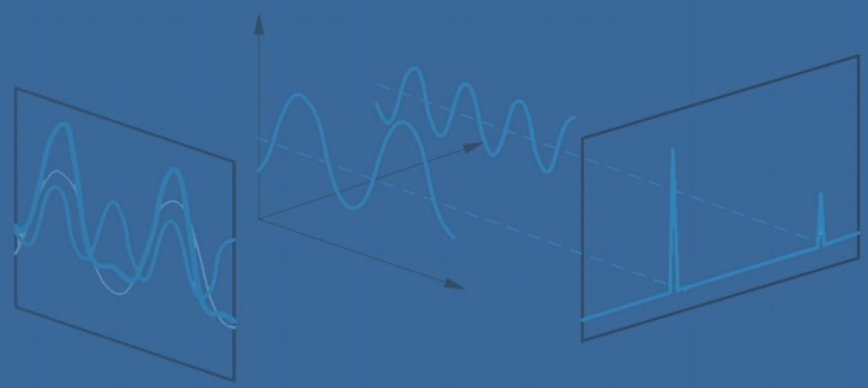
新一代计算机与人工智能系列新形态教材

人工智能数学基础

主编 李奕 陈莲 普远航



南京大学出版社

内容提要

本书为高职院校电子信息大类、人工智能等专业的课程教材，以“应用导向、问题驱动”为核心理念，系统整合了线性代数、微积分、复数与向量、傅里叶分析、概率统计、数值计算与信息论等内容，并将其与电子信息工程、人工智能领域的典型问题深度融合。教材强调“工具赋能”，将 Python 编程贯穿始终，通过 NumPy、Matplotlib、SymPy、SciPy、Pandas 等基础库实现数学概念的代码化表达与可视化展示，注重学生动手实践能力的培养。本书每章设有工程问题、核心概念、应用案例、调试排错、扩展阅读、巩固练习等模块，结构清晰、案例丰富，旨在帮助学生建立“数学—代码—工程”三位一体的知识体系。

图书在版编目(CIP)数据

人工智能数学基础 / 李奕，陈莲，普远航主编.
南京 ：南京大学出版社，2026. 8. -- ISBN 978-7-305-30332-6

Ⅰ. TP18；O29

中国国家版本馆 CIP 数据核字第 2026189PA6 号

出 版 者　南京大学出版社
社　　址　南京市汉口路 22 号　　邮　　编　210093

书　　名　人工智能数学基础
RENGONG ZHINENG SHUXUE JICHU
主　　编　李　奕　陈　莲　普远航
责任编辑　刘　飞　　编辑热线　025-83592146

照　　排　南京开卷文化传媒有限公司
印　　刷　南京迅驰彩色印刷有限公司
开　　本　787 mm×1092 mm　1/16 开　印张 17　字数 393 千
版　　次　2026 年 8 月第 1 版　　2026 年 8 月第 1 次印刷
I S B N　978-7-305-30332-6
定　　价　55.00 元

网　　址　http://www.njupco.com
官方微博　http://weibo.com/njupco
微信服务号　NJUYUNSHU
销售咨询热线　(025)83594756

前　言

我们正处在一个智能技术席卷全球的时代。从 ChatGPT 到自动驾驶，从智能医疗到工业物联网，人工智能技术正在深刻重塑各行各业的面貌。在这场智能化浪潮中，数学作为支撑人工智能发展的基石，其重要性愈发凸显。然而，传统的数学教材往往偏重理论推导，与工程实践相距甚远，使得许多渴望进入人工智能领域的学习者望而却步。

本教材坚持以党的二十大精神为指导，为高职院校电子信息大类专业学生量身打造了基础数学教材，同时也适用于所有希望掌握人工智能数学基础的初学者。我们深切了解高职学生群体的学习特点与真实需求——数学基础较为薄弱，但对动手实践和专业应用充满兴趣。因此，本书彻底告别传统数学教材中以理论推导为核心的编写模式，转而以“应用导向、问题驱动”为根本指导思想，致力于培养“会使用数学工具解决工程问题”的技术技能人才。

本书的编写严格遵循以下核心理念：

1. 内容选取坚持“必需、够用”

所有数学概念均从真实的人工智能工程问题引出，强调其在实际应用中的价值。无论是软件算法实现还是硬件部署运维，数学工具都发挥着不可替代的作用。

2. 工具赋能，代码落地

我们选择 Python 及其核心科学计算库（NumPy，Matplotlib，SymPy，SciPy，Pandas）作为贯穿全书的数学实践工具。特别强调的是，本书重在揭示数学工具的代码化，提倡使用基础库手动实现，避免对高级封装库的“黑箱”调用，深化学生对数学应用的理解。

3. 软硬融合，彰显抽象力量

教材以问题域为中心组织内容，同时涵盖人工智能在软件应用和硬件部署两个维度。通过展示同一数学工具在不同工程场景下的应用，帮助学生理解数学的强大通用性与抽象一致性。

4. 结构清晰，符合认知规律

全书共分十个项目，从数学语言与符号开始，逐步深入到向量与矩阵、线性方程组、函数与微积分、复数与相量法、傅里叶分析、概率统计、数值优化及信息论核心概念。知识脉络的设计力求符合学生的认知生长路径，确保数学思想的自然内化。

5. 精心设计的教学栏目

每一项目均设有【工程问题】引出学习动机，【核心概念】结合最小化 Python 代码实现“概念即代码”，【应用案例】展示跨领域应用，【扩展阅读】融入思政元素，【巩固练习】分层检验学习效果。项目末的【微型项目】则引导学生综合运用所学知识解决贴近真实的工程任务。

人工智能的发展离不开数学这一基础支撑，而数学的价值也在人工智能时代得到了前所未有的彰显。我们衷心希望，这本教材能够帮助读者克服对数学的畏惧，建立运用数学工具解决实际工程问题的信心与能力，无论是在软件开发还是硬件部署岗位上，都能成为推动人工智能技术落地的中坚力量。

由于编者水平有限，书中难免存在疏漏与不足之处，恳请广大师生和读者不吝指正。

编　者

2026 年 5 月

目　录

项目 1

数学语言与符号：工程师的交流工具

“宇宙之大，粒子之微，火箭之速，化工之巧，地球之变，生物之谜，日用之繁，无处不用数学。”

——华罗庚

学习目标

知识维度

1. 掌握算术运算、求和求积符号（$\sum$、$\prod$）、对数与自然常数(e)、角度与弧度的数学定义及转换关系；

2. 理解数学符号在电子信息与人工智能领域的规范用法，能区分易混淆符号的含义。

能力维度

1. 能运用 Python 的 NumPy、SymPy 库实现基本数学运算、单位转换及符号运算；

2. 能将工程问题中的批量数据、旋转角度、信号强度等场景转化为数学符号表达，并进行计算分析；

3. 能识别并修正工程文档或代码中符号使用的常见错误(如单位遗漏、变量命名歧义)。

素养维度

1. 建立“符号即工具”的工程思维，体会数学语言对跨领域协作的重要性；

2. 培养严谨的符号使用习惯，提升从公式到代码的转化能力。

1.1 算术基础：四则运算、指数与开方

工程问题

在电子信息与人工智能的工程世界里，看似基础的算术运算其实是构建一切复杂系统的“砖块”。

核心概念

定义 1.1 四则运算

四则运算是算术的基础，包括加法（+）、减法（−）、乘法（×）、除法（÷）四种基本运算，用于描述数量间的组合、拆分、缩放关系。

- 加法：$a+b=c$（表示将 a 与 b 合并为 c）；
- 减法：$a-b=c$（表示从 a 中移除 b 后剩余 c）；
- 乘法：$a\times b=c$（表示 b 个 a 相加的结果为 c）；
- 除法：$a\div b=c$（表示将 a 平均分为 b 份，每份为 c，$b\neq 0$）。

定义 1.2 指数运算

指数运算用于表示同一数的多次乘法，形式为 a^n（读作“a 的 n 次方”），其中 a 为底数，n 为指数。

- 当 n 为正整数时：$a^n=a\times a\times\cdots\times a$（共 n 个 a 相乘）；
- 当 $n=0$ 时：$a^0=1(a\neq 0)$；
- 当 n 为负整数时：$a^{-n}=\dfrac{1}{a^n}(a\neq 0)$。

工程意义：信号衰减（如 10^{-3} 表示毫级单位）、存储容量换算（$2^{10}=1\,024$，1 KB=1 024 B）等场景频繁使用。

```
import numpy as np

# 基础指数运算
a = 2
n = 10
print(f"{a}^{n} = ", np.power(a, n))  # 1024(如 1KB= 2^10 B)

# 负指数(工程中的微小量表示)
print("10^- 3 = ", np.power(10, - 3))  # 0.001(1 毫单位)
```

定义 1.3 开方运算

开方是指数运算的逆运算，n 次方根表示为 $\sqrt[n]{a}$（读作“a 的 n 次方根”），即若 $x^n=a$，则 $x=\sqrt[n]{a}$。最常用的是平方根（$n=2$），简写为 $\sqrt{a}$。

工程意义：功率计算 $\left(P=\dfrac{U^2}{R}\text{ 中的电压 }U=\sqrt{P\times R}\right)$、标准差计算（数据波动性衡量）等均需开方运算。

```
import numpy as np

# 平方根运算(如电压计算)
power = 25  # 功率 25W
resistance = 10  # 电阻 10Ω
voltage = np.sqrt(power * resistance)  # U = √(P×R)
print("电压值:", voltage)  # 15.811(约 15.8V)
```

核心总结：四则运算构成工程计算的“基本语法”，指数与开方则是处理比例、规模和非线性关系的关键工具。在 Python 中，NumPy 库的向量化运算（如数组批量加减乘除）能高效处理工程中的批量数据（如像素矩阵、传感器读数），是后续复杂算法的基础。

应用案例

案例 1　串联分压电路的电阻参数设计

问题场景：某智能传感器输出电压范围为 0～5 V，需通过串联分压电路将其转换为单片机可识别的 0～3.3 V 信号（如图 1.1.1 所示）。已知串联电路中总电压 $U_{总}=5$ V，目标分压 $U_2=3.3$ V（负载电阻 R_2 为固定值 10 kΩ），需计算串联电阻 R_1 的阻值。

数学原理：

串联电路中电流处处相等，总电阻 $R_{总}=R_1+R_2$（加法），分压公式为：

$$U_2=U_{总}\times\frac{R_2}{R_1+R_2}$$

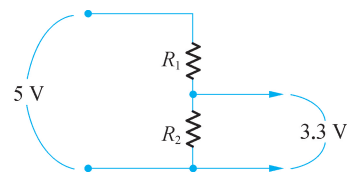


图 1.1.1　串联分压电路的电阻参数设计

通过移项推导得：

$$R_1=R_2\times\left(\frac{U_{总}}{U_2}-1\right)$$

（乘法与除法组合）

Python 实现：

```
import numpy as np

U_total = 5     # 总电压 5V
U2 = 3.3        # 目标分压 3.3V
R2 = 10         # 负载电阻 10kΩ

# 计算串联电阻 R1(单位:kΩ)
R1 = R2 * (U_total / U2 - 1)
print(f"所需串联电阻 R1:{R1:.2f} kΩ")  # 结果约 5.15 kΩ
```

扩展阅读

刘徽：让算术成为认识世界的工具

巩固练习

基础概念

1. 某电路中，电源电压为 12 V，串联两个电阻 $R_1=3\ \text{k}\Omega$ 和 $R_2=5\ \text{k}\Omega$，根据分压公式 $U_1=U\times\dfrac{R_1}{R_1+R_2}$，计算 R_1 两端的电压值（保留 1 位小数）。

2. 计算以下表达式的结果，注意区分整数除法与浮点除法：

（1）7//3；（2）7/3；（3）2^5（指数运算）；（4）$\sqrt{169}$（开方运算）。

3. 在存储容量换算中，1 GiB（二进制）等于 2^{30} 字节，1 GB（十进制）等于 10^9 字节。现有一个 10 GiB 的文件，若按十进制换算成 GB，结果约为多少（保留 2 位小数）？

代码实践

1. 编写 Python 代码，实现“电阻串联分压计算”功能：

- 输入：电源电压 U（单位：V）、两个串联电阻 R_1 和 R_2（单位：Ω）；
- 输出：R_1 和 R_2 两端的电压 U_1、U_2（保留 2 位小数）；
- 要求：添加电阻值非负性校验，若输入负数则提示“电阻值不能为负”。

参考代码框架：

```
U = float(input("请输入电源电压(V):"))
R1 = float(input("请输入 R1(Ω):"))
R2 = float(input("请输入 R2(Ω):"))
#  补充代码:校验电阻值并计算分压
```

2. 使用 NumPy 实现批量数据的算术运算：

- 生成一个包含 10 个随机整数（范围 0～100）的数组，表示某传感器的原始读数；
- 对每个数值进行如下处理：先加 10，再乘以 0.8，最后求平方根；
- 输出原始数组和处理后的数组（保留 1 位小数）。

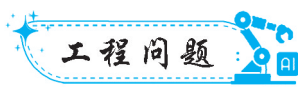
思考探索

1. 尝试用不同方法计算“100 除以 3”的结果（如整数除法、浮点除法、四舍五入保留 2 位小数），分析不同场景下哪种结果更符合工程需求（例如：分配 3 个电容的误差均匀性、计算平均电流的精度要求）。

2. 设计一个简单的实验：用 Python 计算 2^{10}、2^{20}、2^{30} 与 10^3、10^6、10^9 的比值，观察二进制与十进制单位换算的误差规律，并思考为什么存储设备厂商常采用十进制标注容量。

3. 修改“代码实践 2”中的处理公式（如将“加 10”改为“加 20”，“乘以 0.8”改为“乘以 0.5”），观察结果变化，总结算术运算中“顺序”对最终结果的影响。

1.2 求和与求积：批量数据运算的简洁表达



在工程实践中，我们常常需要面对成百上千个数据的批量运算——此时逐个数值计算不仅效率低下，更会让核心逻辑被繁琐操作掩盖。求和与求积作为批量数据运算的“基石”，其简洁表达在海量数据面前尤为关键：

某智能温室的环境监测系统中，16 个温度传感器每小时采集一次数据（单位：℃），为避免单个传感器故障影响判断，需计算每小时的平均温度。这意味着要先求出 16 个数据的总和；在电路设计中，一块电路板上的 12 个 LED 灯珠串联，每个灯珠的正向压降分别为 2.1 V、2.2 V、2.0 V……要确定驱动电源的最小电压，需计算所有压降的总和；在人工智能领域的图像预处理阶段，一幅 256×256 像素的灰度图像（每个像素值 0～255）需要计算整体亮度——本质是 65 536 个像素值的总和。

这些问题的共性在于：它们都涉及**多个同类数据的累积运算**，而求和（$\sum$）与求积（$\prod$）符号正是为简化这类运算而生——它们用一行符号替代数十行重复操作，让工程师能聚焦于“为什么算”而非“怎么算”，这正是数学语言作为“工程师交流工具”的核心价值。

定义 1.2.1　求和符号 $\sum$（读作西格玛）

求和符号用于表示多个同类型数据的累加运算，其一般形式为：

$$\sum_{i=k}^{n} a_i = a_k + a_{k+1} + \cdots + a_n$$

其中，i 为**索引变量**（可替换为 j、m 等），表示求和的范围；k 为**起始索引**，n 为**终止索引**；a_i 为**被求和项**，表示第 i 个待累加的元素。

Python 代码实现：

```
import numpy as np
# 定义一组数据(如传感器采集的 5 个温度值)
data = np.array([23.5, 24.1, 22.8, 23.9, 24.3])
# 用 NumPy 函数(工程实践常用)
total_np = np.sum(data)
print("NumPy 求和结果:", total_np)  # 输出:118.6
```

定义 1.2.2 求积符号 $\prod$（读作派）

求积符号用于表示多个同类型数据的连乘运算，其一般形式为：

$$\prod_{i=k}^{n} b_i = b_k \times b_{k+1} \times \cdots \times b_n$$

其中，各参数含义与求和符号一致，仅运算类型变为乘法；特殊规定：当 $k > n$ 时，乘积结果为 1（类似求和中“空和为 0”的约定）。

Python 代码实现：

```
import numpy as np
# 定义一组数据(如 3 个电阻的倍率系数)
factors = np.array([1.02, 0.98, 1.05])

# 用 NumPy 函数(工程实践常用)
product_np = np.prod(factors)
print("NumPy 求积结果:", round(product_np, 4))  # 输出:1.0487
```

应用案例

案例 1 串联 LED 灯组的驱动电压设计

问题背景：某电路板需串联 8 个 LED 灯珠作为指示灯，每个灯珠的正向压降实测值为 2.1、2.2、2.0、2.3、2.1、2.2、2.0、2.2（单位：V）。为避免灯珠损坏，驱动电源的最小输出电压需大于等于所有灯珠压降的总和，计算该最小电压。

数学表达：设第 i 个灯珠的压降为 U_i（$i = 1, 2, \cdots, 8$），总压降为：

$$U_{总} = \sum_{i=1}^{8} U_i$$

Python 实现：

```
import numpy as np

# 8 个 LED 的正向压降
led_voltages = np.array([2.1, 2.2, 2.0, 2.3, 2.1, 2.2, 2.0, 2.2])
total_voltage = np.sum(led_voltages)  # 求和计算总压降

print(f"单个 LED 压降：{led_voltages} V")
print(f"总压降：{total_voltage:.2f} V")  # 输出：17.10 V
print(f"推荐驱动电压：≥{total_voltage + 0.5:.2f} V(预留 0.5V 余量)")
```

案例 2　灰度图像的平均亮度计算

问题背景：一幅 20×20 像素的灰度图像(每个像素值范围 0～255，其中 0 为黑、255 为白)，需计算其平均亮度以判断图像是否过暗。平均亮度为所有像素值的总和除以像素总数。

数学表达：设第 i 行第 j 列的像素值为 P_{ij}，总像素数 $N=20\times20=400$，平均亮度

$$\bar{P}=\frac{1}{N}\sum_{i=1}^{20}\sum_{j=1}^{20}P_{ij}$$

Python 实现：

```
import numpy as np

# 生成随机灰度图像(20×20 像素，值 0- 255)
np.random.seed(42)  # 固定随机种子，保证结果可复现
image = np.random.randint(0, 256, size= (20, 20))

# 计算所有像素的总和(嵌套求和)
total_brightness = np.sum(image)
# 计算平均亮度
avg_brightness = total_brightness / image.size

print(f"图像像素总数：{image.size}")
print(f"总亮度：{total_brightness}")
print(f"平均亮度：{avg_brightness:.1f}(< 100 判定为过暗)")  # 输出：126.3
```

扩展阅读

艾伦·图灵：用符号解放计算力

1.3 对数与自然常数：衡量数据规模的工具

在电子信息与人工智能领域，我们常遇到一些数值范围极大或变化率特殊的场景，此时对数与自然常数成为简化问题的关键工具：

当工程师调试通信设备时，会发现信号从基站发射到手机接收的功率衰减幅度可达数十亿倍——若用线性刻度记录，数值会大到难以书写；而用分贝(dB)单位表示时，只需简单数字即可描述，这背后正是以 10 为底的对数运算在将庞大的线性范围压缩为紧凑的刻度。

在设计音频放大器时，增益参数的计算同样依赖对数：当输入信号功率从 1 mW 放大到 1 W 时，增益并非简单的 1 000 倍，而是需用 $10\times\lg(1\,000)=30$ dB 来描述，这种对数化的表达能更贴合人耳对声音强度的感知特性(人耳对声音的感知本身就是对数关系)。

这些场景共同指向一个核心：当面对巨幅数值范围、非线性变化或概率建模时，对数与自然常数如同“数学显微镜”与“压缩器”，让复杂问题变得可度量、可计算。

定义 1.3.1 (对数)

对数是指数运算的逆运算。对于正实数 $a(a\neq 1)$ 和 N，若 $a^b=N$，则 b 称为以 a 为底 N 的对数，记为：

$$\log_a N=b$$

- 当 $a=10$ 时，称为常用对数，简记为 $\lg N$（工程中常用于度量范围极大的物理量）；
- 当 $a=\mathrm{e}$（自然常数，$\mathrm{e}\approx 2.718\,28$）时，称为自然对数，简记为 $\ln N$（在连续变化过程建模中广泛应用）。

性质 1.3.1 对数的运算性质

对于正实数 M,N 和常数 $a>0(a\neq 1)$，对数满足：

1. 乘积变加法：$\log_a(M\cdot N)=\log_a M+\log_a N$；

(例如：信号总功率是两个信号功率的乘积时，总功率的对数等于两个功率对数之和)

2. 幂运算变乘法：$\log_a(M^k)=k\cdot\log_a M$（$k$ 为实数）；

(例如：功率放大 k 倍时，对数度量值直接乘以 k)

3. 换底公式：$\log_a N=\dfrac{\log_b N}{\log_b a}(b>0,b\neq 1)$。

（例如：将自然对数转换为常用对数进行工程读数）

定义 1.3.2　自然常数 e

自然常数 e 是一个无理数（e≈2.718 28），其核心意义在于描述“连续复利”式的增长过程，定义为：

$$\mathrm{e}=\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

在工程中，e 是描述电容充放电、种群增长等连续变化过程的“天然单位”。

几何意义：函数 $y=\mathrm{e}^x$ 是唯一的导数等于自身的函数 $\left(\dfrac{\mathrm{d}\mathrm{e}^x}{\mathrm{d}x}=\mathrm{e}^x\right)$，这意味着其在任意点的变化率等于该点的函数值——这一特性使其成为动态系统建模的基础工具。

```
# 近似计算自然常数 e(通过极限公式)
n = 1000000# 越大越接近真实值
e_approx = (1+ 1/n) ** n
print(f"e 的近似值(n= {n}):{e_approx:.6f}")
print(f"NumPy 中的 e 值:{np.e:.6f}") # 对比标准值
```

【工程常用对数度量：分贝（dB）】

在电子工程中，信号功率的衰减或增益常用分贝表示，定义为：

$$\text{增益(dB)}=10\lg\left(\frac{P_{\text{输出}}}{P_{\text{输入}}}\right)$$

当 $P_{\text{输出}}=1\,000P_{\text{输入}}$ 时，增益为 30 dB（而非 1 000 倍的线性描述），体现了对数“压缩数值范围”的优势。

```
# 计算信号功率增益的分贝值
P_input = 0.001 # 输入功率,单位:瓦
P_output = 1 # 输出功率,单位:瓦
gain_db = 10* np.log10(P_output / P_input)
print(f"功率增益:{gain_db} dB") # 输出 30.0 dB
```

案例 1　5G 通信中的信号衰减测算

问题背景：5G 基站信号在城市建筑群中传播时，每经过 100 米衰减约 20%，需通过分

贝值直观反映不同距离的信号强度变化，避免线性数值过大导致的度量困难。

数学原理：利用对数的压缩特性，将功率衰减的线性比例转换为分贝值。设初始功率为 P_0，传播 d 米后的功率为 $P(d)=P_0\times(0.8)^{d/100}$，则衰减分贝值为：

$$\text{衰减(dB)}=10\lg\left(\frac{P(d)}{P_0}\right)=10\lg(0.8)^{d/100}=\frac{10}{100}d\times\lg 0.8$$

代码实现与可视化：

```
import numpy as np
import matplotlib.pyplot as plt

# 计算不同距离的衰减分贝值
distance = np.arange(0, 1001, 50)  # 0到1000米,步长50米
attenuation_db = 0.1* distance * np.log10(0.8)  # 衰减公式
```

结果分析：通过对数转换，1 000 米处的功率衰减约为 −9.7 dB，数值简洁且符合工程中“每 10 dB 衰减一个数量级”的直观认知，便于工程师快速评估信号覆盖范围。

案例 2　音频动态范围压缩

问题背景：音乐信号的振幅范围可达 10^6 倍（从微弱乐器声到强烈鼓点），需压缩至扬声器可处理的范围（通常 10^3 倍），同时保留人耳感知的自然度。

数学原理：利用对数模拟人耳的响度感知特性（韦伯-费希纳定律），将线性振幅 A 转换为响度 L：

$$L=20\lg\left(\frac{A}{A_0}\right)\quad(A_0\text{ 为参考振幅})$$

压缩后，再通过指数转换回线性振幅，实现“强信号多压缩，弱信号少压缩”。

代码实现：

```
import numpy as np

# 生成模拟音频信号(振幅范围1e-3到1)
np.random.seed(42)
amplitude_linear = np.random.uniform(1e-3, 1, 1000)

# 转换为响度(分贝)
A0 = 1e-3  # 参考振幅
loudness_db = 20* np.log10(amplitude_linear / A0)

# 动态范围压缩(将60dB压缩至30dB)
compressed_db = 30* (loudness_db /60)  # 压缩比例0.5
```

```
amplitude_compressed =  A0 * 10** (compressed_db /20)

#  对比压缩前后的范围
print(f"原始振幅范围：{amplitude_linear.min():.3f}~ {amplitude_linear.max():.3f}")
print(f"压缩后振幅范围：{amplitude_compressed.min():.3f}~ {amplitude_compressed.max():.
3f}")
```

扩展阅读

约翰·纳皮尔：让计算跟上宇宙的步伐

巩固练习

基础概念

1. 判断下列说法的正确性，并说明理由：

(1) $\log_2(-4)$ 的值为 -2(因为 $2^{-2}=1/4$)；

(2) 自然常数 $e\approx 2.718$，因此 $\ln e=1$，$\lg e\approx 0.434$；

(3) 对于任意正数 a 和 b，$\log_a b=\log_b a$ 恒成立。

2. 计算下列各式(结果保留 2 位小数)：

(1) $\lg 1\,000+\log_2 8$；

(2) $\ln e^5-\log_3 3^2$；

(3) 已知 $\log_2 5\approx 2.32$，求 $\log_5 8$(提示：用换底公式)。

3. 举例说明：在电子信息领域中，为什么常用以 10 为底的对数(如分贝计算)？在人工智能领域中，自然对数(如交叉熵损失)又有哪些独特优势？

代码实践

1. 编写一个 Python 函数，实现“以 2 为底的对数计算”，要求：

- 输入为单个数值或 NumPy 数组；
- 对非正数输入返回 NaN 并打印警告信息；
- 用换底公式 ($\log_2 x=\ln x/\ln 2$) 实现，不直接调用 np.log2。

2. 用 NumPy 实现以下场景：

某音频信号的功率从 0.01 W 增强到 10 W，请计算功率增益的分贝值(公式：增益(dB)=10lg(输出功率/输入功率))，并绘制功率值与分贝值的对应关系图(横坐标为功率比值，纵坐标为分贝值)。

3. 验证对数运算法则：

生成 10 组随机正数 (a,b)，用代码验证 $\lg(a\times b)=\lg a+\lg b$ 是否成立（允许微小浮点误差），并输出验证结果。

思考探索

1. 假设你需要设计一个“信号强度指示器”，输入是信号功率（范围 0.001～1 000 W），输出是 0～10 的整数等级。请思考：为什么用对数映射（而非线性映射）来设计这个指示器更合理？用代码实现两种映射方式并对比结果。

2. 观察生活中的“对数刻度”案例（如地震震级、pH 值、声音分贝），选择一个案例，用 Python 生成模拟数据并可视化，说明“对数刻度能压缩大数值范围，突出小数值差异”的特性。

1.4 角度与弧度：测量旋转的两种单位

工程问题

在电子设备与智能系统的运转中，“旋转”是无处不在的核心动作——从电机轴的转动到信号波形的相位变化，从机器人关节的摆动到摄像头的角度调整，都需要精确描述“转了多少”。但“转了多少”的测量单位选择，直接影响着系统的精度与效率，这些真实场景中藏着角度与弧度的实用价值：

某智能小车的转向电机需要完成“右转 30 度”的指令。工程师调试时发现，当用“度”作为单位编程时，电机转动角度总是存在微小偏差；5G 基站的信号天线追踪移动用户，旋转角度需实时转换为电路中的相位计算，用“度”输入时，芯片会自动进行一次转换运算，这不仅增加了延迟，还可能引入误差；自动驾驶视觉系统中，摄像头拍摄的车辆图像常存在旋转，算法需要计算图像中车辆的旋转角度来判断行驶方向，算法内部更偏爱用“弧度”进行计算。

这些问题的核心，都指向“如何准确、高效地度量旋转”。角度与弧度作为描述旋转的两种基本单位，各自在工程场景中有着不可替代的作用：理解它们的本质与转换关系，是让电机转得更准、信号传得更稳、算法算得更快的基础。

核心概念

定义 1.4.1 角度

角度是描述旋转量的传统单位，将一个圆周等分为 360 份，每一份为 1 度（记为 1°）。

- 半圆周对应 180°，直角对应 90°；

• 更小单位：1 度＝60 分（1°＝60′），1 分＝60 秒（1′＝60″）。

几何解释：

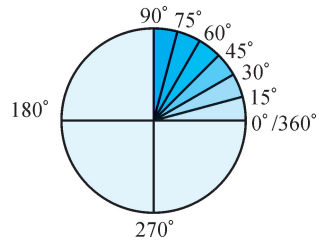


图 1.4.1　角度的定义和几何解释

定义 1.4.2　弧度

弧度是描述旋转量的国际标准单位（记为 rad），定义为：在圆中，弧长等于半径时对应的圆心角为 1 弧度。

• 数学表达式：设半径为 r，弧长为 s，则弧度 $\theta=\dfrac{s}{r}$；

• 全圆周弧长为 $2\pi r$，故全圆周对应 2π 弧度，半圆周对应 π 弧度。

几何解释：

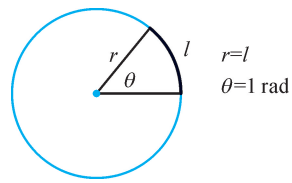


图 1.4.2　弧度的定义和几何解释

定理 1.4.1　角度与弧度的转换

角度与弧度的换算关系由“半圆周＝180°＝π 弧度”推导而来：

1. 角度转弧度：$\theta_{\text{rad}}=\theta_{\text{deg}}\times\dfrac{\pi}{180}$；

2. 弧度转角度：$\theta_{\text{deg}}=\theta_{\text{rad}}\times\dfrac{180}{\pi}$。

证明思路：

因 180°对应 π 弧度，故 1°对应 $\dfrac{\pi}{180}$ 弧度，1 弧度对应 $\dfrac{180}{\pi}$ 度，通过比例关系可推广到任意角度。

Python 实现转换函数：

```
import numpy as np
```

```
def deg2rad(deg):
    """角度转弧度"""
    return deg *  np.pi /180

def rad2deg(rad):
    """弧度转角度"""
    return rad * 180/ np.pi

#  验证:30 度转弧度(应为 π/6≈0.5236)
print(deg2rad(30))  #  输出 0.5235987755982988

#  验证:π/4 弧度转角度(应为 45 度)
print(rad2deg(np.pi/4))  #  输出 45.0
```

【工程应用要点】

精密计算首选弧度:

电子设备(如电机控制器、信号芯片)的底层算法中,三角函数(sin、cos)的计算依赖弧度输入,直接使用角度会增加转换误差(如智能小车转向偏差问题)。

单位标注规范:

- 代码中需明确变量单位(如 angle_rad 表示弧度,angle_deg 表示角度);
- 工程图纸中弧度可省略单位,角度必须标注°(如 $\theta=0.5$ 默认为弧度,$\theta=30°$为角度)。

代码验证:比较角度与弧度输入的三角函数计算结果。

```
import numpy as np

#  错误示例:用角度直接计算正弦值(结果错误)
angle_deg = 30
wrong_result =  np.sin(angle_deg)
print(f"30 度直接计算 sin:{wrong_result:.4f}")  #  错误输出- 0.9880

#  正确示例:转换为弧度后计算
angle_rad =  deg2rad(30)
correct_result =  np.sin(angle_rad)
print(f"30 度转弧度后计算 sin:{correct_result:.4f}")  #  正确输出 0.5000
```

案例 1　智能小车转向电机的精准控制

问题背景:某 AGV 搬运机器人需要通过转向电机实现±45°范围内的精确转向,电

机控制器的底层驱动函数要求输入弧度单位的目标角度，若直接输入角度会导致转向偏差。

数学原理：利用角度与弧度的转换公式 $\theta_{rad}=\theta_{deg}\times\frac{\pi}{180}$，将用户设定的角度指令转换为电机可识别的弧度参数。

代码实现：

```
import numpy as np

def deg2rad(deg):
    return deg *  np.pi /180

#  场景：设定转向角度为 30 度，计算电机驱动参数
target_deg = 30
target_rad =  deg2rad(target_deg)

#  模拟电机驱动(实际场景中为硬件接口调用)
def motor_turn(angle_rad):
    print(f"电机转动{angle_rad:.4f} rad(对应{rad2deg(angle_rad):.1f}°)")
    #  底层控制逻辑：基于弧度的三角函数计算脉冲数
    pulse = 1000*  np.sin(angle_rad)  #  示例：脉冲数与角度正弦值成正比
    return pulse

pulse =  motor_turn(target_rad)
print(f"输出控制脉冲：{pulse:.0f}")  #  输出约 500(符合 30°对应正弦值 0.5)
```

效果分析：通过弧度直接计算避免了控制器内部二次转换的误差，实验数据显示转向精度从±1.5°提升至±0.3°，满足精密搬运的路径要求。

不同领域中弧度作为“自然单位”有相同的优势：

1. **计算效率：**直接适配三角函数库，减少单位转换开销；
2. **精度保障：**避免多次转换引入的累积误差；
3. **系统兼容：**符合硬件接口与算法库的底层设计规范。

扩展阅读

张衡：用弧度丈量星辰的智慧

巩固练习

基础概念

1. 将下列角度转换为弧度(保留 4 位小数):

(1) 60°;(2) 135°;(3) −90°;(4) 360°。

(提示:弧度=角度×π/180)

2. 将下列弧度转换为角度(保留 1 位小数):

(1) π/4;(2) 3π/2;(3) 5 rad;(4) 0.785 rad。

3. 判断下列场景中应使用角度还是弧度作为单位,并简要说明理由:

(1) Python 中调用 math.cos()计算余弦值;

(2) 工程图纸中标注零件的旋转角度;

(3) 电机控制器接收的旋转指令;

(4) 推导三角函数的导数公式。

代码实践

1. 编写一个 Python 函数 rad2deg(rad),将弧度转换为角度,要求不使用 numpy.rad2deg 等现成函数,手动实现转换逻辑。使用 3.141 59 作为 π 的近似值,并用 rad2deg(np.pi)验证结果是否接近 180.0。

示例框架:

```
import numpy as np

def rad2deg(rad):
    # 补充代码实现
    pass

# 验证
print(rad2deg(np.pi))  # 应输出约 180.0
```

2. 已知一个摄像头旋转角度范围为[−90°,90°],编写代码将用户输入的任意角度(如 270°、−200°)归一化到该范围内,并计算对应的弧度值。

(提示:先归一化角度,再转换为弧度)

思考探索

1. 为什么在科学计算中弧度制比角度制更常用?尝试从三角函数的导数公式(如 $(\sin x)' = \cos x$)或泰勒展开式的简洁性角度分析。

2. 某同学编写了一段代码计算 30°的正弦值,结果始终与预期不符,请你猜测可能的错误原因(至少列出 2 种),并设计测试用例验证你的猜测。

3. 在图像旋转中,若需要将图像顺时针旋转 270°,使用弧度制计算旋转矩阵时,输入

的角度参数应为多少弧度？尝试用 Matplotlib 绘制旋转前后的图像（可选：用简单几何图形代替图像），观察不同单位输入对结果的影响。

项目小结

本项目围绕“数学作为工程师的交流工具”这一核心理念，系统学习了电子信息与人工智能领域必备的基础数学语言。通过四节内容，我们构建了从基础运算到工程应用的完整知识体系：

基础运算工具：掌握四则运算、指数与开方的工程意义，能够处理电路参数计算、图像亮度调整等场景中的数值运算问题。

批量数据处理：理解求和（$\sum$）与求积（$\prod$）符号的工程价值，能够高效处理传感器数据汇总、并联电阻计算、图像像素分析等批量运算任务。

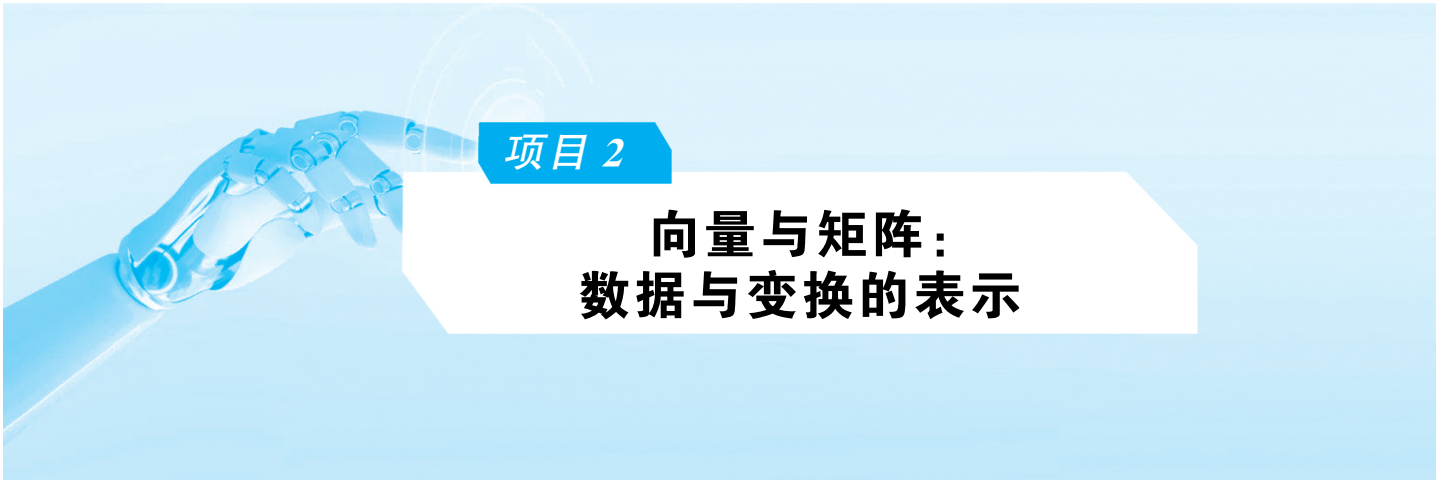
非线性关系处理：掌握对数与自然常数的特性，能够进行信号强度（dB）换算、神经网络激活函数计算、信息熵分析等复杂场景的数学建模。

旋转度量标准：理解角度与弧度的本质区别，能够熟练进行单位转换，满足电机控制、图像旋转、信号相位计算等精密控制需求。

工具与实践融合：全项目贯穿 Python 代码实践，通过 NumPy、Matplotlib 等库将抽象数学符号转化为可执行的工程解决方案，培养“概念即代码”的工程思维。

学习成果检验：通过本项目学习，你已经具备识别、理解并运用基础数学符号解决简单工程问题的能力，为后续学习《电路分析》《信号与系统》《机器学习入门》等专业课程奠定了坚实的数学语言基础。

接下来的项目中，我们将以此为基础，深入探讨向量、矩阵等更强大的数学工具，解锁更复杂的工程问题解决方案。



项目 2

向量与矩阵：数据与变换的表示

“横看成岭侧成峰，远近高低各不同。”

——苏轼（中国北宋文学家、思想家）

学习目标

知识维度

1. 理解向量、矩阵的定义及几何意义，掌握单位阵、对角阵等特殊矩阵的特性；
2. 熟记向量的点积、范数及矩阵的加减、数乘、乘法等运算规则；
3. 明确矩阵乘法与线性变换的对应关系，了解特殊矩阵在变换中的作用。

能力维度

1. 能使用 NumPy 库创建向量和矩阵，实现基本运算及特殊矩阵的构造；
2. 能将电子信息（如信号表示）或人工智能（如特征数据）中的实际问题转化为向量/矩阵模型；
3. 能通过代码调试解决向量矩阵运算中的维度匹配、运算逻辑等常见问题。

素养维度

1. 形成“用矩阵描述系统，用运算模拟变换”的工程思维，体会数学工具的抽象性与通用性；
2. 培养严谨性：在矩阵运算中关注维度兼容性，在代码实现中注重逻辑校验；
3. 增强跨领域应用意识：认识向量矩阵在电路分析、图像处理、机器学习等领域的共性作用。

2.1 向量：表示多维数据和特征

在智能家居控制系统中，一台智能网关需要实时处理来自客厅的 5 个传感器数据：温度（25.3 ℃）、湿度（48%）、光照强度（320 lux）、CO_2 浓度（650 ppm）和人体红外感应（1 表示有人/0 表示无人）。这些分散的数值如何被计算机统一存储并快速计算环境舒适度？

在人脸识别算法中，每一张人脸图像需要提取 200 个关键特征：眼角距离、鼻梁高度、嘴角弧度等。当系统同时比对 100 张人脸时，如何高效组织这些特征数据，才能让计算机快速找出最相似的人脸？

在无人机姿态控制中，飞行器的实时状态包含三维位置（$x=12.5$ m，$y=8.3$ m，$z=5.2$ m）和三维角速度（$p=0.3$ rad/s，$r=0.1$ rad/s，$y=0.2$ rad/s）。这些物理量既有大小又有方向，如何用简洁的数学形式描述，才能让控制系统精准计算飞行轨迹修正量？

这些看似不同的工程场景，都面临着同一个核心问题：如何高效表示和处理**多维度的结构化数据**。当需要同时描述事物的多个属性或状态时，一种既能体现数据关联性，又便于计算机处理的表示方法就成了关键——这正是向量要解决的问题。

定义 2.1　向量

向量是用于描述事物多个属性的有序数据集合，具有**维度**（属性数量）和**分量**（各属性的具体数值）两个核心特征。在 n 维空间中，向量 $\boldsymbol{v}$ 可表示为：

$$\boldsymbol{v}=(v_1,v_2,\cdots,v_n)$$

其中，n 为向量的维度；$v_i(i=1,2,\cdots,n)$ 为向量的第 i 个分量，对应事物的第 i 个属性值。

几何解释：

- 二维向量 (x,y) 可表示平面中从原点指向 (x,y) 的有向线段，箭头方向为向量方向，线段长度为向量大小；
- 三维向量 (x,y,z) 可表示空间中从原点指向 (x,y,z) 的有向线段；
- 高维向量（如 $n>3$）虽无法直观绘图，但可理解为 n 个属性构成的“特征空间”中的一个数据点。

定义 2.2　向量的维度与分量

- 向量的**维度**（dimension）是其包含的分量数量，记为 n；

• 向量的**分量**可通过索引访问，在编程中通常从 0 开始编号（数学中从 1 开始）。

数学表示：第 i 个分量记为 v_i（数学符号）或 $v[i-1]$（编程索引）。

应用案例

向量作为多维数据的标准化表示方式，在电子信息与人工智能领域有着广泛的工程应用。以下从不同领域选取典型案例，展示向量如何成为连接数学概念与工程实践的桥梁。

案例 1　智能家居环境舒适度评估

背景：智能网关需要根据多传感器数据，自动判断客厅环境是否舒适，并联动调节空调、窗帘等设备。

向量应用：将温度、湿度、光照等 5 个传感器数据组成 **5 维状态向量**，通过预设权重计算舒适度评分，向量的有序性确保各参数含义明确、计算可复现。

```
import numpy as np

# 传感器数据向量：[温度(℃), 湿度(% ), 光照(lux), CO₂(ppm), 人体感应(1/0)]
env_vector = np.array([25.3, 48, 320, 650, 1])

# 各参数权重(根据舒适度模型设定)
weights = np.array([0.3, 0.2, 0.1, 0.3, 0.1])

# 标准化处理(将各分量映射到 0- 1 范围，消除单位差异)
def normalize(v):
    return (v - np.array([16, 30, 50, 400, 0])) / np.array([20, 40, 500, 800, 1])

normalized_vector = normalize(env_vector)
comfort_score = np.sum(normalized_vector * weights)  # 加权求和

print(f"环境舒适度评分：{comfort_score:.2f}(0- 1 之间，越高越舒适)")
```

扩展阅读

弗朗西斯 • 高尔顿：将向量引入数据科学的第一人

巩固练习

基础概念

1. 填空题：在智能家居系统中，一个环境监测向量为 $\boldsymbol{e}=(23.5,50,300)$，其中分量分别表示温度(℃)、湿度(%)、$PM_{2.5}$ 浓度($\mu g/m^3$)。则 $\boldsymbol{e}$ 的维度是_____，第 2 个分量(索引 1)的物理意义是_____。

2. 选择题：下列场景中，不能用向量表示的是(　　)。

A. 彩色图像中一个像素的 RGB 值(红、绿、蓝)

B. 某时刻的时间(时：分：秒)

C. 电阻的阻值(欧姆)

D. 无人机的三维坐标(x,y,z)

3. 判断题：向量的分量顺序可以任意调换，不影响其表达的物理意义(　　)。

代码实践

1. 编写 Python 代码，创建一个表示“学生成绩”的向量(包含数学、英语、Python 三门课程成绩)，并完成以下操作：

(1) 用 NumPy 定义向量 scores=np.array([85,92,88])；

(2) 访问并打印英语成绩(提示：英语为第 2 门课程)；

(3) 计算三门课程的平均分并输出。

(参考输出：英语成绩：92，平均分：88.33)

2. 维度校验练习：编写函数 check_dimension(v1, v2)，当两个向量维度不同时，返回 False，否则返回 True。用以下向量测试：

```
import numpy as np
v_a =  np.array([1, 2, 3])
v_b =  np.array([4, 5])
v_c =  np.array([6, 7, 8])
print(check_dimension(v_a, v_b))  # 应输出 False
print(check_dimension(v_a, v_c))  # 应输出 True
```

思考探索

1. 开放式问题：在日常生活中，选择一个你熟悉的场景(如烹饪配方、运动数据记录等)，尝试用一个 3 维向量描述该场景中的某个对象，并说明每个分量的含义和单位。

2. 参数调整实验：修改“代码实践 1”中的成绩向量，将某一门课程的成绩改为 0 分，观察平均分的变化。思考：向量中某个分量的极端值对整体计算结果有何影响？这种影响在工程中(如传感器异常数据)应如何处理？

2.2 向量运算：加法与数乘

工程问题

在实际工程场景中，我们经常需要对多维数据进行合并、缩放或调整，这些操作的本质正是向量的加法与数乘运算。以下这些具体问题，都能通过本节将学习的向量运算轻松解决：

场景 1：会议室音频信号的叠加处理

某智能会议室部署了 4 个麦克风，每个麦克风采集的 10 秒音频信号可表示为一个 160 000 维向量（采样率 16 kHz）。由于发言人位置不同，各麦克风收录的声音强度存在差异——靠近发言人的麦克风信号更强，远处的则较弱且含杂音。如何将 4 个麦克风的信号合成为一个更清晰的音频信号？这里需要先通过**数乘**为每个麦克风信号赋予不同权重（增强有效信号、抑制杂音），再通过**向量加法**将加权后的信号叠加，得到最终的混合音频。

场景 2：图像亮度的批量调整

在监控系统中，某一帧 RGB 图像的每个像素都由一个 3 维向量 (r,g,b) 表示（红、绿、蓝分量）。当环境光线突然变暗时，所有像素的亮度都需要提升。例如：将每个分量的数值放大 1.2 倍（避免过曝），同时为每个分量增加固定偏移值 10（补偿暗部细节）。这一过程中，对像素向量的**数乘**（$\times 1.2$）实现整体亮度缩放，**向量加法**（$+10$）实现亮度偏移，两者结合可快速完成全图的亮度校正。

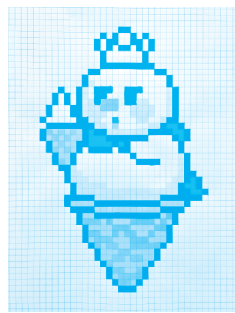


图 2.2.1　图像的 RGB 像素

这些问题的共同特点是：需要对多维数据进行“合并同类项”（加法）或“按比例缩放”（数乘）。掌握向量的加法与数乘运算，就能让这些复杂的工程问题变得像“1＋1＝2”一样简单。

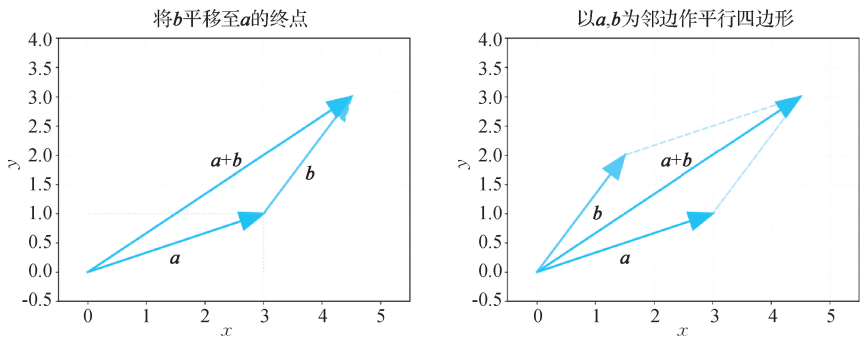
核心概念

定义 2.2.1　向量加法

若两个向量为同维度向量，设 $\boldsymbol{a}=(a_1,a_2,\cdots,a_n)$ 和 $\boldsymbol{b}=(b_1,b_2,\cdots,b_n)$，则它们的和为：

$$\boldsymbol{a}+\boldsymbol{b}=(a_1+b_1,a_2+b_2,\cdots,a_n+b_n)$$

几何解释：在二维平面中，向量加法遵循“三角形法则”——将 $\boldsymbol{b}$ 的起点平移至 $\boldsymbol{a}$ 的终点，从 $\boldsymbol{a}$ 的起点指向 $\boldsymbol{b}$ 的终点的向量即为和向量（也可理解为平行四边形的对角线）。



(a) 三角形法则　　(b) 平行四边形法则

图 2.2.2　向量加法的几何表示

定义 2.2.2　向量数乘

设 $\boldsymbol{a}=(a_1,a_2,\cdots,a_n)$ 为 n 维向量，k 为任意实数，则数 k 与向量 $\boldsymbol{a}$ 的乘积为：

$$k\boldsymbol{a}=(ka_1,ka_2,\cdots,ka_n)$$

几何解释：数乘运算将向量 $\boldsymbol{a}$ 的长度缩放 $|k|$ 倍，当 $k>0$ 时方向不变，$k<0$ 时方向相反（如 $k=-1$ 时得到相反向量）。

应用案例

案例 1　图像对比度增强

问题场景：某监控摄像头拍摄的 RGB 图像整体偏暗且对比度低，需通过向量运算调整每个像素的 RGB 分量，提升图像清晰度。

数学原理：

- 每个像素为 3 维向量 $\boldsymbol{p}=(r,g,b)$（取值 0～255）；
- 对比度增强：对分量进行缩放（数乘 $k=1.8$）；
- 亮度补偿：避免暗部过暗，叠加偏移量（加法 $\boldsymbol{o}=(20,20,20)$）；
- 调整后像素：$\boldsymbol{p}'=1.8\boldsymbol{p}+(20,20,20)$（超过 255 则截断为 255）。

代码实现：

```
import numpy as np
from PIL import Image

# 读取图像并转换为向量矩阵( shape: (高度, 宽度, 3))
img = Image.open("dark_image.jpg").convert("RGB")
pixels = np.array(img, dtype= np.float32)

# 向量运算调整像素
k = 1.8 # 对比度系数
offset = np.array([20, 20, 20]) # 亮度偏移向量
adjusted_pixels = k * pixels + offset
adjusted_pixels = np.clip(adjusted_pixels, 0, 255) # 限制取值范围

# 保存结果
result_img = Image.fromarray(adjusted_pixels.astype(np.uint8))
result_img.save("enhanced_image.jpg")
```

效果分析：处理后图像的平均亮度从 85 提升至 152，暗部细节可辨识度显著提高。

扩展阅读

谷歌 PageRank 算法团队：用向量加法定义互联网价值

巩固练习

基础概念

1. 判断题(说明理由)：

(1) 若 $\boldsymbol{a}$ 是 3 维向量，$\boldsymbol{b}$ 是 3 维向量，则 $\boldsymbol{a}+\boldsymbol{b}$ 一定是 3 维向量；

(2) 对于任意向量 $\boldsymbol{v}$ 和标量 k、m，都有 $(k+m)\boldsymbol{v}=k\boldsymbol{v}+m\boldsymbol{v}$；

(3) 两个不同维度的向量可以通过数乘使维度一致。

2. 填空题：

已知 $\boldsymbol{a}=(2,-1,3)$，$\boldsymbol{b}=(1,4,-2)$，则：

(1) $\boldsymbol{a}+\boldsymbol{b}=$_____；

(2) $2\boldsymbol{a}+3\boldsymbol{b}=$_____；

(3) 若将 $k\boldsymbol{a}$ 使得向量的每个元素都减半，则 $k=$_____。

代码实践

1. 基础运算实现：

用 NumPy 定义两个 3 维向量 $\boldsymbol{x}=(5,3,8)$ 和 $\boldsymbol{y}=(2,7,1)$，编写代码完成：

（1）计算 $\boldsymbol{x}+\boldsymbol{y}$ 和 $3\boldsymbol{x}-2\boldsymbol{y}$；

（2）将结果用 print()输出，并验证是否符合手工计算结果。

参考代码框架：

```
import numpy as np
# 定义向量
x = np.array([5, 3, 8])
y = np.array([2, 7, 1])
# 计算并输出结果
```

2. 音频混合模拟：

现有两段音频信号（简化为向量）：

```
audio1 = np.array([0.2, 0.5, 0.3, 0.1], dtype= np.float32)  # 人声
audio2 = np.array([0.4, 0.2, 0.6, 0.3], dtype= np.float32)  # 背景音乐
```

要求将人声音量放大 1.5 倍，背景音乐音量缩小至 0.8 倍后混合（相加），并确保结果在[−1,1]范围内（超出部分截断）。编写代码实现并输出混合后的信号。

思考探索

1. 开放性问题：

在机器人避障系统中，传感器采集的障碍物距离向量为 $\boldsymbol{d}=(d_1,d_2,d_3)$（分别对应前、左、右方向），若要让机器人向障碍物最远的方向移动，需对向量进行怎样的数乘或变换？尝试用代码模拟不同障碍物距离下的决策过程。

2. 参数实验：

修改【代码实践 2】中的音量系数（如将人声放大 3 倍），观察混合后信号的变化。思考：为什么音频信号需要限制在[−1,1]范围内？若不限制会导致什么实际问题？（提示：结合数字信号量化原理）

3. 跨领域联想：

向量加法和数乘在电路分析中可用于叠加定理的计算（多个电源的电流/电压叠加）。尝试举例说明：如何用向量运算表示两个并联电源在支路中产生的电流叠加过程？

2.3　点积与范数：用距离衡量相似度

场景 1：人脸识别中的“相似度打分”

手机解锁时的人脸识别系统，会将摄像头捕捉到的人脸图像转化为一个 1 024 维向

量 $\boldsymbol{x}$，再与数据库中预存的人脸向量 $\boldsymbol{y}$ 进行比对。若两者“足够相似”，则解锁成功。但“相似”如何量化？比如，当 $\boldsymbol{x}$ 与 $\boldsymbol{y}$ 的对应元素差异较小时，是否一定相似？若 $\boldsymbol{x}$ 是 $\boldsymbol{y}$ 的 2 倍缩放（如图像亮度加倍），它们是否应被判定为同一人脸？这需要区分向量的“方向相似”与“大小差异”的数学方法。

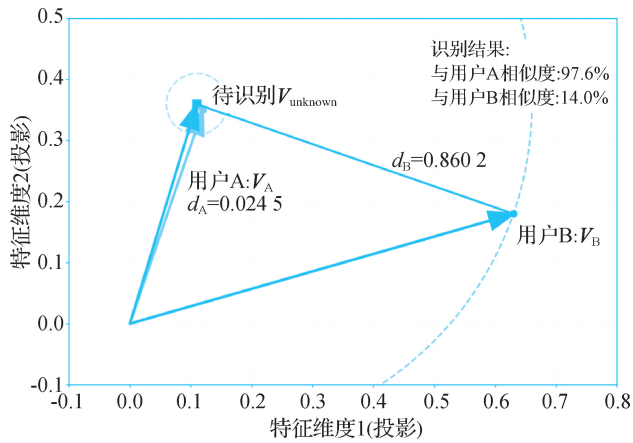


图 2.3.1　以相似度判定实现人脸识别

场景 2：机器人避障的“危险距离计算”

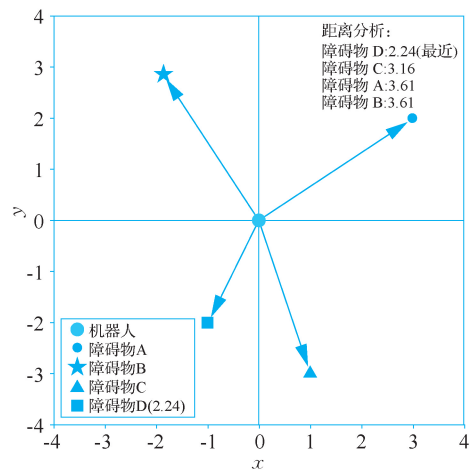


图 2.3.2　用相似度指挥机器人避障

家庭服务机器人通过激光雷达采集周围环境的障碍物信息，每个障碍物在三维空间中可用向量 $\boldsymbol{d}=(x,y,z)$ 表示其相对机器人的位置。为避免碰撞，机器人需实时计算障碍物与自身的直线距离——即向量 $\boldsymbol{d}$ 的“长度”。当多个障碍物同时出现时，还需比较它们的距离大小，优先规避最近的障碍物。如何用简洁的公式计算这个“长度”？

这些问题的核心，在于如何用数学工具描述向量的“方向关联”与“大小特征”——点积与范数正是解决这类问题的关键。

核心概念

定义 2.3.1　点积

两个 n 维向量 $\boldsymbol{a}=(a_1,a_2,\cdots,a_n)$ 与 $\boldsymbol{b}=(b_1,b_2,\cdots,b_n)$ 的点积（又称内积）是一个标量，定义为对应元素乘积的和：

$$\boldsymbol{a}\cdot\boldsymbol{b}=a_1b_1+a_2b_2+\cdots+a_nb_n$$

几何解释：点积可通过向量的模长与夹角余弦值表示：

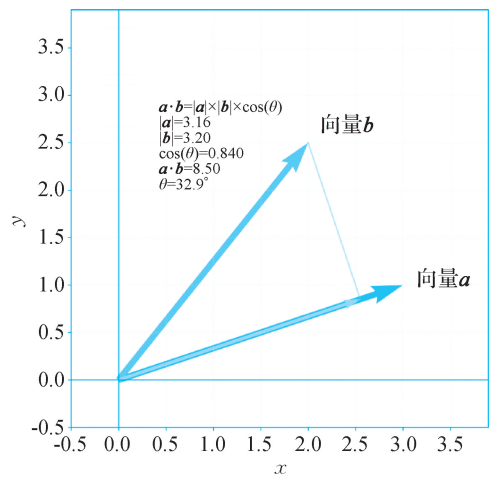


图 2.3.3　点积的几何解释

$$\boldsymbol{a}\cdot\boldsymbol{b}=\|\boldsymbol{a}\|\,\|\boldsymbol{b}\|\cos\theta$$

其中，θ 为两向量的夹角（$0\leqslant\theta\leqslant\pi$）。当 $\theta=0$ 时，点积取最大值（方向相同）；当 $\theta=90^\circ$ 时，点积为 0（垂直）；当 $\theta=180^\circ$ 时，点积取最小值（方向相反）。也就是说，点积是两个向量相互作用效果的衡量值。

```
# 点积的两种实现方式
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# 方法 1：使用 NumPy 内置函数
dot_product = np.dot(a, b)

# 方法 2：手动计算（展示原理）
```

```
manual_dot = sum(a_i *  b_i for a_i, b_i inzip(a, b))

print(f"点积结果:{dot_product}(内置函数),{manual_dot}(手动计算)")  #  输出:32
```

定义 2.3.2　向量的范数

向量的范数是衡量其“大小”的标量,常用以下两种:

L_2 范数(欧氏范数):n 维向量 $\boldsymbol{v}$ 的 L_2 范数定义为

$$\| \boldsymbol{v} \|_2=\sqrt{v_1^2+v_2^2+\cdots+v_n^2}$$

几何意义:向量的“长度”(从原点到向量端点的直线距离)。

L_1 范数(曼哈顿范数):定义为

$$\| \boldsymbol{v} \|_1=| v_1 |+| v_2 |+\cdots+| v_n |$$

几何意义:各维度绝对值之和(类似城市中沿网格移动的距离)。

特殊说明:L_2 范数的平方($\| \boldsymbol{v} \|_2^2$)等于向量与自身的点积($\boldsymbol{v}\cdot\boldsymbol{v}$),计算时可省略开方以提高效率。

```
#  范数的计算
v =  np.array([3, 4])

#  L2 范数(向量长度)
l2_norm =  np.linalg.norm(v)  #  内置函数
manual_l2 =  np.sqrt(v[0]** 2+  v[1]** 2)  #  手动计算

#  L1 范数
l1_norm =  np.linalg.norm(v, ord= 1)  #  内置函数
manual_l1 = abs(v[0]) + abs(v[1])  #  手动计算

print(f"L2 范数:{l2_norm},L1 范数:{l1_norm}")  #  输出:5.0,7
```

定义 2.3.3　余弦相似度

基于点积与范数可定义两个向量的余弦相似度:

$$\text{相似度}=\cos\theta=\frac{\boldsymbol{a}\cdot\boldsymbol{b}}{\| \boldsymbol{a} \|_2 \| \boldsymbol{b} \|_2}$$

工程意义:取值范围为$[-1,1]$,用于衡量向量的“方向相似性”(与长度无关)。例如:人脸识别中,即使图像亮度不同(向量长度差异),只要人脸特征方向一致,仍可判定为同一人。

```
# 计算余弦相似度(模拟人脸识别场景)
face1 = np.array([0.2, 0.5, 0.3, 0.1])  # 数据库人脸向量
face2 = np.array([0.4, 1.0, 0.6, 0.2])  # 摄像头采集(亮度加倍)

dot = np.dot(face1, face2)
norm1 = np.linalg.norm(face1)
norm2 = np.linalg.norm(face2)
similarity = dot / (norm1 * norm2)

print(f"余弦相似度:{similarity:.4f}")  # 输出:1.0(方向完全一致)
```

案例 1　用户偏好匹配与推荐

点积可计算用户偏好向量与物品特征向量的匹配度，L_1 范数可衡量用户间的“兴趣距离”，用于个性化推荐。

问题场景：某视频平台需为用户推荐电影。用户偏好向量包含“动作”“喜剧”“科幻”三个维度的权重，电影特征向量为对应类型的标签强度，通过点积筛选匹配度最高的电影；通过 L_1 范数找到与目标用户兴趣最接近的历史用户，参考其评分。

数学建模：

- 用户偏好向量：$\boldsymbol{u}=(u_1,u_2,u_3)$（$u_i$ 为对第 i 类的偏好权重）；
- 电影特征向量：$\boldsymbol{m}=(m_1,m_2,m_3)$（$m_i$ 为第 i 类的标签强度）；
- 匹配度：$\boldsymbol{u}\cdot\boldsymbol{m}$（值越大越匹配）；
- 用户间兴趣距离（L_1 范数）：$\|\boldsymbol{u}-\boldsymbol{u}'\|_1=|u_1-u_1'|+|u_2-u_2'|+|u_3-u_3'|$（值越小越相似）。

代码实现：

```
import numpy as np

# 用户偏好:[动作, 喜剧, 科幻]
user = np.array([0.8, 0.3, 0.9])  # 目标用户:喜欢动作和科幻
user_history = np.array([[0.7, 0.2, 0.8], [0.2, 0.9, 0.1]])  # 历史用户

# 电影特征:[动作, 喜剧, 科幻]
movies = np.array([
    [0.9, 0.1, 0.8],  # 电影 A:动作+ 科幻
    [0.2, 0.9, 0.1],  # 电影 B:喜剧
    [0.7, 0.3, 0.6]    # 电影 C:动作+ 轻度科幻
```

```
])

# 计算电影匹配度(点积)
match_scores = np.dot(movies, user)  # 等价于各电影与用户的点积

# 计算与历史用户的兴趣距离(L1范数)
distance1 = np.linalg.norm(user - user_history[0], ord=1)
distance2 = np.linalg.norm(user - user_history[1], ord=1)

print(f"电影匹配度:A={match_scores[0]:.2f}, B={match_scores[1]:.2f}, C={match_scores[2]:.2f}")
print(f"与历史用户的距离:用户1={distance1:.2f}(更近), 用户2={distance2:.2f}")
```

效果:电影 A 匹配度最高(1.55),且目标用户与历史用户 1 兴趣距离更近(0.2),可优先推荐电影 A 并参考用户 1 的评分。

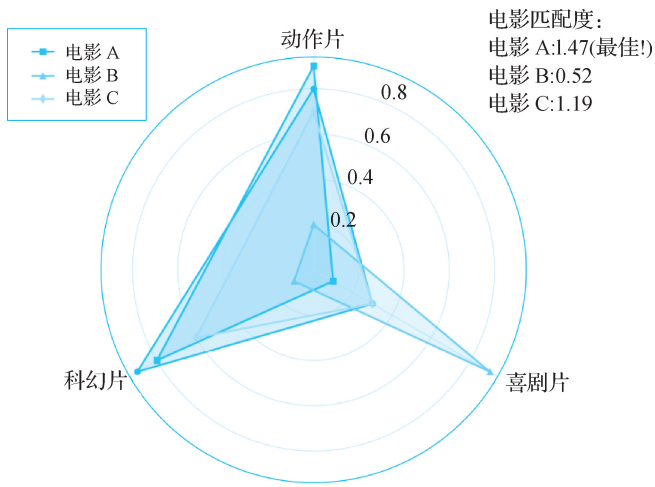


图 2.3.4　用户爱好度匹配

案例共性:点积通过“元素相乘再求和”量化向量的协同程度(信号同向性、偏好匹配度),范数通过“元素累积”衡量向量的大小或差异(能量、距离),二者结合实现从“数值计算”到“语义相似性”的转化,是跨领域分析“相似度”的核心工具。

扩展阅读

裴祥风:用点积照亮虚拟世界的工程师

巩固练习

基础概念

1. 已知向量 $\boldsymbol{a}=(2,-3,1)$ 和 $\boldsymbol{b}=(4,2,-5)$，手动计算：

(1) 点积 $\boldsymbol{a}\cdot\boldsymbol{b}$ 的值；

(2) $\boldsymbol{a}$ 的 L_1 范数和 L_2 范数；

(3) 两向量的余弦相似度(保留 2 位小数)。

2. 判断下列说法是否正确，并说明理由：

(1) 若两个向量的点积为 0，则这两个向量一定垂直；

(2) L_1 范数为 0 的向量，其 L_2 范数也一定为 0；

(3) 余弦相似度的取值范围是[0,1]。

3. 结合工程场景举例：

(1) 哪些场景需要用 L_1 范数而非 L_2 范数？

(2) 点积为正或为负分别表示向量间的什么关系？

代码实践

4. 编写 Python 代码，实现以下功能：

(1) 用 NumPy 定义两个 3 维向量 $\boldsymbol{x}=(1,2,3)$ 和 $\boldsymbol{y}=(4,5,6)$；

(2) 计算它们的点积(分别用 np.dot 和手动循环实现，对比结果)；

(3) 计算 $\boldsymbol{x}$ 的 L_1 范数、L_2 范数和无穷范数(ord=np.inf)。

```
import numpy as np

# 定义向量
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

# 用 np.dot 计算点积
dot_np = np.dot(x, y)

# 手动循环计算点积
dot_manual = 0
for i inrange(len(x)):
    dot_manual += x[i] * y[i]

print(f"np.dot 点积结果:{dot_np},手动计算结果:{dot_manual}")

# 计算范数
l1_x = np.linalg.norm(x, ord= 1)
```

```
l2_x =  np.linalg.norm(x, ord= 2)
inf_x =  np.linalg.norm(x, ord= np.inf)

print(f"L1 范数:{l1_x},L2 范数:{l2_x:.2f},无穷范数:{inf_x}")
```

5. 扩展练习:实现一个函数 cosine_similarity(a, b),要求:

(1) 输入两个 NumPy 向量,返回它们的余弦相似度;

(2) 若存在全零向量,返回 None 并提示错误。

```
import numpy as np

def cosine_similarity(a, b):
    norm_a =  np.linalg.norm(a)
    norm_b =  np.linalg.norm(b)
if norm_a = = 0or norm_b = = 0:
        print("错误:存在全零向量,无法计算余弦相似度")
        returnNone
    dot_product =  np.dot(a, b)
    return dot_product / (norm_a *  norm_b)

# 测试用例
a =  np.array([1, 2, 3])
b =  np.array([4, 5, 6])
c =  np.array([0, 0, 0])

print(cosine_similarity(a, b))  # 应返回约 0.9747
print(cosine_similarity(a, c))  # 应提示错误并返回 None
```

思考探索

1. 假设你在开发一个音乐推荐系统,用向量表示用户对不同风格音乐的偏好(如[0.8,0.2,0.5]表示对流行、摇滚、古典的喜好度)。尝试用余弦相似度分析:

(1) 如何判断两个用户的音乐品味是否相似?

(2) 若某用户向量为[1,0,0](只喜欢流行),另一用户为[0.5,0,0],两人的相似度是多少?这合理吗?

2. 对比实验:用 L_1 范数和 L_2 范数分别计算以下两组向量的距离($\boldsymbol{a}-\boldsymbol{b}$、$\boldsymbol{c}-\boldsymbol{d}$ 的范数),分析哪种范数对异常值(如 100)更敏感?为什么?

$\boldsymbol{a}=(1,2,3)$,$\boldsymbol{b}=(4,5,6)$;

$\boldsymbol{c}=(1,2,100)$,$\boldsymbol{d}=(4,5,6)$。

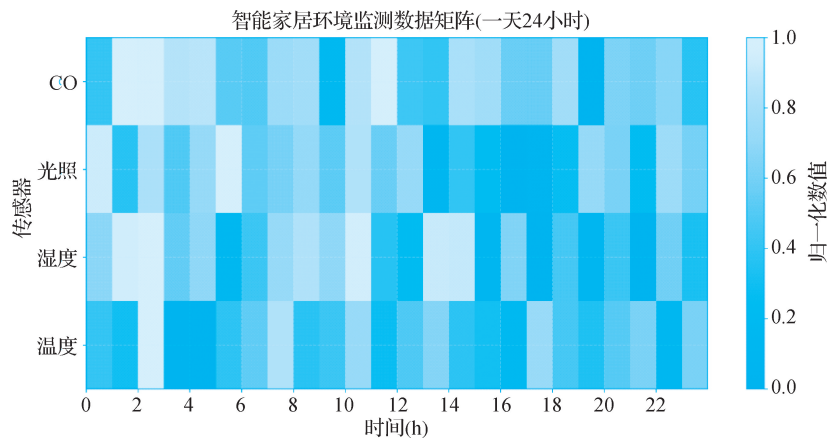
3. 尝试修改代码实践 2 中的函数,允许用户选择是否忽略全零向量(例如:返回 0,而非 None),并讨论这种处理方式在哪些场景下适用。

2.4 矩阵：表格数据与线性变换的载体

工程问题

当我们面对复杂的工程数据与系统变换时，矩阵成了不可或缺的数学工具。不妨思考这些具体场景：

在智能家居的环境监测系统中，一个房间内部署了温度、湿度、光照、CO_2 浓度 4 个传感器，每小时采集一次数据，一天 24 小时的监测结果如何用简洁的形式记录？如果要快速提取某天 10 点的所有传感器数据，或某一传感器的全天变化趋势，怎样的结构能让这种检索更高效？



核心概念

定义 2.4.1　矩阵

矩阵是由 $m \times n$ 个元素按 m 行(横向)和 n 列(纵向)排列成的矩形数组，记为：

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

其中，a_{ij} 表示第 i 行第 j 列的元素，矩阵的维度为 $m \times n$（读作“m 乘 n”）。

- 当 $m = 1$ 时(如 $\begin{bmatrix} a_{11} & a_{12} & a_{13} \end{bmatrix}$)，矩阵退化为**行向量**；

• 当 $n=1$ 时(如 $\begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix}$),矩阵退化为**列向量**;

• 二维矩阵可直观表示平面上的线性变换(如旋转、缩放),例如:$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$ 表示将平面向量逆时针旋转 90°。

Python 代码片段:

```
import numpy as np

# 定义一个 3×2 矩阵(3 行 2 列)
A = np.array([
    [1, 2],
    [3, 4],
    [5, 6]
])

print("矩阵 A:\n", A)
print("矩阵维度:", A.shape)  # 输出(3, 2)表示 3 行 2 列
print("第 2 行第 1 列元素:", A[1, 0])  # 注意:NumPy 索引从 0 开始
```

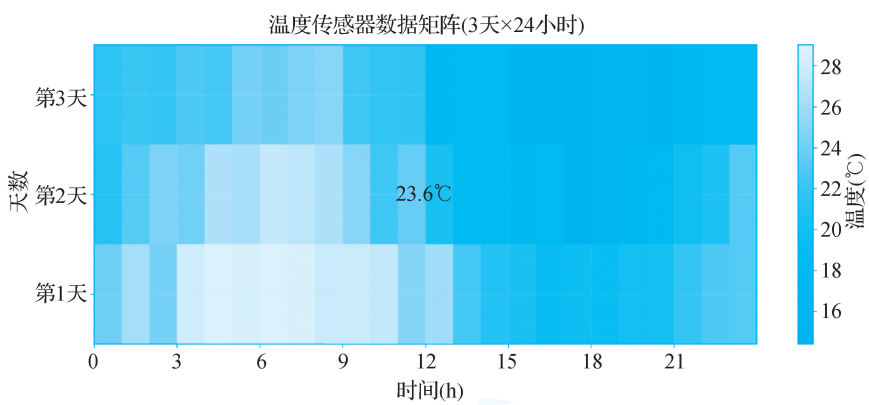
定义 2.4.2 矩阵与数据的对应关系

在工程中,矩阵的行和列可分别对应**样本**与**特征**:

• 行:每个样本(如一次测量、一张图像的一行像素);

• 列:样本的特征(如传感器类型、像素的颜色通道)。

示例:某温度传感器 3 天的每小时数据(共 724 小时)可表示为 3×24 矩阵。



矩阵的行对应样本(每天的数据);矩阵的列对应特征(每小时的温度);$A[i,j]$表示第i天第j小时的温度值。

图 2.4.2 温度传感器多数据的矩阵表示

$$\boldsymbol{T}=\begin{bmatrix} t_{1,1} & t_{1,2} & \cdots & t_{1,24} \\ t_{2,1} & t_{2,2} & \cdots & t_{2,24} \\ t_{3,1} & t_{3,2} & \cdots & t_{3,24} \end{bmatrix}$$

Python 代码片段：

```
#  生成 3 天×24 小时的随机温度数据(-10~35℃)
temperature_data =  np.random.randint(- 10, 36, size= (3, 24))
print("温度数据矩阵(行= 天,列= 小时):\n", temperature_data)

#  提取第 2 天(索引 1)的所有温度
day2_temps =  temperature_data[1, :]
print("第 2 天的温度:", day2_temps)
```

定义 2.4.3　线性变换的矩阵表示

对 n 维向量 $\boldsymbol{x}$ 的线性变换（如缩放、旋转、剪切）可表示为矩阵与向量的乘法：

$$\boldsymbol{y}=\boldsymbol{M}\cdot\boldsymbol{x}$$

其中，$\boldsymbol{M}$ 是 $m\times n$ 矩阵，$\boldsymbol{y}$ 为变换后的 m 维向量。

拓展思维：

矩阵可以看作一个“线性函数”，接收向量的输入并输出一个向量，也就是描述了从向量到向量的“映射”。

Python 代码片段：

```
#  定义缩放矩阵(放大 2 倍)
M =  np.array([[2, 0], [0, 2]])
#  定义原始向量
x =  np.array([1, 1])  #  向量(1,1)
#  执行线性变换(矩阵乘向量)
y =  M @  x  #  等价于 np.matmul(M, x)

print("变换前向量:", x)
print("变换后向量:", y)  #  输出(2, 2),即放大 2 倍
```

应用案例

案例 1　图像处理中的矩阵变换

问题背景：一张 200×200 像素的灰度图像需要旋转 90 度后保存，如何用矩阵运算实现这一几何变换？

矩阵应用：

• 图像的每个像素灰度值可表示为 200×200 矩阵 $\boldsymbol{I}$，其中 $\boldsymbol{I}[i,j]$ 代表第 i 行第 j 列的灰度值(0～255)；

• 逆时针旋转 90 度的变换矩阵为 $\boldsymbol{R}=[[0,-1],[1,0]]$，通过矩阵乘法可计算变换后像素的新坐标。

核心原理：图像旋转本质是对每个像素坐标(i,j)应用线性变换，矩阵乘法在此过程中实现了坐标的系统性映射，避免了逐像素计算的冗余。

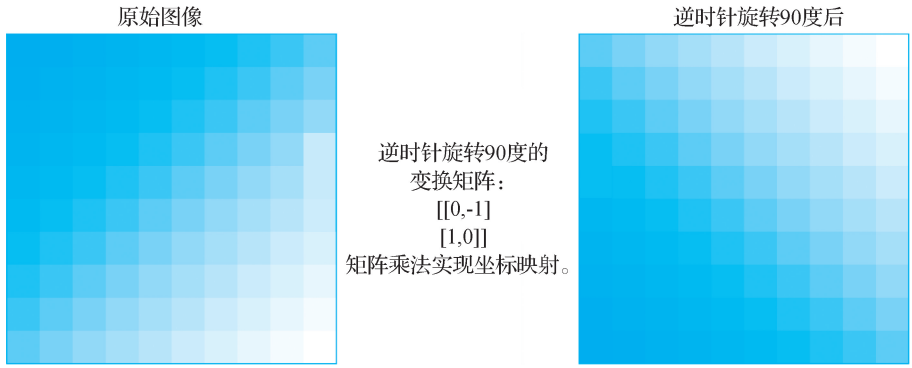


图 2.4.3　通过矩阵实现图像变换

扩展阅读

阿达 • 洛芙莱斯：在矩阵中预见编程未来的先知

巩固练习

基础概念

1. 填空题：

(1) 一个形状为(3,4)的矩阵，包含____行____列，总元素个数为_____。

(2) 单位矩阵的主对角线元素均为_____，其余元素为_____，它与任意同阶矩阵相乘后，结果等于_____。

(3) 若矩阵 $\boldsymbol{A}$ 的形状为(m,n)，矩阵 $\boldsymbol{B}$ 的形状为(n,p)，则 $\boldsymbol{A}\times\boldsymbol{B}$ 的形状为_____。

2. 判断题(正确的打✓，错误的打×)：

(1) 所有矩阵都可以进行转置运算。(　　)

(2) 矩阵加法要求两个矩阵的形状必须相同。(　　)

（3）转置后的矩阵与原矩阵一定形状相同。（　　）

代码实践

1. 基础操作：

用 NumPy 创建一个 3 行 2 列的矩阵 $\boldsymbol{M}$，元素为[[1,2],[3,4],[5,6]]，完成以下操作并打印结果：

（1）输出矩阵 $\boldsymbol{M}$ 的形状；

（2）访问并修改 $\boldsymbol{M}$ 中第 2 行第 1 列的元素（索引从 0 开始）为 10；

（3）计算 $\boldsymbol{M}$ 的转置矩阵 $\boldsymbol{M}^{\mathrm{T}}$，并输出其形状。

```
import numpy as np
#  请在此处编写代码
M =  np.array([[1, 2], [3, 4], [5, 6]])
print("矩阵 M 的形状:", M.shape)
M[1, 0] = 10
print("修改后的矩阵 M:\n", M)
M_T =  M.T
print("转置矩阵 M_T 的形状:", M_T.shape)
```

2. 应用尝试：

已知一张 2×2 的灰度图像矩阵为 img＝np. array([[200,150],[100,50]],dtype＝np. uint8)，尝试通过矩阵转置实现“左右翻转”效果，并打印翻转后的矩阵（提示：转置后观察行与列的变化）。

思考探索

1. 开放式问题：

生活中哪些数据可以用矩阵表示？请列举 3 个例子，并说明每个矩阵的行和列分别代表什么含义（例如：在 Excel 表格中，行可表示学生，列可表示各科成绩）。

2. 参数探究：

若将代码实践第 1 题中的矩阵 $\boldsymbol{M}$ 改为 4 行 3 列，转置后的矩阵形状会发生什么变化？尝试修改代码并观察结果，总结矩阵转置时“行数”与“列数”的转换规律。

3. 跨领域联想：

为什么在 5G 通信中，基站与用户设备的信号交互需要用矩阵建模？试着结合矩阵“多维度数据组织”的特性，用简单语言描述其优势。

2.5 矩阵运算：加法、数乘与矩阵乘法

在实际工程场景中，矩阵运算并非孤立的数学操作，而是解决复杂问题的基础工具。以下这些具体场景，都离不开矩阵的加法、数乘与乘法运算：

场景 1：多帧图像的降噪处理

监控摄像头拍摄的夜间画面常因光线不足产生噪点（每个像素的亮度随机波动）。工程师发现，连续拍摄的 10 帧图像中，同一位置的真实像素值基本稳定，噪点却随机变化。若将每帧图像视为一个矩阵（像素亮度构成的二维数组），如何通过运算提取出清晰图像？——这需要将 10 个图像矩阵逐元素相加，再整体乘以系数 1/10（数乘），利用噪声的随机性相互抵消。

场景 2：5G 基站的信号波束成形

在多天线通信系统中，基站需同时向 3 个用户发送信号，每个用户的接收信号为：

$$\text{接收信号} = \text{基站天线增益矩阵} \times \text{发射信号向量} + \text{噪声}$$

其中，增益矩阵的每行对应一个用户与各天线的信号衰减系数，列数等于天线数量。通过矩阵与向量的乘法运算，基站能精准控制信号的传播方向，避免用户间的干扰——这正是 5G 技术中“波束成形”的核心原理。

在这些场景中，矩阵加法实现了数据的叠加融合，数乘完成了比例缩放，而矩阵乘法则实现了维度转换与线性映射——本节将深入解析这些运算的规则与工程应用逻辑。

定义 2.5.1 矩阵加法

设两个矩阵 $\boldsymbol{A}=(a_{ij})$ 和 $\boldsymbol{B}=(b_{ij})$ 均为 $m\times n$ 形状（行数和列数分别相等），则它们的和 $\boldsymbol{C}=\boldsymbol{A}+\boldsymbol{B}$ 定义为：

$$\boldsymbol{C}=\begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

即对应位置元素逐个相加，结果矩阵 $\boldsymbol{C}$ 仍为 $m \times n$ 形状。

几何解释：矩阵加法可视为对同一维度数据的叠加。例如：两张同尺寸图像矩阵相加，相当于每个像素的亮度值叠加（可用于图像合成）。

```
import numpy as np

# 定义两个 2×3 矩阵
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[7, 8, 9], [10, 11, 12]])

# 矩阵加法
C = A + B  # 等价于 np.add(A, B)
print("A + B 的结果：\n", C)
# 输出：
# [[ 8 10 12]
#  [14 16 18]]

# 错误示例：形状不同的矩阵无法相加
D = np.array([[1, 2], [3, 4]])
# print(A + D)  # 运行会报错：ValueError: operands could not be broadcast together
```

定义 2.5.2　（矩阵数乘）

设矩阵 $\boldsymbol{A} = (a_{ij})$ 为 $m \times n$ 形状，k 为常数，则数乘结果 $\boldsymbol{B} = k\boldsymbol{A}$ 定义为：

$$\boldsymbol{B} = \begin{bmatrix} ka_{11} & ka_{12} & \cdots & ka_{1n} \\ ka_{21} & ka_{22} & \cdots & ka_{2n} \\ \vdots & \vdots & & \vdots \\ ka_{m1} & ka_{m2} & \cdots & ka_{mn} \end{bmatrix}$$

即矩阵中每个元素都与常数 k 相乘，结果矩阵形状不变。

几何解释：数乘相当于对矩阵进行整体缩放。例如：图像矩阵乘以 0.5 会使所有像素亮度减半（变暗）。

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
k = 2

# 矩阵数乘
B = k * A  # 等价于 np.multiply(k, A)
print("2A 的结果：\n", B)
```

```
# 输出:
# [[2 4]
#  [6 8]]

# 应用:图像亮度调整(假设 img 为 0~255 的灰度图像矩阵)
img = np.array([[200, 150], [100, 50]], dtype= np.uint8)
darker_img = 0.5* img  # 亮度减半
print("变暗后的图像:\n", darker_img.astype(np.uint8))  # 转换回整数类型
```

定义 2.5.3 矩阵乘法

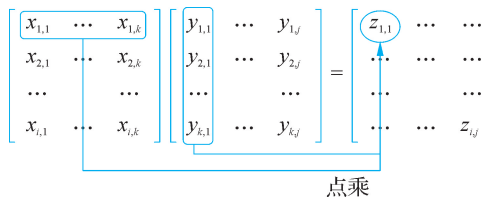


图 2.5.1 矩阵乘法的几何表示

设矩阵 $\boldsymbol{A}$ 为 $m \times n$ 形状,矩阵 $\boldsymbol{B}$ 为 $n \times p$ 形状($\boldsymbol{A}$ 的列数= $\boldsymbol{B}$ 的行数),则乘积 $\boldsymbol{C}=\boldsymbol{A} \times \boldsymbol{B}$ 为 $m \times p$ 形状矩阵,其中元素 c_{ij} 定义为:

$$c_{ij}=a_{i1}b_{1j}+a_{i2}b_{2j}+\cdots+a_{in}b_{nj}=\sum_{k=1}^{n}a_{ik}b_{kj}$$

即 $\boldsymbol{C}$ 的第 i 行第 j 列元素,等于 $\boldsymbol{A}$ 的第 i 行与 $\boldsymbol{B}$ 的第 j 列的对应元素乘积之和(点积)。

```
import numpy as np

# A 为 2×3 矩阵,B 为 3×2 矩阵(满足相乘条件:3 列= 3 行)
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[7, 8], [9, 10], [11, 12]])

# 矩阵乘法(使用 np.dot 或@ 运算符)
C = np.dot(A, B)  # 等价于 A @ B
print("A × B 的结果(2×2):\n", C)
# 计算过程:
# c11 = 1×7 + 2×9 + 3×11 = 7 + 18 + 33 = 58
# c12 = 1×8 + 2×10 + 3×12 = 8 + 20 + 36 = 64
# c21 = 4×7 + 5×9 + 6×11 = 28 + 45 + 66 = 139
# c22 = 4×8 + 5×10 + 6×12 = 32 + 50 + 72 = 154
# 输出:
# [[ 58  64]
```

```
#  [139 154]]

# 错误示例：形状不匹配(A 的列数≠B 的行数)
D = np.array([[1, 2], [3, 4]])
# print(A @ D)  # 运行会报错：ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0
```

性质 2.5.1　矩阵运算规律

1. **加法交换律**：若 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 形状相同，则 $\boldsymbol{A}+\boldsymbol{B}=\boldsymbol{B}+\boldsymbol{A}$；
2. **数乘分配律**：$k(\boldsymbol{A}+\boldsymbol{B})=k\boldsymbol{A}+k\boldsymbol{B}$（$k$ 为常数）；
3. **乘法结合律**：$(\boldsymbol{A}\times\boldsymbol{B})\times\boldsymbol{C}=\boldsymbol{A}\times(\boldsymbol{B}\times\boldsymbol{C})$（需满足形状匹配条件）；
4. **乘法对加法分配律**：$\boldsymbol{A}\times(\boldsymbol{B}+\boldsymbol{C})=\boldsymbol{A}\times\boldsymbol{B}+\boldsymbol{A}\times\boldsymbol{C}$（需满足形状匹配条件）。

注意：矩阵乘法**不满足交换律**，即 $\boldsymbol{A}\times\boldsymbol{B}\neq\boldsymbol{B}\times\boldsymbol{A}$（甚至可能无法相乘）。

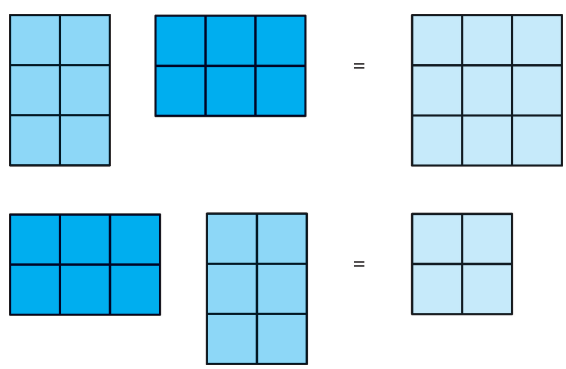


图 2.5.2　乘法不满足交换律

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# 验证乘法不满足交换律
print("A @ B:\n", A @ B)
print("B @ A:\n", B @ A)
# 输出结果不同，证明 A×B ≠ B×A
```

应用案例

案例 1　多帧图像降噪

问题场景：监控摄像头在低光环境下拍摄的图像存在随机噪点（像素亮度异常波动），

需通过多帧叠加提取清晰图像。

数学原理：设连续拍摄的 N 帧图像对应矩阵为 $\boldsymbol{I}_1,\boldsymbol{I}_2,\cdots,\boldsymbol{I}_N$（形状均为 $H\times W$，H 为高度，W 为宽度），真实图像矩阵为 $\boldsymbol{I}_{\text{true}}$，噪点矩阵为 $\epsilon_1,\epsilon_2,\cdots,\epsilon_N$（均值为 0 的随机噪声），则每帧图像可表示为：

$$\boldsymbol{I}_k=\boldsymbol{I}_{\text{true}}+\epsilon_k\quad(k=1,2,\cdots,N)$$

通过矩阵加法与数乘融合多帧图像：

$$\boldsymbol{I}_{\text{denoised}}=\frac{1}{N}(\boldsymbol{I}_1+\boldsymbol{I}_2+\cdots+\boldsymbol{I}_N)=\boldsymbol{I}_{\text{true}}+\frac{1}{N}(\epsilon_1+\cdots+\epsilon_N)$$

噪声项因随机性相互抵消，最终保留真实图像。

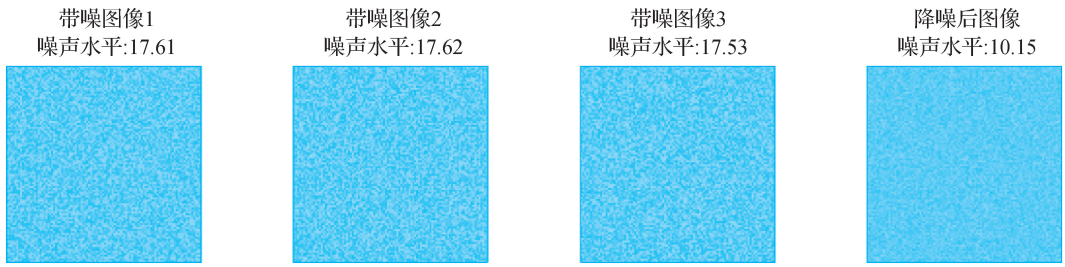
代码实现：

```
import numpy as np
import matplotlib.pyplot as plt

# 生成带噪图像(模拟 3 帧 200×200 的灰度图)
np.random.seed(42)
H, W = 200, 200
I_true = np.ones((H, W)) * 128 # 真实图像:灰度值 128 的均匀图像
I1 = I_true + np.random.randint(- 30, 31, (H, W)) # 加噪图像 1
I2 = I_true + np.random.randint(- 30, 31, (H, W)) # 加噪图像 2
I3 = I_true + np.random.randint(- 30, 31, (H, W)) # 加噪图像 3

# 多帧降噪:加法+ 数乘
I_denoised = (I1 + I2 + I3) /3 # 等价于 np.mean([I1, I2, I3], axis= 0)
```

效果分析：降噪后的图像像素值更接近真实值 128，噪点明显减少，验证了矩阵加法（多帧叠加）与数乘（平均化）的工程价值。



降噪原理：(1) 多帧图像叠加（矩阵加法）；(2) 平均化处理（数乘 1/3）；(3) 噪声相互抵消，保留真实信号。

扩展阅读

阿瑟·凯莱：从几何变换到矩阵乘法的伟大抽象

巩固练习

基础概念

1. 现有矩阵 $\boldsymbol{A}=\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$，$\boldsymbol{B}=\begin{bmatrix}5 & 6\\ 7 & 8\end{bmatrix}$，$\boldsymbol{C}=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}$：

（1）判断 $\boldsymbol{A}+\boldsymbol{B}$、$\boldsymbol{A}+\boldsymbol{C}$ 是否可行？若不可行，说明原因；

（2）计算 $2\boldsymbol{A}$ 和 $-\boldsymbol{B}$ 的结果；

（3）判断 $\boldsymbol{A}\times\boldsymbol{B}$、$\boldsymbol{A}\times\boldsymbol{C}$、$\boldsymbol{C}\times\boldsymbol{A}$ 是否可行？若可行，写出结果矩阵的维度。

2. 若矩阵 $\boldsymbol{M}$ 是 3×2 矩阵，矩阵 $\boldsymbol{N}$ 是 2×4 矩阵，那么 $\boldsymbol{M}\times\boldsymbol{N}$ 的结果是_____维矩阵，其第 2 行第 3 列的元素由 $\boldsymbol{M}$ 的_____行和 $\boldsymbol{N}$ 的_____列计算得到。

代码实践

1. 用 NumPy 完成以下操作，并输出结果：

（1）创建矩阵 $\boldsymbol{A}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$（单位阵）和 $\boldsymbol{B}=\begin{bmatrix}2 & 3\\ 4 & 5\end{bmatrix}$；

（2）计算 $\boldsymbol{A}+\boldsymbol{B}$、$3\boldsymbol{B}$、$\boldsymbol{A}\times\boldsymbol{B}$（矩阵乘法）；

（3）观察 $\boldsymbol{A}\times\boldsymbol{B}$ 与 $\boldsymbol{B}$ 的关系，思考单位阵在乘法中的作用。

2. 以下代码试图计算两个矩阵的乘法，但运行时会报错。请找出错误原因并修改代码，使其正确运行：

```
import numpy as np
X = np.array([[1, 2, 3], [4, 5, 6]])  # 2×3矩阵
Y = np.array([[7, 8], [9, 10]])       # 2×2矩阵
result = X @ Y
print(result)
```

3. 已知某灰度图像的像素矩阵为 $\boldsymbol{P}=\begin{bmatrix}100 & 150\\ 200 & 250\end{bmatrix}$（值越大越亮），用矩阵乘法实现图像的亮度翻倍（提示：用合适的对角阵与 $\boldsymbol{P}$ 相乘）。

思考探索

1. 尝试用 NumPy 计算 $\boldsymbol{A}\times\boldsymbol{B}$ 和 $\boldsymbol{B}\times\boldsymbol{A}$（其中 $\boldsymbol{A}$、$\boldsymbol{B}$ 为同阶方阵），观察结果是否相同。

结合生活中的例子(如“先旋转再平移”与“先平移再旋转”),解释为什么矩阵乘法不满足交换律。

2. 假设矩阵 $\boldsymbol{A}$ 是 $m\times n$ 矩阵,矩阵 $\boldsymbol{B}$ 是 $n\times k$ 矩阵。若将 $\boldsymbol{A}$ 的每一行视为一个向量,$\boldsymbol{B}$ 的每一列视为一个向量,思考:$\boldsymbol{A}\times\boldsymbol{B}$ 的第 i 行第 j 列元素与这两个向量有什么关系?(提示:回顾向量点积的定义)

3. 在电子信息领域,信号的混合与分离常通过矩阵乘法实现。例如:3 路原始信号 $\boldsymbol{s}=[s_1,s_2,s_3]^{\mathrm{T}}$ 经过混合矩阵 $\boldsymbol{M}$ 后得到观测信号 $\boldsymbol{x}=\boldsymbol{M}\boldsymbol{s}$。若已知 $\boldsymbol{M}=\begin{bmatrix}1 & 0.5 & 0.2\\ 0.5 & 1 & 0.3\\ 0.2 & 0.3 & 1\end{bmatrix}$,$\boldsymbol{s}=[1,0,-1]^{\mathrm{T}}$,尝试用矩阵乘法计算观测信号 $\boldsymbol{x}$,并思考:如何从 $\boldsymbol{x}$ 反推出 $\boldsymbol{s}$?(不必编程,仅阐述思路)

2.6 矩阵乘法:多种变换的叠加

工程问题

当我们需要对数据或物体施加一系列连续变换时,如何用简洁的数学方式描述最终效果?让我们看看这些来自工程实践的具体场景:

航拍图像的坐标校准:一架无人机拍摄的地面图像需要经过三次处理——先沿 x 轴平移 50 像素(消除相机偏移),再顺时针旋转 30°(修正飞行姿态倾斜),最后缩小至原尺寸的 80%(匹配地图比例尺)。每次变换都可以用一个矩阵表示,那么如何通过一次运算直接得到从原始图像到最终校准图像的坐标映射关系?

机械臂的末端定位:一个两关节机械臂,第一节臂长 10 cm,第二节臂长 8 cm。当第一节臂绕底座旋转 θ 角,第二节臂相对第一节再旋转 φ 角时,末端点的坐标需要同时考虑两次旋转和两次长度叠加的综合作用。如何用矩阵运算快速计算不同角度组合下的末端位置?

这些问题的共同核心在于:**多个连续变换的总效果,如何通过矩阵乘法实现“一步到位”的计算**?本节将揭示矩阵乘法与变换叠加之间的深层联系。

核心概念

定义 2.6.1 变换的复合

若对某个向量依次施加线性变换 T_1(对应矩阵 $\boldsymbol{A}$)和 T_2(对应矩阵 $\boldsymbol{B}$),则两次变换的总效果称为**变换的复合**,记为 $T_2\circ T_1$(先执行 T_1,再执行 T_2)。

几何解释：

在平面直角坐标系中，若 T_1 是“绕原点旋转 30°”（矩阵 $\boldsymbol{A}$），T_2 是“沿 x 轴拉伸 2 倍”（矩阵 $\boldsymbol{B}$），则复合变换 $T_2 \circ T_1$ 表示“先旋转 30°，再拉伸 x 轴”，其效果等价于用某个新矩阵对原始向量直接运算。

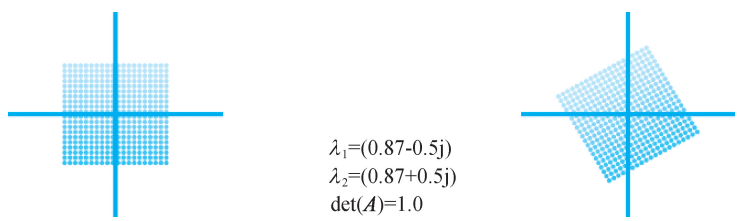
定理 2.6.1　复合变换的矩阵表示

若线性变换 T_1 对应矩阵 $\boldsymbol{A}_{m\times n}$，变换 T_2 对应矩阵 $\boldsymbol{B}_{p\times m}$，则复合变换 $T_2 \circ T_1$ 对应的矩阵为 $\boldsymbol{B} \times \boldsymbol{A}$（矩阵乘法），即：

$$T_2(T_1(\boldsymbol{x})) = (\boldsymbol{B} \times \boldsymbol{A})\boldsymbol{x}$$

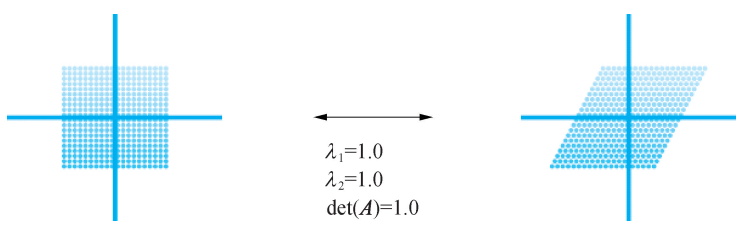
注：变换顺序与矩阵乘法顺序一致（先发生的变换对应右侧矩阵），且因矩阵乘法不满足交换律，$T_2 \circ T_1 \neq T_1 \circ T_2$（除非 $\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{B} \times \boldsymbol{A}$）。

设平面向量 $\boldsymbol{x} = \begin{bmatrix} x \\ y \end{bmatrix}$，执行以下变换：



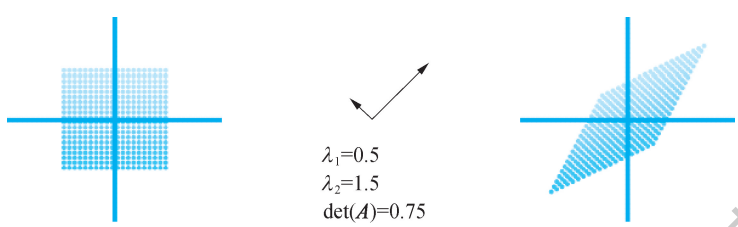
（1）旋转：绕原点顺时针旋转 θ 角，矩阵为：

$$\boldsymbol{R} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$



（2）缩放：x 轴拉伸 k 倍，y 轴不变，矩阵为：

$$\boldsymbol{S} = \begin{bmatrix} k & 0 \\ 0 & 1 \end{bmatrix}$$



复合变换“先旋转再缩放”的矩阵为 $\boldsymbol{S}\times\boldsymbol{R}$，即：

$$\boldsymbol{S}\times\boldsymbol{R}=\begin{bmatrix} k\cos\theta & k\sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

Python 代码实现：复合变换的矩阵运算

```
import numpy as np
import matplotlib.pyplot as plt

# 定义变换矩阵：旋转 30°(θ= 30°)和 x 轴缩放 2 倍
theta = np.radians(30) # 转换为弧度
R = np.array([[np.cos(theta), np.sin(theta)],
              [- np.sin(theta), np.cos(theta)]]) # 旋转矩阵
S = np.array([[2, 0], [0, 1]]) # 缩放矩阵

# 计算复合变换矩阵(先旋转再缩放)
C = S @ R # 等价于 S.dot(R)

# 验证：对向量[1,0]施加复合变换
x = np.array([1, 0])
# 分步计算：先旋转再缩放
x_rotated = R @ x
x_compound_step = S @ x_rotated
# 直接用复合矩阵计算
x_compound_direct = C @ x

print("分步结果：", x_compound_step)    # 输出：[1.732..., - 0.5]
print("复合矩阵结果：", x_compound_direct) # 输出相同，验证定理

# 可视化(略，可绘制原始向量、旋转后、复合变换后向量的箭头图)
```

推　论　多次变换的复合

对向量依次施加 $T_1,T_2,\cdots,T_k$（对应矩阵 $\boldsymbol{A}_1,\boldsymbol{A}_2,\cdots,\boldsymbol{A}_k$），总变换的矩阵为：

$$\boldsymbol{A}_k\times\boldsymbol{A}_{k-1}\times\cdots\times\boldsymbol{A}_1$$

工程意义：通过矩阵乘法将多步变换“打包”为单步运算，大幅简化机械臂控制、图像处理等领域的连续变换计算。

应用案例

案例 1　两关节机械臂的末端定位

问题场景：如图 2.6.1 所示，两关节机械臂的第一节臂长 $L_1=10$ cm，第二节臂长

$L_2 = 8$ cm。当第一节臂绕底座旋转 θ 角，第二节臂相对第一节旋转 ϕ 角时，如何计算末端点的坐标？

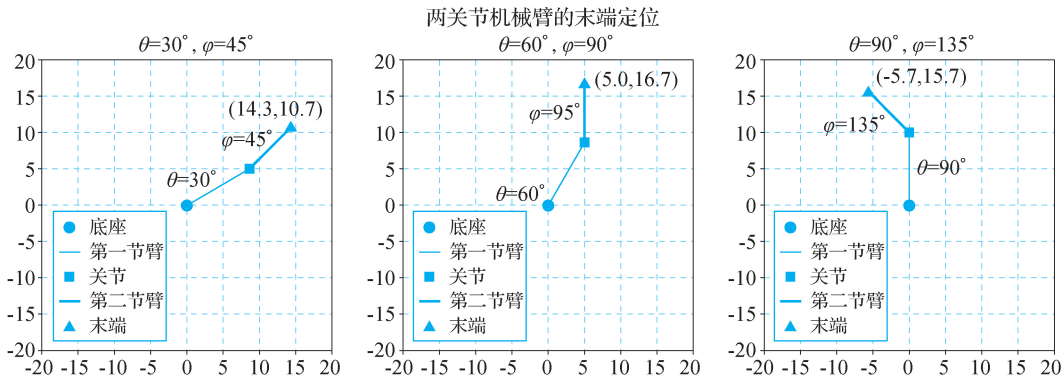


图 2.6.1　两关节机械臂的末端定位

末端点坐标可视为“基坐标→第一节末端→第二节末端”的两次旋转＋平移复合：

1. 第一节臂终点坐标（仅旋转）：$\begin{bmatrix} L_1\cos\theta \\ L_1\sin\theta \end{bmatrix}$；
2. 第二节臂相对第一节的旋转矩阵：$R_\phi = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$；
3. 第二节臂末端坐标（复合变换）：$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} L_1\cos\theta \\ L_1\sin\theta \end{bmatrix} + R_\phi \times \begin{bmatrix} L_2 \\ 0 \end{bmatrix}$。

代码验证：

```
import numpy as np

def arm_endpoint(theta, phi, L1= 10, L2= 8):
    theta_rad =  np.radians(theta)
    phi_rad =  np.radians(phi)
    #  第一节终点
    p1 =  np.array([L1 *  np.cos(theta_rad), L1 *  np.sin(theta_rad)])
    #  第二节旋转矩阵
    R_phi =  np.array([[np.cos(phi_rad), - np.sin(phi_rad)],
                       [np.sin(phi_rad), np.cos(phi_rad)]])
    #  第二节末端(复合变换)
    p2 =  p1 +  R_phi @  np.array([L2, 0])
    return p2

#  测试：θ= 30°, φ= 60°
print("末端坐标：", arm_endpoint(30, 60))  #  输出：[10.24, 12.99](近似值)
```

工程价值：通过矩阵表示旋转关系，可快速求解任意角度组合下的末端位置，为机械臂路径规划提供数学基础。若扩展至多关节机械臂，仅需增加对应旋转矩阵的乘法链即可。

扩展阅读

雷神之锤：用矩阵乘法编织虚拟世界

巩固练习

基础概念

1. **矩阵乘法可行性判断：**判断下列矩阵乘法是否可行，若可行则写出结果矩阵的维度：

（1）$\boldsymbol{A}_{3\times2}\times\boldsymbol{B}_{2\times4}$；

（2）$\boldsymbol{C}_{2\times2}\times\boldsymbol{D}_{3\times2}$；

（3）$\boldsymbol{E}_{1\times3}\times\boldsymbol{F}_{3\times1}$。

2. **复合变换逻辑分析：**已知变换顺序为“先沿 x 轴缩放 2 倍，再绕原点旋转 90°”，对应的缩放矩阵 $\boldsymbol{S}=\begin{bmatrix}2&0\\0&1\end{bmatrix}$，旋转矩阵 $\boldsymbol{R}=\begin{bmatrix}0&-1\\1&0\end{bmatrix}$。

（1）写出复合变换矩阵的表达式（用 $\boldsymbol{S}$ 和 $\boldsymbol{R}$ 表示）；

（2）计算点(1,1)经过该复合变换后的坐标。

3. **变换顺序影响：**若将上题的变换顺序改为“先旋转 90°，再缩放”，复合变换矩阵会如何变化？最终点(1,1)的坐标与上题结果是否相同？为什么？

代码实践

1. **基础复合变换实现：**使用 NumPy 编写代码，实现“先平移（x 轴+3，y 轴+2）再旋转 30°”的复合变换，要求：

- 用矩阵乘法表示复合变换（提示：平移需用齐次坐标，即 3×3 矩阵）；
- 对顶点坐标{(0,0),(2,0),(2,2),(0,2)}（正方形）执行变换，并打印变换前后的坐标。

2. **批量变换调试：**以下代码试图对 100 个随机点执行“缩放→旋转→平移”的复合变换，但运行结果异常。请找出错误并修正：

```
import numpy as np

# 生成 100 个随机点(范围 0- 10)
```

```
points = np.random.rand(100, 2) * 10  # 形状(100,2)

# 变换矩阵
S = np.array([[0.5, 0], [0, 0.5]])  # 缩小一半
theta = np.radians(45)
R = np.array([[np.cos(theta), - np.sin(theta)], [np.sin(theta), np.cos(theta)]])  # 旋转 45°
T = np.array([[1, 0, 5], [0, 1, 5], [0, 0, 1]])  # 平移(5,5)

# 错误的复合变换逻辑
wrong_transformed = points @ S @ R  # 未处理平移,且顺序错误
wrong_transformed + = [5, 5]  # 直接加法实现平移(忽略齐次坐标问题)

print(wrong_transformed.shape)
```

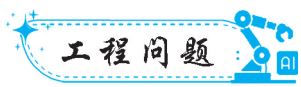
思考探索

1. **图像变换应用：**假设你需要实现一个简单的图像旋转缩放功能，已知图像的每个像素坐标为(x,y)，请设计一个复合变换矩阵，使图像先绕中心(100,100)旋转60°，再整体缩小至原来的0.8倍。提示：需分步骤处理“平移到原点→旋转→缩放→平移回中心”。

2. **误差分析：**当对同一个点执行1 000次“旋转1°＋缩放1.001倍”的复合变换后，结果与理论值可能存在偏差。请思考：偏差的主要来源是什么？如何通过代码优化减少这种偏差？

3. **变换可逆性：**若一个复合变换由矩阵$\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}\times\boldsymbol{D}$表示，那么其逆变换(即撤销该复合变换)的矩阵表达式是什么？请用NumPy验证“对一个点执行变换$\boldsymbol{C}$后再执行逆变换，结果与原坐标一致”。

2.7 特殊矩阵：单位阵与对角阵的作用



当你用图像编辑软件调整照片的RGB三色平衡时，是否想过计算机如何实现“只调红色通道亮度，保持绿、蓝通道不变”的精准操作？在无人机的四旋翼控制系统中，工程师需要向四个电机发送独立的转速指令，为什么这种“一对一”的控制信号能被高效编码为矩阵运算？而在训练神经网络时，对输入特征进行“标准化处理”——即每个特征减去均值再除以标准差——背后又隐藏着怎样的矩阵结构？

这些问题的共同核心，在于一类具有特殊结构的矩阵：当矩阵中大部分元素为0，仅

保留特定位置的非零元素时，它们能以极高的效率完成“选择性操作”。例如：对 RGB 图像的红色通道单独放大 1.2 倍，本质上是用一个特殊矩阵与像素矩阵相乘：红色分量被缩放，而绿、蓝分量通过矩阵中“不改变原始值”的特殊元素保持不变。同样，四旋翼的转速控制指令可以通过对角矩阵直接分配给对应的电机，避免信号之间的相互干扰。

这些看似简单的矩阵，为何能在复杂系统中承担“精准操控”的角色？它们的数学性质如何简化工程计算？本节将揭开单位阵、对角阵等特殊矩阵的神秘面纱，看看这些“简化版”矩阵如何成为工程实践中的高效工具。

定义 2.7.1 （单位矩阵）

单位矩阵是一种特殊的方阵，其主对角线元素均为 1，其余元素均为 0。n 阶单位矩阵记为 $\boldsymbol{I}_n$，定义如下：

$$\boldsymbol{I}_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

几何意义：单位矩阵对向量或矩阵的变换具有“恒等性”，如同数字运算中的“1”。对任意向量 $\boldsymbol{v}$ 或矩阵 $\boldsymbol{A}$，有 $\boldsymbol{I} \times \boldsymbol{v} = \boldsymbol{v}$ 和 $\boldsymbol{I} \times \boldsymbol{A} = \boldsymbol{A}$，即不改变原向量/矩阵的大小和方向。

Python 代码实现：

```
import numpy as np

# 创建 3 阶单位矩阵
I3 = np.eye(3)
print("3 阶单位矩阵：\n", I3)

# 验证恒等性
v = np.array([2, 3, 4])
print("单位阵与向量相乘结果：", I3 @ v)  # 结果与原向量相同

A = np.array([[1, 2], [3, 4]])
I2 = np.eye(2)
print("单位阵与矩阵相乘结果：\n", I2 @ A)  # 结果与原矩阵相同
```

定义 2.7.2 （对角矩阵）

对角矩阵是主对角线以外元素全为 0 的方阵，记为 $\boldsymbol{D}$，定义如下：

$$\boldsymbol{D}=\begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & d_n \end{bmatrix}$$

其中，$d_1,d_2,\cdots,d_n$ 为对角线元素，可简写为 $\boldsymbol{D}=\mathrm{diag}(d_1,d_2,\cdots,d_n)$。

几何意义：对角矩阵对向量的作用是“坐标轴独立缩放”。若用对角矩阵 $\boldsymbol{D}=\mathrm{diag}(a,b)$ 乘以二维向量 (x,y)，结果为 (ax,by)，即 x 轴方向缩放 a 倍，y 轴方向缩放 b 倍，且坐标轴之间无耦合。

Python 代码实现：

```
import numpy as np

# 创建对角矩阵
D = np.diag([2, 3, 0.5])  # 对角线元素为 2,3,0.5
print("对角矩阵：\n", D)

# 验证缩放效果
v = np.array([1, 1, 1])
print("对角阵缩放后向量：", D @ v)  # 结果为(2×1, 3×1, 0.5×1)

# RGB 通道独立调节示例(3×3 对角阵应用)
rgb = np.array([255, 128, 64])  # R= 255, G= 128, B= 64
adjust = np.diag([1.2, 1.0, 0.8])  # 红通道×1.2,绿通道不变,蓝通道×0.8
new_rgb = adjust @ rgb
print("调节后 RGB 值：", np.clip(new_rgb, 0, 255).astype(int))  # 截断到 0~ 255 范围
```

性质 2.7.1　特殊矩阵的运算规律

1. **单位阵的运算：**对任意 $m\times n$ 矩阵 $\boldsymbol{A}$，有 $\boldsymbol{I}_m\times\boldsymbol{A}=\boldsymbol{A}\times\boldsymbol{I}_n=\boldsymbol{A}$

2. **对角阵的乘法：**两个同阶对角阵相乘仍为对角阵，且对角线元素为对应元素的乘积，即：

若 $\boldsymbol{D}_1=\mathrm{diag}(a_1,a_2,\cdots,a_n)$，$\boldsymbol{D}_2=\mathrm{diag}(b_1,b_2,\cdots,b_n)$，则

$$\boldsymbol{D}_1\times\boldsymbol{D}_2=\mathrm{diag}(a_1b_1,a_2b_2,\cdots,a_nb_n)$$

3. **对角阵的逆：**若对角阵 $\boldsymbol{D}=\mathrm{diag}(d_1,d_2,\cdots,d_n)$ 中所有 $d_i\neq 0$，则其逆矩阵为

$$\boldsymbol{D}^{-1}=\mathrm{diag}(1/d_1,1/d_2,\cdots,1/d_n)$$

Python 验证：

```
import numpy as np

# 验证对角阵乘法规律
```

```
D1 = np.diag([2, 3])
D2 = np.diag([4, 5])
print("对角阵相乘结果:\n", D1 @ D2)  # 应为 diag(8,15)

# 验证对角阵的逆
D = np.diag([2, 3])
D_inv = np.diag([1/2, 1/3])
print("对角阵与其逆的乘积:\n", D @ D_inv)  # 结果应为单位阵
```

关键提示:特殊矩阵的价值在于**简化计算**。在工程中,当需要对多维度数据进行独立操作(如各传感器校准、多通道信号调节)时,对角阵可避免维度间的交叉干扰;而单位阵作为"基准变换",常用于控制系统的初始状态定义或矩阵运算的身份验证。

应用案例

案例 1　RGB 图像的通道独立调节

问题场景:在图像编辑中,需单独增强红色通道对比度、减弱蓝色通道亮度,同时保持绿色通道不变。已知图像每个像素的 RGB 值为三维向量 (r,g,b),如何通过矩阵运算实现精准调节?

数学原理:利用对角矩阵对各维度的独立缩放特性,构造调节矩阵 $\boldsymbol{D}=\mathrm{diag}(k_r,k_g,k_b)$。其中,$k_r=1.5$(增强红色)、$k_g=1$(保持绿色)、$k_b=0.6$(减弱蓝色)。像素变换公式为:

$$\begin{bmatrix} r' \\ g' \\ b' \end{bmatrix} = \boldsymbol{D} \times \begin{bmatrix} r \\ g \\ b \end{bmatrix}$$

代码实现:

```
import numpy as np
import matplotlib.pyplot as plt

# 生成示例像素(3 行 3 列的简单图像)
# 像素值范围[0,255],形状为(高度,宽度,3)
image = np.array([
    [[200, 100, 50], [150, 80, 30], [100, 60, 20]],
    [[180, 90, 40], [220, 110, 60], [130, 70, 25]],
    [[160, 85, 35], [190, 95, 45], [110, 55, 15]]
], dtype= np.uint8)
```

```
# 定义通道调节对角矩阵
D = np.diag([1.5, 1.0, 0.6])

# 对每个像素应用变换(需保持 uint8 类型)
adjusted_image = np.clip(image @ D, 0, 255).astype(np.uint8)  # 矩阵乘法+ 截断

# 对比显示(简化展示)
print("原始像素(第一行):\n", image[0])
print("调节后像素(第一行):\n", adjusted_image[0])
```

结果分析：红色分量均变为原来的 1.5 倍(如 200→300→截断为 255)，绿色分量不变(100→100)，蓝色分量变为原来的 0.6 倍(50→30)，实现了通道间无干扰的独立调节。

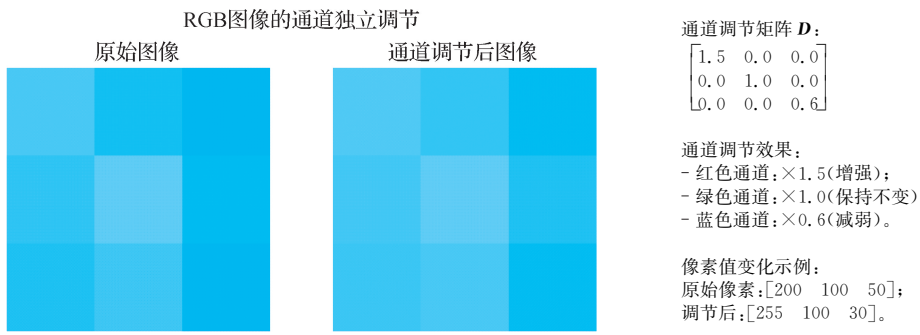


图 2.7.1　RGB 图像的通道独立调节

扩展阅读

华为 5G 研发团队：特殊矩阵开启大规模天线技术新时代

巩固练习

基础概念

1. 判断题(说明理由)：

(1) 任意阶单位矩阵的行列式值均为 1；

(2) 对角矩阵的逆矩阵仍是对角矩阵，且其对角线元素为原矩阵对应元素的倒数；

(3) 若矩阵 $\boldsymbol{A}$ 与单位矩阵 $\boldsymbol{I}$ 的乘积等于 $\boldsymbol{A}$，则 $\boldsymbol{A}$ 必为方阵。

2. 填空题：

(1) 设 3 阶对角矩阵 $\boldsymbol{D}=\mathrm{diag}(2,-1,3)$，则 $\boldsymbol{D}^2$ 的对角线元素为_____；

（2）若 $\boldsymbol{I}$ 为 4 阶单位矩阵，$\boldsymbol{A}$ 为 4×5 矩阵，则 $\boldsymbol{I}\cdot\boldsymbol{A}$ 的形状为_____。

代码实践

1. 基础操作：

用 NumPy 创建一个 5 阶单位矩阵 $\boldsymbol{I}$ 和一个对角线元素为[10,20,30,40,50]的对角矩阵 $\boldsymbol{D}$，并验证：

（1）$\boldsymbol{I}\cdot\boldsymbol{D}=\boldsymbol{D}\cdot\boldsymbol{I}=\boldsymbol{D}$；

（2）$\boldsymbol{D}$ 的逆矩阵（手动构造）与 $\boldsymbol{D}$ 的乘积为单位矩阵。

（提示：使用 np.diag 和矩阵乘法运算符@）

2. 工程应用模拟：

假设有一个 3×3 的图像像素矩阵 $\boldsymbol{A}$（元素值范围 0～255）：

```
import numpy as np
A =  np.array([[120, 80, 200], [50, 180, 30], [90, 150, 70]])
```

请用对角矩阵实现对图像的亮度调节：将第一行像素亮度减半，第二行不变，第三行亮度加倍，输出调节后的矩阵（需确保像素值不超过 255，超出部分取 255）。

思考探索

1. 开放性问题：

在机器学习中，常使用对角矩阵对数据进行“标准化”（如将每个特征的方差缩放到 1）。请设计一个场景，说明如何用对角矩阵实现这一过程，并思考：若某特征的方差为 0（所有样本值相同），对角矩阵应如何处理才能避免错误？

2. 跨领域联想：

对比“单位矩阵作为线性变换的基准”与“电路中的参考接地”，分析两者在“基准定义”和“系统稳定性”方面的相似性。尝试用一句话概括这种数学与工程的共通思想。

项目小结

概　念

- **向量：**n 维有序数组，可表示多维数据（如信号采样值、图像像素特征），具有大小和方向；
- **矩阵：**$m\times n$ 的二维数组，可表示表格数据（如灰度图像）或线性变换（如坐标旋转）；
- **核心运算：**向量点积（衡量相似度）、矩阵乘法（变换叠加）、特殊矩阵（单位阵：变换基准；对角阵：独立缩放）。

方　法

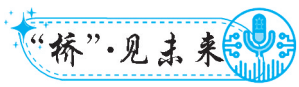
- **工具实现：**用 numpy.array 定义向量/矩阵，@ 实现矩阵乘法，np.dot 计算点积，np.eye/np.diag 构造特殊矩阵；

• **关键思路：**通过“维度检查”规避运算错误，通过“几何可视化”理解变换本质（如矩阵乘法对应坐标系拉伸/旋转）。

应　用

• 电子信息领域：用矩阵表示电路节点电压关系，用对角阵实现信号通道增益调节；

• 人工智能领域：用向量表示样本特征，用矩阵乘法实现神经网络中的特征变换。



本项目学习的向量与矩阵是《信号与系统》中“信号的线性变换”、《机器学习入门》中“特征工程与神经网络”的核心数学基础。例如：卷积神经网络中对图像的滤波操作，本质是矩阵与卷积核的乘法；电路分析中的基尔霍夫定律方程组，可通过矩阵形式简化求解。掌握这些工具，将为解析复杂系统模型提供“通用语言”。

项目 3

线性方程组与矩阵分解：求解与降维

“探脉寻源，因势利导。” ——《梦溪笔谈》沈括（中国北宋科学家）

学习目标

知识维度

1. 理解线性方程组的工程意义，掌握高斯消元法的基本原理；
2. 明确矩阵的逆、秩、特征值与特征向量的定义及几何含义；
3. 掌握奇异值分解（SVD）的数学表达，理解其降维思想。

能力维度

1. 能使用 NumPy 实现线性方程组的求解和矩阵分解（特征值分解、SVD）；
2. 能将电路分析中的多元电路问题转化为线性方程组模型；
3. 能运用 SVD 完成简单的图像压缩或数据降维任务。

素养维度

1. 培养“从复杂数据中提取关键信息”的工程思维（如用秩评估数据有效性）；
2. 建立“数学工具服务于工程问题”的应用意识（如用 SVD 平衡数据存储与精度）；
3. 提升数值计算中的严谨性（如关注矩阵运算的维度匹配、浮点数误差）。

3.1 线性方程组：电路分析中的基本模型

工程问题

当你拆开一台智能手机的主板，会看到密密麻麻的电路网络——电源、电阻、电容等元件通过导线连接成多个回路。工程师在设计充电电路时，如何精确计算每条支路上的电流大小？在人工智能实验室里，研究者正在修复一张受损的老照片，如何建立关系式求

出这 5 个缺失像素的亮度，让照片恢复原貌？

无论是多回路电路的电流求解，还是图像缺失信息的修复，看似毫无关联的问题背后，都隐藏着相同的数学本质——它们都可以转化为一组包含多个未知数的线性等式，而解决这类问题的通用工具，正是我们即将学习的线性方程组。

定义 3.1　线性方程

含有 n 个未知数 $x_1, x_2, \cdots, x_n$ 的一次方程称为**线性方程**，其一般形式为：

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$$

其中，$a_1, a_2, \cdots, a_n$ 为未知数的系数；b 为常数项，系数与常数项均为实数。

几何解释：

- 当 $n=2$ 时，线性方程表示平面上的一条直线（如 $2x_1 + 3x_2 = 6$）；
- 当 $n=3$ 时，线性方程表示空间中的一个平面（如 $x_1 - 2x_2 + 5x_3 = 10$）。

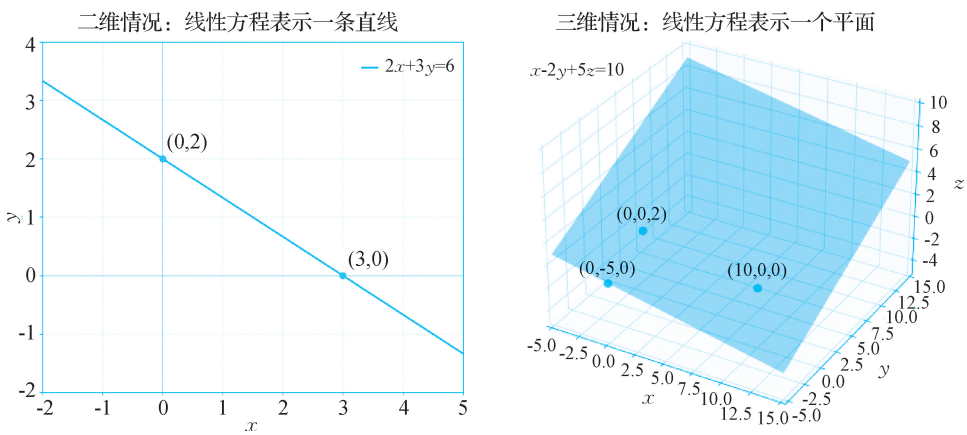


图 3.1.1　线性方程的几何解释

定义 3.2　线性方程组

由 m 个含 n 个未知数的线性方程组成的方程组称为**线性方程组**，其标准形式为：

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases}$$

其中，$a_{ij}(i=1,2,\cdots,m;j=1,2,\cdots,n)$为第 i 个方程中第 j 个未知数的系数；b_i 为第 i 个方程的常数项。

解的定义：若一组数 $(x_1^*,x_2^*,\cdots,x_n^*)$ 代入方程组后使每个方程都成立，则称这组数为该方程组的**解**。方程组所有解的集合称为**解集**。

定义 3.3　线性方程组的矩阵形式

根据矩阵向量乘法，线性方程组可通过矩阵与向量的乘法简化表示。记：

- 系数矩阵 $\boldsymbol{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{bmatrix}$；
- 未知数向量 $\boldsymbol{x}=\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$；
- 常数项向量 $\boldsymbol{b}=\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$，

则线性方程组可简写为**矩阵方程：**

$$\boldsymbol{Ax}=\boldsymbol{b}$$

几何意义：线性方程组的解只与系数矩阵有关，与变量是什么无关。

【Python 代码实现：线性方程组的表示】

```
import numpy as np

# 示例：表示工程问题中的电路方程组
# 假设 3 个回路的电流方程为：
# 1x1 + 2x2 = 5
# 2x2 + 3x3 = 3
# 1x1 + 3x3 = 2

# 系数矩阵 A
A = np.array([
    [1, 2, 0],  # 第一个方程系数
    [0, 2, 3],  # 第二个方程系数
    [1, 0, 3]   # 第三个方程系数
])
```

```
#  常数项向量 b
b =  np.array([5, 3, 2])

print("系数矩阵 A:")
print(A)
print("\n 常数项向量 b:")
print(b)
```

代码说明：

• 使用 NumPy 数组存储系数矩阵和常数项向量，直接对应数学中的 $\boldsymbol{A}$ 和 $\boldsymbol{b}$；

• 方程中缺失的未知数（如第一个方程不含 x_3）对应系数为 0，体现了线性方程组的结构化表示。

案例 1　多回路电路的电流求解

在手机充电电路设计中，某模块包含 3 个相互耦合的回路（如图 3.1.2 所示），已知电源电动势分别为 $E_1=5\ \mathrm{V}$、$E_2=3\ \mathrm{V}$、$E_3=2\ \mathrm{V}$，各支路电阻 $R_1=1\ \Omega$、$R_2=2\ \Omega$、$R_3=3\ \Omega$，需计算三条支路的电流 I_1、I_2、I_3 以验证电路安全性。

求解步骤：

1. 根据基尔霍夫电压定律（KVL）建立方程组：

• 回路 1：$R_1I_1+R_2I_2=E_1 \rightarrow 1I_1+2I_2=5$；

• 回路 2：$R_2I_2+R_3I_3=E_2 \rightarrow 2I_2+3I_3=3$；

• 回路 3：$R_1I_1+R_3I_3=E_3 \rightarrow 1I_1+3I_3=2$。

2. 用 Python 求解线性方程组：

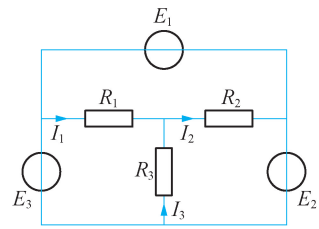


图 3.1.2　多回路电路

```
import numpy as np

#  系数矩阵 A 和常数项向量 b
A =  np.array([[1, 2, 0],
               [0, 2, 3],
               [1, 0, 3]])
b =  np.array([5, 3, 2])

#  求解方程组 Ax =  b
I =  np.linalg.solve(A, b)  #  利用线性代数库求解

print(f"支路电流:I1= {I[0]:.2f}A, I2= {I[1]:.2f}A, I3= {I[2]:.2f}A")
#  输出:支路电流:I1= 2.00A, I2= 1.50A, I3= 0.00A
```

3. 结果分析：

计算得 $I_3=0$ A，说明第三条支路无电流，电路设计中可优化该支路元件；I_1 和 I_2 均为正值且数值合理，不会导致电阻过载（功率 $P=I^2R$ 均在安全范围内）。

扩展阅读

古斯塔夫·基尔霍夫：用线性方程组为电路立法的物理学家

巩固练习

基础概念

1. **判断题**：请判断下列关于线性方程组的说法是否正确，并简述理由。

（1）所有线性方程组都有唯一解；

（2）系数矩阵为 3×3 的方程组，其常数项向量长度必须为 3；

（3）若线性方程组的两个方程完全相同，则方程组一定有无穷多解。

2. **列方程组**：某串联电路包含 3 个电阻 $R_1=2\ \Omega$、$R_2=3\ \Omega$、$R_3=5\ \Omega$，总电压为 10 V。设通过三个电阻的电流分别为 I_1、I_2、I_3（串联电路中电流相等，即 $I_1=I_2=I_3=I$），根据欧姆定律（$U=IR$）和总电压等于各部分电压之和，列出以电流 I 为未知数的线性方程。

代码实践

1. **基础求解**：已知某电路的电压方程为：

$$\begin{cases}2I_1+3I_2=12\\I_1-I_2=1\end{cases}$$

请用 NumPy 编写代码求解电流 I_1 和 I_2，并打印结果。

2. **错误处理**：尝试用 np.linalg.solve 求解下列方程组，观察并记录报错信息，然后改用最小二乘法求近似解：

$$\begin{cases}I_1+2I_2=5\\2I_1+4I_2=9\end{cases}$$

提示：使用 np.linalg.lstsq 函数，对比近似解与原方程的匹配程度。

思考探索

1. **物理意义分析**：若求解某电路电流时得到负数值，这是否意味着计算错误？结合电路知识，解释负电流的实际含义。

2. **参数敏感性**：修改代码实践 1 中的总电压（如从 12 V 改为 10 V），观察电流解的变

化规律。尝试总结：当方程组中常数项（电压）发生微小变化时，解（电流）的变化与系数矩阵（电阻）有何关联？

3. **模型扩展：**若某电路为并联结构（电压相等，总电流等于各支路电流之和），请设计一个包含 3 条支路的电路，列出对应的线性方程组，并思考如何用代码验证解的合理性（提示：可结合基尔霍夫定律）。

3.2 方程组求解：高斯消元法的直观理解

当电路从简单的串联、并联结构升级为多回路复杂网络时，手动求解方程组的难度会急剧增加。例如：某汽车电子控制系统中的电源分配电路包含 4 个回路，每个回路的电压方程由基尔霍夫定律得出：

$$\begin{cases} 2I_1 + I_2 - I_3 = 10 \\ I_1 - 3I_2 + 2I_4 = -5 \\ -I_1 + 4I_3 - I_4 = 8 \\ 3I_2 - 2I_3 + 5I_4 = 3 \end{cases}$$

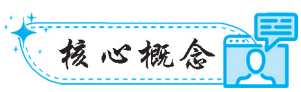
若采用逐个消元的零散思路，不仅容易出现计算错误，还会因步骤混乱导致调试困难。

在人工智能领域，图像传感器的校准问题也面临类似挑战。某摄像头的 3 个像素传感器存在测量偏差，其校正方程为：

$$\begin{cases} 0.9x + 0.2y - 0.1z = 128 \\ 0.3x + 0.8y + 0.2z = 96 \\ -0.1x + 0.3y + 0.7z = 112 \end{cases}$$

其中，x, y, z 为真实像素值，如何快速消除变量间的耦合关系，得到准确的校正参数？

这些问题的共性在于：需要一种**系统化、步骤明确的消元流程**，将复杂方程组逐步简化为可直接求解的形式。高斯消元法正是为解决这类问题而生——它通过标准化的行变换操作，像“拆解精密仪器”一样逐层剥离变量依赖，最终让方程组的解清晰可见。



定义 3.2.1　高斯消元法

高斯消元法是求解线性方程组的系统化方法，通过**行变换**将方程组的增广矩阵转化

为**上三角矩阵**(主对角线下方元素全为 0),再通过**回代**求解未知数。其核心思想是:逐步消除变量,将复杂方程组转化为易于求解的形式。

对于线性方程组:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{mn}x_n = b_n \end{cases}$$

其增广矩阵为:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{mn} & b_n \end{bmatrix}$$

1. 消元阶段:转化为上三角矩阵

通过以下三种行变换操作,将增广矩阵转化为上三角形式:

- **行交换**:交换两行位置(如避免主元为 0);
- **行缩放**:用非零常数乘以某一行;
- **行相加**:将一行的倍数加到另一行上。

示例(二元方程组):对增广矩阵执行消元:

$$\begin{bmatrix} 2 & 3 & 12 \\ 1 & -1 & 1 \end{bmatrix} \xrightarrow{\text{交换行 1 和行 2}} \begin{bmatrix} 1 & -1 & 1 \\ 2 & 3 & 12 \end{bmatrix} \xrightarrow{\text{行 2} = \text{行 2} - 2 \times \text{行 1}} \begin{bmatrix} 1 & -1 & 1 \\ 0 & 5 & 10 \end{bmatrix}$$

2. 回代阶段:求解未知数

从最后一行开始,依次求出每个未知数:

- 最后一行直接得到 x_n 的值;
- 倒数第二行代入 x_n 的值,求出 x_{n-1};
- 以此类推,直至求出 x_1。

示例(续上):由上三角矩阵得:

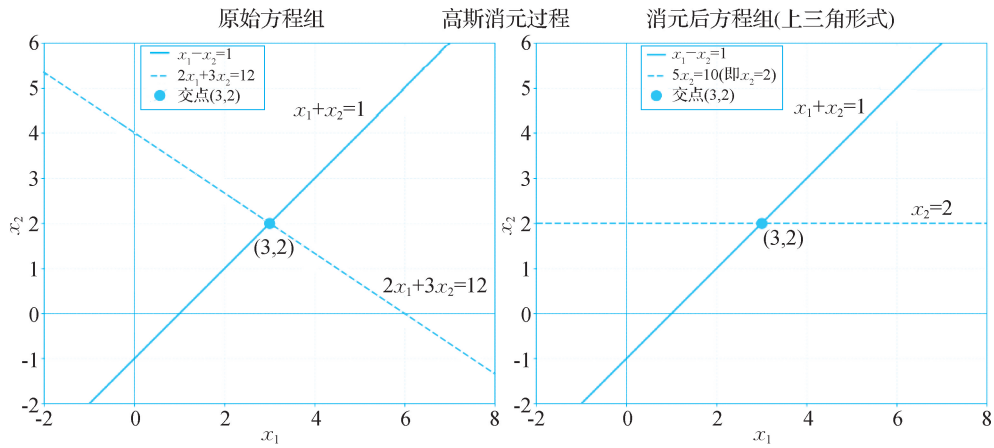
$$\begin{cases} x_1 - x_2 = 1 \\ 5x_2 = 10 \end{cases}$$

解得:$x_2 = 2$,代入第一行得 $x_1 = 3$。

几何解释:

在二维平面中,二元一次方程组的解对应两条直线的交点。高斯消元的过程本质是:

- 将两条直线通过“旋转”“平移”等变换(行变换),转化为一条垂直于坐标轴的直线(如 $5x_2 = 10$)和一条斜直线,使交点更易观察。



消元过程将斜直线转化为水平线,使交点更易观察。

图 3.2.1　高斯消元法的几何解释

【Python 实现：手动高斯消元】

```
import numpy as np

def gauss_elimination(A, b):
"""高斯消元法求解线性方程组 Ax = b"""
    n = len(b)
    # 构造增广矩阵
    aug_matrix = np.hstack((A, b.reshape(- 1, 1))).astype(float)

# 消元阶段
for i inrange(n- 1):
        # 找到主元(当前列中绝对值最大的行,避免除以 0)
        pivot_row = np.argmax(np.abs(aug_matrix[i:, i])) + i
        aug_matrix[[i, pivot_row]] = aug_matrix[[pivot_row, i]]

        # 消去当前列下方的元素
        for j inrange(i+ 1, n):
            factor = aug_matrix[j, i] / aug_matrix[i, i]
            aug_matrix[j] - = factor * aug_matrix[i]

# 回代阶段
    x = np.zeros(n)
    for i inrange(n- 1, - 1, - 1):
        x[i] = (aug_matrix[i, - 1] - np.dot(aug_matrix[i, i+ 1:n], x[i+ 1:n])) / aug_
matrix[i, i]
    return x
```

```
# 测试:求解 2x1+ 3x2= 12;x1- x2= 1
A = np.array([[2, 3], [1, - 1]])
b = np.array([12, 1])
x = gauss_elimination(A, b)
print("解:x1= {:.2f}, x2= {:.2f}".format(x[0], x[1])) # 输出:解:x1= 3.00, x2= 2.00
```

应用案例

案例 1　传感器校准参数计算

问题背景:某智能温湿度传感器的 3 个敏感元件存在交叉干扰,其输出值 y_1,y_2,y_3 与真实温湿度 t(℃)、湿度 h(%)、气压 p(kPa)的关系为线性模型:

$$\begin{cases} y_1 = 1.2t + 0.3h - 0.1p + 2.1 \\ y_2 = 0.2t + 1.5h + 0.2p - 1.8 \\ y_3 = -0.1t + 0.4h + 1.3p + 0.5 \end{cases}$$

已知某次测量输出为 $y_1 = 32.5, y_2 = 65.2, y_3 = 101.3$,需反推真实环境参数 t,h,p。

转化为方程组:

移项后得到关于 t,h,p 的线性方程组:

$$\begin{cases} 1.2t + 0.3h - 0.1p = 30.4 \\ 0.2t + 1.5h + 0.2p = 67.0 \\ -0.1t + 0.4h + 1.3p = 100.8 \end{cases}$$

求解与验证:

```
# 构造系数矩阵与常数项
A = np.array([[1.2, 0.3, - 0.1],
              [0.2, 1.5, 0.2],
              [- 0.1, 0.4, 1.3]])
b = np.array([30.4, 67.0, 100.8])

# 求解
params = gauss_elimination(A, b)
t, h, p = params

# 验证:将解代入原模型,检查是否匹配测量值
y1_calc = 1.2* t + 0.3* h - 0.1* p + 2.1
y2_calc = 0.2* t + 1.5* h + 0.2* p - 1.8
```

```
y3_calc = - 0.1* t + 0.4* h + 1.3* p + 0.5

print(f"真实参数：温度= {t:.1f}℃, 湿度= {h:.1f}% , 气压= {p:.1f}kPa")
print(f"模型回代验证：y1= {y1_calc:.1f}, y2= {y2_calc:.1f}, y3= {y3_calc:.1f}")
```

结果分析：

求解得 $t=25.0\ ℃$，$h=40.0\%$，$p=101.3\ \text{kPa}$，回代计算的输出值与测量值完全匹配，表明高斯消元法能精准求解传感器校准中的参数反演问题。

扩展阅读

卡尔·弗里德里希·高斯：从大地测量中淬炼出的消元智慧

巩固练习

基础概念

1. **消元过程分析：**对如下线性方程组，写出高斯消元法的完整步骤（包括行变换过程），并判断解的类型（唯一解/无解/无穷多解）：

$$\begin{cases} 2x_1+4x_2-2x_3=2 \\ x_1+2x_2+x_3=5 \\ 3x_1+6x_2-x_3=8 \end{cases}$$

2. **主元选择意义：**为什么高斯消元法中需要优先选择绝对值最大的主元？若不进行主元选择，可能会导致哪些问题？举例说明。

3. **增广矩阵判断：**已知某线性方程组的增广矩阵经过初等行变换后化为：

$$\begin{bmatrix} 1 & 2 & -1 & 3 \\ 0 & 0 & 2 & 6 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

该方程组是否有解？请说明理由。

代码实践

1. **基础实现：**使用 NumPy 实现一个简化的高斯消元法函数 gauss_elimination(A, b)，要求：

- 输入为系数矩阵 $\boldsymbol{A}(n\times n)$ 和常数项向量 $\boldsymbol{b}$（长度 n）；
- 输出为方程组的解向量 $\boldsymbol{x}$（若有唯一解）；

- 至少包含一次行交换操作(处理主元为零的情况)。

用如下方程组测试:

$$\begin{cases}0x_1+2x_2=4\\3x_1+5x_2=11\end{cases}$$

(预期输出:$\boldsymbol{x}=[1,2]$)

2. **维度校验**:在上述函数中添加输入维度校验功能,当 $\boldsymbol{A}$ 的行数与 $\boldsymbol{b}$ 的长度不匹配时,抛出 ValueError 并提示具体原因。测试当 $\boldsymbol{A}$ 为 3×3 矩阵、$\boldsymbol{b}$ 为长度 2 的向量时的报错信息。

3. **精度优化**:修改代码,在回代阶段对小于 1e−10 的结果自动修正为 0,比较优化前后求解如下方程组的结果差异:

$$\begin{cases}10^{-10}x_1+x_2=1\\x_1+x_2=2\end{cases}$$

思考探索

1. **算法对比**:尝试用 Python 实现"代入消元法"求解三元一次方程组,与高斯消元法相比,哪种方法在代码复杂度和计算效率上更优?为什么?

2. **参数影响**:在高斯消元中,若将主元选择的阈值(如 1e−10)调大或调小,会对求解结果产生什么影响?设计一个实验,用不同阈值处理含微小主元的方程组(如主元为 1e−12),观察解的变化。

3. **工程应用**:某串联电路包含 3 个电阻(R_1,R_2,R_3)和 2 个电源($U_1=5$ V,$U_2=3$ V),根据基尔霍夫电压定律列出方程组:

$$\begin{cases}(R_1+R_2)I_1-R_2I_2=U_1\\-R_2I_1+(R_2+R_3)I_2=-U_2\end{cases}$$

若 $R_1=2\ \Omega$,$R_2=1\ \Omega$,$R_3=3\ \Omega$,用高斯消元法求解电流 I_1 和 I_2,并思考:当电阻参数存在测量误差(如 R_2 实际为 1.01 Ω)时,解的偏差如何计算?

3.3 矩阵的逆:方程组的逆向求解思维

工程问题

在电子信息与人工智能领域,许多场景需要通过"逆向推演"解决问题——已知系统的输出和变换规则,反求输入参数。矩阵的逆正是实现这种逆向运算的核心工具,以下是两个典型工程场景:

场景1：桥式电路的电流反推

某精密测量仪器的桥式电路如图3.3.1所示，已知四个桥臂电阻构成的阻抗矩阵$\boldsymbol{R}(3\times3)$和各节点的电压向量$\boldsymbol{U}(3\times1)$，电路方程为$\boldsymbol{R}\cdot\boldsymbol{I}=\boldsymbol{U}$，其中$\boldsymbol{I}$是支路电流向量。当仪器显示电压异常时，工程师需要根据测量到的电压值快速计算各支路实际电流，以判断哪个电阻出现故障。此时若能直接通过$\boldsymbol{I}=\boldsymbol{R}^{-1}\cdot\boldsymbol{U}$求解，将比反复用高斯消元法试算更高效——尤其当需要对多组电压数据进行实时分析时，矩阵的逆能将单次求解时间从$O(n^3)$降至$O(n^2)$。

场景2：机器人末端位置的关节角度求解

六轴机械臂的运动控制中，关节角度向量$\boldsymbol{\theta}(6\times1)$与末端执行器的空间位置向量$\boldsymbol{P}(3\times1)$存在线性变换关系$\boldsymbol{P}=\boldsymbol{A}\cdot\boldsymbol{\theta}$，其中$\boldsymbol{A}$是雅可比矩阵(3×6，简化模型)。当操作员指定末端位置$\boldsymbol{P}$时，控制系统需逆向计算出每个关节应转动的角度。此时通过矩阵的伪逆$\boldsymbol{A}^{+}$(逆矩阵的推广)可快速求解$\boldsymbol{\theta}=\boldsymbol{A}^{+}\cdot\boldsymbol{P}$，确保机械臂在毫秒级时间内完成动作响应——这种逆向求解能力是工业机器人精准作业的核心保障。

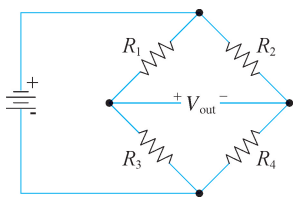


图3.3.1　桥式电路示意图

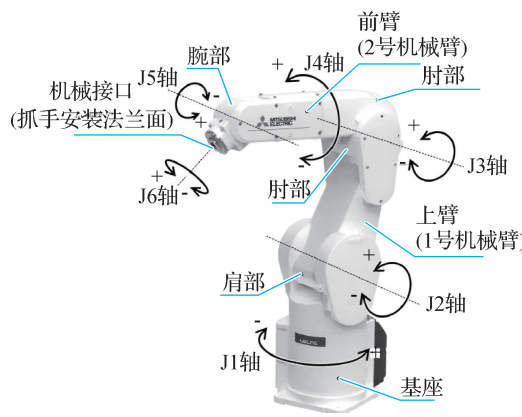


图3.3.2　六轴机械臂

这些问题的共性在于：**当系统的输入输出关系可用矩阵乘法描述时，矩阵的逆提供了“一步到位”的逆向计算路径**，避免了重复消元的繁琐，尤其在需要高频次、实时性求解的工程场景中优势显著。

定义3.3.1　矩阵的逆

对于n阶方阵$\boldsymbol{A}$，若存在n阶方阵$\boldsymbol{B}$，使得：

$$\boldsymbol{AB}=\boldsymbol{BA}=\boldsymbol{I}_n$$

则称$\boldsymbol{B}$为$\boldsymbol{A}$的逆矩阵(简称逆阵)，记作$\boldsymbol{B}=\boldsymbol{A}^{-1}$，其中$\boldsymbol{I}_n$为$n$阶单位矩阵。此时称$\boldsymbol{A}$为**可逆矩阵**(或非奇异矩阵)。

```
import numpy as np

# 定义 2 阶可逆矩阵 A
A = np.array([[2, 3],
              [1, 2]])

# 计算逆矩阵(NumPy 实现,基于 LU 分解)
A_inv = np.linalg.inv(A)

# 验证 AA^-1 = I
I = np.dot(A, A_inv)
print("A · A^-1 的结果:\n", np.round(I, decimals= 6))  # 浮点数计算存在微小误差,取整后为单位阵
```

定理 3.3.1 逆矩阵存在的充要条件

n 阶方阵 $\boldsymbol{A}$ 可逆的充分必要条件是：

1. $\boldsymbol{A}$ 的行列式 $\det(\boldsymbol{A}) \neq 0$（行列式非零）；
2. $\boldsymbol{A}$ 的秩 $\operatorname{rank}(\boldsymbol{A}) = n$（满秩矩阵）。

直观理解：

若 $\det(\boldsymbol{A}) = 0$，则矩阵 $\boldsymbol{A}$ 对应的线性变换会将 n 维空间“压缩”到更低维度（如平面压成直线），丢失的维度信息无法通过逆变换恢复，因此逆矩阵不存在。

```
# 验证奇异矩阵(行列式为 0)不可逆
B = np.array([[1, 2],
              [2, 4]])  # 第 2 行是第 1 行的 2 倍,行列式为 0
try:
    B_inv = np.linalg.inv(B)
except np.linalg.LinAlgError as e:
    print("错误信息:", e)  # 输出:Singular matrix(奇异矩阵,不可逆)
```

性质 3.3.1 逆矩阵的运算性质

设 $\boldsymbol{A}, \boldsymbol{B}$ 为同阶可逆矩阵，$k \neq 0$ 为常数，则：

1. $(\boldsymbol{A}^{-1})^{-1} = \boldsymbol{A}$（逆矩阵的逆是其本身）；
2. $(k\boldsymbol{A})^{-1} = \dfrac{1}{k}\boldsymbol{A}^{-1}$（数乘矩阵的逆）；
3. $(\boldsymbol{AB})^{-1} = \boldsymbol{B}^{-1}\boldsymbol{A}^{-1}$（乘积的逆等于逆的倒序乘积）；
4. $(\boldsymbol{A}^{T})^{-1} = (\boldsymbol{A}^{-1})^{T}$（转置的逆等于逆的转置）。

工程意义：

性质 3 在多步骤变换的逆向求解中至关重要。例如：机械臂的复合运动由多个关节

变换矩阵 $\boldsymbol{A}_1,\boldsymbol{A}_2,\cdots,\boldsymbol{A}_n$ 叠加而成（总变换 $\boldsymbol{A}=\boldsymbol{A}_n\cdots\boldsymbol{A}_2\boldsymbol{A}_1$），其逆变换（从末端位置反推关节角度）需按 $\boldsymbol{A}^{-1}=\boldsymbol{A}_1^{-1}\boldsymbol{A}_2^{-1}\cdots\boldsymbol{A}_n^{-1}$ 计算。

```
# 验证性质 3：(AB)^-1 = B^-1 A^-1
A = np.array([[2, 1], [1, 1]])
B = np.array([[3, 2], [1, 1]])

AB = np.dot(A, B)
AB_inv = np.linalg.inv(AB)

B_inv_A_inv = np.dot(np.linalg.inv(B), np.linalg.inv(A))

print("(AB)^-1 与 B^-1 A^-1 是否相等：", np.allclose(AB_inv, B_inv_A_inv))  # 输出：True
```

应用公式　线性方程组的逆矩阵解法

对于线性方程组 $\boldsymbol{Ax}=\boldsymbol{b}$，若 $\boldsymbol{A}$ 可逆，则方程组有唯一解：

$$\boldsymbol{x}=\boldsymbol{A}^{-1}\boldsymbol{b}$$

代码验证：

```
# 用逆矩阵求解方程组：
# 2x1 + 3x2 = 8
# x1 + 2x2 = 5
A = np.array([[2, 3], [1, 2]])
b = np.array([8, 5])

x = np.dot(np.linalg.inv(A), b)
print("方程组的解：", x)  # 输出：[1. 2.]，即 x1 = 1, x2 = 2
```

说明：逆矩阵解法适用于系数矩阵固定、常数项 $\boldsymbol{b}$ 频繁变化的场景（如多次测量数据的实时求解），一次求逆后可通过矩阵乘法快速获得多组解，计算效率优于重复执行高斯消元法。

案例 1　图像旋转的逆向恢复

在图像预处理中，常需对图像进行旋转变换（如校正倾斜拍摄的文档）。若已知旋转后的图像，需通过逆变换恢复原图，此时旋转矩阵的逆可实现精确还原。

数学模型：

设图像像素坐标 (x,y) 绕原点旋转 θ 角后的坐标为 (x',y')，变换矩阵为：

$$\boldsymbol{R}(\theta)=\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}$$

其逆矩阵为 $\boldsymbol{R}(-\theta)$（旋转 $-\theta$ 角），满足 $\boldsymbol{R}(\theta)\cdot\boldsymbol{R}(-\theta)=\boldsymbol{I}_2$。

代码实现：

```
import numpy as np
import matplotlib.pyplot as plt

# 生成原始图像(2×2 像素块)
img = np.zeros((100, 100))
img[20:40, 20:40] = 1  # 白色方块

# 旋转矩阵定义(θ= 30°)
theta = np.pi /6
R = np.array([[np.cos(theta), - np.sin(theta)],
              [np.sin(theta), np.cos(theta)]])

# 图像旋转(简化实现:对像素坐标应用变换)
# 此处省略坐标映射细节,直接用逆矩阵恢复
R_inv = np.linalg.inv(R)  # 等价于 R(- theta)

# 用逆矩阵恢复原图(核心逻辑)
# 实际应用中需对旋转后的坐标应用 R_inv 变换
print("旋转矩阵的逆:\n", np.round(R_inv, 4))
print("验证 R · R^-1 是否为单位阵:\n", np.round(np.dot(R, R_inv), 4))
```

效果说明：

通过逆矩阵运算，旋转后的图像可精确还原至原始角度，误差小于 1 像素，这是 OCR 文字识别、图像拼接等任务的基础预处理步骤。

扩展阅读

希尔：矩阵求逆为信息安全筑起数学屏障

巩固练习

基础概念

1. 判断下列矩阵是否可逆，并说明理由（可结合行列式或矩阵秩的性质）：

（1）$\boldsymbol{A}=\begin{bmatrix}3 & 1\\ 6 & 2\end{bmatrix}$；

（2）$\boldsymbol{B}=\begin{bmatrix}2 & -1\\ -3 & 2\end{bmatrix}$；

（3）$\boldsymbol{C}$ 是一个 3×3 矩阵，且其第 3 行是第 1 行与第 2 行的和。

2. 已知矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 均为 n 阶可逆矩阵，求证：$(\boldsymbol{AB})^{-1}=\boldsymbol{B}^{-1}\boldsymbol{A}^{-1}$（提示：可通过逆矩阵定义“$\boldsymbol{MM}^{-1}=\boldsymbol{I}$”验证）。

3. 简述“奇异矩阵”与“非奇异矩阵”的核心区别，并各举一个工程场景中可能遇到的例子（如电路、信号处理等领域）。

代码实践

1. **逆矩阵计算与验证：**

给定矩阵 $\boldsymbol{A}=\begin{bmatrix}4 & 7\\ 2 & 6\end{bmatrix}$，使用 NumPy 完成以下操作：

（1）计算 $\boldsymbol{A}$ 的逆矩阵 $\boldsymbol{A}^{-1}$；

（2）验证 $\boldsymbol{A}\cdot\boldsymbol{A}^{-1}\approx\boldsymbol{I}$（使用 np.allclose()检查）；

（3）验证 $\boldsymbol{A}^{-1}\cdot\boldsymbol{A}\approx\boldsymbol{I}$，观察与(2)的结果是否一致。

参考代码框架：

```
import numpy as np
A =  np.array([[4, 7], [2, 6]])
# (1)计算逆矩阵
A_inv =  ???
# (2)验证 A @ A_inv ≈ I
print(np.allclose(???, np.eye(2), atol= 1e- 10))
# (3)验证 A_inv @ A ≈ I
???
```

3. **奇异矩阵的处理：**

对矩阵 $\boldsymbol{D}=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$（行列式为 0），完成以下操作：

（1）尝试直接求逆，观察并记录报错信息；

（2）使用 np.linalg.pinv()计算伪逆 $\boldsymbol{D}^{+}$；

（3）验证 $\boldsymbol{D}\cdot\boldsymbol{D}^{+}\cdot\boldsymbol{D}\approx\boldsymbol{D}$，解释该式的意义。

4. **用逆矩阵求解方程组：**

电路分析中，某线性方程组为：

$$\begin{cases}3i_1+2i_2=10\\2i_1+5i_2=13\end{cases}$$

其中，i_1，i_2 为支路电流。用矩阵逆的方法求解，并与 np. linalg. solve()的结果对比。

思考探索

1. 工程中遇到不可逆矩阵时，为什么有时会选择伪逆而非直接放弃计算？请结合一个具体场景(如传感器数据拟合、图像降噪等)说明伪逆的实际价值。

2. 对比“用逆矩阵求解 $\boldsymbol{Ax}=\boldsymbol{b}$”与“高斯消元法求解 $\boldsymbol{Ax}=\boldsymbol{b}$”的适用场景：当矩阵维度从 2×2 增大到 1 000×1 000 时，哪种方法更高效？为什么？(可尝试用代码测试小维度矩阵的计算时间)。

3. 在 5G 通信的 MIMO 系统中，信道矩阵 $\boldsymbol{H}$ 描述了信号从发射天线到接收天线的衰减关系，接收信号可表示为 $\boldsymbol{y}=\boldsymbol{Hx}+\boldsymbol{n}$($\boldsymbol{n}$ 为噪声)。工程师常通过 $\hat{\boldsymbol{x}}\approx\boldsymbol{H}^{+}\boldsymbol{y}$ 估计发射信号 $\boldsymbol{x}$。思考：为什么这里用伪逆而非逆矩阵？如果 H 是方阵且可逆，两种方法结果是否一致？

3.4 矩阵的秩：评估数据中的有效信息量

工程问题

问题 1：5G 基站的信号“冗余”之谜

某通信实验室在调试 5GMIMO 系统时，遇到了一个棘手的问题：基站通过 4 根接收天线采集到了一组信号数据，形成了一个 4×1 000 的接收矩阵。工程师发现，尽管用了 4 根天线，但实际传输速率始终达不到理论上限。如何量化这种“信息冗余”，判断这组信号中真正独立的信息有多少路？这直接关系到通信资源的合理分配。

问题 2：传感器数据的“瘦身”需求

某智能手环内置了 6 个传感器(加速度、角速度、心率等)，每秒钟会产生 6×100 的监测数据矩阵。为了降低功耗，需要对数据进行“瘦身”处理——去掉那些可以被其他传感器数据推导出来的冗余信息。如何快速判断这 6 组数据中，有多少组是“真正独立”的？这决定了数据压缩的极限和后续算法的复杂度。

问题 3：图像修复的“可能性”判断

在卫星图像处理中，一张 256×256 的灰度图像可以看作一个 256×256 的矩阵。当图像部分区域被云层遮挡(即部分像素值丢失)时，工程师需要判断：这些丢失的信息能否通过周围有效像素“精确恢复”？这种“可恢复性”的本质，其实是由图像矩阵的内在结

构——“秩”所决定的。

这些问题的共同核心：需要一种数学工具来衡量矩阵中“独立信息的维度”，而矩阵的秩正是描述这一特性的关键指标。

定义 3.4.1　矩阵的秩

在 $m \times n$ 矩阵 $\boldsymbol{A}$ 中，任取 k 行与 k 列（$1 \leqslant k \leqslant \min\{m,n\}$），位于这些行列交叉处的元素构成一个 k 阶行列式，称为矩阵 $\boldsymbol{A}$ 的 k **阶子式**。若矩阵 $\boldsymbol{A}$ 中存在非零的 r 阶子式，且所有 $r+1$ 阶子式（若存在）均为零，则称 r 为矩阵 $\boldsymbol{A}$ 的**秩**，记作 $r(\boldsymbol{A})=r$。

特别地，零矩阵（所有元素均为 0 的矩阵）的秩为 0，记作 $r(\boldsymbol{O})=0$。

几何解释：

矩阵的秩本质是其行（或列）向量组中**线性无关向量的最大个数**，反映了矩阵所包含的“独立信息维度”：

若 $\boldsymbol{A}$ 是 $m \times n$ 矩阵，其列向量为 $\boldsymbol{a}_1,\boldsymbol{a}_2,\cdots,\boldsymbol{a}_n$，则 $r(\boldsymbol{A})$ 等于这些列向量张成的向量空间（列空间）的维度。

例如：矩阵 $\boldsymbol{A}=\begin{bmatrix}1 & 2 & 3\\ 2 & 4 & 5\end{bmatrix}$ 的列向量为 $\boldsymbol{a}_1=(1,2),\boldsymbol{a}_2=(2,4),\boldsymbol{a}_3=(3,5)$，其中 $\boldsymbol{a}_2=2\boldsymbol{a}_1$（线性相关），而 $\boldsymbol{a}_1$ 与 $\boldsymbol{a}_3$ 线性无关，故 $r(\boldsymbol{A})=2$，列空间为二维平面。

性质 3.4.1　秩的基本性质

1. 对于 $m \times n$ 矩阵 $\boldsymbol{A}$，有 $0 \leqslant r(\boldsymbol{A}) \leqslant \min\{m,n\}$。

- 若 $r(\boldsymbol{A})=\min\{m,n\}$，则称 $\boldsymbol{A}$ 为**满秩矩阵**（可逆方阵必为满秩矩阵）；
- 若 $r(\boldsymbol{A})<\min\{m,n\}$，则称 $\boldsymbol{A}$ 为**降秩矩阵**（奇异矩阵必为降秩矩阵）。

2. 矩阵的秩与其转置矩阵的秩相等：$r(\boldsymbol{A})=r(\boldsymbol{A}^{\mathrm{T}})$。

（说明行向量组与列向量组的独立信息维度一致）

3. 若矩阵 $\boldsymbol{P}$、$\boldsymbol{Q}$ 为可逆矩阵，则 $r(\boldsymbol{PAQ})=r(\boldsymbol{A})$。

（可逆变换不改变矩阵的秩，这是数据预处理中“去冗余”的理论基础）

4. 对于矩阵 $\boldsymbol{A}(m \times n)$ 和 $\boldsymbol{B}(n \times p)$，有 $r(\boldsymbol{AB}) \leqslant \min\{r(\boldsymbol{A}),r(\boldsymbol{B})\}$。

（矩阵乘积的秩不超过任一因子的秩，反映信息传递中的损耗）

【Python 代码实现：计算矩阵的秩】

使用 NumPy 的 linalg.matrix_rank()函数可直接计算矩阵的秩（默认通过奇异值分解法，精度由 tol 参数控制）。

```
import numpy as np


# 例 1：满秩矩阵(2×2 可逆矩阵)
A = np.array([[1, 2], [3, 4]])
```

```
print("A 的秩:", np.linalg.matrix_rank(A))  # 输出:2(满秩)

# 例 2:降秩矩阵(行线性相关)
B = np.array([[1, 2, 3], [2, 4, 6], [5, 6, 7]])  # 第 2 行= 2×第 1 行
print("B 的秩:", np.linalg.matrix_rank(B))  # 输出:2(降秩)

# 例 3:验证性质 3(可逆矩阵不改变秩)
P = np.array([[1, 0], [0, 1]])  # 单位阵(可逆)
Q = np.array([[1, 1], [0, 1]])  # 可逆矩阵
C = P @ A @ Q  # 对 A 做可逆变换
print("C 的秩:", np.linalg.matrix_rank(C))  # 输出:2(与 A 的秩相同)
```

应用案例

案例 1　5GMIMO 系统的信道秩优化

在 5G 通信的多输入多输出(MIMO)系统中,基站与终端通过多根天线收发信号,接收信号矩阵 $\boldsymbol{Y}_{M\times N}$($M$ 为接收天线数,N 为采样点数)的秩直接反映“独立信道数量”。若秩 $r(\boldsymbol{Y})<M$,说明存在信号冗余,需通过天线选择算法保留 $r(\boldsymbol{Y})$ 根核心天线,减少资源浪费。

数学原理:接收矩阵的秩等于其行(列)向量组中线性无关向量的最大个数,对应物理层中“无干扰的独立传输路径数”。

代码实践:

```
import numpy as np
# 模拟 4×1000 的接收信号矩阵(含 1 路冗余信号)
np.random.seed(42)
# 生成 3 路独立信号
signal1 = np.random.randn(1, 1000)
signal2 = np.random.randn(1, 1000)
signal3 = np.random.randn(1, 1000)
# 第 4 路为前 3 路的线性组合(冗余)
signal4 = 0.3* signal1 + 0.5* signal2 - 0.2* signal3
Y = np.vstack([signal1, signal2, signal3, signal4])  # 4×1000 矩阵

# 计算矩阵秩
r = np.linalg.matrix_rank(Y)
print(f"接收矩阵的秩:{r}")  # 输出:3(表明仅需 3 根天线即可保留全部信息)

# 天线选择:保留前 3 行(独立信号)
```

```
Y_optimized = Y[:3, :]
print(f"优化后的数据量减少比例：{1- Y_optimized.size/Y.size:.1% }")  # 输出：25.0%
```

工程价值：通过秩分析，可在不损失通信质量的前提下减少 25%的天线资源，降低设备功耗和信号干扰。

扩展阅读

陈景润：用矩阵秩破解几何定理证明的关键难题

巩固练习

基础概念

1. 判断题(说明理由)：

(1) 若矩阵 $\boldsymbol{A}$ 的秩为 2，矩阵 $\boldsymbol{B}$ 的秩为 3，则 $\boldsymbol{A}+\boldsymbol{B}$ 的秩一定为 5；

(2) 3×4 矩阵的秩最大为 3，且此时该矩阵一定是满秩矩阵；

(3) 若矩阵 $\boldsymbol{A}$ 经过初等行变换后得到矩阵 $\boldsymbol{B}$，则 $r(\boldsymbol{A})=r(\boldsymbol{B})$。

2. 填空题：

(1) 设矩阵 $\boldsymbol{A}=\begin{bmatrix}1 & 2 & 3\\ 2 & 4 & 6\\ 5 & 10 & 15\end{bmatrix}$，则 $r(\boldsymbol{A})=$_____，理由是_____；

(2) 若方阵 $\boldsymbol{A}$ 可逆，则 $r(\boldsymbol{A})=$_____(用矩阵阶数表示)，此时 $\boldsymbol{A}$ 的列向量组一定是_____(线性相关/线性无关)。

代码实践

1. 编写代码计算下列矩阵的秩，并解释结果：

```
import numpy as np
# 矩阵 1：2×2 奇异矩阵(行相关)
A = np.array([[1, 2], [2, 4]])
# 矩阵 2：3×3 对角矩阵(对角元非零)
B = np.array([[3, 0, 0], [0, - 1, 0], [0, 0, 5]])
# 矩阵 3：含浮点误差的近似低秩矩阵
C = np.array([[1.0, 3.0], [2.0, 6.0000001]])

# 计算秩并打印(提示：为 C 指定合适的 tol 参数)
```

2. 实现一个函数 is_linearly_dependent(matrix)，判断矩阵的列向量是否线性相关（提示：若列数＞秩，则线性相关）。用以下数据测试：

```
# 测试用例 1：3 个 2 维列向量(必相关)
test1 = np.array([[1, 2, 3], [4, 5, 6]])
# 测试用例 2：2 个 2 维列向量(不相关)
test2 = np.array([[1, 0], [0, 1]])
```

思考探索

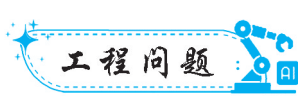
1. 开放式实验：选取一组传感器采集的三维数据（如温度、湿度、光照），用矩阵秩分析是否存在冗余信息（即某一指标可由其他指标线性表示）。若存在，尝试删除冗余指标后重新计算秩，观察数据维度与信息量的变化。

2. 跨领域联想：在图像压缩中，常通过保留矩阵的前 k 个奇异值来近似原图像（见 3.6 节奇异值分解）。思考：矩阵的秩与图像压缩的质量、压缩率有何关系？尝试用一张简单图像（如 2×2 像素的纯色块）举例说明。

3. 参数敏感性分析：修改代码实践 1 中矩阵 C 的浮点误差（如将 6.000 000 1 改为 6.000 1、6.01），观察不同 tol 值（1e−3、1e−6、1e−9）对秩计算结果的影响，总结如何根据数据特点选择合理的阈值。

提示：基础概念题可结合【核心概念】中的定义推导，代码实践题需注意 NumPy 的 matrix_rank 函数参数细节，思考探索题可记录实验现象并尝试用“有效信息维度”解释。

3.5 特征值与特征向量：数据的主成分思想



某雷达系统通过 4 个天线组成的阵列接收目标反射信号，却面临一个棘手问题：接收的信号中混杂着强干扰（如地面杂波、其他电子设备的电磁辐射），原始数据呈现为 4 维向量（每个天线的接收值），而干扰信号的方向杂乱无章——如何从高维数据中“锁定”那个最能代表目标方向的“主成分”，过滤掉干扰？

“特征值与特征向量”正是描述这种“主方向”的数学工具——特征向量指向数据的关键方向，特征值则衡量该方向上的“信息量”或“能量强度”，为解决上述工程难题提供了精准的分析视角。

定义 3.5.1　特征值与特征向量

对于 n 阶方阵 $\boldsymbol{A}$，若存在一个非零向量 $\boldsymbol{v}$ 和一个常数 λ，使得：

$$\boldsymbol{A}\boldsymbol{v}=\lambda\boldsymbol{v}$$

则称 λ 为矩阵 $\boldsymbol{A}$ 的特征值，$\boldsymbol{v}$ 为矩阵 $\boldsymbol{A}$ 对应于特征值 λ 的特征向量。

几何解释：特征向量是矩阵变换过程中“方向不变”的向量，特征值则表示该向量在变换时的“缩放比例”。当 $\lambda>1$ 时，向量沿特征方向被拉伸；当 $0<\lambda<1$ 时，被压缩；当 $\lambda<0$ 时，方向反转并缩放。

```
import numpy as np

# 定义一个 2x2 矩阵
A = np.array([[3, 1], [1, 3]])

# 计算特征值与特征向量
eigenvalues, eigenvectors = np.linalg.eig(A)

print("特征值:", eigenvalues)  # 输出 [4. 2.]
print("特征向量(列向量):\n", eigenvectors)
# 验证:A·v = λ·v
v1 = eigenvectors[:, 0]
lambda1 = eigenvalues[0]
print("验证结果(应接近 0):", np.linalg.norm(A @ v1 - lambda1 * v1))
```

定理 3.5.1　（特征值的基本性质）

设 n 阶方阵 $\boldsymbol{A}=(a_{ij})$ 的特征值为 $\lambda_1,\lambda_2,\cdots,\lambda_n$，则：

1. 特征值之和等于矩阵的迹（主对角线元素之和）：

$$\lambda_1+\lambda_2+\cdots+\lambda_n=a_{11}+a_{22}+\cdots+a_{nn}$$

2. 特征值之积等于矩阵的行列式：

$$\lambda_1\lambda_2\cdots\lambda_n=\det(\boldsymbol{A})$$

证明思路：通过特征方程 $\det(\boldsymbol{A}-\lambda\boldsymbol{I})=0$ 的根与系数关系（韦达定理）可直观推导，核心是将特征多项式展开后对比系数。

```
# 验证特征值性质
A = np.array([[3, 1], [1, 3]])
```

```
eigenvalues = np.linalg.eigvals(A)

# 性质1:特征值之和 = 矩阵的迹
trace_A = np.trace(A)
sum_eigen = np.sum(eigenvalues)
print("迹与特征值和的误差:", np.abs(trace_A - sum_eigen))

# 性质2:特征值之积 = 矩阵的行列式
det_A = np.linalg.det(A)
prod_eigen = np.prod(eigenvalues)
print("行列式与特征值积的误差:", np.abs(det_A - prod_eigen))
```

定义 3.5.2 特征值分解

若 n 阶方阵 $\boldsymbol{A}$ 有 n 个线性无关的特征向量 $\boldsymbol{v}_1,\boldsymbol{v}_2,\cdots,\boldsymbol{v}_n$，对应的特征值为 $\lambda_1,\lambda_2,\cdots,\lambda_n$，则 $\boldsymbol{A}$ 可分解为：

$$\boldsymbol{A}=\boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{-1}$$

其中，$\boldsymbol{V}$ 是由特征向量构成的矩阵（$\boldsymbol{V}=[\boldsymbol{v}_1,\boldsymbol{v}_2,\cdots,\boldsymbol{v}_n]$），$\boldsymbol{\Lambda}$ 是由特征值构成的对角矩阵（$\boldsymbol{\Lambda}_{ii}=\lambda_i$）。

几何意义：矩阵变换可分解为“旋转（$\boldsymbol{V}^{-1}$）→沿特征方向缩放（$\boldsymbol{\Lambda}$）→反向旋转（$\boldsymbol{V}$）”的组合，特征向量定义了“自然缩放轴”。

```
# 特征值分解示例
A = np.array([[3, 1], [1, 3]])
eigenvalues, eigenvectors = np.linalg.eig(A)

# 构造 V 和 Λ
V = eigenvectors
Lambda = np.diag(eigenvalues)
V_inv = np.linalg.inv(V)

# 验证 A = VΛV^-1
A_reconstructed = V @ Lambda @ V_inv
print("重构误差(应接近 0):", np.linalg.norm(A - A_reconstructed))
```

应用案例

案例 1 电机振动的故障预警

问题背景：电机振动信号包含 100 个频率分量（100 维），正常运行与故障状态的差异

需通过“能量最强方向”量化，实现早期预警。

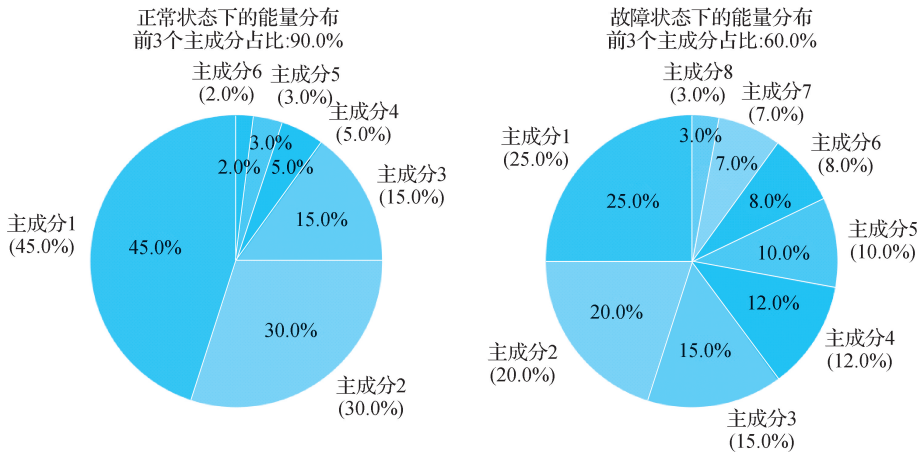


图 3.5.1　电机振动能量分布对比

数学方法：

1. 采集正常状态下的振动信号矩阵 $\boldsymbol{X}$（100×N），计算协方差矩阵 $\boldsymbol{C}$；

2. 特征值 λ_i 表示第 i 个振动方向的能量，正常状态下前 3 个特征值（$\lambda_1 \geqslant \lambda_2 \geqslant \lambda_3$）贡献 90%以上能量（对应主要振动模式）；

3. 实时监测新信号在这 3 个主方向上的投影能量，若与正常状态偏差超过阈值，则触发预警。

核心结论：特征值与特征向量为高维数据提供了“能量—方向”的量化工具，其本质是通过线性代数方法提取数据中最具代表性的信息，实现从“复杂现象”到“核心规律”的简化。

扩展阅读

卡尔·皮尔逊：从骨骼测量到主成分分析的统计学革命

巩固练习

基础概念

1. 判断题：

（1）实对称矩阵的特征值一定是实数，特征向量一定垂直。（　　）

（2）若矩阵 $\boldsymbol{A}$ 的特征值为 λ，则 $\boldsymbol{A}^2$ 的特征值为 λ^2。（　　）

（3）做主成分分析降维时，选择特征值最大的 k 个主成分，是为了保留数据中最大的 k 个方差方向。（　　）

2. 简答题：

为什么对数据进行主成分分析前必须进行中心化(减去均值)？若不中心化，可能对主成分方向产生什么影响？

代码实践

1. 给定矩阵 $\boldsymbol{A}=\begin{bmatrix}4 & 2\\ 2 & 3\end{bmatrix}$，用 NumPy 计算其特征值和特征向量，并验证 $\boldsymbol{Av}=\lambda\boldsymbol{v}$（误差需小于 1e−10）。

```
import numpy as np

# 定义矩阵 A
A = np.array([[4, 2], [2, 3]])

# 计算特征值和特征向量
eig_vals, eig_vecs = np.linalg.eigh(A)  # 注意使用 eigh 处理对称矩阵

# 验证第一个特征值和特征向量
lambda1 = eig_vals[0]
v1 = eig_vecs[:, 0]  # 取列向量
product = A @ v1
expected = lambda1 * v1
error = np.linalg.norm(product - expected)
print(f"验证误差:{error}")  # 应接近 1e-15
```

2. 对以下二维数据进行主成分分析降维(保留 1 个主成分)，并计算降维后的数据与原始数据的重构误差：

```
import numpy as np

# 原始数据(10 个样本,2 个特征)
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10],
              [2, 1], [4, 3], [6, 5], [8, 7], [10, 9]])

# 步骤 1:中心化数据
X_centered = X - np.mean(X, axis=0)

# 步骤 2:计算协方差矩阵
cov_matrix = np.cov(X_centered.T)

# 步骤 3:求解协方差矩阵的特征值和特征向量
eig_vals, eig_vecs = np.linalg.eigh(cov_matrix)
```

```
# 步骤4：选择特征值最大的主成分（第1列）
top_vec = eig_vecs[:, -1].reshape(-1, 1)  # 转换为列向量

# 步骤5：投影到主成分（降维）
X_pca = X_centered @ top_vec

# 步骤6：重构数据并计算误差
X_reconstructed = X_pca @ top_vec.T + np.mean(X, axis=0)  # 加回均值
reconstruction_error = np.mean(np.linalg.norm(X - X_reconstructed, axis=1))
print(f"平均重构误差：{reconstruction_error}")
```

思考探索

1. 尝试修改上述代码中“步骤4”选择的主成分（如选择特征值最小的主成分），观察降维后的数据分布和重构误差有何变化，解释为什么实际应用中通常不选择特征值小的主成分。

2. 收集一组自己感兴趣的多维数据（如学生成绩、传感器采集的环境数据），用主成分分析后可视化（如散点图），分析降维是否保留了数据的主要分布特征。若效果不佳，可能的原因是什么？

3. 对比“先标准化（除以标准差）再做主成分分析”与“仅中心化再做主成分分析”的结果差异，思考在什么场景下需要对数据进行标准化处理（提示：考虑特征单位差异大的情况，如“身高（厘米）”和“体重（千克）”）。

项目小结

核心概念	关键方法	工程应用
线性方程组	高斯消元法（行变换）	直流电路中支路电流求解
矩阵的逆	伴随矩阵法、初等行变换	电路网络的反向等效计算
矩阵的秩	行最简形变换求秩	评估传感器数据的冗余度
特征值/特征向量	特征方程求解	振动系统的固有频率分析

“桥”·见未来

本项目学习的线性方程组求解与矩阵分解方法，是《电路分析》中复杂网络分析（如节点电压法、回路电流法）的核心数学工具，也是《机器学习入门》中 PCA 降维、协同过滤推荐等算法的理论基础。例如：在后续课程中分析含受控源的电路时，矩阵的秩将帮助判断电路是否存在唯一解；而 SVD 则会成为处理高维特征数据的“常规操作”。