事实上，对于包含回环检测的 vSLAM 系统而言，通过人为或自主方法，使其路径中包含闭环非常重要。

VO 和 vSLAM 的概念之间并没有泾渭分明的分界线，它们有时存在一定程度的混用情况。事实上，其区分往往是根据目的和用途。在本书的后续章节中，如果一个 VO 算法包含重定位和回环检测，或者我们关注全局地图的生成，我们就称该算法为 vSLAM 算法；否则我们称其为 VO 算法。

对于 vSLAM 建立的地图，一般认为包含以下几类：稀疏地图、半稠密/稠密地图、隐式地图。**稀疏地图**也称为稀疏点云，是由 3D 特征点直接构成的地图。其点和点的间距一般较大，对于环境中比较均一的区域，特征点会非常稀疏。**稠密地图**是几乎连续的 3D 点构成的地图，要求其中的点也包含颜色信息。稠密地图可以用于新视角的合成 (即渲染)，也可以转化为面元 (mesh)、体素 (volume) 等其他 3D 模型格式。**半稠密地图**介于稀疏地图和稠密地图二者之间，指的是物体边缘处稠密、光度平坦处稀疏的地图。

隐式地图是基于渲染的 VO 使用的 3D 空间表示方法。这种方法不依赖于 3D 点，而是将三维空间建模为不同小块的不透明度和颜色，从而有效实现**可微分渲染**，简化新视角合成，避免传统 CG 中的复杂渲染管线 (pipeline)。目前隐式地图主要是 **3DGS 地图**，详见第29章。

VO/vSLAM 领域经过多年发展，不同的方法、技术、概念较多，且相互间关系比较复杂。本书将挑选几种具有代表性的 VO/vSLAM 算法，在以下的章节中进行详细介绍。

作为本节的结束，我们列举这几种方法对应的传感器类型、图像利用方法 (前端类型)、联合优化方法 (后端类型)。读者可以通过该表清晰地了解后续章节和本节介绍的概念之间的关系。

在本章的后续节中，我们将介绍相机模型这一 VO 基石，并介绍间接法中的特征点和光流相关的处理方法。

表 4: 重要 VO/vSLAM 方法及其特性

方法	传感器类型	前端方法	后端类型	地图类型
MSCKF	RGB+IMU	点特征	滤波	稀疏地图
VINS-Fusion	RGB+IMU	点特征	优化	稀疏地图
DSO	RGB	直接法	优化	稀疏地图
MonoGS	RGB	基于渲染	3DGS	3DGS 地图

24.2 相机模型

相机是 vSLAM/VO 使用的主要传感器。相机的本质是阵列排列的光电传感器，称为感光芯片。感光芯片可以以**像素**为单位测量光的强度，并将其转化为一定的数值。相机的基本参数是感光芯片上横向和纵向的像素数，分别称为 Width(W) 和 Height(H)。

相机得到的数据称为**图像**。图像中某个点对应的横纵像素编号称为**像素坐标**。我们一般以图像左上角为原点，右方和下方为正方向，建立**像素坐标系**。像素坐标系的单位为 1 像素，即 1pix。像素坐标一般以 $[u,v]^T$ 表示，有时也以增广的形式 $[u,v,1]^T$ 表示。在增广表示下，我们记

$$\upsilon = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{VII.24.2}$$

除了像素坐标系，我们还常常建立与相机固联的三维坐标系：**相机坐标系**。相机光路中的所有光线汇聚点称为**光心**。相机坐标系一般是以光心为原点，以相机指向的前方为 z 轴，以感光芯片横向右侧为 x 轴，按照右手定则建立的坐标系。相机坐标系一般表示为 c 系。

相机的成像是一个复杂的光学过程，经过设计的镜头透镜组将外界某个点发出的光线汇聚到焦平面 (即感光芯片所在的平面) 上。根据不同的变换关系，相机镜头分为广角镜头、鱼眼镜头、长焦镜头等等。在视觉 SLAM 中，常用的镜头为鱼眼和广角镜头。为简单起见，我们接下来仅讨论广角镜头。

镜头对光线作用的数学模型称为**相机模型**。对于广角镜头，其成像过程可近似为**小孔成像模型**，即像平面的光线经过相机光心处的小孔后，在焦平面形成倒立的实像。为简单起见，我们将这一光心后方的倒立实像，等效为光心前方同样距离的正立像。这样，相机坐标系的 x,y 轴可以与像素坐标系的 u,v 轴方向保持一致。

在上述建模下，小孔成像相机模型可以表示为

$$\upsilon = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K\nu^c = \frac{1}{z^c}Kp^c = \frac{1}{z^c}K\begin{bmatrix} x^c \\ y^c \\ z^c \end{bmatrix} \tag{VII.24.3}$$

这里，我们记归一化坐标为 ν，即

$$\nu^c = \frac{1}{z^c}p^c = \frac{1}{z^c}\begin{bmatrix} x^c \\ y^c \\ z^c \end{bmatrix} \tag{VII.24.4}$$

其中，p^c 表示相机坐标系下，物体上某个点的坐标。$[u,v,1]^T$ 表示该点成像后，对应的 (增广) 像素坐标。K 称为**相机内参矩阵**，其组成如下所示

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{VII.24.5}$$

式中，c_x, c_y 是横向和纵向相机光心投影距离感光芯片左上角的像素数。理想的相机的 c_x, c_y 是 W, H 值的一半，实际一般是接近的数值。f_x, f_y 是横向和纵向的相机坐标系单位长度 (1m) 和像素坐标系单位 (1pix) 的比值。

实际的广角镜头成像过程中，还会产生**畸变**。畸变是指像点在像平面上和理想位置间的偏移，导致成像后的像失去保角、保直线等性质。畸变也可以用一组参数进行描述，在此不再赘述。

相机的内参和畸变参数仅由相机和镜头光路决定。可以使用一定的方法测量内参和畸变参数，称为**相机内参标定**。相机标定时，使用预先制作的、尺寸精确已知的标定板，配合角点检测和内参标定算法，一般是通过求解优化问题，得到内参参数。

相机一般固定于机器人载体上。习惯上，我们将机器人运动的右/前/上三个方向分别定义为载体系 (即 b 系) 的 $x/y/z$ 轴，或以载体上 IMU 的三个敏感轴为 b 系的 $x/y/z$ 轴；而相机的 z 轴一般指向前方。这样，就需要一个齐次矩阵 T_b^c 表示相机 c 系和载体 b 系间的相对位姿。矩阵 T_b^c 也称为**相机外参**。

如果 VO 系统包含 IMU 输入，可以通过一定的方法确定齐次矩阵 T_b^c，这一过程称为**相机外参标定**。相机外参标定一般依赖于内参标定的结果，其本质也是求解已知环境特征尺寸的优化问题。

在 VO 和 vSLAM 中，相机内参、畸变参数、外参等一般默认为已知，且默认图像已经过畸变补偿。不过在后续章节中，我们介绍的一些方法认为不准确的内参会影响 VO/vSLAM 精度；因此它们也将内参也作为优化变量之一，在算法中进行修正。

25 特征点与光流

在本章中，我们将讨论间接法图像利用的基本原理和方法，即特征点法与光流法。首先，我们将介绍数字图像相关的一些基本概念、符号和记法，随后我们分别介绍一些重要的特征点与光流方法。

25.1 数字图像基础

我们一般所说的**数字图像**是指：在计算机中，以浮点数/整数形式存储的、代表图像光度的矩阵。最常见的数字图像是 **RGB 图像**，即模仿人眼的三种视锥细胞、具有红绿蓝 3 个颜色通道的数字图像，其矩阵形状为 $H \times W \times 3$。在 VO 和 vSLAM 中，一般我们不需要使用颜色信息，此时我们使用的图像为**灰度图像**，其矩阵形状为 $H \times W$。

上述表达式中，H, W 代表图像的高度和宽度，和相机模型一节的定义一致。图像的像素坐标在宽度方向记为 u，高度方向记为 v。(u, v) 可以取整数，也可以为非整数。一般我们认为，$(u, v) \in [0, W] \times [0, H]$。

对于每张灰度图像 I，记 u, v 处的灰度值为 $I[u, v]$。一般取灰度值时默认 u, v 为整数；对非整数情况可以取双线性插值的结果。**灰度图像 I 是一种 2 维离散信号，可以认为是 2 维连续信号（连续函数）的采样。**即：对于每张灰度图像 I，都可以定义一个图像函数 $\mathcal{I}(x, y)$，使得

$$I[u, v] = \mathcal{I}(u, v)$$

对于连续函数 $\mathcal{I}(x, y)$，我们假设它二阶连续可导，并记其一阶和二阶偏导为

$$\begin{aligned}
\mathcal{I}_x(x, y) &:= \frac{\partial \mathcal{I}}{\partial x}(x, y) \\
\mathcal{I}_y(x, y) &:= \frac{\partial \mathcal{I}}{\partial y}(x, y) \\
\mathcal{I}_{xx}(x, y) &:= \frac{\partial^2 \mathcal{I}}{\partial x^2}(x, y) \\
\mathcal{I}_{yy}(x, y) &:= \frac{\partial^2 \mathcal{I}}{\partial y^2}(x, y) \\
\mathcal{I}_{xy}(x, y) &:= \frac{\partial^2 \mathcal{I}}{\partial x \partial y}(x, y)
\end{aligned}$$

以上定义都是连续域中的定义，而我们实际可以处理的图像都是离散的数字图像。我们定义**图像梯度** $I_u, I_v, I_{uu}, I_{vv}, I_{uv}$ 为相应图像函数一阶/二阶导数的离散采样。不过，在实际应用中，我们不会真的从数字图像恢复一个连续函数，而是使用图像卷积近似图像梯度[25]。我们有如下算法

$$\begin{aligned}
I_u[u, v] &:= \mathcal{I}_x(u, v) \approx I \star K_u \\
I_v[u, v] &:= \mathcal{I}_y(u, v) \approx I \star K_v \\
I_{uu}[u, v] &:= \mathcal{I}_{xx}(u, v) \approx I \star K_u \star K_u \\
I_{vv}[u, v] &:= \mathcal{I}_{yy}(u, v) \approx I \star K_v \star K_v \\
I_{uv}[u, v] &:= \mathcal{I}_{xy}(u, v) \approx I \star K_u \star K_v
\end{aligned} \tag{VII.25.1}$$

其中，K_u, K_v 为一阶微分卷积核，此处我们使用了 **Sobel 算子**

[25]我们假设读者已经有图像处理的基本知识，不再详述卷积操作等基本运算；且忽略 padding 等细节

$$
K_u = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \\
K_v = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{VII.25.2}
$$

根据偏导数，我们可以定义图像梯度 ∇I

$$
\nabla I[u,v] := \nabla \mathcal{I}(u,v) = \begin{bmatrix} \mathcal{I}_x(u,v) \\ \mathcal{I}_y(u,v) \end{bmatrix}
$$

对于二元函数，其每个点的梯度是一个 2 维向量。我们将其分为模长和方向角，这两部分都可以使用上面的偏导数近似求解

$$
\begin{aligned} ||\nabla I|| &= \sqrt{I_u \odot I_u + I_v \odot I_v} \\ \theta_{\nabla I} &= \arctan\left(I_u \oslash I_v\right) \end{aligned} \tag{VII.25.3}
$$

式中，$\odot$ 表示矩阵逐元素相乘，$\oslash$ 表示矩阵逐元素相除。除一阶微分和梯度外，我们还可以使用二阶偏导计算图像的二阶 Laplace 算子 ΔI，即

$$
\Delta I[u,v] := \Delta \mathcal{I}(u,v) = \mathcal{I}_{xx}^2(u,v) + \mathcal{I}_{yy}^2(u,v)
$$

其近似计算为

$$
\Delta I = I_{uu}^2 + I_{vv}^2 = I \star K_L \tag{VII.25.4}
$$

其中，卷积核 K_L 有两种取法

$$
K_{L1} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \\
K_{L2} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{VII.25.5}
$$

25.2 特征点基本概念

在上一章中我们提到，不同 VO 有不同的利用图像的方法。间接法 VO 的本质，就是使用特征点/光流等中间表示来提取两帧图像的**匹配关系**。只有根据匹配关系，我们才能估计出两帧间相机的运动。如果两帧图像没有任何匹配点，那么运动估计就无从谈起。本节将介绍的特征点方法，以及下一节介绍的光流法，都是从图像中寻找匹配关系的方法。

所谓**特征点**，就是图像中比较特殊的点，一般是特定物体的边缘点和角点，它们在整个图像中具有一定的代表性，便于进行检测和匹配。对于一幅一般的图像，在数百万个像素点中，一般只有几百至几千个点符合这样的要求。

VO 或 vSLAM 的本质，是从 2D 图像中恢复 3D 信息。因此，在 VO 或 vSLAM 中的特征点一般具有两个含义：2D 图像中的代表性点 (也称为 **2D 特征点**)、3D 空间中的代表性点 (也称为 **3D 特征点**)。2D 特征点是直接从图像中获取的，从单张图像中寻找 2D 特征点的过程就称为**特征点检测**。在本章及后续章节中，2D 特征点有时特指一幅图像中，一组 2D 特征点的像素坐标；相应的，3D 特征点也特指 3 维空间中，一组 3D 特征点的 3 维坐标。

2D 特征点的匹配关系非常重要，是特征点法的核心。这是因为，如果两帧图像观测的是同一 3D 空间，那么 3D 空间中的特征点应该分别在两张图像中，各自对应一个 2D 特征点。如果我们找到了这样的匹配关系，又知道了两帧图像的位姿，就可以恢复出 3D 特征点的坐标。反之，如果我们知道 3D 特征点的坐标，也能恢复两帧图像的位姿关系。这一切都基于 2D 特征点的匹配关系。给定两帧图像及各自检测的 2D 特征点，求解特征点匹配关系的过程，就称为**特征点匹配**。**特征点的检测和特征点的匹配是特征点法前端中的两大核心问题**。

问题	**特征点检测问题**
问题简述	已知单帧图像，求图像中有代表性点的像素坐标
已知	图像 I
求	特征点像素坐标 $v_1, ..., v_n$

问题	**特征点匹配问题**
问题简述	已知两帧图像及其特征点，求特征点匹配关系
已知	图像 I_1, I_2 图 1 的 2D 特征点 $v_1^1, ...v_i^1, ..., v_m^1$ 图 2 的 2D 特征点 $v_1^2, ..., v_j^2, ..., v_n^2$
求	匹配的下标对 $(i_1, j_1), ..., (i_k, j_k)$

对于特征点匹配问题，为了求解匹配关系，我们需要知道特征点区别于其他特征点的特质是什么。无论是物体边角处还是清晰纹理处的特征点，其邻域的像素都与其他点的邻域像素差别很大。因此，我们往往基于特征点的邻域像素，构造一种特征点的描述；再根据这种描述的异同进行匹配。这种描述就称为**特征点描述子**。

下面，我们将介绍常用的特征点检测和特征点匹配方法。

25.3 Harris 角点

如前所述，对于特征点检测问题，我们希望能找到图像中具有代表性、清晰易识别的点。更具体的来说，我们希望找**物体边角或清晰纹理**处的图像点。下面我们介绍 **Harris 角点**的方法。

25.3.1 滑动误差

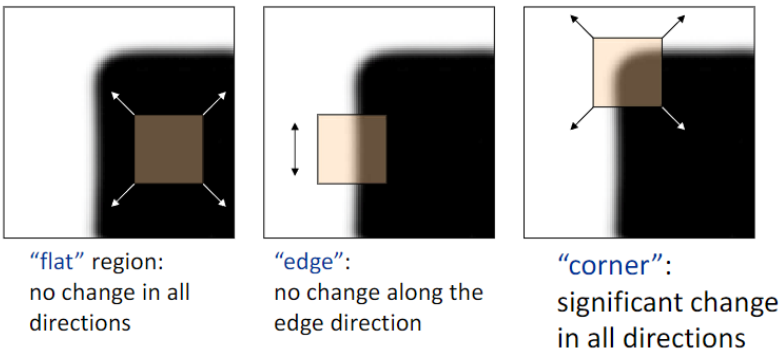

图 VII.25.1: Harris 角点原理

如图VII.25.1所示，假设在某点周围取一个邻域窗口，对邻域中的像素点进行某种加权平均。我们希望寻找的特征点往往满足如下性质：如果该点为角点，则向各个方向滑动窗口时，加权平均值的变化都比较剧烈；如果该点为边缘点，则沿边缘滑动时加权平均值变化较小，而垂直边缘滑动时加权平均值变化较大。

具体来说，在像素点 (u_0, v_0) 周围，我们设计如下滑动误差指标，作为滑动矢量 $(\delta u, \delta v)$ 的函数

$$E[u_0, v_0](\delta u, \delta v) := \sum_{(u,v)\in W} w(u,v)(I[u_0, v_0] - I_{\overrightarrow{\delta u, \delta v}}[u_0, v_0])^2 \tag{VII.25.6}$$

其中，$I_{\overrightarrow{a,b}}$ 表示平移 (a, b) 后的图像，其定义为

$$I_{\overrightarrow{a,b}}[u, v] = \mathcal{I}(u + a, v + b)$$

W 表示加权平均过程中的滑窗的范围。滑窗可以看作是一种卷积，如果卷积核边长为 k，则可以设置 $W = [-\frac{k+1}{2}, \frac{k+1}{2}] \times [-\frac{k+1}{2}, \frac{k+1}{2}]$

对上述平移后图像，在连续域中，我们有一阶泰勒展开近似

$$\mathcal{I}(u + a, v + b) \approx \mathcal{I}(u, v) + \frac{\partial \mathcal{I}}{\partial x}(u, v)a + \frac{\partial \mathcal{I}}{\partial y}(u, v)b$$

因此，在离散域中，我们也有

$$I_{\overrightarrow{a,b}}[u, v] \approx I[u, v] + aI_u[u, v] + bI_v[u, v]$$

即

$$\begin{aligned} E[u_0, v_0](\delta u, \delta v) &= \sum_{(u,v)\in W} w(u,v)(I[u_0, v_0] - I_{\overrightarrow{\delta u, \delta v}}[u_0, v_0])^2 \\ &= (((I - I_{\overrightarrow{\delta u, \delta v}}) \odot (I - I_{\overrightarrow{\delta u, \delta v}})) \star K_w)[u_0, v_0] \\ &\approx (((\delta u I_u + \delta v I_v) \odot (\delta u I_u + \delta v I_v)) \star K_w)[u_0, v_0] \\ &= (\delta u^2 I_u \odot I_u \star K_w + \delta v^2 I_v \odot I_v \star K_w + 2\delta u \delta v I_u \odot I_v \star K_w)[u_0, v_0] \end{aligned}$$

设

$$
\begin{aligned}
A &= I_u \odot I_u \star K_w \in \mathbb{R}^{H\times W} \\
B &= I_v \odot I_v \star K_w \in \mathbb{R}^{H\times W} \\
C &= I_u \odot I_v \star K_w \in \mathbb{R}^{H\times W}
\end{aligned} \tag{VII.25.7}
$$

则有

$$
\begin{aligned}
E[u_0, v_0](\delta u, \delta v) &\approx (\delta u^2 A + \delta v^2 B + 2\delta u \delta v C)[u_0, v_0] \\
&= \delta u^2 A[u_0, v_0] + \delta v^2 B[u_0, v_0] + 2\delta u \delta v C[u_0, v_0] \\
&= \begin{bmatrix} \delta u & \delta v \end{bmatrix} \begin{bmatrix} A[u_0, v_0] & B[u_0, v_0] \\ B[u_0, v_0] & C[u_0, v_0] \end{bmatrix} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix}
\end{aligned}
$$

再设

$$
H[u_0, v_0] = \begin{bmatrix} A[u_0, v_0] & B[u_0, v_0] \\ B[u_0, v_0] & C[u_0, v_0] \end{bmatrix} \tag{VII.25.8}
$$

则有

$$
E[u_0, v_0](\delta u, \delta v) \approx \begin{bmatrix} \delta u & \delta v \end{bmatrix} H[u_0, v_0] \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \tag{VII.25.9}
$$

可见，滑动误差 $E[u_0, v_0](\delta u, \delta v)$ 可近似为矩阵 $H[u_0, v_0]$ 的二次型。更进一步，对 $H[u_0, v_0]$ 进行 SVD 分解，有

$$
H[u_0, v_0] = R^T \begin{bmatrix} \lambda_1 & \\ & \lambda_2 \end{bmatrix} R
$$

其中，R 为正交矩阵，λ_1, λ_2 为矩阵 $H[u_0, v_0]$ 的两个特征值。$H[u_0, v_0]$ 的特征值决定了滑动误差 $E[u_0, v_0]$ 的变化情况。

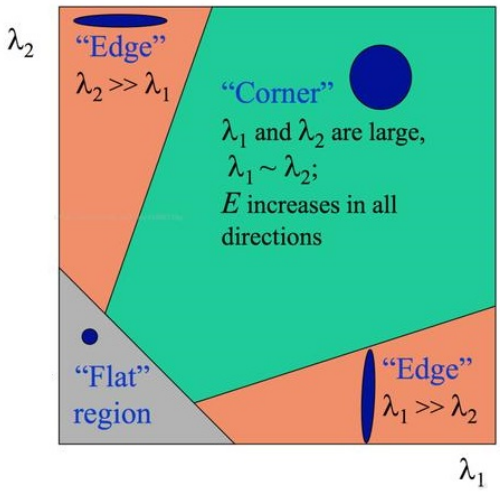

图 VII.25.2: Harris 角点的 λ 值

如图VII.25.2所示，λ_1, λ_2 越大，意味着在相同 $(\delta u, \delta v)$ 下，滑动误差 $E[u_0, v_0](\delta u, \delta v)$ 的变化越大；对照前述讨论，意味着 (u_0, v_0) 是角点。如果 λ_1, λ_2 一大一小，意味着仅在沿某一方向滑动时滑动误差变化较大；对照前述讨论，意味着 (u_0, v_0) 是边缘点。这个变化较大的方向其实就是垂直于边缘的方向，可以由 R 的特征向量给出。

25.3.2 算法流程

综合上述分析，我们的目标就是找到所有 $H[u_0, v_0]$ 中，特征值最大的那些，它们对应我们需要的图像角点。不过，由于有两个特征值，这个标准不好量化。我们可以进一步定义关于两个特征值的指标

$$\begin{aligned} r(u_0, v_0) &:= \lambda_1 \lambda_2 - \alpha(\lambda_1 + \lambda_2)^2 \\ &= \det H[u_0, v_0] - \alpha \operatorname{tr}(H[u_0, v_0]) \end{aligned}$$

如上所述，当 (u_0, v_0) 为边缘点时，有 $\lambda_1 >> \lambda_2 \approx 0$ 或 $\lambda_2 >> \lambda_1 \approx 0$，即图VII.25.2中的橙色区域，此时 $r(u_0, v_0)$ 将是一个绝对值较大的负值；当 (u_0, v_0) 为角点时，有 $\lambda_1 \approx \lambda_2 >> 0$，即图VII.25.2中的绿色区域，此时 $r(u_0, v_0)$ 将是一个绝对值较大的正值；而如果 (u_0, v_0) 附近较为平坦，则 λ_1, λ_2 都将较小，$r(u_0, v_0)$ 将接近 0。

由此，只要计算出每个点对应的 $r(u_0, v_0)$，即可很容易地判别该点为边缘点/角点/平坦点。我们可以直接从 A, B, C 计算 r：

$$r(u_0, v_0) = A[u_0, v_0]C[u_0, v_0] - B^2[u_0, v_0] - \alpha(A[u_0, v_0] + C[u_0, v_0]) \tag{VII.25.10}$$

所有的 $r(u_0, v_0)$ 组成一个矩阵，记为 R_H。在视觉 SLAM 中，相比边缘点，角点一般更加稳定。因此，我们只需要找出 R_H 的极大值，它们对应的 (u_0, v_0) 即为特征点（角点）。该过程中还需要进行非局部极大值抑制 (Non-Maximum Suppression, NMS)，以防特征点”扎堆”。该算法 *get_topn_NMS* 的实现方式在下一小节中介绍。

综合上述推导，我们得到 Harris 角点检测算法

Algorithm 78: Harris 角点检测 (Harris_FP)

Input: 图像 I

Parameter: 滑窗卷积核 K_w, α

Parameter: NMS 区域大小 k，数量 n

Output: 特征点像素坐标 $v_1, ..., v_n$

$I_u, I_v \leftarrow I \star K_u, I \star K_v$

$A \leftarrow I_u \odot I_u \star K_w$

$B \leftarrow I_v \odot I_v \star K_w$

$C \leftarrow I_u \odot I_v \star K_w$

$R_H \leftarrow A \odot C - B \odot B - \alpha(A + C)$

$v_{1:n} \leftarrow \text{get\_topn\_NMS}(R_H, n, k)$

算法	**Harris 角点检测**
问题类型	特征点检测问题
已知	图像 I
求	特征点像素坐标 $\upsilon_1, ..., \upsilon_n$
算法性质	传统 CV

25.3.3 NMS 算法

[本部分内容将在后续版本中更新，敬请期待]

上面介绍的 Harris 角点计算简单、定义清晰，但缺乏尺度不变性 (not scale invariant)。下面介绍的几种特征点检测和描述方法都兼顾了尺度不变性。

25.4 SIFT 特征点

所谓**尺度**，就是图像中某个特征区域的像素宽度。同一 3D 物体在不同的 2D 视角下，会呈现不同尺度的投影。如上节所述，Harris 角点的问题在于缺乏" 尺度不变性"。也就是说，对于同一物体不同尺度投影，Harris 方法无法同时检测其角点。

事实上，我们可以从理论上分析该问题的原因。注意到 Harris 角点的 H 阵仅由 I_u, I_v 以及 K_w 决定，而 I_u, I_v 在前述 Sobel 算子下仅能对 1-2 像素宽度的边缘作出响应。另一方面，K_w 也是固定值。因此，Harris 角点算法能响应的特征尺度是确定的，天然无法检测不同尺度的特征。

1998 年，Lowe 提出了**尺度不变特征变换** (SIFT, Scale Invariant Feature Transform)，即 **SIFT 特征点**。SIFT 特征点是一种尺度空间特征点，每个 (2D) 特征点的参数除了其像素坐标外，还包含对应的尺度信息，有助于更好地在不同投影上匹配特征点。

SIFT 特征点是一种基于**特征图**的特征点，即使用某种方式得到图像空间的一个函数，函数局部极大值的位置就对应某种明显的特征。SIFT 特征点使用了不同尺度的高斯差分卷积核对全图进行卷积，其特征图称为 **DoG 特征金字塔**，可以有效识别图像中的近似椭圆形物体特征 (blob 特征)。此外，该方法还提出了尺度/方向不变的 **SIFT 描述子**。

接下来，我们首先介绍 DoG 特征金字塔，再介绍如何从该金字塔中提取不同尺度的特征点，最后介绍如何对特征进行描述。

25.4.1 多尺度 DoG 算子

SIFT 希望使用**高斯拉普拉斯 (Laplacian of Gaussian, LoG) 卷积核**检测特征，它是一种对圆形特征有针对性响应的卷积核。LoG 核的定义为

$$\Delta\mathcal{G}(x, y; \sigma) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right] e^{-\frac{x^2+y^2}{2\sigma^2}}$$

其中，Laplacian 算子 Δ 的定义已在" 数字图像基础" 一节介绍。如图VII.25.3所示，二维连续域下的 LoG 核是一个四周突起的漏斗形，对某种圆形特征有最大响应。这种最大响应的圆形半径和使用的 LoG 核标准差 σ 成正比。因此，我们也将 LoG 卷积核的标准差称为" 特征尺度" 或" 尺度"。

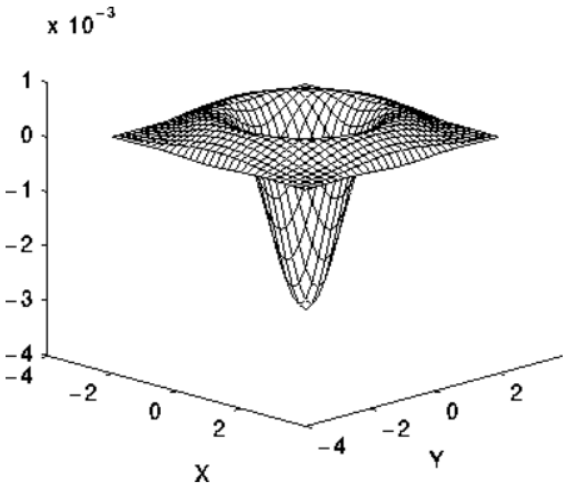

图 VII.25.3: LoG 的形状

LoG 卷积核对原图的卷积结果称为 **LoG 特征图**

$$\mathrm{LoG}[\mathcal{I}](x,y) = \mathcal{I}(x,y) \star \Delta\mathcal{G}(x,y;\sigma)$$

LoG 的卷积计算较为复杂，但单纯的高斯函数卷积比较简单。事实上，LoG 可以通过两个不同标准差的高斯核的差分，即**高斯差分 (Difference of Gaussian, DoG) 卷积核**进行近似

$$\Delta\mathcal{G}(x,y;\sqrt{k-1}\sigma) \approx \mathcal{G}(x,y;k\sigma) - \mathcal{G}(x,y;\sigma)$$

上式中 $k > 1$[26]。这样，我们的 LoG 特征图也可以通过 DoG 特征图来近似

$$\mathrm{LoG}[\mathcal{I}](x,y) \approx \mathrm{DoG}[\mathcal{I}](x,y) = \mathcal{I}(x,y) \star \mathcal{G}(x,y;k\sigma) - \mathcal{I}(x,y) \star \mathcal{G}(x,y;\sigma)$$

因此，如果我们想要获取不同尺度的 LoG 特征图，我们就可以先计算一系列 Gauss 模糊，然后进行两两差分，如图VII.25.4所示。

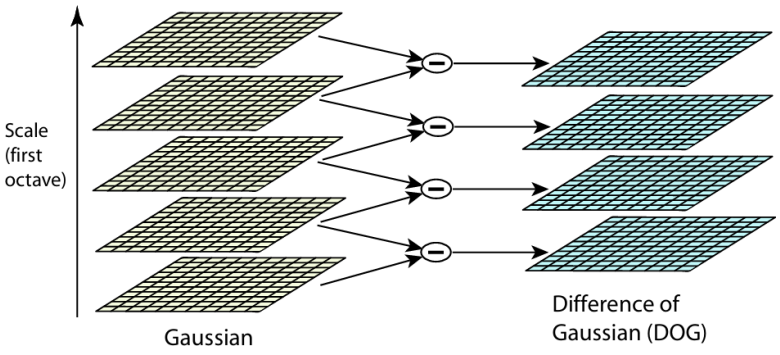

图 VII.25.4: DoG 近似多尺度 LoG

[26]在 SIFT 特征点中，我们往往选取 $k = \sqrt{2}$，原因在下一小节中介绍

为简单起见，我们希望 LoG 特征图的尺度呈等比数列，也就是对应的 DoG 的尺度也呈等比数列，也就是用于差分的高斯核的尺度也呈等比数列。具体来说，假设最小的高斯核标准差为 σ_0，比例系数为 k，总数为 n，则有

$$\mathcal{I}_i(x,y) := \mathcal{I}(x,y) \star \mathcal{G}(x,y;k^{i-1}\sigma_0), \quad i=1,...,n$$

则各尺度的 DoG 为

$$\mathrm{DoG}_i[\mathcal{I}](x,y) = \mathcal{I}_{i+1}(x,y) - \mathcal{I}_i(x,y), \quad i=1,...,n-1 \tag{VII.25.11}$$

上述的计算已经可以得到一组不同尺度的 DoG 特征图，也就是近似 LoG 特征图。然而，这里有一个工程问题：σ 越大的高斯核，其卷积计算量也就越大。利用高斯函数的性质，该问题可以通过将大高斯核拆分为小高斯核来缓解

$$\mathcal{G}(x,y;\sqrt{\sigma_1^2+\sigma_2^2}) = \mathcal{G}(x,y;\sigma_1) \star \mathcal{G}(x,y;\sigma_2)$$

作为等比数列的高斯特征图，事实上我们不需要每次都对原图 $\mathcal{I}(x,y)$ 进行卷积，而只需要迭代地对上一尺度的特征图进行卷积，这样我们使用的卷积核可以相对小一些

$$\begin{aligned}\mathcal{I}_1(x,y) &= \mathcal{I}(x,y) \star \mathcal{G}(x,y;\sigma_0)\\ \mathcal{I}_i(x,y) &= \mathcal{I}_{i-1}(x,y) \star \mathcal{G}(x,y;\sqrt{k^2-1}k^{i-1}\sigma_0), \quad i=2,...,n\end{aligned} \tag{VII.25.12}$$

25.4.2 图像金字塔

上述的迭代方法，确实可以节省一些多尺度 LoG 特征图的计算量。然而，即使如此，其使用的卷积核尺寸 $\sqrt{k^2-1}k^{i-1}\sigma_0$ 仍然是随着 i 指数增长的。为了更彻底地解决这一问题，我们可以使用**降采样**技巧。

具体来说，降采样是指将大小为 $H \times W$ 的图像矩阵等比例缩小为一个大小为 $\rho H \times \rho W$ 的矩阵，其中 $0<\rho<1$。从连续函数的角度上，即

$$\mathbf{DS}_{1/\rho}[\mathcal{I}](x,y) = \mathcal{I}\left(\frac{x}{\rho},\frac{y}{\rho}\right)$$

对于离散域，记图像 I 的 ρ 倍降采样为 $\mathbf{DS}_{1/\rho}[I]$。对于 Gauss 模糊，在 ρ 倍降采样的图像上应用高斯核 $G(\sigma)$，相当于在原图上应用高斯核 $G\left(\frac{\sigma}{\rho}\right)$，随后再进行降采样

$$\mathbf{DS}_{1/\rho}[I] \star \mathcal{G}(\sigma) = \mathbf{DS}_{1/\rho}\left[I \star \mathcal{G}\left(\frac{\sigma}{\rho}\right)\right]$$

可以看到，降采样大大简化了大 σGauss 卷积的计算。事实上，对离散图像而言，最简单的降采样是 $\rho=\frac{1}{2}$ 的降采样，此时仅需对原图像的相邻 4 个像素取平均，作为新的像素。在无特殊说明的情况下，我们一般默认 $\rho=\frac{1}{2}$。

$$\mathbf{DS}_2[I][u,v] = \frac{1}{4}(I[2u,2v]+I[2u+1,2v]+I[2u,2v+1]+I[2u+1,2v+1])$$

这样，如果我们想得到大 σ 尺度的 LoG 特征图，我们仅需进行一些小尺度的 Gauss 卷积，再通过多次降采样，即可等效为更大核的模糊。

假设原图像 I 经过一系列 (如 $n-1$ 次)Gauss 模糊，形成一组 Gaussian 特征图 $I_{1,1},I_{1,2},...,I_{1,n}$；对该组特征图迭代地进行 $L-1$ 次 $\rho=1/2$ 降采样，形成 L 组尺寸不同的特征图，有

$$I_{l+1,i} = \mathbf{DS}_2[I_{l,i}], \quad l = 1, ..., L-1, i = 1, ..., n \tag{VII.25.13}$$

上述特征图组构成一个金字塔的结构，称为图像的 **Gauss 特征金字塔**。金字塔中，每组尺寸相同的特征也称为一个 **Octave**。在 Gauss 特征金字塔中，对每一 Octave 中的特征图两两差分，形成一组 DoG 特征

$$D_{l,i} = I_{l,i+1} - I_{l,i}, \quad l = 1, ..., L, i = 1, ..., n-1 \tag{VII.25.14}$$

多组不同尺寸的 DoG 特征，也组成一个金字塔，称为 **DoG 特征金字塔**，如图VII.25.5所示。

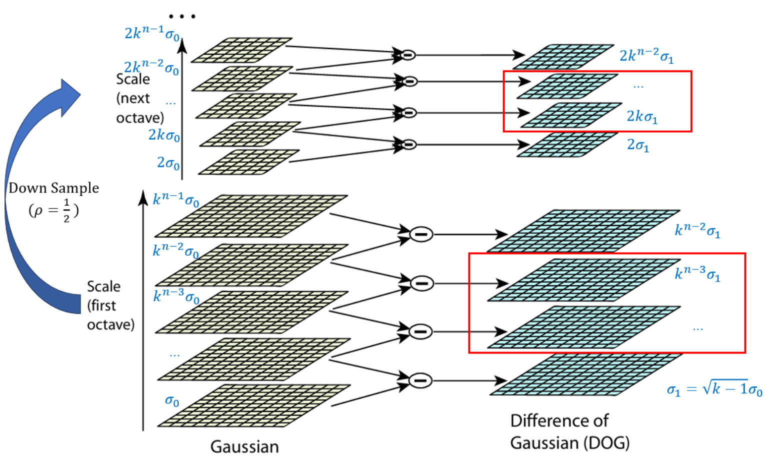

图 VII.25.5: DoG 特征金字塔

对于一个 DoG 特征金字塔，我们有如下参数：最小高斯核标准差 σ_0、相邻高斯核间的标准差倍数 k、每个 Octave 的 Gauss 特征数量 n、总 Octave 数量 L。由此，我们可以算出 $D_{l,i}$ 的等效 LoG 核的标准差 $\sigma_{l,i}$

$$\sigma_{l,i} = 2^{l-1}k^{i-1}\sigma_1, \quad l = 1, ..., L, i = 1, ..., n-1 \tag{VII.25.15}$$

其中 $\sigma_1 = \sqrt{k-1}$ 是最底层的 DoG 特征等效的 LoG 核的标准差。

在 DoG 特征金字塔中，并不是每层特征图都会用于生成 (初始) 特征点 (详见下一小节)。在每个 Octave 中，初始特征点仅会从最上层和最下层以外的其他特征图 (即 $D_{l,2}, ..., D_{l,n-2}$) 中生成，如图VII.25.5的红框所示。我们希望：所有 Octave 中，中间特征图的等效 LoG 核标准差构成一个等比数列，也就是

$$k\sigma_{l,n-2} = \sigma_{l+1,2}$$

这样，我们得到了一个关于 k 和 n 的方程。解方程，我们有 k 和 n 的关系

$$k = 2^{1/(n-3)} \tag{VII.25.16}$$

事实上，我们一般希望一个 Octave 中有 2 个中间 DoG 特征图。也就是希望 $(n-2)-2=1$，即取 $n=5$。在这样的情况下，就有 $k=\sqrt{2}$。

至此，我们已完成不同尺度 DoG 计算方法的推导。假设标准差为 σ 的离散高斯核 $G(\sigma)$ 已知，总结上述公式，我们有 **DoG 金字塔算法**

Algorithm 79: DoG 金字塔 (DoG_pyramid)

Input: 图像 I

Parameter: 初始核标准差 σ_0

Parameter: 金字塔层数 L，每层高斯核数 n

Output: DoG 特征图 $D_{1:L,1:n-1}$

$k \leftarrow 2^{1/(n-3)}$

$I_{1,1} \leftarrow I \star G(\sigma_0)$

for $i \in 2, ..., n$ **do**

  $I_{1,i} \leftarrow I_{1,i-1} \star G(k^{i-1}\sigma_0)$

for $l \in 2, ..., L$ **do**

  for $i \in 1, ..., n$ **do**

    $I_{l,i} \leftarrow \mathbf{DS}_2[I_{l-1,i}]$

for $l \in 1, ..., L$ **do**

  for $i \in 1, ..., n-1$ **do**

    $D_{l,i} \leftarrow I_{l,i+1} - I_{l,i}$

25.4.3 极大值提取

上面我们已经获取了 DoG 金字塔，即不同尺度的 LoG 特征图的近似。接下来，我们要从中寻找特征点的位置。

假设存在一个 3 维的特征空间，其三个维度分别对应图像的长、宽，以及 LoG 核的尺度，我们称为**尺度空间**。对于每张图像 $\mathcal{I}$，在尺度空间中都存在标量值函数 $f_{\mathcal{I}}(x, y, \sigma)$，表示该图像中心点为 (x, y) 处对尺度为 σ 的 LoG 核的响应强度，我们可以称其为**特征函数**。那么，DoG 金字塔的本质，就是特征函数 $f_{\mathcal{I}}(x, y, \sigma)$ 的一种近似与离散化。

如前所述，SIFT 特征点是一种基于特征图的特征点，其核心思想就是：**将特征函数局部极大值的位置作为特征点的位置**。DoG 金字塔是特征函数的离散化，因此特征函数局部极大值就近似于 DoG 金字塔中特征图的极大值。注意：由于特征函数是三维函数，极大值的求取也要同时考虑像素位置与尺度。

具体来说，在每个 Octave 内，SIFT 同时考虑相邻三张特征图 $D_{l,i-1}, D_{l,i}, D_{l,i+1}$，其组成一个 $\frac{H}{2^{l-1}} \times \frac{W}{2^{l-1}} \times 3$ 的 3 维矩阵。在这个矩阵中，对于每个 $D_{l,i}[u_0, v_0]$，如果其值为相邻 27 个格点中的局部极大值，则认为它就是该 Octave 中的局部极大值。这种算法求出的极值位置，仅存在于每个 Octave 中最上层与最下层之外的特征图中。

不过，上述假设是连续域的假设。我们获得的 DoG 金字塔仅仅是特征函数的离散采样矩阵 (并且随着尺度的增加，采样分辨率将越来越低)。如果仅仅寻找矩阵中的局部极大值，未必对应连续空间的极大值位置。

对此，我们可以使用**亚像素优化**技巧。具体来说，在上述方法搜索出的局部极大值附近，我们对 $f_{\mathcal{I}}(x, y, \sigma)$ 进行二阶泰勒展开近似，使用第3章介绍的牛顿法求解局部极值。牛顿法要求函数一阶和二阶导数已知，我们通过格点差分近似计算各导数值。经过几轮迭代，就可以求取更精确的、亚像素级别的极值点 (即特征点位置)。

在求出精确的特征点位置后，我们还需进行一步筛选，因为我们不希望提取图像的边缘，而仅希望提取角点。这里只需使用 Harris 角点相同的方法，求取 r 值即可。

综合上述介绍，我们可以总结 **SIFT 特征点检测算法**如下

算法	**SIFT 角点检测**
问题类型	特征点检测问题
已知	图像 I
求	特征点像素坐标 $v_1, ..., v_n$
算法性质	传统 CV

Algorithm 80: SIFT 角点检测 (SIFT_FP)

Input: 图像 I

Parameter: 初始核标准差 σ_0

Parameter: 金字塔层数 L，每层高斯核数 n

Output: 特征点像素坐标 $v_1, ..., v_n$

$D_{1:L,1:n-1} \leftarrow \text{DOG\_pyramid}(I; \sigma_0, L, n)$

$res \leftarrow []$

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 描述子

[主要内容：SIFT 描述子：梯度直方图 HoG；总结 SIFT 特征点匹配]

[本部分内容将在后续版本中更新，敬请期待]

25.5 ORB 特征点

25.5.1 FAST 特征点

[主要内容：FAST 角点；灰度阈值对比思想；4 点初筛/N 点判别；NMS 过程]

[本部分内容将在后续版本中更新，敬请期待]

25.5.2 BRIEF 描述子

[BRIEF 描述子；ORB 匹配算法；RANSAC]

[本部分内容将在后续版本中更新，敬请期待]

25.6 光流基本概念

如前所述，间接法 VO 要求快速准确地找到图像中的特征点并进行匹配。对于匹配问题，上面的几节中介绍的方法都依赖于描述子的计算与匹配，复杂度较高。相比之下，光流是一种无需计算描述子，就可完成点对匹配的方法。

光流 (Optic Flow) 是**物体相对相机的速度在相机平面投影构成的平面场**，是观测对象和观测者的相对运动的结果。

具体来说，假设观测者观测某些物体得到图像 $\mathcal{I}$。当观测值相对视野范围内的 (部分或全部) 物体产生运动，图像 $\mathcal{I}$ 就是时间和像素空间的函数 $\mathcal{I}(x,y,t)$。假设在极短时间 dt 内，视野内所有物体的亮度均不改变，即**光度不变假设**。则视野内每个点 (x,y) 处，物体相对相机的速度都将在相机平面上产生一个投影，这些投影构成的连续场 $\mathcal{V}(x,y,t)$ 即称为光流。

光流与图像梯度具有密切的关系。考虑同一空间中的 3D 点由于相机和目标相对运动产生像点偏移。具体来说，假设在时间 t 时，点 P 投影的像素坐标为 (p_x,p_y)；在时间 $t+dt$ 时，投影像素坐标为 (p_x+dp_x,p_y+dp_y)。根据光度不变假设，有

$$\mathcal{I}(p_x,p_y,t)=\mathcal{I}(p_x+dp_x,p_y+dp_y,t+dt)$$

假设 $\mathcal{I}$ 有较好的光滑性，对其进行一阶泰勒展开，有

$$\mathcal{I}(p_x+dp_x,p_y+dp_y,t+dt)\approx\mathcal{I}(p_x,p_y,t)+\mathcal{I}_x(p_x,p_y,t)dp_x+\mathcal{I}_y(p_x,p_y,t)dp_y+\mathcal{I}_t(p_x,p_y,t)dt$$

忽略小量并代入上式，有

$$\mathcal{I}_x(p_x,p_y,t)dp_x+\mathcal{I}_y(p_x,p_y,t)dp_y+\mathcal{I}_t(p_x,p_y,t)dt=0$$

即

$$\begin{bmatrix}\mathcal{I}_x & \mathcal{I}_y\end{bmatrix}(p_x,p_y,t)\begin{bmatrix}dp_x\\dp_y\end{bmatrix}=-\mathcal{I}_t(p_x,p_y,t)dt$$

由于光流定义为

$$\mathcal{V}[\mathcal{I}](p_x,p_y,t)=\begin{bmatrix}\frac{dp_x}{dt}(p_x,p_y,t)\\\frac{dp_y}{dt}(p_x,p_y,t)\end{bmatrix}\tag{VII.25.17}$$

因此有

$$(\nabla\mathcal{I})^T(p_x,p_y,t)\mathcal{V}[\mathcal{I}](p_x,p_y,t)=-\mathcal{I}_t(p_x,p_y,t)\tag{VII.25.18}$$

式VII.25.18也称为**光流方程**，它揭示了光流和图像梯度、图像光度变化之间的关系。

在实际工程中，我们需要在离散域中估计光流 $\mathbf{V}[u,v]$，也就是估计相邻两帧图像间 (相机和被观测对象) 相对运动的投影。这一问题称为**光流估计问题**。

问题	光流估计问题
问题简述	已知两帧图像，求离散光流
已知	图像 I_1,I_2
求	离散光流 $\mathbf{V}[u,v]$

在离散域下，光流是 $\delta u, \delta v$ 组成的两个矩阵，大小和图像 I_1, I_2 一致。

$$\mathbf{V}[u_0, v_0] = \begin{bmatrix} \delta u[u_0, v_0] \\ \delta v[u_0, v_0] \end{bmatrix} \tag{VII.25.19}$$

我们可以写出离散条件下的光流方程

$$(\nabla I)^T[u_0, v_0]\mathbf{V}[\mathbf{I}][u_0, v_0] = (I_2 - I_1)[u_0, v_0] \tag{VII.25.20}$$

在各类 VO/vSLAM 方法中，如果我们需要用到两帧图像光流的全部信息，称其为**稠密光流**；如果我们仅关注几个特殊点处的光流，称为**稀疏光流**。

对于间接法，我们使用光流的目的是给出特征点的匹配关系。假设我们有分别在图像 I_1, I_2 上检测的特征点像素坐标 $\upsilon^1_{1:m}, \upsilon^2_{1:n}$，又已知 I_1 到 I_2 的光流 $\mathbf{V}$，我们可以很容易地进行特征点匹配：我们只需在 I_1 的特征点处求解光流，将 υ^1 加上光流，随后在附近寻找 υ^2 即可。

算法	光流特征点匹配
问题类型	特征点匹配问题
已知	图像 I_1, I_2，光流 $\mathbf{V}$ 图 1 的 2D 特征点 $\upsilon^1_1, ...\upsilon^1_i, ..., \upsilon^1_m$ 图 2 的 2D 特征点 $\upsilon^2_1, ..., \upsilon^2_j, ..., \upsilon^2_n$
求	匹配的下标对 $(i_1, j_1), ..., (i_k, j_k)$
算法性质	传统 CV

Algorithm 81: 光流特征点匹配 (match_FP_optflow)

Input: 图像 I_1, I_2，光流 $\mathbf{V}$
Input: 图 1 的 2D 特征点 $\upsilon^1_{1:m}$
Input: 图 2 的 2D 特征点 $\upsilon^2_{1:n}$
Parameter: 搜索半径 r
Output: 匹配的下标对 $(i_1, j_1), ..., (i_k, j_k)$

```
pairs ← []
candidates ← [1, 2, ..., n]
for i ∈ 1, ..., m do
    υ̂ ← υ_i^1 + V[υ_i^1]
    r_min ← r + 1
    for j ∈ candidates do
        r_j ← ||υ_j^2 − υ̂||
        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)]
```

因此，间接法中使用的光流一般是稀疏光流。相应的，直接法使用的光流一般是稠密光流或半稠密光流。

25.7 L-K 光流

25.7.1 最小二乘求解

下面，我们考虑离散情况下的光流求解问题。根据式VII.25.20可知，对点 $[u_0, v_0]$ 处的光流，有两个变量 $\delta u, \delta v$，但仅有 1 个方程，条件不足。对此，我们可以假设 $[u_0, v_0]$ 周围的一个窗口 $W[u_0, v_0]$ 内，运动投影均相同。这样，我们就可以构造基于窗口的方程组

$$\begin{bmatrix} I_u[u_0, v_0] & I_v[u_0, v_0] \\ I_u[u_1, v_1] & I_v[u_1, v_1] \\ \cdots & \cdots \\ I_u[u_w, v_w] & I_v[u_w, v_w] \end{bmatrix} \mathbf{V}[u_0, v_0] = \begin{bmatrix} (I_2 - I_1)[u_0, v_0] \\ (I_2 - I_1)[u_1, v_1] \\ \cdots \\ (I_2 - I_1)[u_w, v_w] \end{bmatrix}, \quad (u_i, v_i) \in W[u_0, v_0]$$

这是一个超定的线性方程组，可以求最小二乘解。省略推导过程，我们直接给出结果：

$$\mathbf{V}[u_0, v_0] = H_{LK}[u_0, v_0]^{-1} g_{LK}[u_0, v_0] \tag{VII.25.21}$$

其中

$$\begin{aligned} H_{LK}[u_0, v_0] &= \sum_{(u_i, v_i) \in W} \begin{bmatrix} I_u^2[u_i, v_i] & I_u \odot I_v[u_i, v_i] \\ I_u \odot I_v[u_i, v_i] & I_v^2[u_i, v_i] \end{bmatrix} \\ g_{LK}[u_0, v_0] &= \sum_{(u_i, v_i) \in W} \begin{bmatrix} I_u \odot (I_2 - I_1)[u_i, v_i] \\ I_v \odot (I_2 - I_1)[u_i, v_i] \end{bmatrix} \end{aligned} \tag{VII.25.22}$$

该算法要求 $H_{LK}[u_0, v_0]$ 非奇异。值得注意的是，矩阵 $H_{LK}[u_0, v_0]$ 和此前"Harris 角点" 一节中的 $H[u_0, v_0]$ 极为相似，等效于"Harris 角点" 中 K_W 取平均核。因此，该方程存在非奇异解，基本等同于 u_0, v_0 取 Harris 角点。

上述基于窗口的最小二乘光流求解方法由 Lucas 和 Kanade 于 1981 年提出，因此也称为 **L-K 光流法**。

Algorithm 82: L-K 光流 (calc_optflow_LK)

Input: 图像 I_1, I_2
Parameter: 滑窗卷积核 K_w
Output: 离散光流 $\mathbf{V}$
$I_u, I_v \leftarrow I_1 \star K_u, I_1 \star K_v$
$A, B, D, E, F \leftarrow 0_{5 \times H \times W}$
$A, B, D \leftarrow (I_u \odot I_u) \star K_w, (I_u \odot I_v) \star K_w, (I_v \odot I_v) \star K_w$
$E, F \leftarrow (I_u \odot (I_2 - I_1)) \star K_w, (I_v \odot (I_2 - I_1)) \star K_w$
$\delta u \leftarrow D \odot E - B \odot F$
$\delta v \leftarrow A \odot F - B \odot E$
$H_{det} \leftarrow A \odot D - B \odot B$
$\mathbf{V} \leftarrow \begin{bmatrix} \delta u \oslash H_{det} \\ \delta v \oslash H_{det} \end{bmatrix}$

算法	L-K 光流
问题类型	光流估计问题
已知	图像 I_1, I_2
求	离散光流 $\mathbf{V}$
算法性质	传统 CV

25.7.2 迭代求解

上述的最小二乘光流算法依赖于图像函数的一阶泰勒展开近似，即默认运动 $(\delta u, \delta v)$ 较小。当运动较大时，该算法无法很好估计光流。

针对这一问题，我们可以使用迭代求解的方法。对于点 $[u_0, v_0]$，假设在迭代第 k 轮，我们已有光流估计 $\mathbf{V}_k[u_0, v_0]$。此时，我们将光流估计补偿到 I_1，即

$$I_{1,k}[u_0, v_0] = \mathbf{Sft}[I_1, \mathbf{V}_k] := I_1[u_0 - (\delta u)_k(u_0), v_0 - (\delta v)_k(v_0)] \tag{VII.25.23}$$

随后我们用上述方法求解 $I_{1,k}$ 到 I_2 的光流 $\delta\mathbf{V}_k$，随后进行迭代更新

$$\mathbf{V}_{k+1}[u_0, v_0] = \mathbf{V}_k[u_0, v_0] + \delta\mathbf{V}_k[u_0, v_0] \tag{VII.25.24}$$

由此，我们可以总结**迭代 L-K 光流算法**如下

算法	迭代 L-K 光流
问题类型	光流估计问题
已知	图像 I_1, I_2
求	离散光流 $\mathbf{V}$
算法性质	传统 CV

Algorithm 83: 迭代 L-K 光流 (calc_optflow_iterLK)

```
Input: 图像 I_1, I_2
Parameter: 光流初值 V_0 = 0_{2×H×W}
Output: 离散光流 V
for k ∈ 0, 1, ..., K_n do
    I_{1,k} ← Sft[I_1, V_k]
    δV_k ← calc_optflow_LS(I_{1,k}, I_2)
    V_{k+1} ← V_k + δV_k
V ← V_k
```

25.7.3 金字塔迭代求解

上述问题实际上是一个非凸的优化问题，优化目标是补偿后的光度误差，优化变量是全图的光流。对于运动较大的情况，即使使用了迭代求解的方法，其效果也不够好。

事实上，对于非凸优化问题，优化初值的选取往往非常重要。上面的算法中，所有光流初值均为 0。但实际上，我们可以采用图像金字塔的方法，获得更好的光流估计。

具体来说，根据光流的定义，降采样图像的光流相当于光流降采样再进行放缩，即

$$\mathbf{V}[\mathbf{DS}_{1/\rho}[I]] = \rho\mathbf{DS}_{1/\rho}[\mathbf{V}[I]] \tag{VII.25.25}$$

因此，我们可以先在 ρ 倍降采样的图像 $\mathbf{DS}_{1/\rho}[I]$ 上估计光流 $\mathbf{V}_\rho$，再将 $\mathbf{V}_\rho$ 上采样，作为原图的光流初值。

更进一步，我们可以使用类似 SIFT 特征点的思路，首先从高倍降采样的图像 (即金字塔塔尖) 开始，随后逐层向下估计，每层都以上一层的估计结果作为初值，最后可得到原分辨率图像的较准确光流估计。上述思路的光流求解称为**金字塔 L-K 光流法**。我们总结该算法如下

算法	**金字塔 L-K 光流**
问题类型	光流估计问题
已知	图像 I_1, I_2
求	离散光流 $\mathbf{V}$
算法性质	传统 CV

Algorithm 84: 金字塔 L-K 光流

Input: 图像 I_1, I_2

Parameter: 金字塔层数 L

Output: 离散光流 $\mathbf{V}$

$I_{1,1}, I_{1,2} \leftarrow I_1, I_2$

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**

  $\mathbf{V}_l \leftarrow$ 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 点对位姿求解

26.1 问题建立

上一章中，我们已经介绍了使用间接法进行特征点提取和匹配的方法，可以得到两幅图像中匹配的特征点了。下面，我们就要根据这些信息来估计位姿，即**点对位姿求解问题**。

具体来说，我们已知一组 N 个点在两个不同相机位姿下的像素坐标或相机系坐标，需要求解两个位姿间的齐次矩阵 T。一般 VO/vSLAM 在初始化前，没有任何已知 3D 信息，此时所有特征点坐标都是像素坐标 (2D)。当我们已经有了一些 3D 信息后，就可以使用已有的相机系坐标 (3D)。根据已知量是像素坐标 (2D) 或相机系坐标 (3D)，我们可以定义三种不同的点对位姿求解问题

问题	**2D-2D 点对位姿求解**
问题简述	已知两个位姿的匹配点像素坐标，求位姿间齐次矩阵
已知	位姿 1 下的各点像素坐标 $v_1^1, v_2^1, ..., v_N^1$ 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$ 相机内参 K
求	位姿间齐次矩阵 T_2^1

问题	**3D-3D 点对位姿求解**
问题简述	已知两个位姿的匹配点相机系坐标，求位姿间齐次矩阵
已知	位姿 1 下的各点相机系坐标 $p_1^1, p_2^1, ..., p_N^1$ 位姿 2 下的各点相机系坐标 $p_1^2, p_2^2, ..., p_N^2$
求	位姿间齐次矩阵 T_2^1

问题	**3D-2D 点对位姿求解**
问题简述	已知一组匹配点的 3D 坐标和某位姿下像素坐标，求位姿间齐次矩阵
已知	位姿 1 下的各点相机系坐标 $p_1^1, p_2^1, ..., p_N^1$ 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$ 相机内参 K
求	位姿间齐次矩阵 T_2^1

在这三个问题中，我们均假设内参矩阵 K 已知，且成像过程无畸变 (畸变已被补偿)。

下面，我们首先关注 2D-2D 问题和 3D-3D 问题，然后再关注 3D-2D 问题。

26.2 2D-2D 问题

26.2.1 对极约束

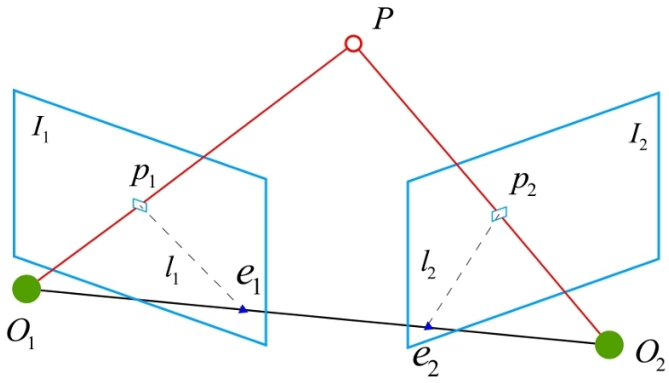

图 VII.26.1: 对极约束

首先，我们对 2D-2D 问题进行建模，这种建模方法又称为对极几何。

如图VII.26.1所示，空间中的点 P 和两个位姿 1, 2 处的相机光心 O_1, O_2 各有连线，即光线。光线和像平面 I_1, I_2 各自的交点 p_1, p_2 即为各自图像上 2D 点的位置。P 点、O_1, O_2 在不共线的情况下构成一个平面，称为**极平面**。连线 O_1O_2 称为**基线**，基线和像平面 I_1, I_2 的交点 e_1, e_2 称为**极点**，极点和像点的连线 p_1e_1, p_2e_2 称为**极线**。

假设 p_1, p_2 各自的 2D 像素坐标为 v^1, v^2，P 点在相机坐标系 1 和 2 下的 3D 坐标分别为 p^1, p^2。假设 p^1, p^2 焦平面深度 z^1, z^2 非 0，根据相机模型和坐标变换规则，我们有

$$v^1 = \frac{1}{z^1} K p^1$$
$$v^2 = \frac{1}{z^2} K p^2$$
$$p^1 = R_2^1 p_2 + t_{12}^1$$

代入像素坐标，有

$$z^1 K^{-1} v^1 = R_2^1 (z^2 K^{-1} v^2) + t_{12}^1$$

即

$$K^{-1} v^1 = \frac{z^2}{z^1} R_2^1 (K^{-1} v^2) + \frac{1}{z^1} t_{12}^1$$

两边叉乘 t_{12}^1，有

$$t_{12}^1 \times K^{-1} v^1 = t_{12}^1 \times \frac{z^2}{z^1} R_2^1 (K^{-1} v^2)$$

两边左乘 $(K^{-1}v^1)^T$

$$(K^{-1}v^1)^T t_{12}^1 \times K^{-1} v^1 = \frac{z^2}{z^1} (K^{-1}v^1)^T t_{12}^1 \times R_2^1 (K^{-1} v^2)$$

注意到向量 a 垂直于 $a \times b$，因此一定有 $a^T(a \times b) = 0$，即

$$0 = \frac{z^2}{z^1}(K^{-1}v^1)^T t_{12}^1 \times R_2^1(K^{-1}v^2)$$

整理，得

$$(v^1)^T K^{-T}(t_{12}^1)_\times R_2^1 K^{-1} v^2 = 0 \tag{VII.26.1}$$

式VII.26.1也称为对极约束，它描述了同一个空间点在两个不同相机位姿下的 (增广) 像素坐标的关系。我们定义

$$F(T_2^1, K) := K^{-T}(t_{12}^1)_\times R_2^1 K^{-1} \tag{VII.26.2}$$

式中的矩阵 F 称为**基础矩阵** (Fundamental Matrix)。定义

$$E(T_2^1) := (t_{12}^1)_\times R_2^1 \tag{VII.26.3}$$

式中的矩阵 E 称为**本质矩阵** (Essential Matrix)。

可以看到，基础矩阵和本质矩阵均为和相对位姿相关的 3×3 矩阵。

26.2.2 三角测量

在使用对极几何求解 2D-2D 问题之前，我们首先介绍三角测量方法。

三角测量也称深度估计，是点对位姿求解问题的逆问题。位姿求解是已知相机系坐标/像素坐标的条件下，求解位姿齐次矩阵。而三角测量是在已知齐次矩阵和像素坐标的条件下，求解相机系的坐标，或焦平面深度。

问题	三角测量问题
问题简述	已知两位姿间齐次矩阵、一组匹配点的像素坐标，求 3D 坐标
已知	位姿 1,2 下的像素坐标 v^1, v^2 位姿间齐次矩阵 T_2^1 相机内参 K
求	位姿 1,2 下的相机系坐标 p^1, p^2

为什么要首先介绍三角测量呢？因为 2D-2D 问题存在多解性。这个多解性来自于一个隐含约束——相机成像时，所有像平面距离均为正。然而，相机模型和基础矩阵、本质矩阵自身的数学描述并不包含该约束，我们需要人为执行这一约束。在执行这一约束的过程中，需要根据待定的位姿反求深度，再使用深度来排除或选择位姿。

实际的 VO 和 vSLAM 中，三角测量也非常重要，因为在没有深度先验的情况下，一组特征点首次观测时的深度信息必然要通过纯 2D 信息反推。三角测量正是这样的反推方法。事实上，三角测量的本质，就是通过归一化坐标和相对位姿的关系，来求解两个深度。

具体来说，根据上小节的对极几何关系，我们有

$$z^1 K^{-1} v^1 = R_2^1(z^2 K^{-1} v^2) + t_{12}^1$$

两边使用 $K^{-1}v^1$ 叉乘，有

$$K^{-1}v^1 \times R_2^1 K^{-1} v^2 z^2 + K^{-1} v^1 \times t_{12}^1 = 0$$

该式是一个关于标量 z^2 的代数方程。记

$$a = K^{-1}v^1 \times R_2^1 K^{-1} v^2$$
$$b = K^{-1}v^1 \times t_{12}^1$$

则有

$$z^2 = -\frac{b^T a}{a^T a}$$

随后，可使用相同的技巧求出 z^1，并使用相机模型计算 p^1, p^2。上述算法总结如下

算法	**三角测量**
问题类型	三角测量问题
已知	位姿 1,2 下的像素坐标 v^1, v^2 位姿间齐次矩阵 T_2^1 相机内参 K
求	位姿 1,2 下的相机系坐标 p^1, p^2
算法性质	近似解

Algorithm 85: 三角测量 (triangulate)

Input: 位姿 1,2 下的像素坐标 v^1, v^2
Input: 位姿间齐次矩阵 T_2^1
Input: 相机内参 K
Output: 位姿 1,2 下的相机系坐标 p^1, p^2

$a \leftarrow K^{-1}v^1 \times R_2^1 K^{-1} v^2$
$b \leftarrow K^{-1}v^1 \times t_{12}^1$
$z^2 \leftarrow -b^T a / a^T a$
$c \leftarrow K^{-1} v^1$
$d \leftarrow R_2^1 (z^2 K^{-1} v^2) + t_{12}^1$
$z^1 \leftarrow d^T c / c^T c$
$p^1 \leftarrow z^1 K^{-1} v^1$
$p^2 \leftarrow z^2 K^{-1} v^2$

对应的 python 代码如下所示

```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 位姿恢复

上小节中，我们介绍了对极约束。可以看到，在已知同一组位姿对应的点对的条件下，只要求解出基础矩阵或本质矩阵，就可以解决 2D-2D 位姿求解问题。

我们将在下一小节中详细介绍 F 阵和 E 阵的求解。在此之前，我们先介绍如何从 E 阵和 F 阵恢复 T_2^1，即 R_2^1 和 t_{12}^1。

事实上，F 阵可以转化为 E 阵

$$E = K^T F K \tag{VII.26.4}$$

因此, 我们只需从 E 阵恢复 T_2^1 即可。注意到 E 阵的特殊结构，我们可以使用如下所示的算法恢复旋转和平移。这里需要用到上述介绍的三角测量方法来去除多解。

算法	本质矩阵恢复位姿
问题类型	2D-2D 点位姿求解
已知	本质矩阵 E 位姿 1 下的各点像素坐标 $v_1^1, v_2^1, ..., v_N^1$ 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$
求	位姿间齐次矩阵 T_2^1
算法性质	近似解

Algorithm 86: 本质矩阵恢复位姿 (E_to_Rt)

Input: 本质矩阵 E

Input: 位姿 1 下的各点像素坐标 $\upsilon_1^1, \upsilon_2^1, ..., \upsilon_N^1$

Input: 位姿 2 下的各点像素坐标 $\upsilon_1^2, \upsilon_2^2, ..., \upsilon_N^2$

Output: 位姿间齐次矩阵 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**

    $d_i^1, d_i^2 \leftarrow \text{triangulate}(R, t, \upsilon_i^1, \upsilon_i^2)$

  if $d_{1:N}^1 > 0$ *and* $d_{1:N}^2 > 0$ **then**

    $R_2^1, t_{12}^1 \leftarrow R, t$

$$T_2^1 \leftarrow \begin{bmatrix} R_2^1 & t_{12}^1 \\ 0_{1\times 3} & 1 \end{bmatrix}$$

对应的 python 代码如下所示

```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 点法位姿求解

之前的小节中，我们介绍了对极约束。可以看到，在已知同一组位姿对应的点对的条件下，只要求解出基础矩阵或本质矩阵，就可以结合上一小节介绍的位姿恢复方法，解决 2D-2D 位姿求解问题。

这里需要特别注意：无论是 E 阵还是 F 阵，它们并没有 9 个自由度。事实上，仅就求解出的 T_2^1 来说，对其平移部分 t_{12}^1 乘以任一尺度因子 $r \neq 0$ 后，对极约束均成立。事实上，这就是著名的**单目无尺度性：在没有已知基线长度的情况下，仅依靠单目视觉传感器，不可能恢复运动的尺度信息**。

因此，F 矩阵实际上是有 8 个而非 9 个未知数。我们可以强制令右下角的值为 1，最后求出的 F 阵整体乘以尺度因子 r 后，仍满足全部约束。欲求解 8 个未知数的线性方程，需要 8 组非线性相关的约束，因此该方法要求点数 $N \geq 8$。实际使用中，我们一般选取 $N > 8$，使用超定方程求解误差最小的 F 阵。因此，这一方法也称为 **8 点法**。

在实际的 VO 和 vSLAM 算法中，解决单目无尺度性有两种思路。一是使用基线已知的双目相机增加尺度约束。二是使用其他传感器，如 IMU 等，进行多传感器融合，解决尺度问题。

此外，即使不考虑无尺度性，F 矩阵也没有 8 个自由度，因此计算得到的 F' 阵还需进行一步秩处理得到 F 阵。处理的方法是去除其最小的特征值对应的部分，使其满足秩为 2.

综合上述内容，将 8 点法位姿求解算法总结如下

算法	**8 点法位姿求解**
问题类型	2D-2D 点位姿求解
已知	位姿 1 下的各点像素坐标 $v_1^1, v_2^1, ..., v_N^1$ 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$ 相机内参 K
求	位姿间齐次矩阵 T_2^1
算法性质	近似解

Algorithm 87: 8 点法位姿求解

Input: 位姿 1 下的各点像素坐标 $v_1^1, v_2^1, ..., v_N^1$

Input: 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$

Input: 相机内参 K

Output: 基础矩阵 F，位姿间齐次矩阵 T_2^1

$A \leftarrow 0_{N\times 9}$

for $i \in 1, \ldots, N$ **do**

$[u_2, v_2, 1]^T \leftarrow v_i^2$

$A_{i,:} \leftarrow [u_2(v_i^1)^T \quad v_2(v_i^1)^T \quad (v_i^1)^T]$

$f \leftarrow \text{linear\_solve}(Af = 0)$

$$F' \leftarrow \begin{bmatrix} f_1 & f_4 & f_7 \\ f_2 & f_5 & f_8 \\ f_3 & f_6 & 1 \end{bmatrix}$$

$U, \text{diag}\{r, s, t\}, V^T \leftarrow \text{SVD\_calc}(F')$

$F \leftarrow U\,\text{diag}\{r, s, 0\}V^T$

$E \leftarrow K^T F K$

$T_2^1 \leftarrow \text{E\_to\_Rt}(E, v_:^1, v_:^2)$

对应的 python 代码如下所示

```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 点法 H 阵求解

基础矩阵/本质矩阵求解位姿要求 8 点不共面，否则求解过程中的 A 矩阵不可逆。那么，对于所有特征点共面情况，能否求解位姿呢？

单应矩阵定义

假设所有特征点均在平面 $\mathbf{n}^T\mathbf{x}^n + d = 0$ 上。其中，$\mathbf{n}$ 为平面法向量，x^n 表示世界坐标系下点的 3 维坐标。假设分别有 $c1$ 和 $c2$ 两个不同的相机坐标系，对应的位姿分别为 $R_{c1} = R_n^{c1}, t_{c1} = t_{nc1}^n$ 和 $R_{c2} = R_n^{c2}, t_{c2} = t_{nc2}^n$。对于 n 系下的点 $\mathbf{x}_i, i = 1, ..., N$，则有

$$
\begin{aligned}
0 &= \mathbf{n}^T\mathbf{x}_i + d \\
\mathbf{x}_i^{c1} &= R_{c1}(\mathbf{x}_i - t_{c1}) \\
\mathbf{x}_i^{c2} &= R_{c2}(\mathbf{x}_i - t_{c2})
\end{aligned}
\tag{VII.26.5}
$$

设

$$
\begin{aligned}
R &= R_{c1}R_{c2}^T \\
t &= R_{c1}(t_{c2} - t_{c1})
\end{aligned}
$$

则有

$$
\mathbf{x}_i^{c1} = R\mathbf{x}_i^{c2} + t \tag{VII.26.6}
$$

将式VII.26.5的第一行代入第三行，有

$$
\mathbf{n}^T(R_{c2}^T\mathbf{x}_i^{c2} + t_{c2}) + d = 0
$$

即

$$(R_{c2}\mathbf{n})^T\mathbf{x}_i^{c2} + \mathbf{n}^T t_{c2} + d = 0$$

记

$$\mathbf{n}_2 = R_{c2}\mathbf{n}$$
$$d_2 = \mathbf{n}^T t_{c2} + d$$

则有

$$\mathbf{n}_2^T\mathbf{x}_i^{c2} = -d_2$$

即

$$\frac{\mathbf{n}_2^T\mathbf{x}_i^{c2}}{-d_2} = 1$$

代入式VII.26.5中，有

$$\mathbf{x}_i^{c1} = R\mathbf{x}_i^{c2} + t\frac{\mathbf{n}_2^T\mathbf{x}_i^{c2}}{-d_2}$$

即

$$\mathbf{x}_i^{c1} = (R - \frac{t\mathbf{n}_2^T}{d_2})\mathbf{x}_i^{c2}$$

现在假设我们没有 3 维坐标，而是只有像素坐标，即

$$v_{i,1} = K\mathbf{x}_i^{c1}/z_i^{c1}$$
$$v_{i,2} = K\mathbf{x}_i^{c2}/z_i^{c2}$$

则有

$$K^{-1}v_{i,1}z_i^{c1} = (R - \frac{t\mathbf{n}_2^T}{d_2})K^{-1}v_{i,2}z_i^{c2}$$

即

$$v_{i,1} = \frac{z_i^{c2}}{z_i^{c1}}K(R - \frac{t\mathbf{n}_2^T}{d_2})K^{-1}v_{i,2}$$

记

$$H = K(R - \frac{t\mathbf{n}_2^T}{d_2})K^{-1} \tag{VII.26.7}$$

以及

$$\lambda_i = \frac{z_i^{c2}}{z_i^{c1}}$$

则有

$$v_{i,1} = \lambda_i H v_{i,2} \tag{VII.26.8}$$

其中，H 称为单应矩阵，为 3*3 的矩阵，包含相对位姿、平面位置等信息。λ_i 为尺度因子。式VII.26.8说明，一组共面点的 2D 坐标满足线性关系，线性系数为单应矩阵和尺度因子的乘积。

和本质矩阵类似，在已知多对 2D 点坐标的情况下，也可以求解单应矩阵，这就是单应矩阵求解问题。

问题	**2D-2D 点求解单应矩阵**
问题简述	已知两个位姿的多对共面点像素坐标，求单应矩阵
已知	位姿 1 下的各点像素坐标 $v_1^1, v_2^1, ..., v_N^1$ 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$ 相机内参 K
求	单应矩阵 H

单应矩阵求解

H 矩阵是 3 行 3 列的矩阵，我们把 H 矩阵展开成元素形式

$$\begin{bmatrix} u_{i;1} \\ v_{i;1} \\ 1 \end{bmatrix} = \hat{\lambda}_i \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{bmatrix} \begin{bmatrix} u_{i;2} \\ v_{i;2} \\ 1 \end{bmatrix}$$

注意，这里我们将原本 h_9 的位置设置为 1，本质上是将整个单应矩阵除以 h_9，同时将 λ_i 乘以 h_9 得到 $\hat{\lambda}_i$。即

$$\begin{aligned} u_{i;1} &= \hat{\lambda}_i(h_1 u_{i;2} + h_2 v_{i;2} + h_3) \\ v_{i;1} &= \hat{\lambda}_i(h_4 u_{i;2} + h_5 v_{i;2} + h_6) \\ 1 &= \hat{\lambda}_i(h_7 u_{i;2} + h_8 v_{i;2} + 1) \end{aligned}$$

使用第三行消去 $\hat{\lambda}_i$，有

$$\begin{aligned} 0 &= (h_1 u_{i;2} + h_2 v_{i;2} + h_3) + u_{i;1}(h_7 u_{i;2} + h_8 v_{i;2} + 1) \\ 0 &= (h_4 u_{i;2} + h_5 v_{i;2} + h_6) + v_{i;1}(h_7 u_{i;2} + h_8 v_{i;2} + 1) \end{aligned}$$

可以看到，目前我们有 8 个未知数和 2 个方程。这是第 i 对点的 2 组 2D 坐标提供的信息。如果我们有不少于 4 对点，我们就能得到不少于 8 个方程进行联立，从而实现单应矩阵的求解，也就是 4 点法。我们将该算法总结如下

算法	**4 点法求解单应矩阵**
问题类型	2D-2D 点对求解单应矩阵
已知	位姿 1 下的各点像素坐标 $v_1^1, v_2^1, ..., v_N^1$ 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$ 相机内参 K
求	单应矩阵 H
算法性质	近似解

Algorithm 88: 4 点法求解单应矩阵

Input: 位姿 1 下的各点像素坐标 $v_1^1, v_2^1, ..., v_N^1$

Input: 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$

Input: 相机内参 K

Output: 单应矩阵 H

$A \leftarrow 0_{2N\times 8}$

$b \leftarrow 0_{2N}$

for $i \in 1, \dots, N$ **do**

$[u_1, v_1, 1]^T \leftarrow K^{-1}v_i^1$

$[u_2, v_2, 1]^T \leftarrow K^{-1}v_i^2$

$A_{2i,:} \leftarrow [u_2, v_2, 1, 0, 0, 0, -u_1u_2, -u_1v_2]^T$

$A_{2i+1,:} \leftarrow [0, 0, 0, u_2, v_2, 1, -v_1u_2, -v_1v_2]^T$

$b_{2i:2i+2} \leftarrow [u_1, v_1]$

$h \leftarrow \text{linear\_solve}(Ah = b)$

$$H \leftarrow \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{bmatrix}$$

不过需要注意，这样求解得到的单应矩阵，和真正的单应矩阵相差一个比例因子，对应于原本单应矩阵的右下角元素。

26.2.6 6 点法速度求解

上面我们介绍了两种基于 2D 点对的位姿估计方法，其中 E 阵需要至少 8 对点，而 H 阵需要至少 4 对共面点。除此之外，还有一种使用至少 6 对点的光流估计相机 3 维速度和角速度的方法。

光流方程组

首先，我们来推导光流和相机速度及角速度的关系。假设导航系为 n，相机系为 c。某点 P 在相机系下的 3 维坐标为

$$p^c = R_n^c(p^n - p_{nc}^n)$$

假设相机正在运动，相机在 n 系下的速度和角速度分别为 v_{nc}^n 和 ω_{nc}^n，根据式I.2.50，有

$$\begin{aligned} \dot{p}_{nc}^n &= v_{nc}^n \\ \dot{R}_n^c &= -R_n^c(\omega_{nc}^n)_\times \end{aligned}$$

此时，考虑 p^c 的导数，有

$$\begin{aligned} \dot{p}^c &= \dot{R}_n^c(p^n - p_{nc}^n) - R_n^c\dot{p}_{nc}^n \\ &= -R_n^c(\omega_{nc}^n)_\times(p^n - p_{nc}^n) - R_n^c v_{nc}^n \\ &= -(R_n^c\omega_{nc}^n) \times (R_n^c(p^n - p_{nc}^n)) - R_n^c v_{nc}^n \\ &= -\omega_{nc}^c \times p^c - v_{nc}^c \end{aligned}$$

记

$$v = v_{nc}^c = [v_x, v_y, v_z]^T$$
$$\omega = \omega_{nc}^c = [w_x, w_y, w_z]^T$$

我们有相机系下的特征点微分方程

$$\dot{p}^c = -\omega \times p^c - v \tag{VII.26.9}$$

假设

$$p^c = \begin{bmatrix} x^c \\ y^c \\ z^c \end{bmatrix}$$

则式VII.26.9可以写为

$$\begin{bmatrix} \dot{x}^c \\ \dot{y}^c \\ \dot{z}^c \end{bmatrix} = - \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix} \begin{bmatrix} x^c \\ y^c \\ z^c \end{bmatrix} - \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

即

$$\begin{aligned} \dot{x}^c &= w_z y^c - w_y z^c - v_x \\ \dot{y}^c &= -w_z x^c + w_x z^c - v_y \\ \dot{z}^c &= w_y x^c - w_x y^c - v_z \end{aligned} \tag{VII.26.10}$$

p^c 对应的归一化坐标为 $[x^c/z^c, y^c/z^c]$，归一化光流即为归一化坐标在 dt 内的变化量，即

$$\begin{aligned} o_x &= dt(\frac{\dot{x}^c}{z^c}) \\ o_y &= dt(\frac{\dot{y}^c}{z^c}) \end{aligned}$$

即

$$\begin{aligned} o_x &= dt\frac{z^c\dot{x}^c - x^c\dot{z}^c}{(z^c)^2} \\ o_y &= dt\frac{z^c\dot{y}^c - y^c\dot{z}^c}{(z^c)^2} \end{aligned}$$

代入式VII.26.10，有

$$\begin{aligned} \frac{o_x}{dt} &= \frac{1}{(z^c)^2}(z^c(w_z y^c - w_y z^c - v_x) - x^c(w_y x^c - w_x y^c - v_z)) \\ \frac{o_y}{dt} &= \frac{1}{(z^c)^2}(z^c(-w_z x^c + w_x z^c - v_z) - y^c(w_y x^c - w_x y^c - v_z)) \end{aligned}$$

接下来，我们用逆深度 ρ 和归一化坐标 α, β 代替 x^c, y^c, z^c，即

$$\begin{bmatrix} \alpha \\ \beta \\ \rho \end{bmatrix} = \frac{1}{z^c} \begin{bmatrix} x^c \\ y^c \end{bmatrix} \tag{VII.26.11}$$

此时我们有

$$\begin{aligned}\frac{o_x}{dt} &= -(v_x\rho - \alpha v_z - \alpha\beta w_x + (1+\alpha^2)w_y\rho - \beta w_z)\\ \frac{o_y}{dt} &= -(v_y\rho - \beta v_z\rho - (1+\beta^2)w_x + \alpha\beta w_y + \alpha w_z)\end{aligned} \tag{VII.26.12}$$

该方程组中，$v_x, v_y, v_z, w_x, w_y, w_z$ 为变量，ρ 为未知量，其他全为已知。也就是说，一个点的归一化坐标和光流信息可以提供 2 个方程和 1 个未知数。N 个点会有 $2N$ 个方程和 $6+N$ 个未知数。因此，$N \geq 6$ 时该方程组可以求解。

我们将这一问题总结如下

问题	**2D-2D 点求解 3 维速度**
问题简述	已知两个位姿的匹配点像素坐标，求 3 维速度和角速度
已知	各点像素坐标 $v_1, v_2, ..., v_N$ 像素坐标光流 $o_1, o_2, ..., o_N$ 时间间隔 δt，相机内参 K
求	相机系下速度 v_{nc}^c，角速度 ω_{nc}^c

6 点法优化算法

观察方程组VII.26.12，我们发现，这是一个非线性方程组，但非线性项仅存在于乘法交叉项。因此，我们考虑使用非线性优化的方法进行数值求解。

具体来说，假设给定 N 个点 $p_1^c, ..., p_N^c$，对于每个点 p_i^c，我们都有已知量 $o_{x,i}, o_{y,i}, \alpha_i, \beta_i$，以及未知量 ρ_i。也就是说，联合优化变量

$$\mathbf{x} = [v_x, v_y, v_z, w_x, w_y, w_z, \rho_1, ..., \rho_N] \tag{VII.26.13}$$

我们可以构造第 i 组残差项

$$\mathbf{e}_i = \begin{bmatrix} -(v_x\rho - \alpha v_z\rho - \alpha\beta w_x + (1+\alpha^2)w_y - \beta w_z) - \frac{o_x}{dt} \\ -(v_y\rho - \beta v_z\rho - (1+\beta^2)w_x + \alpha\beta w_y + \alpha w_z) - \frac{o_y}{dt} \end{bmatrix} \tag{VII.26.14}$$

此时，$\mathbf{e}_i$ 对 $\mathbf{x}$ 的导数 $J_i(\mathbf{x})$ 为

$$J_i(\mathbf{x}) = \begin{bmatrix} -\rho_i & 0 & \alpha\rho & \alpha\beta & -(1+\alpha^2) & \beta & \dots & -v_x + \alpha v_z & \dots \\ 0 & -\rho & \beta\rho & 1+\beta^2 & -\alpha\beta & -\alpha & \dots & -v_y + \beta v_z & \dots \end{bmatrix} \tag{VII.26.15}$$

这样，我们就可以使用第3.6节介绍的高斯牛顿法等非线性优化方法，对该问题进行联合优化求解。我们将算法总结如下

算法	**6 点法求解三维速度**
问题类型	2D-2D 点对求解三维速度
已知	各点像素坐标 $v_1, v_2, ..., v_N$ 像素坐标光流 $o_1, o_2, ..., o_N$ 时间间隔 δt，相机内参 K
求	相机系下速度 v_{nc}^c，角速度 ω_{nc}^c
算法性质	近似解

Algorithm 89: 6 点法求解三维速度

Input: 各点像素坐标 $\upsilon_1, \upsilon_2, ..., \upsilon_N$

Input: 像素坐标光流 $o_1, o_2, ..., o_N$

Input: 时间间隔 δt，相机内参 K

Output: 速度 $v_{nc}^c = [v_x, v_y, v_z]^T$，角速度 $\omega_{nc}^c = [w_x, w_y, w_z]^T$

$\mathbf{x}_k \leftarrow [0_{1\times 6}, 1_{1\times N}]^T$

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**

$[\alpha_i, \beta_i, 1]^T \leftarrow K^{-1}\upsilon_i$

$[o_{x,i}, o_{y,i}] \leftarrow o_i$

$\mathbf{e}[2i] \leftarrow -(v_x\rho_i - \alpha_i v_z\rho_i - \alpha_i\beta_i w_x + (1+\alpha_i^2)w_y - \beta_i w_z) - \frac{o_{x,i}}{\delta t}$

$\mathbf{e}[2i+1] \leftarrow -(v_y\rho_i - \beta_i v_z\rho_i - (1+\beta_i^2)w_x + \alpha_i\beta_i w_y + \alpha_i w_z) - \frac{o_{y,i}}{\delta t}$

$J[2i:2i+1, 0:6] \leftarrow \begin{bmatrix} -\rho_i & 0 & \alpha_i\rho_i & \alpha_i\beta_i & -(1+\alpha_i^2) & \beta_i \\ 0 & -\rho_i & \beta_i\rho_i & 1+\beta_i^2 & -\alpha_i\beta_i & -\alpha_i \end{bmatrix}$

$J[2i:2i+1, 6+i] \leftarrow \begin{bmatrix} -v_x + \alpha_i v_z \\ -v_y + \beta_i v_z \end{bmatrix}$

$H_n \leftarrow J^T J$

$g_n \leftarrow J^T\mathbf{e}$

if $||g_n|| < \epsilon$ **then**

break

$\mathbf{x}_k \leftarrow \mathbf{x}_k - H_n^{-1}g_n$

$[v_x, v_y, v_z, w_x, w_y, w_z] \leftarrow (\mathbf{x}_k)0:6$

对应的 python 代码如下所示（使用了 pyceres 库）

```python
class OpticalFlowResidual(pyceres.CostFunction):
    def Evaluate(self, parameters, residuals, jacobians):
        rho = parameters[0][0]  # 逆深度
        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 # 对 rho 的雅可比
            if jacobians[1] is not None:
                jacobians[1][:] = J_v  # 对 v 的雅可比
            if jacobians[2] is not None:
                jacobians[2][:] = J_w  # 对 w 的雅可比
        return True
def calc_vw_from_optflow(obs, optflow, dt):
    problem = pyceres.Problem()
    rhos = {}
    for pid in obs:
        rhos[pid] = np.array([0.3]) # 初始逆深度
        problem.add_parameter_block(rhos[pid], 1)
    v_param = np.array([0.0, 0.0, 0.0])  # 初始速度设为 0
    w_param = np.array([0.0, 0.0, 0.0])  # 初始角速度设为 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 问题

26.3.1 ICP 算法

对 2D-2D 问题，我们已经介绍了基于对极几何的求解方法。下面我们介绍 3D-3D 点对匹配的方法。

一组 3D 点也称为点云。如果已知两个位姿下的点云信息，但并不知道各点的匹配关系，要求对点云进行匹配并求出两个位姿的齐次矩阵，这一问题也称为**点云配准问题**，是 Lidar-based SLAM 领域的基础问题。解决这类问题的方法也称为迭代最近点 (ICP, Iterative Closest Points)。

ICP 算法中的迭代是指，在未知匹配关系的情况下，迭代优化匹配关系和位姿。由于我们考虑的 3D-3D 问题已经假设点云匹配完成，因此我们实际只需要使用 ICP 中的位姿求解步骤。不过，我们仍称该方法为 ICP。

具体来说，我们需要求解 R_2^1 和 t_{12}^1。为简单起见，以下推导中简写 $R = R_2^1, t = t_{12}^1$。我们需要最小化如下匹配误差

$$J(R,t) = \frac{1}{2}\sum_{i=1}^{N} ||Rp_i^2 + t - p_i^1||^2 \tag{VII.26.16}$$

假设两组点云坐标的均值分别为 $\bar{p}^1$ 和 $\bar{p}^2$，有

$$\bar{p}^1 = \frac{1}{N}\sum_{i=1}^{N} p_i^1$$

$$\bar{p}^2 = \frac{1}{N}\sum_{i=1}^{N} p_i^2$$

对第 1 组点云，假设每个点相对中心的偏移 $\tilde{p}_i^1 = p_i^1 - \bar{p}^1$，第 2 组同理，则有

$$\begin{aligned}
J(R,t) &= \frac{1}{2}\sum_{i=1}^{N} ||R(\tilde{p}_i^2 + \bar{p}^2) + t - (\tilde{p}_i^1 + \bar{p}^1)||^2 \\
&= \frac{1}{2}\sum_{i=1}^{N} ||(R\bar{p}^2 + t - \bar{p}^1) + (R\tilde{p}_i^2 - \tilde{p}_i^1)||^2 \\
&= \frac{N}{2}||R\bar{p}^2 + t - \bar{p}^1||^2 + \frac{1}{2}\sum_{i=1}^{N}(R\bar{p}^2 + t - \bar{p}^1)^T(R\tilde{p}_i^2 - \tilde{p}_i^1) + \frac{1}{2}\sum_{i=1}^{N}||R\tilde{p}_i^2 - \tilde{p}_i^1||^2 \\
&= \frac{N}{2}||R\bar{p}^2 + t - \bar{p}^1||^2 + \frac{1}{2}\sum_{i=1}^{N}||R\tilde{p}_i^2 - \tilde{p}_i^1||^2 + \frac{1}{2}(R\bar{p}^2 + t - \bar{p}^1)^T\sum_{i=1}^{N}(R\tilde{p}_i^2 - \tilde{p}_i^1) \\
&= \frac{N}{2}||R\bar{p}^2 + t - \bar{p}^1||^2 + \frac{1}{2}\sum_{i=1}^{N}||R\tilde{p}_i^2 - \tilde{p}_i^1||^2
\end{aligned}$$

可以看到，匹配误差被分为了两个部分。由于误差 J 对 R, t 均为二次型，存在全局最优解，且全局最优解 R^*, t^* 满足一阶导数条件

$$\frac{\partial J}{\partial t} = (R\bar{p}^2 + t - \bar{p}^1)^T = 0 \tag{VII.26.17}$$

因此有

$$t^* = \bar{p}^1 - R^*\bar{p}^2 \tag{VII.26.18}$$

将上述最优平移量和最优旋转矩阵关系代入误差表达式VII.26.16，有

$$
\begin{aligned}
R^* &= \arg\min_R \frac{1}{2}\sum_{i=1}^{N} ||R\tilde{p}_i^2 - \tilde{p}_i^1||^2 \\
&= \arg\min_R \frac{1}{2}\sum_{i=1}^{N} (R\tilde{p}_i^2 - \tilde{p}_i^1)^T (R\tilde{p}_i^2 - \tilde{p}_i^1) \\
&= \arg\min_R \frac{1}{2}\sum_{i=1}^{N} (\tilde{p}_i^1)^T \tilde{p}_i^1 + (\tilde{p}_i^2)^T (R)^T R\tilde{p}_i^2 - 2(\tilde{p}_i^1)^T R\tilde{p}_i^2 \\
&= \arg\min_R \frac{1}{2}\sum_{i=1}^{N} (\tilde{p}_i^1)^T \tilde{p}_i^1 + (\tilde{p}_i^2)^T \tilde{p}_i^2 - \sum_{i=1}^{N} (\tilde{p}_i^1)^T R\tilde{p}_i^2 \\
&= \arg\min_R -\sum_{i=1}^{N} (\tilde{p}_i^1)^T R\tilde{p}_i^2
\end{aligned}
$$

由于 A 为方阵时，有

$$
b^T Ac = \mathrm{tr}(Acb^T)
$$

因此

$$
R^* = \arg\max_R \sum_{i=1}^{N} \mathrm{tr}(R\tilde{p}_i^2 (\tilde{p}_i^1)^T) = \arg\max_R \mathrm{tr}(R\sum_{i=1}^{N} \tilde{p}_i^2 (\tilde{p}_i^1)^T)
$$

记

$$
W = \sum_{i=1}^{N} \tilde{p}_i^2 (\tilde{p}_i^1)^T
$$

则有

$$
R^* = \arg\max_R \mathrm{tr}(RW)
$$

假设 W 的 SVD 分解为

$$
W = U\Sigma V^T \tag{VII.26.19}
$$

则有

$$
R^* = \arg\max_R \mathrm{tr}(RU\Sigma V^T)
$$

由于 W 为 3×3 矩阵，V^T, U 均为三阶正交矩阵，满足 $V^T V = U^T U = I_3$。记

$$
B = RUV^T
$$

则有

$$
R = BVU^T
$$

代入式中，有

$$
\begin{aligned}
B^* &= \arg\max_B \operatorname{tr}(BVU^TU\Sigma V^T) \\
&= \arg\max_B \operatorname{tr}(BV\Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}}V^T) \\
&= \arg\max_B \operatorname{tr}(BV\Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}}V^T)
\end{aligned}
$$

我们引入一条引理，稍后证明：

$$
\operatorname{tr}(AA^T) \geq \operatorname{tr}(BAA^T), \quad \forall A \in \mathbb{R}^{n\times n}, B^TB = I_n
$$

因此，

$$
\operatorname{tr}(BV\Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}}V^T) \leq \operatorname{tr}(V\Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}}V^T), \forall B
$$

即

$$
B^* = I
$$

因此，

$$
R^* = B^*VU^T = VU^T \tag{VII.26.20}
$$

综上，总结 ICP 求解算法如下

算法	**ICP 算法**
问题类型	3D-3D 点位姿求解
已知	位姿 1 下的各点相机系坐标 $p_1^1, p_2^1, ..., p_N^1$ 位姿 2 下的各点相机系坐标 $p_1^2, p_2^2, ..., p_N^2$
求	位姿间齐次矩阵 T_2^1
算法性质	近似解

Algorithm 90: ICP 算法 (ICP_calc)

Input: 位姿 1 下的各点相机系坐标 $p_1^1, p_2^1, ..., p_N^1$

Input: 位姿 2 下的各点相机系坐标 $p_1^2, p_2^2, ..., p_N^2$

Output: 位姿间齐次矩阵 T_2^1

$\bar{p}^1 \leftarrow \frac{1}{N}\sum_{i=1}^N p_i^1$

$\bar{p}^2 \leftarrow \frac{1}{N}\sum_{i=1}^N p_i^2$

$W \leftarrow \sum_{i=1}^N (p_i^2 - \bar{p}^2)(p_i^1 - \bar{p}^1)^T$

$U, \Sigma, V^T \leftarrow \text{SVD\_calc}(W)$

$R_2^1 \leftarrow VU^T$

$t_{12}^1 \leftarrow \bar{p}^1 - R_2^1\bar{p}^2$

$T_2^1 \leftarrow \begin{bmatrix} R_2^1 & t_{12}^1 \\ 0_{1\times 3} & 1 \end{bmatrix}$

对应的 python 代码如下所示

```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 引理证明

下面我们简单证明上一小节中的引理

$$\mathrm{tr}(AA^T) \geq \mathrm{tr}(BAA^T), \quad \forall A \in \mathbb{R}^{n\times n}, B^TB = I_n$$

假设矩阵 A 可以写为

$$\begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix}$$

则

$$
\begin{aligned}
\mathrm{tr}(BAA^T) &= \mathrm{tr}(A^T(BA)) \\
&= \mathrm{tr}\left(\begin{bmatrix} a_1^T \\ a_2^T \\ \dots \\ a_n^T \end{bmatrix} B \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix}\right) \\
&= \sum_{i=1}^{n} a_i^T B a_i = \sum_{i=1}^{n} a_i^T (B a_i) \\
&\leq \sum_{i=1}^{n} \sqrt{(a_i^T a_i)(a_i^T B^T B a_i)} \\
&= \sum_{i=1}^{n} \sqrt{(a_i^T a_i)(a_i^T a_i)} = \sum_{i=1}^{n} (a_i^T a_i) \\
&= \mathrm{tr}\left(\begin{bmatrix} a_1^T \\ a_2^T \\ \dots \\ a_n^T \end{bmatrix} \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix}\right) \\
&= \mathrm{tr}(A^T A)
\end{aligned}
$$

证明过程中用到了 Cauchy-Schwartz 不等式，此处不再赘述。

26.4 3D-2D 问题

26.4.1 6 点法 DLT 求解

对于 3D-2D 问题，考虑空间点 P 在位姿 1 和位姿 2 相机系下的坐标 p^1 和 p^2，显然有

$$
p^2 = R_1^2 p^1 + t_{21}^2
$$

考虑位姿 2 下的成像过程，有

$$
v^2 = \frac{1}{z^2} K p^2
$$

结合起来，我们有位姿 2 下的像素坐标和位姿 1 下的相机系坐标的关系

$$
\begin{bmatrix} z^2 u^2 \\ z^2 v^2 \\ z^2 \end{bmatrix} = K[R_1^2 \quad t_{21}^2] p_{ext}^1
$$

该式中的焦平面深度 z^2 是未知数，我们使用线性变换方法对其进行消元，有

$$
\begin{bmatrix} 1 & 0 & -u^2 \\ 0 & 1 & -v^2 \end{bmatrix} K[R_1^2 \quad t_{21}^2] p_{ext}^1 = 0 \tag{VII.26.21}
$$

记

$$
\begin{bmatrix} 1 & 0 & -u^2 \\ 0 & 1 & -v^2 \end{bmatrix} K = A = [a_1 \quad a_2 \quad a_3] \in \mathbb{R}^{2\times 3}
$$

且

$$\begin{bmatrix} t_1^T \\ t_2^T \\ t_3^T \end{bmatrix} = [R_1^2 \quad t_{21}^2] \in \mathbb{R}^{3\times 4}$$

则方程

$$[a_1 \quad a_2 \quad a_3] \begin{bmatrix} t_1^T \\ t_2^T \\ t_3^T \end{bmatrix} p_{ext}^1 = 0$$

可以被重新写为

$$[a_1(p_{ext}^1)^T \quad a_2(p_{ext}^1)^T \quad a_3(p_{ext}^1)^T] \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} = 0$$

再记

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \in \mathbb{R}^{12}$$

可以看到，待求的 $\mathbf{t}$ 共包含 12 个未知数，而每个已知的 P 点可以构成 2 个方程。因此，在总点数 $N \geq 6$ 的条件下，可以进行线性求解。

由于这种方法通过求解线性方程直接解出齐次矩阵的元素，因此也称为**直接线性变换** (DLT, Direct Linear Transform) 方法。我们将该算法总结如下

算法	**DLT 算法**
问题类型	3D-2D 点位姿求解
已知	位姿 1 下的各点相机系坐标 $p_1^1, p_2^1, ..., p_N^1$ 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$ 相机内参 K
求	位姿间齐次矩阵 T_2^1
算法性质	近似解

Algorithm 91: DLT 算法 (DLT_calc)

Input: 位姿 1 下的各点相机系坐标 $p_1^1, p_2^1, ..., p_N^1$

Input: 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$

Output: 位姿间齐次矩阵 T_2^1

$M \leftarrow 0_{2N\times 12}$

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

$$[a_1 \quad a_2 \quad a_3] \leftarrow \begin{bmatrix} 1 & 0 & -u^2 \\ 0 & 1 & -v^2 \end{bmatrix} K$$

$$M_{2i:2i+1,:} \leftarrow \begin{bmatrix} a_1(p_{ext}^1)^T & a_2(p_{ext}^1)^T & a_3(p_{ext}^1)^T \end{bmatrix}$$

$$\begin{bmatrix} t_1^T & t_2^T & t_3^T \end{bmatrix}^T \leftarrow \text{linear\_solve}(M^T M\mathbf{t} = 0)$$

$$[R_1^2 \quad t_{21}^2] \leftarrow \begin{bmatrix} t_1^T \\ t_2^T \\ t_3^T \end{bmatrix}$$

$R_1^2 \leftarrow R_1^2 / |(R_1^2)|^{-\frac{1}{3}}$

$U, \Sigma, V^T \leftarrow \text{SVD\_calc}(R_1^2)$

$R_1^2 \leftarrow V^T U$

$R_2^1, t_{12}^1 \leftarrow (R_1^2)^T, -R_1^2 t_{21}^2$

对应的 python 代码如下所示

```python
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])
    ])
```

```
25      return T_2_to_1
```

26.4.2 EPnP 位姿求解

另一种求解 3D-2D 问题的方法是 EPnP(Effective Perspective-n-Points) 算法。这一算法的核心是将 3D-2D 问题转化为 3D-3D 问题，随后使用前面介绍的 ICP 进行求解。

具体来说，回顾点在三维空间中的坐标表示：假设某坐标系 n 的原点为 O，其三个正交单位坐标基分别为 $\overrightarrow{OA}, \overrightarrow{OB}, \overrightarrow{OC}$。对空间中的任意一点 P，假设其在该坐标系下的坐标为 $[a_1, a_2, a_3]$，则有空间向量关系

$$\overrightarrow{OP} = a_1\overrightarrow{OA} + a_2\overrightarrow{OB} + a_3\overrightarrow{OC}$$

对于任意坐标系 b(可以不是 n 系)，都有如下坐标关系

$$p^b - o^b = a_1(a^b - o^b) + a_2(b^b - o^b) + a_3(c^b - o^b)$$

整理上式，我们有任意点 p 在任意坐标系 b 下的坐标表达关系

$$p^b = a_1a^b + a_2b^b + a_3c^b + (1 - a_1 - a_2 - a_3)o^b$$

令 $a_4 = 1 - a_1 - a_2 - a_3$，记 $A = F_1, B = F_2, C = F_3, O = F_4$，则有

$$p^b = \sum_{i=1}^{4} a_i f_i^b, \quad \sum_{i=1}^{4} a_i = 1 \tag{VII.26.22}$$

注意，该表达式中 P 点和 b 系的选取与 $F_1 \sim F_4 4$ 点的选取完全无关，且 a_i 的取值也不受 b 系选取的影响。

事实上，$F_1 \sim F_4$ 不必须是构成单位正交基的 4 个点。**只要 $F_1 \sim F_4$ 是不共面的 4 点，空间点 P 在任意坐标系下的坐标都可以由唯一的一组参数确定的 4 点坐标线性表出**。这四个点称为**参考点**。

那么，任给一点 P，假如已知 P 点和**参考点**在某坐标系下的坐标，如何求取系数 $a_1 \sim a_4$ 呢？事实上，我们仅需列出方程

$$F_r\mathbf{a} = \mathbf{b}$$

其中

$$F_r = \begin{bmatrix} f_1 & f_2 & f_3 & f_4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \in \mathbb{R}^{4\times4}$$

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \in \mathbb{R}^4, \mathbf{b} = \begin{bmatrix} p \\ 1 \end{bmatrix} \in \mathbb{R}^4$$

即可解得

$$\mathbf{a} = F_r^{-1}b$$

那么，对于一组已知 3D 坐标的点 (例如位姿 1 相机系下的各点坐标)，如何选取控制点呢？我们可以使用类似 PCA 算法中的思想，使矩阵 A 尽量避免奇异。

具体来说，对于第一个控制点，我们仍取平均值

$$f_1^1 = \bar{p}^1 = \frac{1}{N}\sum_{i=1}^{N} p_i^1 \tag{VII.26.23}$$

定义归零化坐标矩阵

$$\bar{P} = \begin{bmatrix} (p_1^1 - f_1^1)^T \\ (p_2^1 - f_1^1)^T \\ \cdots \\ (p_N^1 - f_1^1)^T \end{bmatrix} \tag{VII.26.24}$$

计算维度协方差矩阵的特征向量

$$\begin{aligned} \bar{P}^T\bar{P}v_1 &= \lambda_1 v_1 \\ \bar{P}^T\bar{P}v_2 &= \lambda_2 v_2 \\ \bar{P}^T\bar{P}v_3 &= \lambda_3 v_3 \end{aligned}$$

这样，就可以选取三个控制点

$$\begin{aligned} f_2^1 &= f_1^1 + \sqrt{\frac{\lambda_1}{N}}v_1 \\ f_3^1 &= f_1^1 + \sqrt{\frac{\lambda_2}{N}}v_2 \\ f_4^1 &= f_1^1 + \sqrt{\frac{\lambda_3}{N}}v_3 \end{aligned}$$

现在，我们考虑在 3D-2D 问题的位姿 2 相机系下进行坐标表达。对每个匹配的点 P，由于

$$p^2 = \sum_{i=1}^{4} a_i f_i^2$$

由相机模型，有

$$v^2 = \frac{1}{z^2}K\sum_{i=1}^{4} a_i f_i^2$$

其中控制点在位姿 2 相机系下的坐标是未知数，共 12 个。记

$$\mathbf{f} = \begin{bmatrix} f_1^2 \\ f_2^2 \\ f_3^2 \\ f_4^2 \end{bmatrix} \tag{VII.26.25}$$

因此，有

$$\begin{bmatrix} z^2u^2 \\ z^2v^2 \\ z^2 \end{bmatrix} = K[a_1I \quad a_2I \quad a_3I \quad a_4I]\mathbf{f}$$

式中，z^2 为未知，u^2, v^2 已知。将第三行代入前两行进行消元，得到关于 $\mathbf{f}$ 的方程

$$\begin{bmatrix} 1 & 0 & -u^2 \\ 1 & -v^2 & 0 \end{bmatrix} K[a_1I \quad a_2I \quad a_3I \quad a_4I]\mathbf{f} = 0$$

对 $N \geq 6$ 对点联立该方程，即可求出 f_i^2，从而根据每个点的系数 $\mathbf{a}$ 求出在位姿 2 下的 3D 点坐标，从而使用 ICP 求解相对位姿 T_2^1。

综合上述推导，我们可以总结 EPnP 算法如下。

Algorithm 92: EPnP 算法

Input: 位姿 1 下的各点相机系坐标 $p_1^1, p_2^1, ..., p_N^1$

Input: 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$

Output: 位姿间齐次矩阵 T_2^1

// 计算控制点

$f_1^1 \leftarrow \frac{1}{N}\sum_{i=1}^{N} p_i^1$

$\bar{P} \leftarrow \begin{bmatrix} (p_1^1 - f_1^1)^T \\ (p_2^1 - f_1^1)^T \\ ... \\ (p_N^1 - f_1^1)^T \end{bmatrix}$

$v_{1:3}, \lambda_{1:3} \leftarrow \text{calc\_feature\_vector}(\bar{P}^T\bar{P})$

for $i \in 1, 2, 3$ **do**

  $f_{i+1}^1 \leftarrow f_1^1 + \sqrt{\frac{\lambda_i}{N}} v_i$

// 计算线性系数

$F_r \leftarrow \begin{bmatrix} f_1^1 & f_2^1 & f_3^1 & f_4^1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$

$A \leftarrow 0_{4\times N}$

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

  $A_{i,:} \leftarrow F_r^{-1}[(p_i^1)^T 1]^T$

// 求解控制点坐标

$M \leftarrow 0_{2N\times 12}$

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

  $V_i \leftarrow \begin{bmatrix} 1 & 0 & -u_i^2 \\ 1 & -v_i^2 & 0 \end{bmatrix}$

  $M_{2i:2i+1,:} \leftarrow V_i K[a_{1,i}I \quad a_{2,i}I \quad a_{3,i}I \quad a_{4,i}I]$

$\mathbf{f} \leftarrow \text{linear\_solve}(M^TM\mathbf{f} = 0)$

// 求解样本 3D 坐标

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

  $p_i^2 \leftarrow [a_{1,i}I \quad a_{2,i}I \quad a_{3,i}I \quad a_{4,i}I]\mathbf{f}$

// ICP 求解相对位姿

$T_2^1 \leftarrow \text{ICP\_calc}(p_:^1, p_:^2)$

算法	**EPnP 算法**
问题类型	3D-2D 点位姿求解
已知	位姿 1 下的各点相机系坐标 $p_1^1, p_2^1, ..., p_N^1$ 位姿 2 下的各点像素坐标 $v_1^2, v_2^2, ..., v_N^2$ 相机内参 K
求	位姿间齐次矩阵 T_2^1
算法性质	近似解

对应的 python 代码如下所示


```python
def get_inside_receiving_radius(points):
    assert points.shape == (4, 3), "points 应为形状为 (4, 3) 的 numpy 数组"
    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) · ((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 位姿求解

BA 的全称是 Bundle Adjustment，中文翻译为**光束平差法**。它的核心思想是：通过将多个视角的位姿误差和观测点的位置误差同时建模为优化变量，求解非线性优化问题，实现对于二者的同时优化。由于从不同相机观察某一个特征点的光线会汇成一束，而优化后光束交点的误差会减小，因此也称为光束平差法。

由于目前我们仅关心 3D-2D 问题的求解，因此我们建模的变量仅包含 1 个仅有 2D 观测的位姿估计的误差，不包括更多位姿，也不包括观测点位置的误差。事实上，该方法可以很容易推广到多个位姿优化的情况。

让我们先构造一个优化问题。具体来说，仍假设 $R = R_2^1, t = t_{12}^1$，则有

$$\begin{aligned} p^2 &= R_1^2 p^1 + t_{21}^2 \\ &= (R_2^1)^T p^1 - (R_2^1)^T t_{12}^1 \\ &= R^T(p^1 - t) \end{aligned}$$

则位姿 2 下的像素坐标

$$\hat{v}^2(R,t) = \frac{1}{z^2} K R^T (p^1 - t)$$

可见，位姿 2 下的像素坐标 $\hat{v}^2(R,t)$ 是 R,t 的函数。如果我们构造一个优化目标，就可以构成一个对 R,t 的最优化问题。接下来，进行最优化迭代求解，即可解决 3D-2D 问题。我们选择观测值 v^2 和估计值 $\hat{v}^2(R,t)$ 之间的误差平方作为损失函数，即

$$J(R,t) = \sum_{i=1}^{N} \frac{1}{2} ||v_i^2 - \hat{v}_i^2(R,t)||^2$$

在2.5章中我们介绍过，直接将旋转矩阵 R 作为优化变量不易求解导数，而以旋转向量 ϕ 作为替代更加容易。因此，我们实际求解的优化问题为

$$J(\phi,t) = \sum_{i=1}^{N} \frac{1}{2} ||e_i(\phi,t)||^2$$

其中

$$e_i(\phi,t) = v_i^2 - \hat{v}_i^2(\exp(\phi_\times), t)$$

总结上述计算流程，我们有

$$
\begin{aligned}
J(\phi,t) &= \sum_{i=1}^{N} \frac{1}{2}||e_i(\phi,t)||^2 \\
e_i(\phi,t) &= I_{2\times 3}(\upsilon_i^2 - \hat{\upsilon}_i^2) \\
\hat{\upsilon}_i^2 &= K\nu_i^2 \\
\nu_i^2 &= \frac{1}{z_i^2}p_i^2 \\
p_i^2 &= R^T(p^1 - t) \\
R &= \exp(\phi_\times)
\end{aligned}
\tag{VII.26.26}
$$

因此，根据链式法则，我们有优化目标对优化变量的导数

$$
\begin{aligned}
\frac{\partial J}{\partial \phi} &= \sum_{i=1}^{N}\left(\frac{\partial J}{\partial e_i}\frac{\partial e_i}{\partial \hat{\upsilon}_i^2}\frac{\partial \hat{\upsilon}_i^2}{\partial \nu_i^2}\frac{\partial \nu_i^2}{\partial p_i^2}\frac{\partial p_i^2}{\partial \phi}\right) \\
\frac{\partial J}{\partial t} &= \sum_{i=1}^{N}\left(\frac{\partial J}{\partial e_i}\frac{\partial e_i}{\partial \hat{\upsilon}_i^2}\frac{\partial \hat{\upsilon}_i^2}{\partial \nu_i^2}\frac{\partial \nu_i^2}{\partial p_i^2}\frac{\partial p_i^2}{\partial t}\right)
\end{aligned}
$$

其中

$$
\begin{aligned}
\frac{\partial J}{\partial e_i} &= e_i^T \\
\frac{\partial e_i}{\partial \hat{\upsilon}_i^2} &= \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} \\
\frac{\partial \hat{\upsilon}_i^2}{\partial \nu_i^2} &= K \\
\frac{\partial \nu_i^2}{\partial p_i^2} &= \frac{1}{z_i^2}\begin{bmatrix} 1 & 0 & -\frac{x_i^2}{z_i^2} \\ 0 & 1 & -\frac{y_i^2}{z_i^2} \\ 0 & 0 & 0 \end{bmatrix} \\
\frac{\partial p_i^2}{\partial \phi} &= (R^T(p_i^1 - t))_\times \\
\frac{\partial p_i^2}{\partial t} &= -R^T
\end{aligned}
$$

这里我们使用了式I.2.43求解李代数的导数。

记

$$
J_\nu(p) = \frac{1}{z}\begin{bmatrix} 1 & 0 & -\frac{x}{z} \\ 0 & 1 & -\frac{y}{z} \\ 0 & 0 & 0 \end{bmatrix}
\tag{VII.26.27}
$$

因此，有

$$
\begin{aligned}
\frac{\partial J}{\partial \phi} &= \sum_{i=1}^{N} \mathrm{diag}(-e_i)KJ_\nu(p_i^2)(R^T(p_i^1 - t))_\times \\
\frac{\partial J}{\partial t} &= \sum_{i=1}^{N} \mathrm{diag}(-e_i)KJ_\nu(p_i^2)R^T
\end{aligned}
\tag{VII.26.28}
$$

此时，我们已经完成关于 ϕ, t 的最小二乘非线性优化问题的构造。根据3.6节，我们可以选择梯度下降、高斯-牛顿、L-M 等方法进行求解。

需要注意的是，由于上面选择的是对旋转向量的右扰动，因此更新优化变量时，对 R 的更新应为

$$R_{k+1} \approx R_k(I + (\delta\phi)_\times) \tag{VII.26.29}$$

在这里，我们以高斯-牛顿法为例，整理求解算法全过程。由于该方法需要一个初值，我们可以使用 DLT 计算这个初值。

Algorithm 93: GN-BA 位姿求解

Input: 位姿 1 下的各点相机系坐标 $p_1^1, p_2^1, ..., p_N^1$

Input: 位姿 2 下的各点像素坐标 $\upsilon_1^2, \upsilon_2^2, ..., \upsilon_N^2$

Parameter: 阈值 ϵ

Output: 位姿间齐次矩阵 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**

$\hat{p}_i^2, \nu^2, \hat{\upsilon}_i^2 \leftarrow R_k^T(p_i^1 - t_k), \frac{1}{z_i^2} K\hat{p}_i^2, K\nu^2$

$(e_k)_{2i:2i+1} \leftarrow I_{2\times 3}(\upsilon_i^2 - \hat{\upsilon}_i^2)$

$J_1 \leftarrow \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$

$J_3 \leftarrow \frac{1}{z_i^2} \begin{bmatrix} 1 & 0 & -x_i^2/z_i^2 \\ 0 & 1 & -y_i^2/z_i^2 \\ 0 & 0 & 0 \end{bmatrix}$

$J_4 \leftarrow [(R_k^T(p^1 - t_k))_\times \quad R_k^T]$

$(J_k)_{2i:2i+1,:} \leftarrow J_1 K J_3 J_4$

$H_n, g_n \leftarrow J_k^T J_k, J_k^T e_k$

$\begin{bmatrix} \delta\phi \\ \delta t \end{bmatrix} \leftarrow -H_n^{-1} g_n$

$t_{k+1}, R_{k+1} \leftarrow t_k + \delta t, R_k(I + (\delta\phi)_\times)$

$k \leftarrow k + 1$

$R_2^1, t_{12}^1 \leftarrow R_k, t_k$

在上述 BA 算法中（即仅估计 1 个位姿），点数 N 需要满足 $2N \geq 6$，即 $N \geq 3$ 且不全共线。

算法	GN-BA 位姿求解
问题类型	3D-2D 点位姿求解
已知	位姿 1 下的各点相机系坐标 $p_1^1, p_2^1, ..., p_N^1$ 位姿 2 下的各点像素坐标 $\upsilon_1^2, \upsilon_2^2, ..., \upsilon_N^2$ 相机内参 K
求	位姿间齐次矩阵 T_2^1
算法性质	迭代解

对应的 python 代码如下所示

```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
```

实际使用中，高斯牛顿法收敛较慢，且算法收敛比较依赖于一个好的学习率。学习率过小则收敛过慢，且容易陷入局部极小值；学习率过大则算法可能不收敛。因此，我们往往使用其改进版本，即 L-M 法进行 BA 位姿求解。

26.5 RANSAC 方法

在本章中，我们已经介绍了不少用特征点信息恢复位姿的方法，分别针对 2D-2D/2D-3D/3D-3D 等不同的问题。然而，到目前为止，我们都默认使用的信息是准确的，即特征点的观测没有错误或异常值。事实上，实际数据不可能是理想的，一定存在一些错误的观测点，它们往往来自前端的错误匹配和跟踪。在所有的错误中，离群值往往会带来很大的估计误差。

针对这样的问题，有一类算法可以预先剔除数据中的离群值，防止结果被野值干扰造成误差很大。只要所有数据满足某种形式的线性约束，就可以通过随机采样的方式识别数据中的离群值。这类算法统称为

RANSAC(RANdom SAmple Consensus, 随机样本一致性)。根据数据的不同以及满足的线性约束的不同，VSLAM 中常常使用以下几种不同的 RANSAC：**4 点法 H 阵 RANSAC**、**8 点法 F 阵 RANSAC**，等等。

26.5.1 RANSAC 原理

在介绍具体的 RANSAC 之前，我们首先以线性拟合问题为例，介绍 RANSAC 的基本原理。

[本部分内容将在后续版本中更新，敬请期待]

26.5.2 4 点法 H 阵 RANSAC

[本部分内容将在后续版本中更新，敬请期待]

26.5.3 8 点法 F 阵 RANSAC

[本部分内容将在后续版本中更新，敬请期待]

26.5.4 6 点法 DLT-RANSAC

[本部分内容将在后续版本中更新，敬请期待]

27 间接 VO 方法

27.1 MSCKF 方法

MSCKF 是一种基于光流前端和滑窗滤波后端的 VIO 方法，2007 年由 Mourikis 和 Roumeliotis 共同提出，2017 年由 Sun Ke 等扩展为双目 VIO。该方法通过建立误差状态模型，将 VIO 问题建模为一个非线性滤波问题，并使用 EKF 进行滤波求解。

27.1.1 总体框架

MSCKF 是一个前后端分离结构的 VIO。其中，前端输入 IMU 和图像信息，输出已匹配 (分配统一 id) 的 2D 特征点坐标；后端输入 2D 特征点坐标，输出位姿速等状态估计，同时维护地图。MSCKF 的后端不含回环检测、重定位等部分，因此一般仅认为是一个 VIO。

MSCKF 的前端由以下部分构成：FAST 特征点检测、基于金字塔 L-K 光流法的特征点匹配、RANSAC 外点剔除、特征点注册等。双目版本的 MSCKF 还包括双目的特征匹配和剔除。

MSCKF 后端的核心是一个第11章介绍过的 ESKF，包括：状态预测、状态更新、滑窗管理与边缘化等部分。

MSCKF 的坐标系包括：载体系 b、导航系 n、相机系 c。

- n 系定义为：$x-y-z$ 轴分别指向东/北/天方向，原点相对于地面静止的坐标系。
- b 系定义为：与移动机器人 (载体) 的 IMU 固联的坐标系。
- c 系定义为：与相机固联的坐标系，定义与”相机模型”一节中的介绍一致。

27.1.2 光流法前端

MSCKF 的前端属于光流间接法，首先用 FAST 方法检测特征点，然后通过金字塔 L-K 光流法。MSCKF 的前端算法总结如下

[本部分内容将在后续版本中更新，敬请期待]

27.1.3 ESKF 状态定义

MSCKF 的系统状态分为两部分：**IMU 状态**和**相机状态**。在所有状态中，均使用使用从 b 系/c 系到 n 系坐标变换的四元数表示旋转。

IMU 状态是直接和 IMU 相关的状态变量，包括：载体相对 n 系的姿态 q_b^n, 载体相对 n 系的位置 p_{nb}^b, 载体相对 n 系的速度 v_{nb}^b，陀螺仪零偏 b_g，加速度计零偏 b_a。IMU 状态满足运动学微分方程，该方程推导见下一小节。

IMU 状态的估计状态为

$$\hat{x}_{IMU} := \begin{bmatrix} q_{nb'}^{b'} & p_{nb'}^{n} & v_{nb'}^{n} & \hat{b}_g & \hat{b}_a \end{bmatrix}^T$$

式中的 b' 系表示：对载体位姿估计不准确时，估计值对应的载体坐标系。估计状态也满足一个微分方程，该方程推导见下一小节。

我们也可以认为，b' 系是 b 系施加微小扰动后得到的坐标系，此扰动即为姿态的误差状态。对于旋转，假设 $\delta\theta$ 表示坐标轴从 b' 系旋转到 b 系的 3 维小角度向量。根据式I.2.21和式I.2.2，我们有

$$q_b^{b'} \approx \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix}$$
$$R_b^{b'} \approx I + (\delta\theta)_\times$$

也就是说，相当于在四元数 $q_{b'}^n$ 和旋转矩阵 $R_{b'}^n$ 上各自施加的右扰动

$$q_b^n \approx q_{b'}^n \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix}$$
$$R_b^n \approx R_{b'}^n (I + (\delta\theta)_\times)$$

由此，IMU 状态的误差状态定义为

$$\tilde{x}_{IMU} := \begin{bmatrix} \delta\theta^T & (\tilde{p}_{nb}^n)^T & (\tilde{v}_{nb}^n)^T & (\tilde{b}_g)^T & (\tilde{b}_a)^T \end{bmatrix}^T \in \mathbb{R}^{15}$$

其中

$$\begin{aligned} q_b^n &\approx q_{b'}^n \otimes \begin{bmatrix} 1 \\ -\frac{1}{2}\delta\theta \end{bmatrix} \\ p_{nb}^n &= \tilde{p}_{nb}^n + p_{nb'}^n \\ v_{nb}^n &= \tilde{v}_{nb}^n + v_{nb'}^n \\ b_g &= \tilde{b}_g + \hat{b}_g \\ b_a &= \tilde{b}_a + \hat{b}_a \end{aligned} \tag{VII.27.1}$$

IMU 误差状态满足线性化方程，该方程见下一小节。

相机状态是相机的历史位姿。一个历史位姿包括相机姿态 q_c^n 与相机位置 p_{nc}^n。全状态包括 IMU 状态与多个相机历史位姿。

$$x_{c_j} = [(q_{c_j}^n)^T \quad (p_{nc_j}^n)^T]^T$$

$$x = \begin{bmatrix} x_{IMU}^T & x_{c_m}^T & x_{c_{m+1}}^T & \dots & x_{c_{m+l}}^T \end{bmatrix}^T$$

和 IMU 状态相同，我们认为估计的相机状态和真实状态之间也有误差。我们使用旋转向量扰动 $\delta\theta_{c_j}$ 表示 $q_{c_j'}^{c_j}$ 即姿态估计误差，用 $\tilde{p}_{nc_j}^n$ 表示位置估计误差

$$\begin{aligned} q_{c_j'}^n &= q_{c_j}^n \otimes q_{c_j'}^{c_j} = q_{c_j}^n \otimes \begin{bmatrix} 0 \\ \frac{1}{2}\delta\theta_{c_j} \end{bmatrix} \\ \tilde{p}_{nc_j}^n &= p_{nc_j}^n - p_{nc_j'}^n \end{aligned} \tag{VII.27.2}$$

二者组成相机的误差状态

$$\tilde{x}_{c_j} = [(\delta\theta_{c_j})^T \quad (\tilde{p}_{nc_j}^n)^T]^T \tag{VII.27.3}$$

所有相机误差状态和 IMU 误差状态共同构成总误差状态，即

$$\tilde{x} = \begin{bmatrix} \tilde{x}_{IMU}^T & \tilde{x}_{c_m}^T & \tilde{x}_{c_{m+1}}^T & \dots & \tilde{x}_{c_{m+l}}^T \end{bmatrix}^T \in \mathbb{R}^{15+6l}$$

27.1.4 系统方程

状态方程

在本小节中，我们具体推导 IMU 状态满足的微分方程。

我们知道，位置和姿态是 VIO 问题的输出。位置的一阶导数为速度，二阶导数为加速度；姿态的一阶导数可以认为是角速度。作为惯性器件的 IMU，能够对角速度和加速度进行测量，也就是对目标的二阶微分进行测量。因此，我们首先需要构建一个包含位置/姿态/速度/加速度/角速度等物理量的二阶微分方程，从而说明以上物理量的关系。

具体来说，我们**使用从 b 系到 n 系坐标变换的四元数表示旋转**。在忽略科里奥利力的理想情况下，IMU 状态满足的载体运动学微分方程为

$$\begin{aligned}\dot{q}_b^n &= q_b^n \otimes \frac{1}{2}(\omega^b - b_g)\\ \dot{v}_{nb}^n &= R_b^n(f^b - b_a) + \mathbf{g}^n\\ \dot{p}_{nb}^n &= v_{nb}^n\end{aligned}$$

式中，$q_b^n, p_{nb}^b, v_{nb}^b$ 分别表示载体相对 n 系的姿态、位置和速度。f^b 表示 IMU 加速度计的直接测量量，即比力。ω^b 表示 IMU 陀螺仪的直接测量量，即角速度。$\mathbf{g}^n$ 表示 n 系下的重力加速度矢量。b_g 和 b_a 分别表示陀螺仪和加速度计的零偏，均为 3 维向量。对于 MEMS 的 IMU 而言，每次上电时它们的取值都会变化。

对于实际系统，IMU 传感器的测量会有噪声；且陀螺仪和加速度计的零偏会进行随机游走。考虑噪声因素后，系统方程为

$$\begin{aligned}\dot{q}_b^n &= q_b^n \otimes \frac{1}{2}(\omega^b - b_g - n_g)\\ \dot{v}_{nb}^n &= R_b^n(f^b - b_a - n_a) + \mathbf{g}^n\\ \dot{p}_{nb}^n &= v_{nb}^n\\ \dot{b}_g &= n_{wg}\\ \dot{b}_a &= n_{wa}\end{aligned}$$

可以看到，这是一个非线性的随机系统。对于我们的 VIO 任务，我们取 $q_b^n, p_{nb}^n, v_{nb}^n, b_g, b_a$ 为真实状态。由于我们需要将姿态作为状态，因此采用11.3.5节介绍的 ESKF 方法。除上述真实状态外，我们还应定义估计状态 $\hat{x}$ 和误差状态 $\tilde{x}$(即 δx，此处符号与 MSCKF 论文保持一致)。

具体来说，MSCKF 定义的估计状态 $q_{nb'}^{b'}, p_{nb'}^n, v_{nb'}^n, \hat{b}_g, \hat{b}_a$ 应满足方程

$$\begin{aligned}\dot{q}_{b'}^n &= q_{b'}^n \otimes \frac{1}{2}(\omega^b - \hat{b}_g)\\ \dot{v}_{nb'}^n &= R_{b'}^n(f^b - \hat{b}_a) + \mathbf{g}^n\\ \dot{p}_{nb'}^n &= v_{nb'}^n\\ \dot{\hat{b}}_g &= 0\\ \dot{\hat{b}}_a &= 0\end{aligned}$$

误差状态方程

上面介绍的 IMU 估计状态的微分方程是非线性的，而误差状态满足下列线性微分方程

$$\dot{\tilde{x}}_{IMU} = F_{IMU}\tilde{x}_{IMU} + G_{IMU}\mathbf{n}_{IMU} \tag{VII.27.4}$$

其中

$$F_{IMU} = \begin{bmatrix} -(\omega^b - \hat{b}_g)_\times & 0_{3\times3} & 0_{3\times3} & -I_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & I_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ -R^n_{b'}(f^b - \hat{b}_a)_\times & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & -R^n_{b'} \\ & & 0_{3\times15} & & \\ & & 0_{3\times15} & & \end{bmatrix}$$

$$G_{IMU} = \begin{bmatrix} -I_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & -R^n_{b'} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & I_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & I_{3\times3} \end{bmatrix} \tag{VII.27.5}$$

$$\mathbf{n}_{IMU} = [n_g \quad n_a \quad n_{wg} \quad n_{wa}]^T \sim \mathcal{N}(0, Q)$$

上述形式是误差状态微分方程的矩阵表达形式。将其拆开，我们可以得到如下的线性微分方程形式

$$\begin{aligned} \dot{\delta\theta} &= -(\omega^b - \hat{b}_g)_\times(\delta\theta) - \tilde{b}_g - n_g \\ \dot{\tilde{p}}^n_{nb} &= \tilde{v}^n_{nb} \\ \dot{\tilde{v}}^n_{nb} &= -R^n_{b'}(f^b - \hat{b}_a)_\times(\delta\theta) - R^n_{b'}\tilde{b}_a - R^n_{b'}n_a \\ \dot{\tilde{b}}_g &= n_{wg} \\ \dot{\tilde{b}}_a &= n_{wa} \end{aligned} \tag{VII.27.6}$$

误差状态方程推导

式VII.27.6中的第 2,4,5 行易得。下面证明其中的第一行和第三行，即关于旋转矢量扰动和速度误差的微分方程。

对速度误差，根据上述误差关系，有

$$\begin{aligned} \dot{\tilde{v}}^n_{nb} &= \dot{v}^n_{nb} - \dot{v}^n_{nb'} \\ &= R^n_b(f^b - b_a - n_a) + \mathbf{g}^n - (R^n_{b'}(f^b - \hat{b}_a) + \mathbf{g}^n) \\ &= (R^n_b - R^n_{b'})f^b - R^n_b b_a + R^n_{b'}\hat{b}_a - R^n_b n_a \\ &= (R^n_b - R^n_{b'})f^b - R^n_b b_a + R^n_b\hat{b}_a - R^n_b\hat{b}_a + R^n_{b'}\hat{b}_a - R^n_b n_a \\ &= (R^n_b - R^n_{b'})f^b - R^n_b(b_a - \hat{b}_a) + (R^n_{b'} - R^n_b)\hat{b}_a - R^n_b n_a \\ &= (R^n_b - R^n_{b'})(f^b - \hat{b}_a) - R^n_b\tilde{b}_a - R^n_b n_a \\ &\approx (R^n_{b'}(I + (\delta\theta)_\times) - R^n_{b'})(f^b - \hat{b}_a) - R^n_{b'}\tilde{b}_a - R^n_{b'}n_a \\ &\approx R^n_{b'}(\delta\theta)_\times(f^b - \hat{b}_a) - R^n_{b'}\tilde{b}_a - R^n_{b'}n_a \\ &= -R^n_{b'}(f^b - \hat{b}_a)_\times(\delta\theta) - R^n_{b'}\tilde{b}_a - R^n_{b'}n_a \end{aligned}$$

在证明旋转矢量扰动的微分方程前，我们首先推导一下 $\dot{q}^n_b$。根据前述定义，我们有近似关系

$$q^n_b \approx q^n_{b'} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix}$$

根据四元数乘法定义，两个四元数乘积的微分也类似于普通乘法。对上式左右两边求微分，有

$$\dot{q}_b^n = \dot{q}_{b'}^n \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} + q_{b'}^n \otimes \begin{bmatrix} 0 \\ \frac{1}{2}\dot{\delta\theta} \end{bmatrix}$$

由于

$$\dot{q}_b^n = q_b^n \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - b_g - n_g) \end{bmatrix}$$
$$\dot{q}_{b'}^n = q_{b'}^n \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - \hat{b}_g) \end{bmatrix}$$

因此

$$q_b^n \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - b_g - n_g) \end{bmatrix} = q_{b'}^n \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - \hat{b}_g) \end{bmatrix} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} + q_{b'}^n \otimes \begin{bmatrix} 0 \\ \frac{1}{2}\dot{\delta\theta} \end{bmatrix}$$

继续展开 q_b^n，

$$q_{b'}^n \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - b_g - n_g) \end{bmatrix} = q_{b'}^n \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - \hat{b}_g) \end{bmatrix} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} + q_{b'}^n \otimes \begin{bmatrix} 0 \\ \frac{1}{2}\dot{\delta\theta} \end{bmatrix}$$

消去 q_b^n，有

$$\begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} \otimes \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - b_g - n_g) \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{1}{2}(\omega^b - \hat{b}_g) \end{bmatrix} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\delta\theta \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{2}\dot{\delta\theta} \end{bmatrix}$$

由式I.2.12，取向量部分，有

$$\frac{1}{2}(\omega^b - b_g - n_g) + \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(\omega^b - b_g - n_g) = \frac{1}{2}(\omega^b - \hat{b}_g) + \frac{1}{2}(\omega^b - \hat{b}_g) \times \left(\frac{1}{2}\delta\theta\right) + \frac{1}{2}\dot{\delta\theta}$$

移项，有

$$\begin{aligned}
\frac{1}{2}\dot{\delta\theta} &= \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(\omega^b - b_g - n_g) - \frac{1}{2}(\omega^b - \hat{b}_g) \times \left(\frac{1}{2}\delta\theta\right) + \frac{1}{2}(\hat{b}_g - b_g - n_g) \\
&= \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(\omega^b - b_g - n_g) + \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(\omega^b - \hat{b}_g) + \frac{1}{2}(\hat{b}_g - b_g - n_g) \\
&= \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(2\omega^b - 2\hat{b}_g - \tilde{b}_g - n_g) + \frac{1}{2}(-\tilde{b}_g - n_g) \\
&\approx \left(\frac{1}{2}\delta\theta\right) \times \frac{1}{2}(2\omega^b - 2\hat{b}_g) - \frac{1}{2}(\tilde{b}_g + n_g)
\end{aligned}$$

即

$$\begin{aligned}
\dot{\delta\theta} &= (\delta\theta) \times (\omega^b - \hat{b}_g) - \tilde{b}_g - n_g \\
&= -(\omega^b - \hat{b}_g) \times (\delta\theta) - \tilde{b}_g - n_g \\
&= -(\omega^b - \hat{b}_g)_\times (\delta\theta) - \tilde{b}_g - n_g
\end{aligned}$$

以上就是系统误差状态的连续时间线性微分方程。实际使用时，会将其离散化，详见下面的” 滤波预测步” 小节。

27.1.5 滤波初值

MSCKF 的估计状态需要给出初值。一般我们默认初始时载体静止，且以初始位置为 n 系原点。因此我们实际仅需估计 q_b^n, b_g 和 b_a。其中，b_g 的估计较为简单，只需采集一段时间的陀螺仪数据平均即可。

对于加速度计的 (平均) 测量值 f^b，它同时涉及姿态和 b_a。我们有

$$f^b = -g^b + b_a = -(R_b^n)^T g^n + b_a$$

其中，n 系下的重力加速度

$$g^n = \begin{bmatrix} 0 \\ 0 \\ -g_0 \end{bmatrix}$$

对于 R_b^n，由于航向不影响测量值，我们仅考虑其包含横滚角 r 和俯仰角 y

$$R_b^n = R_x(p)R_y(r) = \begin{bmatrix} \cos r & 0 & \sin r \\ \sin p \sin r & \cos p & -\sin p \cos r \\ -\cos p \sin r & \sin p & \cos p \cos r \end{bmatrix}$$

因此，我们有

$$f^b = -\begin{bmatrix} \cos r & 0 & \sin r \\ \sin p \sin r & \cos p & -\sin p \cos r \\ -\cos p \sin r & \sin p & \cos p \cos r \end{bmatrix}^T \begin{bmatrix} 0 \\ 0 \\ -g_0 \end{bmatrix} + b_a$$

即

$$f^b = g_0 \begin{bmatrix} -\cos p \sin r \\ \sin p \\ \cos p \cos r \end{bmatrix} + b_a$$

假设 b_a 和 f^b 共线，有

$$f^b / ||f^b|| = \begin{bmatrix} -\cos p \sin r \\ \sin p \\ \cos p \cos r \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}$$

即

$$\begin{aligned} p &= \arcsin(f_2) \\ r &= \arctan\left(\frac{-f_1}{f_3}\right) \end{aligned} \tag{VII.27.7}$$

此时，有

$$b_a = f^b - g_0 ||f^b|| \tag{VII.27.8}$$

27.1.6 滤波预测步

MSCKF 的滤波预测步是指：在不考虑量测的情况下，通过系统方程由 $\hat{x}_k$ 和 P_k 推测 $\hat{x}_k^-$ 和 P_{k+1}^-，也就是进行状态预测和协方差矩阵预测。

状态预测 ESKF 要求给出离散时间的系统方程及其一阶导数。IMU 状态的系统方程 (式VII.27.5) 是连续形式，并且不包括相机位姿。因此，我们需要首先将其扩展为全状态，再将其离散化。

注意到滑窗中的相机位姿是历史位姿，它们不随当前时间变化。因此，在连续域下，全状态的微分方程为

$$\dot{\hat{x}} = f(\hat{x}) = \begin{bmatrix} q_{b'}^n \otimes \frac{1}{2}(\omega^b - \hat{b}_g) \\ R_{b'}^n (f^b - \hat{b}_a) + \mathbf{g}^n \\ v_{nb'}^n \\ 0_{6\times 1} \\ 0_{6l\times 1} \end{bmatrix}$$

接下来，我们考虑将其离散化。假设离散时间间隔为 t_s，有

$$\hat{x}_{k+1}^- = \hat{x}_k + \int_{kt_s}^{(k+1)t_s} f(x_k) dt$$

实际使用时，我们往往使用 RK4 等数值积分方法进行离散化的状态预测。

协方差预测 对于协方差预测，我们其实需要同时考虑**线性化**和**离散化**两个部分。我们考虑先在连续域进行线性化近似，然后对线性化系统进行离散化。具体来说，对上面的非线性连续状态方程，其线性化为

$$\dot{\hat{x}} \approx F_c(\hat{x})\hat{x}$$

其中角标 c 表示 continuous，有

$$F_c(\hat{x}) = \begin{bmatrix} F_{IMU}(\hat{x}_{IMU}) & 0_{15\times 6l} \\ 0_{6l\times 15} & 0_{6l\times 6l} \end{bmatrix}$$

参考连续时间卡尔曼滤波的结论，线性化近似情况下，协方差 P 的微分方程为

$$\dot{P} = F_c P + P F_c^T + G_c Q G_c^T$$

其中，Q 为系统噪声 $\mathbf{n}_{IMU}$ 的协方差，G_c 表示噪声对于系统导数的贡献，有

$$G_c = \begin{bmatrix} G_{IMU}(\hat{x}_{IMU}) \\ 0_{6l\times 12} \end{bmatrix}$$

假设连续状态协方差 P 满足 $P = P^T$，且有分块

$$P = \begin{bmatrix} P_{11} & P_{12} \\ P_{12}^T & P_{22} \end{bmatrix} \tag{VII.27.9}$$

将 F_c, G_c 表达式代入上述微分方程，有

$$\begin{bmatrix} \dot{P}_{11} & \dot{P}_{12} \\ \dot{P}_{12}^T & \dot{P}_{22} \end{bmatrix} = \begin{bmatrix} F_{IMU}P_{11} + P_{11}F_{IMU} + G_{IMU}QG_{IMU}^T & F_{IMU}P_{12} \\ P_{12}^T F_{IMU}^T & 0_{6l\times 6l} \end{bmatrix} \tag{VII.27.10}$$

即

$$\begin{aligned}\dot{P}_{11} &= F_{IMU}P_{11} + P_{11}F_{IMU} + G_{IMU}QG_{IMU}^T \\ \dot{P}_{12} &= F_{IMU}P_{12} \\ \dot{P}_{22} &= 0_{6l\times 6l}\end{aligned}$$

对于 P_{11} 和 P_{12}，假设忽略 F_{IMU} 和 G_{IMU} 的微小变化，这是一个初值已知的线性常微分方程，其解析解为

$$\begin{aligned}P_{k,11}^- &= \exp(F_{IMU}t_s)P_{k,11}\exp(F_{IMU}t_s)^T + Q_{11} \\ Q_{11} &= \int_{kt_s}^{kt_s+t}\exp(F_{IMU}(t-\tau))G_{IMU}QG_{IMU}^T\exp(F_{IMU}(t-\tau))^T d\tau \\ \dot{P}_{k,12}^- &= \exp(F_{IMU}t_s)P_{k,12}\end{aligned}$$

这里面的 $\exp(\cdot)$ 表示矩阵指数。记

$$\Phi_k = \exp(F_{IMU}(\hat{x}_k)t_s) \tag{VII.27.11}$$

则有

$$\begin{aligned}P_{k,11}^- &= \Phi_k P_{k,11}\Phi_k + Q_{11} \\ Q_{11} &\approx \Phi_k G_{IMU}(Qt_s)G_{IMU}^T\Phi_k^T \\ P_{k,12}^- &= \Phi_k P_{k,12} \\ P_{k,22}^- &= P_{k,22}\end{aligned} \tag{VII.27.12}$$

实际计算时，矩阵指数 Φ_k 可使用 3 阶泰勒展开近似

$$\Phi_k \approx I + F_{IMU}t_s + \frac{t_s^2}{2}F_{IMU}^2 + \frac{t_s^3}{6}F_{IMU}^3$$

综上，我们总结 MSCKF 的预测步算法如下

Algorithm 94: MSCKF 预测步

Input: k 步估计状态 $\hat{x}_k$, 估计协方差 P_k

Input: 系统噪声协方差矩阵 Q

Input: 比力 $\mathbf{f}^b$，角速度 $\mathbf{w}_{nb}^b$

Output: k 步预测状态 $\hat{x}_k^-$, 预测协方差 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 相机状态扩张

滑窗内相机的误差状态和 IMU 误差状态一起，组成总的**误差状态**

$$\tilde{x} = \begin{bmatrix} \tilde{x}_{IMU}^T & \tilde{x}_{c_m}^T & \tilde{x}_{c_{m+1}}^T & \dots & \tilde{x}_{c_{m+l}}^T \end{bmatrix}^T \in \mathbb{R}^{15+6l} \tag{VII.27.13}$$

假设我们已经有当前载体的位姿估计 $\hat{x}_{IMU}$。当相机采集一幅新图像 I_k 时，如何生成对应的相机位姿估计 $\hat{x}_{c_j}$ 呢？显然，根据坐标系间关系，我们有

$$\begin{aligned} q_{c_j'}^n &= q_{b'}^n \otimes q_c^b \\ p_{nc_j'}^n &= p_{nb'}^n + R_{b'}^n p_{bc}^b \end{aligned} \tag{VII.27.14}$$

其中，q_c^b 可由 T_c^b 中的 R_c^b 计算。

由于 $R_{b'}^n/q_{b'}^n$ 和 $p_{nb'}^n$ 均为估计值，IMU 误差状态 x_{IMU} 就被传递到了相机误差状态 $\tilde{x}_{c_j}$ 中。具体来说，对于相机位置误差

$$\begin{aligned} \tilde{p}_{nc_j}^n &= p_{nc_j}^n - p_{nc_j'}^n \\ &= p_{nb}^n + R_b^n p_{bc}^b - (p_{nb'}^n + R_{b'}^n p_{bc}^b) \\ &= \tilde{p}_{nb}^n + (R_b^n - R_{b'}^n) p_{bc}^b \\ &\approx \tilde{p}_{nb}^n + (R_b^n - R_b^n(I + (\delta\theta)_\times)) p_{bc}^b \\ &= \tilde{p}_{nb}^n - R_b^n (\delta\theta)_\times p_{bc}^b \\ &= \tilde{p}_{nb}^n + R_b^n (p_{bc}^b \times (\delta\theta)) \\ &= \tilde{p}_{nb}^n + R_b^n (p_{bc}^b)_\times (\delta\theta) \end{aligned}$$

对于相机姿态误差

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

因此

$$q_{c_j'}^{c_j} = q_b^c q_{b'}^b \otimes (q_b^c)^{-1}$$

即

$$\begin{bmatrix} 0 \\ \frac{1}{2}\delta\theta_{c_j} \end{bmatrix} = q_b^c \begin{bmatrix} 0 \\ \frac{1}{2}\delta\theta \end{bmatrix} \otimes (q_b^c)^{-1}$$

由式I.2.21，向量间有旋转变换关系

$$\delta\theta_{c_j} = R_b^c(\delta\theta)$$

总结一下，我们有

$$\begin{aligned} \delta\theta_{c_j} &= R_b^c(\delta\theta) \\ \tilde{p}_{nc_j}^n &= \tilde{p}_{nb}^n + (p_{bc'}^b)_\times(\delta\theta) \end{aligned} \tag{VII.27.15}$$

我们记表示 $\tilde{x}_{c_i}$ 和 $\tilde{x}_{IMU}$ 之间线性关系的矩阵为 J_{c_i}，有

$$J_{c_i} = \begin{bmatrix} R_b^c & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ R_b^n(p_{bc}^b)_\times & I_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \end{bmatrix} \in \mathbb{R}^{6\times15} \tag{VII.27.16}$$

27.1.8 滤波修正步

在 ESKF 框架中，修正步的核心是求出量测方程/输出方程的线性化矩阵 H_k，它表示了观测误差和误差状态之间的线性化关系。类似于此前 BA 问题的设定 (见26.4.3小节)，我们以重投影误差 (即预测步的估计位姿下特征点的像素坐标和实际观测像素坐标间的误差) 作为观测量。

假设对第 j 个相机位姿，简写其估计状态为 $p_{c,j} = p_{nc_j'}^n, R_j = R_{c_j'}^n$；且对第 i 个 3D 特征点，$p_i = p_{ni}^n$ 为 n 系下的 3D 坐标。假设第 j 个相机对第 i 个特征点的观测量为 $z_{i,j}$，则

$$z_{i,j} = v_i^{c_j} - \hat{v}_i(R_j, p_{c,j}) \tag{VII.27.17}$$

其中 $v_i^{c_j}$ 为测量值，且

$$\begin{aligned} \hat{v}_i(R_j, p_{c,j}) &= \frac{1}{z_j^{c_j}} K p_i^{c_j} \\ p_i^{c_j} &= R_j^T(p_i - p_{c,j}) \end{aligned} \tag{VII.27.18}$$

考虑量测 $z_{i,j}$ 对全误差状态 $\tilde{x}$ 的导数，可知该导数中，仅第 j 个相机状态 $\tilde{x}_{c_j}$ 对应的块不为 0。虽然 IMU 状态 $\tilde{x}_{IMU}$ 也和 $\tilde{x}_{c_j}$ 相关，但我们在计算 H 阵时认为二者独立。设

$$H_{i,j} = \frac{\partial z_{i,j}}{\partial \tilde{x}_{c_j}} \tag{VII.27.19}$$

参考第26.4.3小节的推导，有

$$H_{i,j} = \frac{\partial \hat{v}_i^{c_j}}{\partial \nu_i^{c_j}} \frac{\partial \nu_i^{c_j}}{\partial p_i^{c_j}} \frac{\partial p_i^{c_j}}{\partial \tilde{x}_{c_j}}$$

且

$$\begin{aligned}
\frac{\partial \hat{v}_i^{c_j}}{\partial \nu_i^{c_j}} &= K \\
\frac{\partial \nu_i^{c_j}}{\partial p_i^{c_j}} &= \frac{1}{z_i^{c_j}} \begin{bmatrix} 1 & 0 & -x_i^{c_j}/z_i^{c_j} \\ 0 & 1 & -y_i^{c_j}/z_i^{c_j} \end{bmatrix} \\
\frac{\partial p_i^{c_j}}{\partial \tilde{x}_{c_j}} &= \begin{bmatrix} (R_j^T(p_i - p_{c,j}))_\times & -R_j^T \end{bmatrix}
\end{aligned}$$

这样，假设了特征点坐标 p_i 已知，我们就可以求出矩阵 H_k。

27.1.9 滑窗管理与边缘化

[本部分内容将在后续版本中更新，敬请期待]

27.2 VINS-Fusion

[本部分内容将在后续版本中更新，敬请期待]

28 直接 VO 方法

28.1 DSO 方法

[本部分内容将在后续版本中更新，敬请期待]

29 基于渲染的 VO 方法

29.1 3D 高斯泼溅

[本部分内容将在后续版本中更新，敬请期待]

29.2 MonoGS

[本部分内容将在后续版本中更新，敬请期待]