

第 01 章 绪论与应用背景

1.1 课程内容

课程的八章内容按依赖关系排列：第 2、3 章给出线性代数与矩阵计算基础；第 4、5 章分别讨论线性方程组的直接法和迭代法；第 6 章讨论最小二乘；第 7 章以 Krylov 子空间统一解释大规模迭代过程；第 8 章讨论特征值问题。

学习时，可以依次从三个层次理解这些内容。

1. **代数层**：矩阵分解、特征值、奇异值和投影。
2. **算法层**：有限精度下如何稳定地得到近似解。
3. **实现层**：如何在存储层次、并行通信和异构硬件上高效实现算法。

这些章节中的方法经常一起使用。例如，最小二乘可用 QR 或 SVD 求解；Krylov 方法既可求线性方程组，也可求特征值和矩阵函数作用；预条件则把矩阵分解与迭代方法连接起来。

 Tip

本页要点： 先辨认课程中的几类问题及其相互联系。

1.2 本章主要内容

本章先介绍线性空间、线性变换和矩阵问题，再用线性回归、聚类、PCA 和 PageRank 说明数据与图如何产生这些问题。随后讨论计算中的四类约束：浮点精度、稀疏结构、数据移动和并行效率，最后介绍可靠数值软件。

对任何数值结果，都应区分以下三类问题。

- **问题条件数**：输入发生微小扰动时，精确解会变化多少。
- **算法稳定性**：计算结果是否等于某个邻近问题的精确解，或其误差是否可控。
- **实现性能**：在给定硬件上，算术、访存和通信分别消耗多少时间。

条件性是问题本身的属性，稳定性是算法的属性，性能则同时取决于算法、实现、数据和硬件。三者不能相互替代。

 Tip

本页要点： 从应用中认出矩阵问题，再分别判断问题的条件性、算法的稳定性和实现的性能。

第1.1节 基础知识

1.4 Numerical Linear Algebra

数值线性代数研究线性代数问题的可靠、高效计算。这里的“数值”至少包含两层含义：**输入和运算都以有限精度表示**；问题规模通常**大到不能只依靠符号推导或显式公式**。

典型应用包括：ODE/PDE/积分方程离散后的求解，利用 SVD 的主成分分析，利用离散余弦变换的图像压缩，利用低秩分解的推荐系统，利用主特征向量的 PageRank，以及深度学习中的矩阵乘法和优化。

这些应用反复使用几类核心计算：矩阵乘法、三角求解、正交化、矩阵分解和谱计算。一个基础核的稳定性和并行效率，会影响许多调用它的上层应用。

Tip

本页要点： 数值线性代数研究线性代数问题的可靠、高效计算。

1.5 Classical Problems in NLA

设 $A \in \mathbb{C}^{m \times n}$ 、 $\mathbf{b} \in \mathbb{C}^m$ 。第一类问题是线性方程组

$$A\mathbf{x} = \mathbf{b}. \quad (1.1)$$

若 $m = n$ 且 A 非奇异，则解唯一；若 A 奇异或非方阵，则可能无解或有多解。计算后应检查残量

$$\mathbf{r} = \mathbf{b} - A\hat{\mathbf{x}}, \quad (1.2)$$

但小残量并不自动意味着前向误差 $\|\hat{\mathbf{x}} - \mathbf{x}\|$ 小，二者之间还受条件数控制。

第二类问题是最小二乘：

$$\min_{\mathbf{x} \in \mathbb{C}^n} \|A\mathbf{x} - \mathbf{b}\|_2. \quad (1.3)$$

它不要求残量为零，而是在所有候选向量中使残量的二范数最小。超定系统 $m > n$ 中，这通常表示对含噪观测进行拟合。

第三类问题是方阵 $A \in \mathbb{C}^{n \times n}$ 的特征值问题：

$$A\mathbf{x} = \lambda\mathbf{x}, \quad \mathbf{x} \neq \mathbf{0}. \quad (1.4)$$

其中 λ 是特征值， $\mathbf{x}$ 是右特征向量。等式说明线性变换在方向 $\mathbf{x}$ 上只进行倍率为 λ 的伸缩与旋转。

Tip

本页要点： 三类经典问题分别是线性方程组、最小二乘和特征值问题。

1.6 Other Problems in NLA

对 $A \in \mathbb{C}^{m \times n}$ ，奇异三元组 $(\sigma_i, \mathbf{u}_i, \mathbf{v}_i)$ 满足

$$A\mathbf{v}_i = \sigma_i\mathbf{u}_i, \quad A^H\mathbf{u}_i = \sigma_i\mathbf{v}_i, \quad \sigma_i \geq 0, \quad (1.5)$$

且 $\|\mathbf{u}_i\|_2 = \|\mathbf{v}_i\|_2 = 1$ 。把第一式左乘 A^H 得

$$A^H A \mathbf{v}_i = \sigma_i^2 \mathbf{v}_i, \quad (1.6)$$

因此右奇异向量是 $A^H A$ 的特征向量，奇异值平方是其特征值。数值上通常直接计算 SVD，而不显式形成 $A^H A$ ，以免条件数平方。

秩不超过 k 的低秩近似问题写为

$$\min_{\text{rank}(B) \leq k} \|A - B\|. \quad (1.7)$$

低秩矩阵只需存储两个瘦矩阵，例如 $B = XY^H$ ，其中 $X \in \mathbb{C}^{m \times k}$ 、 $Y \in \mathbb{C}^{n \times k}$ ，存储量由 mn 降至约 $k(m+n)$ 。

矩阵函数作用问题是

$$\mathbf{y} = f(A)\mathbf{b}. \quad (1.8)$$

目标通常不是显式形成稠密矩阵 $f(A)$ 。若 $A = V\Lambda V^{-1}$ 可对角化，则形式上

$$f(A)\mathbf{b} = Vf(\Lambda)V^{-1}\mathbf{b}, \quad (1.9)$$

但大规模问题更常用多项式、Krylov 子空间或有理逼近直接计算作用结果。矩阵补全、稀疏表示、谱图学习和核方法都可由这些基本问题与分解组合而成。

Tip

本页要点： SVD 统一处理长方矩阵、低秩近似和矩阵函数等问题。

第1.2节 典型应用问题

1.8 Linear Regression

分类（classification）预测离散类别，例如“正常/异常”；回归（regression）预测数值大小，例如温度或压力，这些量通常按连续变量建模，可以在一定范围内取任意实数值。比如，预测明天的气温是多少属于回归，判断明天是否为高温天气属于分类。即使用 0、1 表示类别，预测目标仍是类别，不会因此变成回归。

给定样本

$$\{(\mathbf{x}_i, y_i)\}_{i=1}^n, \quad \mathbf{x}_i \in \mathbb{R}^d, \quad y_i \in \mathbb{R}, \quad (1.10)$$

线性模型假设

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_d x_{id} + \varepsilon_i, \quad (1.11)$$

其中 β_0 是截距， β_j 是第 j 个特征的系数， ε_i 汇集测量噪声和模型未解释部分。把参数与观测写成

$$\boldsymbol{\beta} = [\beta_0 \quad \beta_1 \quad \cdots \quad \beta_d]^T, \quad \mathbf{y} = [y_1 \quad \cdots \quad y_n]^T, \quad (1.12)$$

并定义设计矩阵

$$X = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1d} \\ 1 & x_{21} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nd} \end{bmatrix} \in \mathbb{R}^{n \times (d+1)}, \quad (1.13)$$

便得到 $\mathbf{y} = X\boldsymbol{\beta} + \boldsymbol{\varepsilon}$ 。最小二乘估计为

$$\hat{\boldsymbol{\beta}} = \arg \min_{\boldsymbol{\beta} \in \mathbb{R}^{d+1}} \|\mathbf{y} - X\boldsymbol{\beta}\|_2^2. \quad (1.14)$$

令目标函数 $\phi(\boldsymbol{\beta}) = \|\mathbf{y} - X\boldsymbol{\beta}\|_2^2$ 。展开得

$$\phi(\boldsymbol{\beta}) = \mathbf{y}^T \mathbf{y} - 2\boldsymbol{\beta}^T X^T \mathbf{y} + \boldsymbol{\beta}^T X^T X \boldsymbol{\beta}. \quad (1.15)$$

求梯度并令其为零：

$$\nabla \phi = -2X^T \mathbf{y} + 2X^T X \boldsymbol{\beta} = \mathbf{0}, \quad (1.16)$$

于是得到正规方程

$$X^T X \boldsymbol{\beta} = X^T \mathbf{y}. \quad (1.17)$$

当 X 列满秩时 $X^T X$ 正定，解唯一。

但是， $\kappa_2(X^T X) = \kappa_2(X)^2$ ，显式形成正规方程会放大病态性；实际求解通常优先用 QR，近秩亏时用 SVD。

💡 Tip

本页要点： 线性回归的关键是把模型拟合写成矩阵最小二乘。

1.9 Clustering and Eigen Decomposition

聚类是在没有预先标签时，把样本分为内部相似、彼此分离的若干组。 k -means 的典型目标是

$$\min_{\{C_\ell\}, \{\boldsymbol{\mu}_\ell\}} \sum_{\ell=1}^k \sum_{\mathbf{x}_i \in C_\ell} \|\mathbf{x}_i - \boldsymbol{\mu}_\ell\|_2^2, \quad (1.18)$$

其中 C_ℓ 是第 ℓ 个簇， $\boldsymbol{\mu}_\ell$ 是该簇质心。固定质心时按最近距离分配样本；固定分组时，令导数为零可得质心是簇内样本均值。这解释了交替迭代的两步，但该目标非凸，结果依赖初值。

谱聚类把每个样本看作图上的一个顶点，用边权表示两个样本有多相似。设样本为 $\mathbf{x}_1, \dots, \mathbf{x}_n$ ，一种常见的构造是：当 $i \neq j$ 时，取 $w_{ij} = \exp(-\|\mathbf{x}_i - \mathbf{x}_j\|_2^2 / (2\sigma^2))$ ，并令 $w_{ii} = 0$ 。这里 $\sigma > 0$ 是距离尺度：两个样本的距离远小于 σ 时，权重接近 1；距离远大于 σ 时，权重接近 0。于是，相似的样本之间连接强，不相似的样本之间连接弱。这个定义给出对称、非负的相似度矩阵 $W = (w_{ij})$ 。样本很多时，也可以只保留近邻之间的边，并同时保留反向边，使 $w_{ij} = w_{ji}$ ，从而得到稀疏矩阵。

定义度矩阵

$$D = \text{diag}(d_1, \dots, d_n), \quad d_i = \sum_j w_{ij}, \quad (1.19)$$

其中 d_i 是与第 i 个顶点相连的边权之和，衡量该样本与其它样本的总连接强度。再定义图拉普拉斯矩阵

$$L = D - W. \quad (1.20)$$

把向量 $\mathbf{z} = (z_1, \dots, z_n)^T$ 理解为给每个顶点赋一个数值， z_i 就是第 i 个顶点上的值。利用 $w_{ij} = w_{ji}$ 和度的定义，可得

$$\begin{aligned} \mathbf{z}^T L \mathbf{z} &= \sum_i d_i z_i^2 - \sum_{i,j} w_{ij} z_i z_j \\ &= \frac{1}{2} \sum_{i,j} w_{ij} (z_i^2 + z_j^2 - 2z_i z_j) \\ &= \frac{1}{2} \sum_{i,j} w_{ij} (z_i - z_j)^2 \geq 0. \end{aligned} \quad (1.21)$$

式 (1.21) 把每条边两端的数值差平方，再按边权加起来，因此可以衡量 $\mathbf{z}$ 在图上的变化大小。它也说明实对称矩阵 L 半正定。若 $L\mathbf{z} = \lambda\mathbf{z}$ ，并将特征向量归一化为 $\|\mathbf{z}\|_2 = 1$ ，则 $\mathbf{z}^T L \mathbf{z} = \lambda$ 。因此，特征值越小，对应特征向量的加权变化总量越小。特别是在权重较大的边上， z_i 与 z_j 必须比较接近，否则这一条边就会产生较大的代价。这就是“小特征值对应图上变化缓慢的方向”的含义；这里的“方向”指特征向量，“变化缓慢”指强连接顶点之间的取值接近。

例如，假设样本形成两组，组内连接强，组间只有少量弱连接。一个向量若在各组内部近似取常数、两组之间取不同的值，那么组内边几乎不产生代价，差异主要落在权重很小的组间边上，总能量仍可以很小。小特征值对应的特征向量因而有助于识别这类分组。还要注意，常数向量 $\mathbf{1} = (1, \dots, 1)^T$ 总满足 $L\mathbf{1} = \mathbf{0}$ ，因为所有顶点取值相同，能量为零；它本身不能区分样本，分组信息要从其它特征向量中取得。

具体计算时，若要分成 k 组，可取 L 的 k 个最小特征值对应的标准正交特征向量 $\mathbf{u}_1, \dots, \mathbf{u}_k$ ，组成矩阵 $U = [\mathbf{u}_1, \dots, \mathbf{u}_k] \in \mathbb{R}^{n \times k}$ 。第 i 个样本的新坐标是 $\mathbf{y}_i = (u_1(i), \dots, u_k(i))^T$ ，其中 $u_j(i)$ 表示 $\mathbf{u}_j$ 的第 i 个分量，也就是取 U 的第 i 行作为该样本的特征。然后对 $\mathbf{y}_1, \dots, \mathbf{y}_n$ 做 k -means，得到原样本的分组。这就是“特征向量作为嵌入坐标”：用样本在几个特征向量中的取值，替代原来的坐标进行聚类。

图割希望把顶点分组，使组间被切断的边权之和较小。为了同时考虑簇大小，一种目标将每组连向其它组的边权之和除以该组样本数，再对各组求和。分组标签原本只能取离散值；将它们放宽为实数，并保留相应的归一化、正交约束，就得到上述最小特征值对应的特征向量问题。最后的 k -means 再把连续坐标变为离散分组。因此，这里求解的是这一图割模型的连续松弛，并不保证给出原图割问题的精确最优解。

两条历史线索

k -means 没有唯一的“发明年份”：平方距离划分、迭代算法和名称是在不同工作中逐渐形成的。

- 1956 年，Hugo Steinhaus 研究空间中物体的划分问题，是 k -means 的早期数学源头之一。
- 1957 年，Stuart Lloyd 在 Bell Labs 的信号量化研究中提出后来称为 Lloyd 算法的交替迭代思想；这项早期工作直到 1982 年才以 *Least Squares Quantization in PCM* 正式发表于期刊。今天常见的“按最近中心分配，再更新均值”与这条量化研究路线密切相关。
- 1965 年，Edward Forgy 的类似方法以摘要形式见于 *Biometrics*。MacQueen 后来的论文也说明，Forgy 与 Robert Jennrich 曾独立注意到这种方法；这不是某一个人一次性完成的发明。
- 1967 年，James MacQueen 在 Berkeley 统计与概率讨论会论文集中系统介绍 k -means，采用逐点加入最近

的簇并立即更新该簇均值的顺序算法。这个年份不宜称为名称的严格“首次出现”：该论文第 297 页的参考文献已经列出他本人 1965 年题名含有“k-means”的摘要。

相关出处：[Lloyd \(1982\)](#)；[MacQueen \(1967\)](#)，第 281–283、297 页；历史综述可读 [Jain \(2010\)](#)，*Data Clustering: 50 Years Beyond K-means*。

谱聚类也经历了多项工作的发展，以下几个年份对应不同阶段。

- 1973 年，Miroslav Fiedler 用图拉普拉斯的第二小特征值定义“代数连通度”，建立图的连通性与谱之间的重要联系；对应特征向量后来被称为 Fiedler 向量。他 1975 年关于特征向量性质及其图论应用的工作进一步发展了这条路线。因此，1973 年适合作为数学基础的一个节点，不宜把它说成完整现代谱聚类算法的诞生年。
- 1997/2000 年，Jianbo Shi 与 Jitendra Malik 将 normalized cuts 用于图像分割：会议论文发表于 CVPR 1997，期刊版发表于 2000 年。这一模型按顶点的总连接权重衡量簇的规模，把图划分问题与广义特征值问题联系起来。
- 2001/2002 年，Andrew Ng、Michael Jordan 与 Yair Weiss 提出并分析了一种影响广泛的归一化谱聚类方法。它属于 NIPS 2001 的会议工作，常按论文集出版年记为 2002 年；这是现代常用算法的一种，而不是谱聚类唯一的形式。

前面的推导使用未归一化的 $L = D - W$ 。Ng - Jordan - Weiss 方法则先构造 $D^{-1/2}WD^{-1/2}$ ，取其最大的 k 个特征值对应的标准正交特征向量。这等价于取归一化拉普拉斯 $L_{\text{sym}} = I - D^{-1/2}WD^{-1/2}$ 的最小的 k 个特征值对应的标准正交特征向量。随后，把特征向量矩阵的**每一行归一化为单位长度**，再对这些行做 k -means。计算要求 $d_i > 0$ ，孤立点应另行处理；行归一化前还要检查是否存在零行，例如图的连通分量数超过 k 时就可能出现零行。“求特征向量后做 k -means”只是概括，具体算法还取决于是否采用矩阵归一化和行归一化。

相关出处：[Fiedler \(1973\)](#)、[Fiedler \(1975\)](#)；[Shi - Malik 的研究组页面（列出 1997、2000 两个版本）](#)；[Ng - Jordan - Weiss, On Spectral Clustering: Analysis and an Algorithm](#)。

Tip

本页要点： 谱聚类用小特征值对应的特征向量构造新坐标，在这些坐标中识别图上强连接的样本群。

1.10 A More Involved Example

比利时与荷兰的边界地区存在飞地和飞地中的飞地：属于一个国家的区域可能被另一个国家的区域包围，内部还可能再次出现另一国的区域。如果仅以二维地理坐标 $\mathbf{x} = (x_1, x_2)^T$ 为输入，判断这一区域内一个地点属于哪个国家，就得到一个二分类问题。可以用 $y = +1$ 表示比利时，用 $y = -1$ 表示荷兰。

一个线性分类器按坐标的仿射函数正负给出预测：

$$\hat{y}(\mathbf{x}) = \begin{cases} +1, & \mathbf{w}^T \mathbf{x} + b \geq 0, \\ -1, & \mathbf{w}^T \mathbf{x} + b < 0. \end{cases} \quad (1.22)$$

其中 $\mathbf{w} \neq \mathbf{0}$ 决定分界线的法向方向， b 控制分界线的位置。它的分类边界是直线 $\mathbf{w}_1 x_1 + \mathbf{w}_2 x_2 + b = 0$ ，两类预测区域分别位于直线的两侧。因此，无论怎样调整 $\mathbf{w}$ 和 b ，单条直线都无法准确表示一类区域包围另一类区域、内部又嵌套其它区域的结构。

这个例子说明，有些分类问题的边界具有**很强的非线性**，即使输入只有两个坐标，线性分类器的表达能力也可能不够。要刻画这样的分区，需要允许弯曲、闭合或包含多个分离部分的分类边界。也可以先对坐标作非线性特征变换，再在新特征空间中进行线性分类；这种方法在原始坐标空间中的边界仍然可以是非线性的。

Tip

本页要点： 飞地和嵌套飞地无法用单条直线分开，分类方法必须能够表达相应的非线性边界。

1.11 Principal Component Analysis

设中心化数据矩阵 $X_c \in \mathbb{R}^{n \times p}$ ，其中 $n \geq 2$ ，每行是一个样本，每列是一个特征，并且每列均值为零。取单位方向 $\mathbf{v}$ ，样本在该方向上的得分为 $X_c \mathbf{v}$ ，总投影能量为

$$\|X_c \mathbf{v}\|_2^2 = \mathbf{v}^T X_c^T X_c \mathbf{v}. \quad (1.23)$$

第一主方向满足

$$\mathbf{v}_1 = \arg \max_{\|\mathbf{v}\|_2=1} \mathbf{v}^T X_c^T X_c \mathbf{v}. \quad (1.24)$$

使用拉格朗日函数

$$\mathcal{L}(\mathbf{v}, \lambda) = \mathbf{v}^T X_c^T X_c \mathbf{v} - \lambda(\mathbf{v}^T \mathbf{v} - 1), \quad (1.25)$$

对 $\mathbf{v}$ 求导得到

$$X_c^T X_c \mathbf{v} = \lambda \mathbf{v}. \quad (1.26)$$

因此最大方差方向是协方差矩阵

$$C = \frac{1}{n-1} X_c^T X_c \quad (1.27)$$

的最大特征值对应特征向量。取前 k 个正交方向组成 V_k ，可统一写成

$$V_k = \arg \max_{V^T V = I_k} \|X_c V\|_F^2. \quad (1.28)$$

直接对同一份中心化数据计算 SVD，也能得到这些主方向。这里采用薄 SVD：记 $q = \min(n, p)$ ， $U \in \mathbb{R}^{n \times q}$ 和 $V \in \mathbb{R}^{p \times q}$ 的列均标准正交， $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_q)$ ，并按 $\sigma_1 \geq \dots \geq \sigma_q \geq 0$ 排列。代入协方差矩阵的定义，得到

$$\begin{aligned} X_c &= U \Sigma V^T, \\ C &= \frac{1}{n-1} (U \Sigma V^T)^T (U \Sigma V^T) \\ &= \frac{1}{n-1} V \Sigma U^T U \Sigma V^T = \frac{1}{n-1} V \Sigma^2 V^T. \end{aligned} \quad (1.29)$$

因此 $C \mathbf{v}_i = (\sigma_i^2 / (n-1)) \mathbf{v}_i$ ：SVD 的第 i 个右奇异向量 $\mathbf{v}_i$ 就是协方差矩阵的特征向量，对应的主成分方差为 $\lambda_i(C) = \sigma_i^2 / (n-1)$ 。注意，式 (1.26) 中的 λ 是 $X_c^T X_c$ 的特征值，还没有除以 $n-1$ 。若 $p > q$ ， C 还具有薄 SVD 未显式列出的 $p-q$ 个零特征值；零特征值对应的方向没有数据方差。

保留前 $k \leq q$ 个主成分时，利用 $V^T V_k$ 选出前 k 列，得分矩阵为

$$T = X_c V_k = U_k \Sigma_k. \quad (1.30)$$

秩 k 重构为 $\hat{X}_c = X_c V_k V_k^T$ 。由于

$$\|X_c - \hat{X}_c\|_F^2 = \|X_c\|_F^2 - \|X_c V_k\|_F^2, \quad (1.31)$$

最大化投影方差等价于最小化正交重构误差。

两种方法的结果是否相同

对同一份数据，采用相同的中心化、标准化和协方差归一化约定，在精确算术下，协方差矩阵的 EVD 与数据矩阵的 SVD 给出相同的主成分方差；按降序选取时，它们描述的是同一个 PCA 问题。上面的对应关系也说明，主方向来自 V ，而 $U_k \Sigma_k$ 给出各样本在这些方向上的得分，两者的作用不能混淆。

① Note

备注：PCA 主方向为何可能不同

“结果相同”不要求软件输出的向量逐元素相同。对一个单特征值， v_i 与 $-v_i$ 都是合法的主方向，得分也会随之变号，但投影和重构不变。若某个特征值重复，对应特征子空间内可以选取不同的标准正交基，此时得分坐标可能相差一个正交变换；只要完整保留这个子空间，重构仍然相同。

若恰好在正的重特征值中间截断，例如 $\lambda_k(C) = \lambda_{k+1}(C) > 0$ ，所选的前 k 维子空间可能不同，重构矩阵也可能不同，但它们达到的最小重构误差相同。相反，当 $k < p$ 且 $\lambda_k(C) > \lambda_{k+1}(C)$ 时，前 k 维主子空间及其投影重构是唯一的。若已保留全部非零方差方向，再选取哪些零方差方向都不影响重构。

实际计算有什么区别

协方差 EVD 先形成 $X_c^T X_c$ ，再分解得到的 $p \times p$ 矩阵；直接 SVD 则分解 X_c 本身。前一种路线把奇异值先平方：当 X_c 列满秩时， $\kappa_2(C) = \kappa_2(X_c)^2$ 。如果奇异值相差很大，较小的主成分方差在形成协方差矩阵时就更容易被舍入误差淹没。因此，需要识别较弱的主成分或判断数值秩时，直接 SVD 通常更可靠。秩亏时通常的条件数为无穷，不能把上述有限条件数比较照搬过去；但形成协方差矩阵仍会把非零奇异值之间的比例平方。若两个主成分的方差本来就十分接近，单个方向还会对数据扰动敏感，这是 PCA 问题本身的性质，直接 SVD 也不能消除。

计算代价则取决于数据形状。对稠密数据，若样本数 n 远大于特征数 p ，且 p 不大，构造较小的协方差矩阵再做 EVD 是常用且可能高效的选择；若 p 远大于 n ，显式存储和分解 $p \times p$ 矩阵就很昂贵，直接薄 SVD 往往更合适。只需少数主成分时，两条路线都可以使用截断或迭代算法，无须计算完整分解，选择还要考虑数据的稀疏性和所需精度。

🔔 Tip

本页要点： 协方差 EVD 与数据 SVD 在数学上对应同一个 PCA 问题，但直接 SVD 通常能更可靠地保留较弱的主成分。

1.12 PageRank and Eigenvectors

PageRank 通常归于斯坦福大学的 Larry Page 和 Sergey Brin: 他们在 1998 年的论文 *The Anatomy of a Large-Scale Hypertextual Web Search Engine* 中介绍了这一方法及其在 Google 搜索中的应用。PageRank 的思想受到“学术论文可由被引用次数评价”的启发, 只是把“引用”换成网页之间的超链接。原始论文的完整信息是: Brin, S., & Page, L. (1998), *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, **Computer Networks and ISDN Systems**, 30(1--7), 107--117, DOI: 10.1016/S0169-7552(98)00110-X。下面的矩阵推导说明, 这个网页排名问题为什么会变成特征值问题。

设有 n 个网页, 先假定每个网页至少有一条出链; 没有出链的情形在下文单独处理。若网页 j 有 $C(j)$ 条出链, 则定义列随机链接矩阵

$$M_{ij} = \begin{cases} 1/C(j), & j \rightarrow i, \\ 0, & \text{其他}. \end{cases} \quad (1.32)$$

记 $\mathbf{p}$ 为各网页当前访问概率组成的列向量, 且 $p_j = PR(j)$ 。由于 M 的第 j 列记录从网页 j 出发的一步转移概率, 矩阵乘积的第 i 个分量为 $(M\mathbf{p})_i = \sum_{j=1}^n M_{ij}p_j = \sum_{j \rightarrow i} p_j/C(j)$ 。它正好收集了所有指向网页 i 的页面贡献。

页面 A 的标量递推式

$$PR(A) = \frac{1-d}{n} + d \sum_{B_i \rightarrow A} \frac{PR(B_i)}{C(B_i)} \quad (1.33)$$

表示随机浏览者以概率 d 沿链接移动, 以概率 $1-d$ 均匀跳转。对每个页面 i , 式 (1.33) 都可以写成 $p_i = (1-d)/n + d(M\mathbf{p})_i$ 。把这 n 个分量合并, 得到 $\mathbf{p} = (1-d)\mathbf{1}/n + dM\mathbf{p}$ 。由于 $\mathbf{1}^T\mathbf{p} = 1$, 有 $\mathbf{1}\mathbf{1}^T\mathbf{p} = \mathbf{1}$, 于是定义 Google 矩阵

$$G = dM + \frac{1-d}{n}\mathbf{1}\mathbf{1}^T, \quad 0 < d < 1, \quad (1.34)$$

并得到 PageRank 方程

$$G\mathbf{p} = \mathbf{p}, \quad \mathbf{p} \geq \mathbf{0}, \quad \mathbf{1}^T\mathbf{p} = 1. \quad (1.35)$$

这里的特征向量是**右特征向量**: $G\mathbf{p} = \mathbf{p}$ 表示经过一次随机浏览转移后, 网页访问概率分布保持不变。反过来, 若 $G\mathbf{p} = \mathbf{p}$ 且 $\mathbf{1}^T\mathbf{p} = 1$, 逐分量展开就恢复式 (1.33), 所以标量递推和 Google 矩阵写法完全等价。

为什么不把式 (1.33) 当作普通递推式直接“算一次”就结束? 因为右端包含未知的 PageRank 分量: 算出 p_i 需要先知道其它页面的 p_j , 而这些 p_j 又依赖更多页面; 链接还可能形成环。因此它不是由已知量驱动的前向递推, 而是一个相互耦合的固定点方程。给定任意满足概率约束的初始向量 $\mathbf{p}^{(0)}$, 反复计算 $\mathbf{p}^{(k+1)} = G\mathbf{p}^{(k)}$ 就是在用递推求这个固定点; 这正是后面幂迭代的含义。收敛后 $\mathbf{p}^{(k+1)} \approx \mathbf{p}^{(k)}$, 于是 $G\mathbf{p} \approx \mathbf{p}$, 所以“用递推计算”和“求特征向量”是同一问题的两种说法。

从代数上也可以把固定点方程改写为线性系统。令 $\mathbf{u} = \mathbf{1}/n$, 由 $\mathbf{p} = dM\mathbf{p} + (1-d)\mathbf{u}$ 得 $(I - dM)\mathbf{p} = (1-d)\mathbf{u}$ 。当 $0 < d < 1$ 且 M 为列随机矩阵时, $\rho(dM) = d < 1$, 所以 $I - dM$ 非奇异, 理论上可以用直接法或迭代线性求解器求解。若直接写成 $(I - G)\mathbf{p} = \mathbf{0}$, 则系数矩阵必然奇异, 因为我们正是在寻找特征值 1; 此时必须另外加入 $\mathbf{1}^T\mathbf{p} = 1$ 这样的归一化条件。

真实网页图往往极大且 M 稀疏。对 $(I - dM)$ 做直接分解可能产生大量填充，存储和计算代价很高；幂迭代每一步只需一次稀疏矩阵-向量乘法，并自动保持非负性和概率和为 1，因此通常更适合大规模 PageRank。特征向量表述主要用来说明稳定分布的存在、唯一性和理论性质，实际算法仍可采用幂迭代或其它适合稀疏系统的方法。

若网页 j 没有任何出链，它称为 dangling node（悬挂节点）：浏览器到达它以后，不能按照网页链接继续前进。此时按原定义，矩阵 M 的第 j 列全为零，列和不再是 1，因而不能直接表示概率转移。通常把这个列改为 $M_{:,j} = \mathbf{1}/n$ ，等价于规定浏览器从该网页均匀跳到所有页面，从而恢复列和为 1。

记悬挂节点集合为 $D = \{j : C(j) = 0\}$ 。式 (1.33) 中也需计入这些节点的贡献： $p_i = (1 - d)/n + d \sum_{j \rightarrow i} p_j / C(j) + (d/n) \sum_{j \in D} p_j$ 。最后一项把所有悬挂节点的概率均分给各网页；没有悬挂节点时，这一项为零，就回到原式。使用修正后的 M ，Google 矩阵和固定点方程仍保持式 (1.34) 至 (1.35) 的形式。实现时只需累加悬挂节点的概率，再均匀分配，不必显式存储这些稠密列。

修正后 $\mathbf{1}^T G = \mathbf{1}^T$ ，故 1 是 G 的特征值；又因跳转项使 $G_{ij} \geq (1 - d)/n > 0$ ， G 是正矩阵。由 **Perron—Frobenius 定理**（见 02线性代数基础.pdf 第 2.28—2.29 页），谱半径 $\rho(G) = 1$ 对应的正特征方向唯一；再加上 $\mathbf{1}^T \mathbf{p} = 1$ 的归一化条件，才得到唯一的 PageRank 向量 $\mathbf{p}$ 。

幂迭代

$$\mathbf{p}^{(k+1)} = G\mathbf{p}^{(k)} \tag{1.36}$$

保持非负性和分量和为 1。阻尼项不仅描述随机跳转，也消除网络不连通和周期性造成的非唯一性。

 **Tip**

本页要点： PageRank 的标量递推逐分量合并后就是 $G\mathbf{p} = \mathbf{p}$ ，因此网页排名是 Google 矩阵的归一化右特征向量。

1.13 Landscape of Parallel Computing

把循环分给多个处理器只是并行计算的一步。不同问题的数据依赖、访存规律、同步频率和负载均衡特征不同，需要采用相应的并行模式。

“计算小矮人”分类用少量反复出现的计算模式描述大量应用中的典型难点或瓶颈。它的意义在于：如果两个应用共享同一种核心模式，那么针对数据布局、通信规避和硬件映射形成的经验往往可以迁移。数值线性代数同时覆盖稠密、稀疏、谱和图计算，是多类并行应用的公共基础。

 **Tip**

本页要点： 并行模式要适应问题的数据依赖、访存规律和同步需求。

1.14 Thirteen Dwarfs in Parallel Computing

十三个“小矮人”可按主要数据访问特征理解。

- 稠密线性代数：** 矩阵元素大多参与运算，访问规则，适合 BLAS-3 和 GPU；常受计算吞吐限制。
- 稀疏线性代数：** 只存非零元，但需要索引和间接访存；SpMV 常受内存带宽限制。

3. **谱方法**: 在频域或谱空间计算, 例如 FFT; 全局数据重排和缓存局部性很关键。
4. **N -体方法**: 计算大量点之间的相互作用; 树方法和快速多极子方法通过分层近似降低复杂度。
5. **结构化网格**: 规则邻域便于分块、向量化和 halo 交换。
6. **非结构化网格**: 连接关系显式存储, 需要图划分并面临不规则访存。
7. **MapReduce**: 局部任务相对独立, 最后聚合; 规模增大时 shuffle 和归约通信可能占主导。
8. **组合逻辑**: 以逻辑操作和有限状态为核心, 适合 SIMD 或专用硬件。
9. **图遍历**: 沿边访问节点; 前沿大小变化造成负载不均衡和细粒度同步。
10. **动态规划**: 子问题之间有依赖, 可通过波前并行, 但常受同步约束。
11. **回溯与分支定界**: 搜索树形状动态变化, 剪枝导致工作量难预测。
12. **图模型**: 变量和依赖构成图, 推断常包含稀疏计算与大量消息传递。
13. **有限状态机**: 下一状态依赖当前状态, 串行依赖强, 通常难以直接并行化。

分类中的“性能限制”不是绝对结论; 矩阵规模、稀疏模式、硬件和实现都会改变瓶颈。

💡 Tip

本页要点: 十三种模式可按主要数据访问特征理解。

1.15 Dwarf Popularity (Red Hot → Blue Cool)

热力图按 HPC、嵌入式、专用计算、机器学习、游戏和数据库等领域展示不同计算模式的使用程度。红色表示更常见, 蓝色表示相对较少。

读图时应关注两点。第一, 同一算法可能同时包含多种模式, 例如 PDE 求解器既包含结构化或非结构化网格, 也包含稀疏线性代数和全局归约。第二, 某种模式“常见”不等于“容易优化”; 图遍历和稀疏计算虽然应用广泛, 却常因不规则访问而难以接近峰值性能。

💡 Tip

本页要点: 一个应用可能包含多种计算模式, 使用广泛的模式也未必容易优化。

1.16 Foundation & Future of Scientific Computing

气候、量子化学、连续介质模拟和数据科学都需要矩阵计算, 相关计算核可能占据较大比例的运行时间。因此, 数值线性代数是科学计算的基础之一。面向不同硬件, 算法设计都需要考虑以下三个方面。

- **规模上的稳定性**: 维数、迭代次数和并行归约次数增加时, 舍入误差会积累。
- **硬件感知算法**: 算法应减少数据移动、暴露并行性, 并适配多核、GPU 和新型体系结构。
- **跨领域统一**: 科学模型和机器学习都需要可解释的误差、稳定的优化与可靠线性代数内核。

硬件吞吐提高不能替代数值分析。提高运行速度不能弥补不稳定算法造成的精度损失。

💡 Tip

本页要点： 矩阵计算服务于多类科学应用，计算内核的稳定性和性能会影响整体结果。

第1.3节 数值挑战

1.18 Considerations in Numerical LA

设计算法时，以下四项需要一起考虑。

1. **计算存在误差：** 实数必须舍入到有限集合，运算也需舍入。
2. **矩阵具有稀疏结构：** 只处理非零元可降低算术和存储，但会使访问不规则。
3. **实现效率：** 理论 FLOP 数相同的两个实现，wall time 可能相差很大。
4. **现代计算硬件：** 缓存、向量单元、多核、GPU 和网络决定可实现性能。

因此，算法评价需要同时报告**误差或残量、问题规模、运算量、数据移动量、并行资源和实测时间**。

💡 Tip

本页要点： 评价一个数值算法时，误差、规模、通信和实测时间要一起看。数值代数领域关注的重点：算法精度、问题结构、实现效率、硬件特点。

第1.3a节 数值挑战1：浮点运算精度

1.20 Some Historical Comments

早期机器可能使用定点数，且不同机器的浮点格式并不统一。IEEE 754 标准于 1985 年首次发布，2008 年和 2019 年先后修订，统一了基本格式、舍入模式和特殊值，使同一算法能够在不同硬件上获得可分析的行为。SP、DP 和 QP 分别指单、双和四倍精度；多精度软件则以更高成本模拟更多有效位。

求解 $Ax = b$ 时，低精度结果差可能来自两种不同原因：矩阵病态使问题本身对扰动敏感，或算法在舍入中引入了过大的后向误差。提高精度能减小舍入，但成本更高；迭代改进则尝试用高精度残量修正低精度分解，是混合精度算法的重要原型。

补充材料：CPU如何计算加法和乘法

[https://www.bilibili.com/video/BV1nL411x7jH/?](https://www.bilibili.com/video/BV1nL411x7jH/?share_source=copy_web&vd_source=a5ea9947c3dabdfbdfddde593e3a7b45)

[share_source=copy_web&vd_source=a5ea9947c3dabdfbdfddde593e3a7b45](https://www.bilibili.com/video/BV1nL411x7jH/?share_source=copy_web&vd_source=a5ea9947c3dabdfbdfddde593e3a7b45)

💡 Tip

本页要点： 早期机器可能使用定点数，且不同机器的浮点格式并不统一。

1.21 Floating-Point Number Storage

一个规范化二进制浮点数可写成

$$x = (-1)^s (1.f)_2 2^{e-b}, \quad (1.37)$$

其中 s 是符号位, e 是存储的指数, b 是指数偏置, $(1.f)_2$ 是有效数。规范数的首位恒为 1, 因而无需显式存储, 称为隐藏位。

若有效数共有 p 个二进制位, 并采用 round-to-nearest, 则单位舍入误差为

$$u = 2^{-p}. \quad (1.38)$$

对处于正规范范围内的实数, 通常有

$$fl(x) = x(1 + \delta), \quad |\delta| \leq u. \quad (1.39)$$

④ Note

备注: Machine epsilon 与 unit roundoff

“Machine epsilon”常指从 1 到下一个更大浮点数的距离 $\varepsilon = 2^{1-p} = 2u$ 。不同软件对 `eps` 与 `unit roundoff` 的命名可能不同, 使用误差界时必须先确认定义。

大端序 (Big-endian) 和小端序 (Little-endian) 是两种字节序 (Endianness), 规定一个多字节数的各个字节按什么顺序放入内存。Big-endian 把最高有效字节放在最低地址处, Little-endian 则把最低有效字节放在最低地址处。这里的“高”和“低”指字节在整个数中的位权, 不是字节本身数值的大小。

例如, 32 位无符号整数 `0x12345678` 占 4 个字节, `0x` 表示十六进制, 每两位十六进制数字对应一个字节。其中 `0x12` 是最高有效字节, `0x78` 是最低有效字节。假设它从地址 `0x1000` 开始存储, 两种排列如下, 表中的地址从左到右递增:

字节序	<code>0x1000</code>	<code>0x1001</code>	<code>0x1002</code>	<code>0x1003</code>
Big-endian (大端序)	<code>0x12</code>	<code>0x34</code>	<code>0x56</code>	<code>0x78</code>
Little-endian (小端序)	<code>0x78</code>	<code>0x56</code>	<code>0x34</code>	<code>0x12</code>

两种方式保存的是同一个数, 只是字节排列相反, 并不把每个字节内部的比特也倒过来。多字节浮点数同样存在这个存储顺序问题: 字节序不改变符号位、指数位和有效数位的含义, 也不改变浮点值的数学定义; 对于同一种浮点格式, 按写入时的字节序读取, 就能还原相同的值。跨平台读写二进制文件或交换原始字节时, 需要约定数值格式、位宽和字节序, 否则同一串字节可能被解释成不同的数。指数位决定范围, 有效数位决定相对精度, 这与大端序、小端序无关。

在现在常见的机器上, 两种字节序的使用情况如下。表中指程序在对应系统中通常采用的内存字节序, 不是文件或网络协议规定的字节序。

机器或系统组合	通常采用的字节序
Intel、AMD 的 x86-64 电脑和服务	Little-endian（小端序）
Apple Silicon（M 系列）Mac	Little-endian（小端序）
主流 ARM64 手机和 Linux 服务器	Little-endian（小端序）
IBM Z 大型主机，例如运行 <code>s390x</code> Linux 的系统	Big-endian（大端序）
运行 AIX 的 IBM Power 服务器	Big-endian（大端序）
运行 <code>ppc64le</code> Linux 的 IBM Power 服务器	Little-endian（小端序）

因此，学生日常使用的 PC、Mac 和主流手机通常采用 Little-endian，Big-endian 则仍用于部分大型主机和企业服务器。ARM 和 POWER 家族中存在支持两种字节序的架构或实现，称为双端序（bi-endian）；实际采用哪一种，还取决于具体硬件、操作系统和应用二进制接口（ABI）的约定，并非每台机器都能在两种模式间切换。例如，`ppc64le` 中的 `le` 就表示 little-endian，所以不能笼统地说“所有 Power 机器都是大端”或“所有 ARM 机器都是小端”。在 Python 中可查看 `sys.byteorder`，它返回 `little` 或 `big`，表示当前运行平台的字节序。

 **Tip**

本页要点： 浮点数由符号、有效数和指数共同决定，可表示范围与精度彼此制约。

1.22 Examples of Double FP Numbers

IEEE binary64 使用 1 个符号位、11 个指数位和 52 个显式小数位；加上隐藏位后精度为 $p = 53$ 。例如

$$0.5 = 1.0_2 \times 2^{-1},$$

(1.40)

所以符号为 0，实际指数为 -1 ，存储指数为 $-1 + 1023 = 1022$ ，小数字段全零。

十进制 0.1 的二进制展开是无限循环：

$$0.1 = (0.0001100110011 \dots)_2.$$

(1.41)

binary64 只能截取并舍入到 53 位，因此存储值不是精确的 $1/10$ 。同理， π 也只能近似表示。

指数和小数字段的特殊组合用于表示：

- 正负零：指数字段和小数字段均为零，符号位可不同；
- 正负无穷：指数字段全一，小数字段全零；
- NaN：指数字段全一，小数字段非零；
- 次正规数：指数字段全零、小数字段非零，不使用隐藏的首位 1，用于逐渐下溢。

次正规数扩大了接近零处的表示范围，但相对误差模型 $fl(x) = x(1 + \delta)$ 在逐渐下溢区需要额外处理。

 **Tip**

本页要点： binary64 使用 52 位显式小数加 1 位隐藏位，精度和可表示范围由位宽共同决定。

1.23 Examples of FP Arithmetic (4 digits)

下面用十进制四位有效数模拟浮点运算。对

$$A = 9.999 \times 10^1, \quad B = 1.610 \times 10^{-1}, \quad (1.42)$$

做加法时必须先对齐指数。把 B 改写为

$$B = 0.0161 \times 10^1 \quad (1.43)$$

对阶本身不改变数值。为说明正确舍入，先保留对阶后的低位，完成精确相加：

$$A + B = (9.999 + 0.0161) \times 10^1 = 10.0151 \times 10^1 = 1.00151 \times 10^2. \quad (1.44)$$

最后舍入到四位有效数，得到

$$fl(A + B) = 1.002 \times 10^2. \quad (1.45)$$

浮点加法依次完成指数对齐、有效数相加、规格化和舍入。实际硬件利用保护位等信息完成最终舍入；若在中间步骤丢弃低位，得到的就只是近似和。当两个数尺度差异很大时，较小加数的贡献可能在最终舍入时完全丢失。

对乘法

$$A = 1.110 \times 10^{10}, \quad B = 9.200 \times 10^{-5}, \quad (1.46)$$

先相加指数并相乘有效数：

$$10 + (-5) = 5, \quad 1.110 \times 9.200 = 10.212. \quad (1.47)$$

规格化后

$$10.212 \times 10^5 = 1.0212 \times 10^6, \quad (1.48)$$

舍入得到

$$fl(AB) = 1.021 \times 10^6. \quad (1.49)$$

乘法不需要对齐输入指数，但要检查结果的指数是否溢出，并在规格化后舍入有效数。

💡 Tip

本页要点： 舍入模式和特殊值是 IEEE 754 可复现性的基础。

1.24 IEEE754 2008 Version

常见二进制格式的关键参数如下。 p 包含隐藏位，表中的 u 对应 round-to-nearest。

格式	指数位	显式小数位	p	单位舍入误差 $u = 2^{-p}$
binary16	5	10	11	4.88×10^{-4}
binary32	8	23	24	5.96×10^{-8}
binary64	11	52	53	1.11×10^{-16}
binary128	15	112	113	9.63×10^{-35}

表中较低位宽的格式具有更小的表示范围和更少有效位数。范围不足会导致上溢或下溢，精度不足会增大舍入误差；这两种风险必须分别分析。BF16 与 binary16 的总位数相同，但 BF16 保留更宽的指数、减少有效位，因而更强调范围而非精度。

 **Tip**

本页要点： 不同浮点格式分别权衡指数范围和有效位数，位宽相同也不代表数值性质相同。

1.25 Why Mixed-Precision Is Important

混合精度是指同一算法在存储、乘法、累加、残量计算或修正步骤中**使用不同浮点格式**。低精度数据占用更少字节，因此同一缓存可容纳更多数据，内存和网络传输量也更小。

以 V100 的这组规格数据为例，FP64、FP32 和 Tensor Core FP16 的峰值吞吐分别约为 7、14 和 112 TFLOP/s，而内存带宽约为 900 GB/s。这些峰值吞吐数据说明：如果算法能用低精度完成大部分工作，再以高精度校正，就可能同时**减少算术运算时间和数据移动**。

把变量类型改为 FP16 后，还必须重新分析缩放、上溢、正定性丢失、停止准则和最终精度，才构成完整的低精度算法。

 **Tip**

本页要点： 混合精度是指同一算法在存储、乘法、累加、残量计算或修正步骤中使用不同浮点格式。

1.26 Driving Force: Deep Learning

深度学习的大量工作由规则的稠密矩阵乘法和卷积构成，对吞吐要求高，并且训练过程在一定程度上能容忍局部舍入噪声。这推动了 Tensor Core 等混合精度单元。

融合乘加 FMA 计算

$$\text{fma}(a, b, c) = \text{round}(ab + c) \tag{1.50}$$

时只在最终和处舍入一次，而分开的乘法和加法通常舍入两次。Tensor Core 可**在较低精度输入上执行矩阵乘法，并在更高精度累加器中累加**；输入精度、乘积精度和累加精度需要分别说明。

低精度适用于一些 AI 训练任务，但不能无条件地用于所有科学计算。守恒性、正定性和严格残量要求可能对舍入更敏感。

补充材料: How do Graphics Cards Work?

https://www.bilibili.com/video/BV1RwL1zkEdQ/?spm_id_from=333.337.search-card.all.click&vd_source=f370c1df022887e8da2ccbf59c7ec2ff

Tip

本页要点: 深度学习的规则矩阵运算适合高吞吐硬件, 但容忍低精度的程度仍取决于具体任务。

1.27 HPL-MxP Benchmark (Nov, 2025)

HPL-MxP 用低精度 LU 因子作为预条件器, 并以 GMRES 型迭代改进恢复目标精度。其抽象过程是: 先在低精度中计算

$$PA_L \approx LU, \quad (1.51)$$

再在较高精度中形成当前残量

$$\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k, \quad (1.52)$$

求近似修正方程

$$A\mathbf{d}_k = \mathbf{r}_k, \quad (1.53)$$

并更新 $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k$ 。低精度因子通过 $U^{-1}L^{-1}P$ 提供成本较低的预条件作用, GMRES 则处理该预条件器并不精确的情况。

若预条件后的系统足够良态、低精度分解没有破坏关键结构, 并且残量与更新使用足够高的精度, 迭代才可能达到高精度答案。基准通过低精度完成主要计算, 并用高精度修正控制误差, 以兼顾性能和精度。

Tip

本页要点: HPL-MxP 的核心是低精度因子加高精度迭代精校。

第1.3b节 数值挑战2: 稀疏矩阵运算

1.29 Sparse Matrices

对 $A \in \mathbb{R}^{n \times n}$, 定义

$$\text{nnz}(A) = \#\{(i, j) : a_{ij} \neq 0\}. \quad (1.54)$$

若一族问题满足 $\text{nnz}(A) = O(n)$, 则平均每行只有 $O(1)$ 个非零元, 称为稀疏矩阵族。局部 PDE 离散、有限邻接图和 PageRank 链接矩阵通常具有这种性质。

稠密存储需要 n^2 个数值; 稀疏存储理想上需要 $O(\text{nnz}(A) + n)$ 个数值与索引。基本运算包括

- SpMV: 稀疏矩阵乘向量, 约 $2 \text{nnz}(A)$ FLOP;
- SpMM: 稀疏矩阵乘稠密多向量;

- SpGEMM：两个稀疏矩阵相乘，输出稀疏模式需动态确定。

“稀疏”本身不足以选择格式；还要看每行长度分布、块结构、对角结构以及目标硬件。

💡 Tip

本页要点： 稀疏矩阵的价值在于非零元远少于总元素，算法应避免处理零。

1.30 Storing Sparse Matrices

C00 用三元组 (i_k, j_k, a_k) 存储每个非零元，便于逐项装配，但同一行元素未必连续。CSR 把非零元按行排列，用三个数组表示：

- `values[k]`：第 k 个非零值；
- `colInd[k]`：该值的列号；
- `rowPtr[i]`：第 i 行在前两数组中的起点。

第 i 行对应区间

$$k = \text{rowPtr}[i], \dots, \text{rowPtr}[i + 1] - 1. \quad (1.55)$$

页中 4×4 矩阵的数组为

$$\begin{aligned} \text{rowPtr} &= [0, 2, 4, 7, 9], \\ \text{colInd} &= [0, 1, 1, 2, 0, 2, 3, 1, 3], \\ \text{values} &= [1, 7, 2, 8, 5, 3, 9, 6, 4]. \end{aligned} \quad (1.56)$$

例如第 2 行（零基编号）使用位置 4, 5, 6，所以非零元位于列 0, 2, 3，数值为 5, 3, 9。

CSRx 在 CSR 上增加隐式顺序信息。页中

$$\text{diagPtr} = [0, 2, 5, 8] \quad (1.57)$$

直接指向各行对角元。这样可快速访问对角、严格下三角和严格上三角部分，又不改变 $O(\text{nnz} + n)$ 的渐近存储量，适合三角求解和预条件。

💡 Tip

本页要点： 稀疏存储的选择取决于矩阵结构，也取决于后续访问方式。

1.31 Sparse Matrix Formats

不同格式在存储开销、并行规则性和填充浪费之间取舍。

- **CSR/CSC**：分别按行/列压缩，通用且适合 SpMV 或列访问。
- **DIA**：按对角线存储，适合带状或规则 stencil；非规则矩阵会浪费填充。
- **ELL**：把每行补齐到相同长度，便于 SIMD/GPU，但长短行差异大时浪费严重。
- **HYB**：常将主要规则部分放在 ELL，少量例外放在 C00。

- **块 CSR**: 一次存储一个小稠密块, 减少索引并提高数据复用, 前提是矩阵确有块结构。
- **tile/CSR5 类格式**: 把非零流划为更规则的并行任务, 以改善 GPU 负载均衡。

选择格式时应测量有效带宽、索引开销、负载不均衡和格式转换成本, 而不能只比较 `nnz`。

Tip

本页要点: 不同格式在存储开销、并行规则性和填充浪费之间取舍。

第1.3c节 数值挑战3: 计算、存储与移动

1.33 Performance Models

wall time 是用户真正等待的时间, 但只有在问题、精度和硬件口径一致时才可公平比较。一个求解器更快, 可能只是因为停止容差更松、离散规模更小或使用了不同预处理。

完整性能描述至少包含: 问题维数与 `nnz`、误差或残量目标、算法和参数、处理器/GPU 数量、精度格式、编译与库环境、wall time、迭代数, 以及必要时的能耗。

硬件模型提供峰值 FLOP/s、内存带宽、缓存容量和网络参数; 算法模型提供 FLOP、字节数、消息数和依赖关系。程序实测性能是两者及具体数据共同作用的结果。

Tip

本页要点: wall time 是用户真正等待的时间, 但只有在问题、精度和硬件口径一致时才可公平比较。

1.34 Moore's Law

Moore 定律描述集成电路晶体管数长期增长的规律; 晶体管变小后, 为什么曾经能够同时提高密度和频率, 却不增加单位面积的动态功耗? 这需要用 Dennard 恒电场缩放来解释, 它是与 Moore 定律相关、但不同的物理缩放模型。

设每一代把晶体管的各个线性尺寸缩为原来的 $s = 0.7$ 。下面的比例建立在理想长沟道 MOS 晶体管模型上, 假定材料性质不变, 电压和阈值电压也能按同一比例缩小。

1. **尺寸与密度**。沟道长度 L 、宽度 W 和栅氧化层厚度 t_{ox} 都缩为原来的 s , 晶体管占用面积便缩为 $s^2 = 0.49$, 即减少约 51%。因此, 相同芯片面积上可容纳约 $1/s^2 \approx 2.04$ 倍的晶体管。
2. **电压、电容与电流**。为保持电场 $E = U/d$ 不变, 距离 d 缩为 s 倍时, 电压 U 也缩为 s 倍。栅电容近似满足 $C \propto WL/t_{\text{ox}}$, 所以缩为 s 倍。理想导通电流满足 $I_{\text{on}} \propto (W/L)V_{\text{ov}}^2/t_{\text{ox}}$, 其中 V_{ov} 是栅源电压与阈值电压之差; 它也缩为 s 倍, 因此电流的缩放比例为 $s^2/s = s$ 。
3. **延迟与频率**。电路翻转需要给电容充放电, 延迟近似为 $t_d \propto CU/I_{\text{on}}$, 所以缩为 $s \cdot s/s = s = 0.7$ 倍, 即减少 30%。相应的工作频率可提高到 $1/s \approx 1.43$ 倍, 即提高约 43%。这里的速度提升来自尺寸、电压、电容和电流的共同缩放, 不能仅凭面积减半推出。
4. **能量与功率**。单次翻转能量满足 $E_{\text{sw}} \propto CU^2$, 缩为 $s^3 = 0.343$ 倍, 即减少约 65.7%。若翻转活动率不

变，并让频率提高到 $1/s$ 倍，单个晶体管的平均动态功率便缩为 $s^3/s = s^2 = 0.49$ 倍。相同芯片面积内晶体管数量增至 $1/s^2$ 倍，两者相乘为 1：**晶体管约多一倍、频率约高四成，而总动态功率及功耗密度保持不变。**

以上是理想缩放关系，未计入漏电功耗、互连开销等因素，也不意味着任意电路的实际功耗都能按这一比例降低。现代工艺中，电压不能一直随尺寸同比下降，漏电和散热约束使这种缩放难以维持；再加上存储延迟和指令级并行限制，传统单核频率路线因而放缓。

现代性能增长更多依赖多核、宽向量单元、GPU 和专用加速器。这**对算法的并行性提出了要求**：必须暴露足够并行性，并控制线程同步、NUMA 访问和主机-设备传输。串行算法不会因为晶体管更多就自动获得过去那样的加速。

💡 Tip

本页要点：理想 Dennard 缩放使晶体管约多一倍、频率约高四成而总动态功率不变，关键是电压也随尺寸降低。

1.35 Memory Hierarchy

处理器按速度和容量组织存储层次：L1 最小最快，L2 次之，多个核心可能共享 L3/LLC，DRAM 容量大但延迟高。典型量级从 L1 的约 1 ns、几十 KB，到 DRAM 的 50—100 ns、GB 级容量。

程序并不显式为每次加载选择缓存层；硬件按 cache line 搬运数据。算法通过连续访问、分块和时间局部性提高命中率。矩阵分块的本质是让一个子块在被逐出快速存储前参与尽可能多的乘加。

这里的“逐出”是缓存替换或驱逐（cache eviction），不是数据从整个存储系统中消失。缓存通常采用**组相联（set-associative）**结构：地址先确定所属的组（set），该内存块可以存放在组内若干个位置之一，每个位置称为一路（way）。查找时，硬件检查该组各路的有效位（valid bit）和标签（tag）；若有有效行的标签与请求匹配，就发生命中，否则就是未命中（cache miss）。

以采用**写回（write-back）策略**的缓存为例，一次读请求未命中的简化处理流程如下。

1. **选择位置。** 若目标组中有空路（无效行），直接使用；若该组已满，按替换策略选出一条被替换行（victim line）。不是从整个缓存中随意挑选，因此即使别的组还有空间，也可能发生冲突替换。
2. **必要时写回。** 如果被替换行是脏行（dirty line），即本级中的数据已修改、尚未写回下一级，就先把它写回下一级；如果是干净行，可以直接丢弃本级副本。这一步只针对写回缓存中的脏行，并非每次替换都要写回。
3. **取回新行。** 从下一级存储取得请求地址所在的整条 cache line，而不只是所需的一个数。L1 未命中可能由 L2 满足，只有继续向下查找仍未命中时，才需要访问 DRAM。
4. **装入缓存。** 把新行放入选定位置，更新标签和有效位；在这一简化读流程中，新装入的行是干净的。
5. **返回数据。** 根据请求地址在行内的偏移，取出所需数据并交给处理器。

📌 Note

备注：缓存处理中的操作重叠

这是便于理解的逻辑顺序；实际硬件可以借助写回缓冲区等机制重叠处理，并不一定逐步等待完成。第 1.38 页进一步说明替换策略、相联度和多核一致性的影响。

多核还会共享最后级缓存和内存带宽，因此单核优化结果不能直接乘以核心数。

补充材料：内存工作原理

https://www.bilibili.com/video/BV1WwJAz7E45/?share_source=copy_web&vd_source=a5ea9947c3dabdfbdfddde593e3a7b45

 **Tip**

本页要点： 处理器按速度和容量组织存储层次：L1 最小最快，L2 次之，多个核心可能共享 L3/LLC，DRAM 容量大但延迟高。

1.36 Improving Memory Performance

早期预测与 El Capitan 节点及系统指标的比较显示，系统峰值和带宽的增长幅度较大。内存延迟仍约为几十到上百纳秒，改善幅度小于算术吞吐。DDR 各代提高了峰值带宽和预取宽度，**却没能让基本访问延迟按同一比例下降**。带宽回答“稳定流式访问每秒可搬多少数据”，延迟回答“一次依赖性访问要等多久”。SpMV 的连续值数组可能受带宽限制，而由列索引引起的随机 x_j 访问还会受到延迟影响。两者不能用同一个数字替代。

 **Note**

备注：Memory latency 与 CAS latency。 本页的 Memory latency 指系统级主存访问延迟，通常包括互连传输、内存控制器排队与调度、DRAM 读操作及数据返回；表中约 100 ns 未注明具体测量口径。CAS latency（Column Address Strobe latency，列地址选通延迟）只是其中一个环节：目标行已打开后，从 DRAM 接收 READ 命令到输出首个数据的时间。若目标行尚未打开，还要等待行激活；若当前打开的是另一行，还需先关闭原来的行。规格中的 CL 通常以时钟周期数表示，例如 DDR4-3200 的内存时钟为 1600 MHz，CL16 对应 10 ns；本页下表已经换算为纳秒。上表是 HBM3 系统，下表是 DDR，两组数值不能直接相减来估计其他开销。

几台率先突破 E 级的机器可以帮助理解这一性能量级。1 EFlop/s = 10^{18} FLOP/s，即每秒百亿亿次浮点运算。以下按 TOP500 的双精度 HPL 实测性能 $R_{\max}$ 判断，不与理论峰值 R_{peak} 或混合精度成绩混用。

机器	首次突破 E 级的 TOP500 榜单	当期 HPL 实测性能	历史位置
Frontier（橡树岭国家实验室）	2022 年 6 月	1.102 EFlop/s	榜单上第一台突破 E 级的机器，并位居第一
Aurora（阿贡国家实验室）	2024 年 6 月	1.012 EFlop/s	第二台突破 E 级的机器，当期排名第二
El Capitan（劳伦斯利弗莫尔国家实验室）	2024 年 11 月	1.742 EFlop/s	第三台突破 E 级的机器，首次上榜即超过 Frontier、登上榜首

“第一台、第二台、第三台”说的是突破门槛的先后顺序，不是同一期的性能排名。HPL 主要考察适合高效并行的稠密线性代数计算；达到 E 级不意味着低算术强度、访存不规则或通信密集的应用也能达到同样吞吐率。

💡 Tip

本页要点： 算术吞吐和内存带宽大幅提高时，内存延迟仍可能改善缓慢。

1.37 Machine Balance Over Time

定义机器平衡点

$$I_* = \frac{P_{\max}}{B_{\text{mem}}} \text{ FLOP/Byte}, \quad (1.58)$$

其中 $P_{\max}$ 是峰值计算吞吐， B_{mem} 是内存带宽。它表示为了让计算单元满载，每移动一个字节至少要完成多少 FLOP。

若 $P_{\max}$ 增长快于 B_{mem} ，则 I_* 随时间增大。过去可能是计算受限的核，在新机器上可能转为带宽受限。低精度把每个数的字节数减小，可提高相同算法的算术强度，但前提是精度仍满足要求。

💡 Tip

本页要点： 在 Roofline 模型中，计算核的算术强度与机器平衡点的相对位置决定性能上界由哪一项限制。

1.38 A Simplified Cache Mechanism

请求地址时，缓存先根据映射规则定位可能的 cache line，并比较标签。命中时直接从缓存读；未命中时从下一级存储取回整条 line，产生较大延迟。

若目标位置已被占用，替换策略选择一条 line 驱逐。对于 write-back 缓存，若被驱逐 line 是 dirty 的，必须先写回下一级；否则可直接覆盖。相联度决定一个内存块可放入多少个候选位置：相联度过低容易冲突，相联度更高则控制逻辑更复杂。

多线程写入同一 cache line 中的不同变量仍可能触发一致性流量，称为 false sharing。通过对齐、填充或改变数据划分可避免这种非必要通信。

补充材料：CPU中的缓存是如何工作的？

[https://www.bilibili.com/video/BV1MdZJYyEUF/?](https://www.bilibili.com/video/BV1MdZJYyEUF/?share_source=copy_web&vd_source=a5ea9947c3dabdfbdfddde593e3a7b45)

[share_source=copy_web&vd_source=a5ea9947c3dabdfbdfddde593e3a7b45](https://www.bilibili.com/video/BV1MdZJYyEUF/?share_source=copy_web&vd_source=a5ea9947c3dabdfbdfddde593e3a7b45)

💡 Tip

本页要点： 缓存命中靠映射与标签匹配；未命中后的替换和写回会增加数据移动。

1.39 SpMV, A Simplified Example

稀疏矩阵向量运算写为

$$\mathbf{y} \leftarrow A\mathbf{x} + \mathbf{y}, \quad y_i \leftarrow y_i + \sum_{j:a_{ij} \neq 0} a_{ij}x_j. \quad (1.59)$$

按行划分 A 时，每个进程拥有一组 y_i 和相应矩阵行，因此输出不冲突；但这些行可能引用其他进程拥有的 x_j 。行块的进程归属决定输出分布，**Access X** 表示各非零块所需的向量分段。

Tip

本页要点： SpMV 的性能通常由访存和通信而不是乘加次数决定。

1.40 Measuring Performance

评价程序性能，需要同时测量运行时间及其与**适用性能上界**的差距。设一个算法执行 F 次浮点运算，从目标存储层移动 D 字节，实测运行时间为 T 。算术运算吞吐率与数据传输率分别为

$$P = \frac{F}{T}, \quad B_{\text{obs}} = \frac{D}{T}. \quad (1.60)$$

这里 B_{obs} 是实测传输率，不是硬件带宽上限。计算单元和 DRAM 之间的数据通路也会限制速度；FLOP 数相同，搬运的字节数不同，耗时就可能不同。

若适用于该精度和运算类型的计算峰值为 P_{max} ，目标存储层的带宽上限为 B ，则计算和数据移动分别至少需要

$$T_{\text{comp}} \geq \frac{F}{P_{\text{max}}}, \quad T_{\text{mem}} \geq \frac{D}{B}. \quad (1.61)$$

即使计算和数据移动可以完全重叠，也必须等待较慢的一项完成，因此 wall time 满足

$$T \geq \max\left(\frac{F}{P_{\text{max}}}, \frac{D}{B}\right). \quad (1.62)$$

只有在计算与访存均达到各自性能上限，且能完全重叠的理想模型中，时间下界才取等号。实际运行还可能有启动、依赖和同步开销，所以这里写成下界，而不把它当作实测时间的恒等式。式中单位必须统一：FLOP 除以 FLOP/s 得秒，Byte 除以 Byte/s 也得秒；若使用 GFLOP 和 GB，则分子、分母都要采用对应单位。

定义**算术强度** (arithmetic intensity, 也记为 AI 或 CI)

$$I = \frac{F}{D} \text{ FLOP/Byte}. \quad (1.63)$$

将时间下界代入 $P = F/T$ ，再用 $F/(D/B) = B(F/D) = BI$ ，得到

$$P = \frac{F}{T} \leq \frac{F}{\max(F/P_{\text{max}}, D/B)} = \min(P_{\text{max}}, BI). \quad (1.64)$$

沿相同算术强度向上，表示提高已有计算和数据移动的执行效率；通过缓存复用、分块等减少 D ，则能向更高算术强度移动。不要为了增大 I 而添加无用运算。

Tip

本页要点： 判断程序是否够快，要比较其算术强度下的性能上界，而不只看机器峰值。

1.41 Compute-Bound vs Memory-Bound

第 1.40 页的式 (1.64) 给出两类性能上界。本页用它区分两种情况：计算受限 (compute-bound) 与内存带宽受限 (memory-bound)；前者指处理器的计算吞吐限制，并不只针对 CPU，也适用于 GPU 等处理器。

$P = BI$ 给出带宽限制下的性能上界：当带宽 B 固定时，该上界与算术强度 I 成正比。 $P = P_{\max}$ 则给出计算能力限制下的性能上界，不随 I 增大而提高。两个上界相等时的算术强度

$$I_* = P_{\max}/B \quad (1.65)$$

是**机器平衡点**。5 至 10 FLOP/Byte 仅是特定硬件背景下的示意值，并非所有机器的固定常数；使用时应按硬件、计算精度和存储层重新计算。

若 $I < I_*$ ，Roofline 上界由带宽项 BI 决定；若 $I > I_*$ ，由计算项 $P_{\max}$ 决定。这先区分的是模型中的受限区域。若实测点明显低于相应屋顶，还要检查指令依赖、访存延迟、分支、负载不均衡和通信，不能仅凭算术强度断定实际瓶颈。页中“50% of Peak”和“50% of STREAM”也采用了不同基准：前者比较计算吞吐，后者比较内存流式访问带宽。

常见计算核的算术强度各不相同：SpMV、BLAS-1/2 和 stencil 通常偏低，FFT 居中，能充分复用数据的 BLAS-3 矩阵运算可以更高。这些只是相应工作量与数据移动假设下的粗略趋势；规模、缓存容量和分块方式都会改变实际算术强度。

④ Note

备注：向量更新的 FLOP 数

$$a_i = b_i + \alpha c_i \quad (1.66)$$

每个元素执行 1 次乘法和 1 次加法，共 2 FLOP。若 b_i, c_i 各读一次、 a_i 写一次，binary64 数据量为 $3 \times 8 = 24$ Byte，故

$$I = \frac{2}{24} \approx 0.083 \text{ FLOP/Byte}. \quad (1.67)$$

这通常远低于机器平衡点，因此增加浮点计算能力不会明显加速。

④ Note

备注：7 点 stencil 计算内核的 FLOP 数

页中 7 点 stencil 的 7 FLOP 可以从常系数三维 Laplacian 的计算式看出来。对均匀网格，记中心值为 u_0 ，六个相邻网格点的值为 $u_1, \dots, u_6$ 。暂不计网格尺度因子 h^{-2} ，负 Laplacian 的差分核为 $6u_0 - (u_1 + u_2 + u_3 + u_4 + u_5 + u_6)$ ：六个邻点求和需要 5 次加法，中心值乘以 6 需要 1 次乘法，两者相减需要 1 次减法，共 $5 + 1 + 1 = 7$ FLOP。若最后还要乘 h^{-2} ，则再增加 1 FLOP；变系数、残量计算和边界处理也要按实际计算式另行计数。这里数的是浮点算术操作，不是机器指令数。在假设邻居值大多缓存命中、每点只计一次有效读和一次写时，估算

$$I = \frac{7}{2 \times 8} \approx 0.438 \text{ FLOP/Byte}. \quad (1.68)$$

若缓存复用没有实现，实际字节数会更大，算术强度更低。Roofline 诊断应针对明确的存储层和实测流量，而不是只用源代码中的数组个数。

从理论Roofline到实际测量

用硬件规格中的峰值计算吞吐和峰值带宽画出的屋顶，通常是偏乐观的上界。实际能达到多高，还取决于工作频率、指令组合、并发程度和访存方式。为给性能比较选取更贴近所用机器的基准，可以使用以下工具。

- **LBL Empirical Roofline Toolkit (ERT)**：其中的 Empirical Roofline Tool 通过一组微基准，测量计算吞吐及不同存储层的带宽，构建经验 Roofline。测量时应说明计算精度、线程或进程配置及所针对的存储层；微基准测出的上界也不意味着每个应用都能达到。参见 [LBL 工具介绍](#)及[代码与说明的公开镜像](#)。
- **NVIDIA Nsight Compute**：用于分析 CUDA 等 GPU 计算核，查看计算吞吐、访存流量等性能指标，并通过 Roofline 图比较计算核的实测性能与相应上界。其分层 Roofline 可以帮助区分不同存储层的限制；使用时需确认图中的计算精度、带宽层次和计数口径。参见 [Nsight Compute 官网](#)与 [Roofline 图的官方说明](#)。

两者的侧重点不同：ERT 用微基准刻画机器的经验上界，Nsight Compute 用于观察具体 GPU 计算核在这些性能约束下的表现。

💡 Tip

本页要点： Roofline 思想用算术强度把计算受限与带宽受限区分开。

1.42 Communications Are Costly

这里的 communication 广义地指数据移动。把数据从 DRAM 送到处理器、从 CPU 送到 GPU、或从一个 MPI 进程送到另一个进程，都需要时间和能量。

物理距离决定信号传播和互连成本；技术上，算术单元可通过复制和流水线提高吞吐，而存储容量、带宽和延迟不能同比例改善；经济上，高速缓存和高带宽互连价格昂贵。因此计算峰值的增长通常快于通信能力。

对数值算法，这意味着减少同步、增加数据复用、融合算子和使用通信规避算法，往往比单纯减少少量乘加更重要。

💡 Tip

本页要点： 数据移动包括访存和进程间通信，都要计入时间与能量成本。

1.43 Modelling Communication Costs

移动一条包含 m 字节的消息，最简单模型为

$$T_{\text{comm}}(m) = \alpha + \frac{m}{\beta}, \quad (1.69)$$

其中 α 是启动延迟， β 是有效带宽。“Latency + Bytes/Bandwidth”概括了这一模型。

若算法发送 q 条消息、总字节数为 D ，近似成本为

$$T_{\text{comm}} \approx q\alpha + \frac{D}{\beta}. \quad (1.70)$$

因此把许多小消息合并可减少延迟项；压缩或避免重复数据可减少带宽项。串行机器中，fast/slow memory 对应缓存与 DRAM/HBM；分布式机器中，对应本地与远端存储。两种情形都可用“快速存储容量有限、慢速存储足够大”的模型分析。

Tip

本页要点： 通信时间可拆成启动延迟和随消息长度增长的传输时间。

1.44 Example: Matrix Multiplication

矩阵乘法

$$C = AB, \quad c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}, \quad (1.71)$$

对 $n \times n$ 稠密矩阵约需 $2n^3$ FLOP (n^2 次向量内积)。若三重循环顺序不合适， A 或 B 的元素会被反复从慢速存储加载，算术单元反而等待数据。

分块把矩阵写为子块并计算

$$C_{IJ} \leftarrow C_{IJ} + A_{IK} B_{KJ}. \quad (1.72)$$

选择能放入缓存的块后，同一 A_{IK} 和 B_{KJ} 可参与块内大量乘加，提高算术强度。高性能 GEMM、LAPACK 的 blocked factorization 以及 GPU tile 都依赖这一思想。

优化时，要在快速存储容量的限制下尽量复用数据、减少慢速层之间的传输，同时保持足够大的并行任务；循环结构应服务于这些目标。

Tip

本页要点： 矩阵乘法的算术量约为立方级，分块和数据复用决定实际性能。

第1.3d节 数值挑战4：并行计算效率

1.46 Richardson's Fantastic Forecast Factory

Richardson 在 1922 年设想由大量“人工计算员”分别负责地球不同区域，通过中央调度协调边界数据。这一构想已经包含现代区域分解“分而治之”的基本元素：**子域、局部计算、邻域通信、同步和负载调度**。

若一个子域提前完成，它仍可能等待邻域边界值；若各区域工作量不同，则出现负载不均衡。红蓝灯调度对应今天的运行时系统和消息状态。这个历史例子说明，处理单元数量很大并不自动解决数据依赖问题。

Tip

本页要点： Richardson 在 1922 年设想由大量“人工计算员”分别负责地球不同区域，通过中央调度协调边界数据。

1.47 Computational Complexity

从时间推进、线性化、空间离散到线性求解和并行化，总成本可概念性写为

$$\text{Complexity} \sim N_{\text{model}} \times N_{\text{time}} \times N_{\text{nl}} \times C_d \left(N_{\text{space}}^d \right)^k \div P. \quad (1.73)$$

这里 N_{model} 是模拟运行次数， N_{time} 是每次模拟的时间步数， N_{nl} 是每个时间步内的非线性迭代数， N_{space}^d 表示 d 维空间自由度量级，指数 k 概括线性求解器的复杂度， P 是处理器数。

除以 P 只代表完美负载均衡且无通信的理想情况。更现实的时间模型应写成

$$T(P) = \frac{T_{\text{work}}}{P} + T_{\text{comm}}(P) + T_{\text{sync}}(P) + T_{\text{imbalance}}(P). \quad (1.74)$$

自适应可降低必要自由度，快速/最优复杂度求解器控制 k 及迭代数，并行化再分摊工作；三者针对不同的成本来源，应结合问题选择。复杂度也不是 wall time，因为常数、数据移动和硬件利用率同样重要。

💡 Tip

本页要点： 复杂度分析要同时记录算术量、数据移动和通信次数。

1.48 An Illustration of Parallel Computing

大量低功率设备的总额定功率可以超过飞机发动机，但它们不能直接替代引擎，因为接口、能量密度、同步与控制方式不匹配。并行计算同理：处理器峰值之和只是资源上限，不是应用可实现性能。

要把资源转化为加速，需要可分解的算法、足够粗的任务粒度、合适的互连、均衡的数据分布和可扩展的软件栈。异构或性能不一致的处理单元还会使调度更困难。

💡 Tip

本页要点： 大量低功率设备的总额定功率可以超过飞机发动机，但它们不能直接替代引擎，因为接口、能量密度、同步与控制方式不匹配。

1.49 75 Years, 14 Orders of Magnitude

从 ENIAC、104 机、Cray-I、银河-I、富岳到百亿亿次系统，计算吞吐从每秒数千次提高到 10^{18} 次量级。十四个数量级的硬件提升使过去不可计算的模型成为可能，但也推动问题规模、分辨率和模型复杂度同步增长。

一个标志性节点是 2022 年 5 月底公布的[2022 年 6 月版 TOP500 榜单](#)：位于美国橡树岭国家实验室的 Frontier（“前沿”）以 HPL 实测性能 1.102 EFlop/s 登顶，即每秒约 1.102×10^{18} 次浮点运算，约为 1.1 百亿亿次。它是 TOP500 榜单上第一台实测性能突破 E 级的超级计算机。这里的“第一”限定为公开榜单和相同 HPL 测试口径，不代表任何程序都能达到这一速度；后续 Aurora、El Capitan 的节点见第 1.36 页。

历史峰值不能直接用于比较应用时间：不同年代的“每秒运算”口径、精度和基准不同。课程中更关心算法如何把硬件能力转化为达到指定误差的解。

本页要点： 从 ENIAC、104 机、Cray-I、银河-I、富岳到百亿亿次系统，计算吞吐从每秒数千次提高到 10^{18} 次量级。

1.50 中国早期电子计算机发展史

中国计算机事业起步于科学研究和工程计算的需求。回顾这段历史，可以分别看设计能力和技术路线：前者从学习、仿制走向自行设计，后者从电子管、晶体管走向集成电路和大规模并行系统。两者相关，但不能等同；自行设计的机器也可能沿用上一代的电子器件。

起步：先培养队伍，再研制机器

1952 年，中国科学院数学研究所成立，华罗庚等人推动计算数学研究与人才培养。1956 年，计算技术被列入国家科学技术发展规划中的紧急措施，中科院开始筹建计算技术研究所。筹备工作同时涉及整机、元件电路、计算数学和程序设计，并通过训练班等途径培养人才。这说明，计算机事业从一开始就不只是制造硬件，还要有人能够把实际问题转化为算法和程序。

几台代表性机器及其特点如下。

时间与机器	技术特点	在发展中的意义
1958 年： 103 机 (八一型)	按苏联 M-3 图纸仿制的小型电子管数字计算机；初期约每秒 30 次，改进后达到每秒 1800 次	中国第一台通用数字电子计算机；通过研制和调试，建立整机、生产与应用队伍，并开始承担实际工程计算
1959 年： 104 机	仿制苏联 Б Э С М-II 的大型电子管计算机，平均每秒约 1 万次	把计算能力扩展到更大的科学与工程任务，曾用于大地测量、水坝应力等计算
1960 年： 107 机	夏培肃主持研制的自行设计通用电子计算机，交付中国科学技术大学使用	从按照外来图纸研制，走向掌握自己的整机设计；计算机也进入高校教学和科研
1964 年： 119 机	自行设计的大型通用计算机，采用电子管和晶体二极管构成逻辑元件，运算速度约每秒 5 万次	不仅提高速度，也改进编程和使用条件，强调实际解题效率；它仍属于电子管计算机，不能因年份较晚就称为晶体管机
1973 年： 150 机	百万次级集成电路计算机，配套操作系统与应用程序	用于石油地震勘探资料处理等任务，计算机与重要行业的生产需求进一步结合
1983 年： 银河-I	国防科技大学研制的亿次级巨型计算机	国内高性能计算进入新的规模，之后的银河系列继续发展，并为更大规模科学计算积累系统研制经验

表中的“每秒多少次”沿用当时的历史资料，不同机器的指令、字长和测试方法并不相同，不能直接当作今天统一测量的双精度 HPL 成绩来比较。

从 119 机到 150 机之间，还有晶体管计算机的发展阶段。例如，中科院计算所于 1965 年研制成功 109 乙机，1967 年研制成功 109 丙机。电子管和晶体管都可以构成逻辑开关；晶体管的体积与功耗较小，集成电路则进一步把多个器件和连线做在芯片上。这些变化使机器能够集成更多功能，但整机性能仍受存储、输入输出和软件效率制约。

从科研设备走向产业与大规模并行计算

1980 年代以后，微型计算机、汉字信息处理和个人计算机逐步推广，计算机的使用范围从少数科研机构扩展到企业、学校和日常办公。高性能计算则沿另一条路线继续发展：随着单个处理器的能力受到限制，需要用并行算法、互连网络和系统软件组织更多处理器共同工作。个人计算机的普及与超级计算机的发展，是这一时期并行推进的两方面。

此后，中国高性能计算又有三项进展：

- **2010 年 11 月，天河一号 A** 位居 TOP500 第一，这是中国超级计算机首次登上该榜单榜首；它采用 CPU 与 GPU 协同计算的异构结构。
- **2013 年 6 月，天河二号** 登上 TOP500 榜首，并连续六期排名第一，体现了更大规模的并行系统集成能力。
- **2016 年 6 月，神威·太湖之光** 接替天河二号位居第一，采用国产申威 SW26010 众核处理器。这也带来新的软件要求：程序必须适应其并行层次和数据移动方式，才能用好硬件。

这些榜单记录的是当时的性能排名，不是今天的名次；后续突破 E 级计算的国际历史节点见第 1.36 页。

为什么这段历史与数值线性代数有关

高性能计算设备的获取与使用还受出口许可、最终用户和用途限制等条件影响。“买到机器”并不总等于可以不受限制地使用和维护它；具体要求取决于当时的合同与管制条件，不能概括成所有进口计算机都采用同一种管理方式。

“自主可控”不仅指处理器，也包括编译器、通信库、BLAS/LAPACK、稀疏求解器和应用软件。任何一层正确性或性能缺陷都可能成为整个模拟链条的瓶颈。

硬件、数值方法和软件库相互促进：更大机器要求更稳定、更可扩展的算法，新的算法又让有限硬件能够求解更大问题。无论早期工程计算还是今天的大规模模拟，评价计算能力都要看能否在给定时间和资源内得到精度可信的解。

参考资料：[中科院计算所《历史沿革》](#)；TOP500 的 [2010 年 11 月](#)、[2013 年 6 月](#)和 [2016 年 6 月](#)历史榜单。

Tip

本页要点： 从“造出机器”到“用好机器”，硬件研制、数值算法和软件系统始终需要共同发展。

1.51 HPC Top500 List 2026.06

这份榜单展示了系统规模、核心数、加速器和能耗的变化。解读时还要考虑：系统能否持续扩张，购置与运行成本是多少，目标应用是否需要这种规模，软件能否有效利用这些资源。

同一台机器上，HPL、HPCG 和应用程序的效率差异反映了计算强度、通信模式和数据局部性的不同。榜单峰值不应直接作为应用性能模型中的 $P_{\max}$ ，更可靠的做法是使用与目标核相符的实测峰值和带宽。

Tip

本页要点： 解读榜单时，要同时看系统规模、能耗与成本，以及目标应用能否用好这些资源。

1.52 How to Measure Parallel Efficiency

记 $T(W, C)$ 为工作量 W 在 C 个处理器上的运行时间。强可扩展性固定总工作量 W_1 ，以 C_1 个处理器为基准，将处理器数增加到 pC_1 。记 $T_1 = T(W_1, C_1)$ 、 $T_p = T(W_1, pC_1)$ ，其中下标 p 表示资源相对基准的倍数。加速比与并行效率定义为

$$S_{\text{strong}}(p) = \frac{T_1}{T_p}, \quad E_{\text{strong}}(p) = \frac{S_{\text{strong}}(p)}{p}. \quad (1.75)$$

理想情况下每个处理器工作量由 W_1/C_1 降为 $W_1/(pC_1)$ ，所以 $T_p \approx T_1/p$ 、 $S_{\text{strong}}(p) \approx p$ 、 $E_{\text{strong}}(p) \approx 1$ 。当本地工作量太小，通信和同步不再可忽略，效率会下降。

弱可扩展性把总工作量和处理器数同时增大 p 倍，即从 (W_1, C_1) 变为 (W_p, pC_1) ，其中 $W_p \approx pW_1$ 。这里 T_p 表示 $T(W_p, pC_1)$ ，与强扩展时不同，工作量也随 p 改变。理想目标是

$$T(W_p, pC_1) \approx T(W_1, C_1). \quad (1.76)$$

弱扩展效率可定义为

$$E_{\text{weak}}(p) = \frac{T(W_1, C_1)}{T(W_p, pC_1)}. \quad (1.77)$$

报告可扩展性时必须写明问题如何随 p 增长，因为三维网格的线性尺寸、自由度和通信表面积具有不同的增长率。

Tip

本页要点： 强扩展与弱扩展必须明确基准问题和处理器规模后再比较。

1.53 Three Mountains on Parallel Simulation

本页取单处理器为基准，即第 1.52 页中的 $C_1 = 1$ ，此时资源倍数 p 就是处理器数。

Amdahl：固定问题能算得多快

固定问题规模，设在单处理器运行时间 T_1 中，不可并行部分占比为 α ，可并行部分占比为 $1 - \alpha$ 。若并行部分能够理想地均分给 p 个处理器，忽略通信、同步和负载不均衡，则

$$T_p = \alpha T_1 + \frac{1 - \alpha}{p} T_1, \quad (1.78)$$

所以

$$S_A(p) = \frac{T_1}{T_p} = \frac{1}{\alpha + (1 - \alpha)/p}. \quad (1.79)$$

当 $\alpha > 0$ 时，令 $p \rightarrow \infty$ ，得到加速上限 $1/\alpha$ 。若 $\alpha = 0.1$ ，则这个固定问题最多加速 10 倍，因为单处理器时间中的那 10% 无法通过增加处理器来缩短。

Gustafson—Barsis：在相近时间内能算多大的问题

实际模拟中，增加处理器后，用户往往还会加密网格、增加样本或扩大计算区域。Gustafson—Barsis 定律考虑的就是这种情形：允许问题规模随处理器数增大，希望在大致相同的可接受运行时间内完成更多计算，而不是始终加速同一个小程序。

记 W_p 为准备交给 p 个处理器的放大后的工作量， $T(W_p, p)$ 为这次并行运行的时间。把它分成两段：不可并行阶段耗时 $t_s(W_p)$ ，可并行阶段耗时 $t_{\text{par}}(W_p, p)$ 。后者是这一阶段实际经过的时间，不是把所有处理器的时间相加。定义 $\beta(p) = t_s(W_p) / (t_s(W_p) + t_{\text{par}}(W_p, p))$ ，它是在这次并行运行时间中测得的串行占比。这里显式写出 W_p 和 p ，是因为串行阶段耗时及其占比都可能随问题规模改变。

现在固定这个放大后的问题 W_p ，比较用一个处理器和用 p 个处理器完成它所需的时间。在同一算法、同等单处理器性能且忽略额外并行开销的理想模型下，两种执行方式中的不可并行工作相同，因此都需要 $t_s(W_p)$ 。这里不是拿大问题和小问题比较：若用 W_1 表示原来小问题的 workload，并没有假设 $t_s(W_p) = t_s(W_1)$ ；问题变大后，串行阶段完全可能耗时更长。

若可并行工作能够理想均分，原来由 p 个处理器共同完成的这一阶段，改由一个处理器执行需要 $p t_{\text{par}}(W_p, p)$ 。于是，对同一个问题有

$$\begin{aligned} T(W_p, p) &= t_s(W_p) + t_{\text{par}}(W_p, p), \\ T(W_p, 1) &= t_s(W_p) + p t_{\text{par}}(W_p, p), \\ S_G(p) &= \frac{T(W_p, 1)}{T(W_p, p)} \\ &= \frac{t_s(W_p) + p t_{\text{par}}(W_p, p)}{t_s(W_p) + t_{\text{par}}(W_p, p)} \\ &= \beta(p) + p(1 - \beta(p)) \\ &= p - (p - 1)\beta(p). \end{aligned} \quad (1.80)$$

“把并行时间归一化为 1”，就是将时间除以本次并行运行总时间 $T(W_p, p)$ ，于是串行阶段占 $\beta(p)$ ，并行阶段占 $1 - \beta(p)$ 。这只是更换计时单位，不意味着不同规模的问题实际耗时相同，也没有假设串行阶段的耗时不随规模增长。

若并行运行中的串行占比 $\beta(p) = 0.1$ ，则 $S_G(p) = 0.9p + 0.1$ ；例如 $p = 100$ 时，加速比为 90.1。这里的 10% 是并行运行中的占比，不是 Amdahl 定律中单处理器运行的占比 α 。

如果串行工作也随规模增长

换一种 workload 增长方式：原来小问题在单处理器上需要串行工作 10 秒、可并行工作 90 秒；现在假设这两部分 workload 都增大 100 倍。扩大后的问题在单处理器上需要 $1000 + 9000 = 10000$ 秒，而在 100 个处理器上的理想时间为 $1000 + 9000/100 = 1090$ 秒。因而加速比只有 $10000/1090 \approx 9.17$ ，并不能达到前例中的 90.1。

此时 $\beta(100) = 1000/1090 \approx 0.9174$ ，已不再是 0.1。把这个比例代入式 (1.80)，得到的仍是 $S_G(100) = 100 - 99 \times (1000/1090) \approx 9.17$ 。而单处理器总时间中的串行占比仍是 $\alpha = 1000/10000 = 0.1$ ，所以 Amdahl 的 10 倍加速上限也仍然适用。因此，Gustafson--Barsis 定律并没有绕过串行瓶颈；它也不能保证任意问题扩大后都能在相同时间内完成。

要保持多处理器总运行时间大致不变，需要选择合适的问题增长方式，使新增工作主要能由新增处理器承担，并使串行阶段和额外并行开销的增长受控。若 $\beta(p)$ 仍然较小，式 (1.80) 允许加速比接近理想值 p ；常见的直线图示例则采用了 $\beta(p)$ 近似不变的简化。这与第 1.52 页弱扩展的思路相近，但 $S_G(p)$ 是放大后同一问题的单、多处理器时间之比，不是式 (1.77) 定义的弱扩展效率。

HPC Paradox: 硬件增长也会带来更大的计算需求

HPC paradox 描述一种需求增长现象：硬件扩大十倍后，用户可能希望求解十倍甚至更大的问题。若求解器复杂度超线性或迭代数随网格恶化，增加硬件只能部分抵消增长；网格无关迭代数、近线性复杂度预条件和软件投入因此至关重要。

💡 Tip

本页要点： Gustafson--Barsis 比较同一放大后问题的单、多处理器耗时，并不假设串行工作不会随问题规模增长。

1.54 Implementation of Algorithms

同一数学算法可因实现和优化方式不同产生数量级性能差异。优化层次包括：编译器优化，改变循环顺序，向量化和 AVX intrinsic，多线程并行，分块，缓存无关递归和专用数学库。

这些优化分别针对不同瓶颈：向量化提高计算单元利用率，循环交换改善连续访问，tiling 提高缓存复用，并行循环利用多核。正确实现算法不意味着已充分利用硬件。性能提升倍率依赖具体基线，不能脱离问题和硬件配置推广。

优化必须在正确性基线上进行：先保存参考结果和误差指标，再逐步改变实现并比较。否则编译器重排、并行归约顺序或低精度优化可能改变数值行为而不被发现。

💡 Tip

本页要点： 同一数学算法可因实现不同产生数量级性能差异。

1.55 Heterogeneous Cluster Architecture

典型集群由多个节点通过高速网络连接；每个节点包含 CPU 核、内存和一个或多个 GPU，CPU-GPU 或 GPU-GPU 之间可能用 PCIe/NVLink 连接。不同层次具有不同带宽和延迟。

并行算法通常同时包含：节点内多线程、GPU 核函数、设备间通信和跨节点 MPI。数据若频繁在主机与设备之间往返，即使 GPU 内核很快，总时间也可能由传输主导。

可扩展实现需要决定矩阵和向量放在哪一层、每层由谁拥有、halo 如何交换、归约在哪个通信域执行，并尽量让通信与独立计算重叠。

💡 Tip

本页要点： 异构集群的性能取决于数据放置、设备间传输和通信重叠，而不只取决于 GPU 峰值算力。

1.56 Parallel Matrix Data Layout

二维块循环分布先把矩阵划分为 $b_r \times b_c$ 的块，再把块 (I, J) 分给进程网格中的

$$(I \bmod P_r, J \bmod P_c). \quad (1.81)$$

相邻块循环落到不同进程，既避免一整行或一整列集中在单个进程，也让矩阵尾部的不完整块较均匀地分散。

ScaLAPACK 采用这种分布支持并行分解。

块过小会增加消息数和元数据，块过大则减少并行度并可能负载不均衡。布局还应匹配算法的通信方向：矩阵乘法需要行/列广播，SpMV 常按行分块并交换远端向量分量。

💡 Tip

本页要点： 二维块循环分布要同时考虑负载均衡、局部性和通信边界。

第1.4节 软件实现

1.58 Why Results Differ from Expectations?

数值实验与预期不一致时，可以从三个层面查找原因：问题本身（模型、离散或数据改变）、数值方法（条件数、舍入、停止准则或随机性）以及程序实现（索引、数据布局、并行归约或库调用）。

建议按以下顺序定位：先固定输入并用小规模已知解检查建模；再计算残量和后向误差，确认程序是否确实求解了目标方程；随后改变精度、缩放和线程数，观察结果是否按舍入和非确定归约的预期变化；最后用独立实现或高精度参考结果交叉核对。这些检查用于区分模型、数值方法和实现层面的原因；对尚未排除的因素，不宜作确定归因。

💡 Tip

本页要点： 结果偏离预期时，分别检查问题本身、数值过程和程序实现。

1.59 Chaotic Dynamics

《礼记·经解》引《易》：“君子慎始，差若毫厘，缪以千里。”在数值计算中，很小的初始差异也可能造成很大的长期差异，原因是什么？

MIT 数学家、气象学家 Edward N. Lorenz 在研究简化的大气对流模型时，发现确定性方程也可能产生非周期、对初始条件敏感的运动。他的代表论文是 1963 年发表于 *Journal of the Atmospheric Sciences* 的

[“Deterministic Nonperiodic Flow”](#)。后来广为人知的“蝴蝶扇动翅膀会不会引起龙卷风”这一比喻，与他 1972 年的演讲联系在一起。它强调的是长期预测对微小扰动的敏感性，并不是说任意微扰都必然造成某个特定的大事件。

当时的机器输出保留三位小数，而机器内部保留六位进行计算。Lorenz 用三位小数输入，微小初值差最终导致完全不同的轨迹。对具有正最大 Lyapunov 指数 $\lambda_{\max}$ 的混沌系统，微小扰动在一段时间内可近似满足

$$\|\delta \mathbf{x}(t)\| \approx e^{\lambda_{\max} t} \|\delta \mathbf{x}(0)\|. \quad (1.82)$$

因此，多保留几位只能把可预测时间向后推移，不能保证无限期逐轨道预测。混沌也不意味着数值计算毫无意义；统计量、吸引子结构和短期预报仍可能收敛，但验证指标必须与研究问题一致。

在这个例子中，舍入产生初始微扰，模型动力学使扰动放大，放大速率由正 Lyapunov 指数描述。

不过，看到两次计算的结果不同，并不能断言“方程是混沌的”。排查时至少要分清以下几类原因。

- **程序实现问题。** 逻辑错误、数组越界、并行读写的数据竞争或同步遗漏，都可能改变结果。找到可定位的 bug 反而意味着有明确的修复方向，但它不是数值误差。
- **计算顺序与编译选项。** 浮点加法不满足结合律，归约顺序变化、表达式重结合和 FMA 融合可能改变末位；允许近似运算的选项还可能降低精度。这不等于编译器必然“算错”，应检查它承诺的浮点语义以及是否启用了 `fast-math` 等选项。
- **建模与数值近似。** 模型误差、离散误差、求解误差和舍入误差都可能产生扰动；若动力学不稳定或具有混沌特性，这些扰动又可能被放大。
- **硬件故障。** 内存位翻转、计算或通信部件故障也可能引入异常，应结合硬件日志、纠错信息和重复实验排查，参见第 1.61 页。

Margaret Hamilton 早期也曾在 MIT 参与 Lorenz 的气象计算工作，后来领导阿波罗计划的机载飞行软件团队，也是软件工程观念的重要倡导者。她的经历涉及科学计算和高可靠软件两个领域，但不能据此认定 Lorenz 的每一次数值实验都由她编程。可参考 [MIT 关于 Hamilton 与阿波罗软件的介绍](#)。

Tip

本页要点： 混沌系统会放大微小扰动，数值轨迹的差异需要结合动力学来解释。

1.60 Pay Attention to Numerical Error

结果差异可按来源分为：模型误差、离散误差、迭代/截断误差、舍入误差和软件实现差异。常用控制手段分别包括模型校准、网格/阶次加密、残量与后验估计、稳定性分析和软件测试。

数值误差分析以程序正确实现算法为前提；软件错误属于实现问题，应通过测试和调试排除，不列入数值误差的分类。

误差之间可能相互遮蔽。例如，把线性求解器容差设得远小于离散误差只会增加成本；反之，若代数误差大于离散误差，网格加密也不会改善最终解。合理策略是建立误差预算，使各层误差与目标精度匹配。

Tip

本页要点： 数值误差要按来源分析并合理分配误差预算，软件错误则需另行排查。

1.61 Hardware Failures

MTTF 通常表示不可修复部件失效前的平均时间；MTBF 常用于可修复系统两次故障之间的平均时间。若简化为常失效率 λ 的指数模型，则

$$R(t) = \Pr(T > t) = e^{-\lambda t}, \quad \text{MTTF} = \frac{1}{\lambda}. \tag{1.83}$$

若系统由大量近似独立部件组成，任一关键部件失效即导致作业失败，则系统失效率近似为各部件失效率之和。部件数增加会缩短整机无故障时间，这解释了大规模作业为何更容易遭遇 CPU、内存或网络故障。

长时间并行模拟应采用 checkpoint/restart、冗余检测和容错归约。检查点过频会浪费 I/O，过疏则在故障后丢失更多计算，需要按故障率和写入成本权衡。

 **Tip**

本页要点： 可靠性指标必须说明失效率模型以及 MTTF、MTBF 的适用含义。

1.62 The Patriot Missile Failure

Dhahran（达兰）位于沙特阿拉伯东部省、波斯湾沿岸地区，是沙特阿美公司（Saudi Aramco）的总部所在地，也是该国石油工业的重要中心，周边区域包括加瓦尔油田（Ghawar Oil Field）等大型油气开发区。1991 年 2 月 25 日，本页所述的爱国者导弹拦截失败事故就发生在达兰。

十进制 0.1 在二进制中为无限循环数。系统用截断的 24 位定点数近似它，每个十分之一秒约产生

$$e_{0.1} \approx 9.5 \times 10^{-8} \text{ s} \tag{1.84}$$

的时间误差。连续运行约 100 小时后，计数约为

$$N = 100 \times 3600 \times 10 = 3.6 \times 10^6. \tag{1.85}$$

累计误差约为

$$Ne_{0.1} \approx 3.6 \times 10^6 \times 9.5 \times 10^{-8} \approx 0.34 \text{ s}, \tag{1.86}$$

对高速目标，这一时间偏差可使跟踪窗口偏离真实位置。

关键问题是系统没有根据最长连续运行时间分析**截断误差的累积**，没有及时重新校准时间基准。

 **Tip**

本页要点： Patriot 事故说明时间表示中的微小截断误差可随运行时间增长，进而影响跟踪结果。

1.63 The Loss of the Sleipner A Platform

该案例中，有限元网格含畸变单元，剪力估计又使用二次多项式向边缘外推，设计规范模型也未准确预测实际破坏压力。最终剪力被低估约 47%，平台在测试中沉没。

外推的目标点位于数据区间之外，高阶项可能放大误差，因此需要额外验证。有限元云图本身不能证明网格、单元或应力恢复正确；应检查网格质量、局部加密收敛、平衡条件和独立手算/模型。

这个案例强调离散误差与建模假设可能叠加。软件成功结束只说明程序没有在运行时失败，并不说明物理解可信。

💡 Tip

本页要点： 该案例中，有限元网格含畸变单元，剪力估计又使用二次多项式向边缘外推，设计规范模型也未准确预测实际破坏压力。

1.64 The Legendary “Nine Calculations”

“九次计算”是邓稼先、周光召等人参与原子弹理论研究期间的一段经历。经过多轮复算以及物理、力学、数学等学科的讨论，计算出的压力始终低于苏联教学模型给出的数值。这项研究有两层进展：九次计算支持了所用的流体假定；周光召随后借理想模型和最大功原理，说明对方给出的压力值超过了理论上限。

1961 年，周光召回国后加入相关理论研究工作，与邓稼先等人共同攻关。他们不仅关心计算结果，也强调民主、宽容的学术氛围，鼓励年轻研究人员讲真话、提出不同意见。这为质疑参考数据提供了讨论条件：面对专家给出的参考数据，研究人员需要有条件质疑，也需要拿出可检验的依据。

当时研究团队从教学模型出发，自行摸索计算方法和相关数据。早期结果能够吻合，深入计算后却在关键量上出现持续差异。各专业的研究人员反复审议，提出修改条件，再进行下一轮计算；前后九轮仍不能消除分歧。周光召听取汇报并检查计算材料后，没有把“专家给出的数值”当作必须凑出的答案，而是转向独立的理论约束，在彭桓武的支持下完成验证。这一结果帮助团队走出了僵局；到 1963 年底，他又协助邓稼先完成了相关理论设计工作。

复算与理论上限承担不同的检验任务：复算在给定假设下检查计算结果，理论上限则用独立的物理约束判断目标值是否可能。上界不必给出实际压力的精确值；只要目标值超过它，就能判定目标值与这些约束不相容。这个案例中，多轮复算和理论分析都发挥了作用。

科学计算中的验证既包括数值验证，也包括模型验证。若独立算法、加密网格和更高精度都给出一致结果，而结果仍与预期不符，应重新检查假设、数据和理论上限，而不是无休止调整程序以逼近预设答案。

这次突破还依赖前期的复算、跨学科讨论和材料整理，它们为独立判断提供了基础，不能只归于某个人的灵感。周光召后来把自己的贡献比作庞大集体中的“十万分之一”。讲课时可以提醒学生，严谨推理和允许不同意见、尊重集体工作的学术环境，都是可信计算所需要的。

💡 Tip

本页要点： 复算检查计算结果，独立的理论界还能检查预设目标本身是否可信。

1.65 Reliability of Numerical Simulator

数值模拟可靠性分为三个层次。

- **模型可靠性：** 方程、参数、边界条件和数据能否以所需精度描述真实物理。
- **算法可靠性：** 离散是否一致并收敛，求解算法是否稳定，停止准则是否与目标误差相关。
- **软件可靠性：** 代码是否正确实现算法，并能在给定硬件、并发和故障条件下持续运行。

一个与实验不符的结果不能直接归因于“数值误差”。应按模型误差、离散误差、迭代误差、舍入误差和软件错误逐层定位，并用网格加密、残量、守恒量、人造解或独立实现交叉验证。

💡 Tip

本页要点： 模拟可靠性要同时检查模型、算法和软件三个层次。

1.66 Legacy Implementation

LINPACK 项目始于 1970 年代，目标是提供可移植、高质量的线性方程组软件。它把成熟数值算法封装为标准接口，使用户不必为每台机器重写全部分解与求解过程，也为后来的 LINPACK benchmark 和 LAPACK 奠定基础。

早期实现主要围绕当时的存储层次和向量机器设计。随着缓存和多级并行成为主流，算法必须从以 BLAS-1 为主的细粒度操作转向能复用数据的 BLAS-3 块算法。

💡 Tip

本页要点： LINPACK 项目始于 1970 年代，目标是提供可移植、高质量的线性方程组软件。

1.67 Modern Implementation

从 Python 用户的角度看，现代稠密线性代数软件可分为以下几层：

$$\text{NumPy/SciPy} \rightarrow \text{LAPACK} \rightarrow \text{BLAS} \rightarrow \text{CPU/GPU kernels.} \quad (1.87)$$

BLAS 提供向量、矩阵-向量和矩阵-矩阵核；LAPACK 用这些核实现分解、求解和谱算法；ScaLAPACK 扩展到分布式内存；Python 库提供高层数组和接口。

可靠软件需要兼顾：错误返回和边界情况、用户接口、后向稳定性、不同矩阵结构的专用算法，以及多线程/MPI/GPU 上的性能。库的层次化使底层优化可被大量上层算法共享。

💡 Tip

本页要点： 数值库把稳定性处理和硬件优化封装起来，但调用者仍要检查输入与返回状态。

1.68 Why Bother Using Numerical Libraries

教学用的三重循环或消元程序可以表达算法思想；实际使用的数值库还要处理缩放、选主元、异常参数、工作空间、缓存分块、向量化、多线程和不同硬件后端。性能与可靠性来自长期的算法分析、测试和体系结构优化。

高质量库常采用：缓存友好的 blocked algorithm，特征值问题中的分治或多阶段方法，以及用高精度残量改善解的迭代改进。多线程、MPI 和 GPU 后端让同一高层算法映射到不同平台。

使用库不等于放弃理解。 用户仍需选择与矩阵结构匹配的例程，理解输入是否被覆盖，检查 `info`、残量和条件估计，并控制线程和数据布局。库负责可靠地实现通用内核，应用代码则负责模型、结构和验证。

💡 Tip

本页要点： 成熟数值库把稳定性、异常处理和硬件优化一起封装起来。

1.69 LAPACK

LAPACK 面向稠密和带状矩阵，线性系统部分根据结构选择算法：一般矩阵用带部分主元的 LU，对称正定矩阵用 Cholesky，对称不定矩阵用带主元的 LDL^T 或 LDL^H 。结构专用算法通常减少存储和运算，并保留矩阵性质。

最小二乘部分覆盖超定与欠定系统，主要使用 QR 或 LQ 分解，也支持带约束变体。以列满秩的 $A \in \mathbb{C}^{m \times n}$ 、 $m \geq n$ 为例，采用完整 QR 分解，其中 $Q \in \mathbb{C}^{m \times m}$ 为酉矩阵， $R \in \mathbb{C}^{n \times n}$ 为非奇异上三角矩阵：

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}, \quad Q^H Q = I, \quad (1.88)$$

由于左乘酉矩阵 Q^H 不改变二范数，

$$\|Ax - b\|_2 = \left\| \begin{bmatrix} R \\ 0 \end{bmatrix} x - Q^H b \right\|_2, \quad (1.89)$$

将 $Q^H b$ 的前 n 个分量记为 c_1 ，其余分量记为 c_2 ，则残量范数的平方为 $\|Rx - c_1\|_2^2 + \|c_2\|_2^2$ 。第二项不依赖 x ，所以只需解三角系统 $Rx = c_1$ ；第 6.14 页会详细讨论这一方法。这里的 **TBC** 表示 LAPACK 功能列表在第 1.70 页继续。

💡 Tip

本页要点： LAPACK 按矩阵结构选择 LU、Cholesky 或对称不定分解，并通过分块实现高性能。

1.70 LAPACK, Continued

标准特征值问题为 $Ax = \lambda x$ ，广义特征值问题为

$$Ax = \lambda Bx. \quad (1.90)$$

对称/Hermitian 矩阵与一般非对称矩阵使用不同算法和存储；用户还可选择全部谱或指定区间的部分谱。

SVD 支持完整或经济型分解，并用于伪逆、数值秩和最小二乘。LAPACK 还提供 LU、Cholesky、QR/QL/RQ/LQ、Schur 和对称谱分解。不同分解不是同一操作的别名：它们利用不同结构，并服务于不同稳定性和复杂度目标。

💡 Tip

本页要点： 按矩阵结构和所需谱信息选择 LAPACK 例程，各种分解的用途并不相同。

1.71 LAPACK XYZZZZ Names

LAPACK 例程名可按 **XYZZZZ** 解读。

- **X** 表示数据类型：**S** 为单精度实数，**D** 为双精度实数，**C** 为单精度复数，**Z** 为双精度复数。

- **YY** 表示矩阵类型：**GE** 一般矩阵，**GB** 一般带状，**SY** 对称，**HE** Hermitian，**TR** 三角，**PO** 对称/Hermitian 正定等。
- **ZZZ** 表示操作：**SV** 求解，**SVX** 带条件估计和误差界的专家驱动，**TRF** 分解，**TRI** 求逆，**EV** 特征值，**SVD** 奇异值分解等。

例如，**DGESV** 表示双精度一般矩阵线性求解，**DGELS** 表示双精度一般最小二乘，**DGESVD** 表示双精度一般矩阵 SVD，**DGEQRF** 表示双精度一般矩阵 QR 分解。

命名帮助定位例程，但不能替代接口文档；原地覆盖、leading dimension、workspace 和 **info** 的含义仍需逐项确认。

💡 Tip

本页要点： LAPACK 例程名可按 **XYZZZ** 解读。

1.72 ScaLAPACK

ScaLAPACK 把 LAPACK 的部分稠密算法扩展到分布式内存。BLACS 提供进程网格上的广播、点对点和归约通信，PBLAS 提供分布式 BLAS，ScaLAPACK 在其上实现分解与求解。

二维块循环分布让矩阵块均匀分散到 $P_r \times P_c$ 进程网格，降低单个进程成为行或列瓶颈的风险。分布式算法的结果仍由同一代数分解定义，但浮点归约顺序、通信拓扑和块大小会影响性能及末位结果。

扩展性取决于局部 BLAS-3 工作能否覆盖面板分解、广播和同步成本。矩阵太小时增加进程反而会降低效率。

💡 Tip

本页要点： ScaLAPACK 把 LAPACK 的部分稠密算法扩展到分布式内存。

1.73 NumPy and SciPy

NumPy 提供多维数组、广播、基本线性代数与底层 BLAS/LAPACK 的连接；SciPy 在此基础上提供更丰富的稠密、稀疏、优化、积分和特征值接口。

高层调用如 `numpy.linalg.solve` 适合常规任务；`scipy.linalg.blas` 和 `scipy.linalg.lapack` 的低层包装允许明确调用 `dgemm`、`dgetrf` 等例程。数组的 `dtype`、存储顺序和是否复制会影响最终调用及性能。

使用库时仍应检查输入结构和返回状态。例如，对称正定问题应选 Cholesky 接口；稀疏矩阵不应无意转为稠密；求解后应计算残量而不是只相信函数未抛出异常。

💡 Tip

本页要点： NumPy 和 SciPy 提供了从数组运算到专业线性代数例程的现成工具。

1.74 Set Up A Python Environment

为课程代码建立一个“干净”的 Python 环境，是指不混用其它项目已经安装的包，只加入本课程所需的依赖。不同项目可能需要不同版本的 NumPy 或 SciPy；若共用同一个环境，为一个项目升级包就可能使另一个项目无法运行。单独安装课程依赖，可以避免改动系统 Python 的包或影响其它项目，也更容易确认代码需要哪些依赖，便于排查问题和重建环境。

在 macOS 上可用 Homebrew 安装 Python，再在课程项目目录中创建并使用独立环境：

```
brew install python
python3 -m venv .venv
source .venv/bin/activate
python -m pip install numpy scipy
```

`python3 -m venv .venv` 在当前目录创建虚拟环境，默认不继承基础 Python 环境中已经安装的第三方包；`source .venv/bin/activate` 让当前终端优先使用该环境中的 Python 和命令。`python -m pip` 用当前解释器执行 pip，避免把包安装到另一个 Python 环境中。

这里隔离的主要是 Python 包的安装和查找，并不是建立一台虚拟机；程序仍使用本机的操作系统、系统库和硬件。运行结束后使用

```
deactivate
```

退出环境，但不会删除其中已经安装的包，下次可再次激活使用。为便于复现，应记录 Python、NumPy、SciPy 版本以及 BLAS 后端和线程数；同一 Python 代码连接 Accelerate、OpenBLAS 或 MKL 时，性能及浮点结果的末位可能不同，独立环境本身并不保证结果逐位一致。

Tip

本页要点： 用独立环境避免项目间的依赖冲突，并记录版本和 BLAS 配置以便复现。

1.75 Examples of SciPy

代码先展示 BLAS Level 1，即 $O(n)$ 的向量-向量操作：

- `ddot(x,y)` 计算 $x^T y$ ；
- `daxpy(x,y,a=2)` 计算 $y \leftarrow 2x + y$ ；
- `dcopy` 复制向量，`dswap` 交换两个向量；
- `dnrm2(x)` 稳健地计算 $\|x\|_2$ 。

Level 2 包含矩阵-向量与秩一更新。页中的

```
y_mv = blas.dgemv(1.0, A, x, beta=0.5, y=y.copy())
```

对应

$$y_{mv} = Ax + 0.5y. \quad (1.91)$$

`dtrmv` 计算三角矩阵乘向量; `dger(1.0,x,y,a=A0)` 计算

$$A_0 \leftarrow A_0 + xy^T. \quad (1.92)$$

Level 3 是矩阵-矩阵操作: `dgemm(1.0,A,B)` 计算 AB , `dtrmm` 计算三角矩阵与矩阵的乘积, `dsyrk(1.0,A)` 更新对称矩阵的一个三角部分, 对应 AA^T 的上三角。随后用该三角部分及其严格转置恢复完整对称矩阵, 避免对角线重复相加。

Level 3 的算术强度通常高于 Level 1/2, 因此更容易发挥缓存和并行硬件。页末 **TBC** 表示代码在第 1.76 页继续。

Tip

本页要点: 将 BLAS 调用与对应的向量、矩阵运算逐一对照, 区分各级操作。

1.76 Examples of SciPy, Continued

一般线性系统部分先调用

```
lu, piv, info = lapack.dgetrf(A.copy())
```

计算带部分主元的 LU 分解。`lu` 在同一数组中存储 L 和 U , `piv` 存储主元交换, `info` 报告参数或奇异主元信息。随后

```
solution_lu, info = lapack.dgetrs(lu, piv, b.copy())
```

必须使用同一次分解返回的 `piv` 求解。`assert_close(A @ solution_lu, b)` 检查计算残量; 更严格时应报告缩放残量

$$\frac{\|b - A\hat{x}\|}{\|A\| \|\hat{x}\| + \|b\|}. \quad (1.93)$$

Cholesky 示例构造

$$A_{\text{spd}} = M^T M + 0.5I. \quad (1.94)$$

对任意非零 z ,

$$z^T A_{\text{spd}} z = \|Mz\|_2^2 + 0.5\|z\|_2^2 > 0, \quad (1.95)$$

所以矩阵确为对称正定。`dpotrf(...,lower=True)` 计算下三角 Cholesky 因子, `dpotrs` 用同一三角约定求解。每次调用后检查 `info`, 可避免把分解失败后的数组继续当作有效因子使用。

Tip

本页要点: 分解与求解必须使用一致的主元或三角约定, 并检查返回状态和残量。

1.77 Finite Element Circus Poem

Rick Falk 于 1996 年秋写下短诗《Ivo's Lament》（《伊沃叹》）。诗借 Ivo Babuška 的口吻，打趣有限元学术聚会 Finite Element Circus 的变化：从抽签排报告顺序，到电脑生成随机数，再设想未来只靠电脑和电话相聚。

Tip

本页要点： 计算机不仅改变计算方法，也改变学术交流的方式。

本章小结

Important

第 1 章回顾

1. **先认清要解的矩阵问题。** 线性方程组、最小二乘、特征值与奇异值问题，是本课程的几条主线。回归、PCA、谱聚类 and PageRank 说明，看起来不同的应用可以归结为这些问题；但模型、约束和矩阵结构仍决定了具体的算法选择。
2. **数学等价不等于计算等价。** 浮点表示只有有限精度，运算次序、舍入和中间量都会影响结果。PCA 的协方差 EVD 与数据 SVD 在数学上对应同一问题，形成协方差矩阵却可能损失较弱方向的信息；混合精度也需要可靠的误差控制，不能只比较运算速度。
3. **存储方式是算法的一部分。** 稀疏矩阵应按非零结构存储和计算，格式选择还要考虑访问顺序、填充和并行规则性。CSR、COO 等格式各有适用场景，不能只凭非零元数量判断运行效率。
4. **性能受限于计算与数据移动。** 缓存、分块和数据复用会改变实际流量；Roofline 用算术强度联系计算吞吐与带宽上界。SpMV 往往受访存限制，矩阵乘法更容易复用数据，因此 FLOP 更少不一定耗时更短，还要计入通信延迟与同步。
5. **增加处理器不保证同比加速。** 强扩展固定问题规模，弱扩展随处理器数扩大问题；Amdahl 与 Gustafson—Barsis 分析的是不同的比较方式。负载不均衡、串行部分和通信都会限制扩展。
6. **可信的结果需要分层检查。** 区分模型误差、离散与求解误差、浮点误差，以及另行排查的程序和硬件故障。成熟数值库可以提供稳定算法和优化实现，但仍须选对例程、检查返回状态并验证结果；历史案例提醒我们，也要审视输入数据和预期答案本身。

读完本章，应能回答： 为什么运算次数更少或机器峰值更高，程序仍可能更慢？当结果偏离预期时，你会怎样区分模型、数值方法和实现的问题？