计算物理讲义-第一章

冯旭

1 数值计算的基础

过去我给学生上《理论力学》的时候，用到过朗道写的《力学》。《力学》这本书是朗道与栗弗席兹合著的《理论物理学教程》中的第一本，倾注了朗道很多的心血，后续很多本都是朗道列大纲，由栗弗席兹执笔。只有《力学》这本书，可以说是朗道亲手写的。在这本书里面，朗道感叹说"理论物理的目标是建立物理定律，即建立物理量之间的关系。确定物理量的具体数值一般不是理论物理的任务，实验在处理这些问题方面相对比较容易，因为在绝大多数情况下，实验不必进行花费大量时间和人力的计算。当然，用理论可以直接算出数值的简单的情况除外"。在朗道看来，从理论出发算出数值的结果是很困难，朗道之所以有这样的论断是因为他写《力学》这本书的时候是在上世纪中叶，那时的计算手段很落后，数值计算比实验还要困难。但我们今天借助庞大的计算机，可以做很多前人想都无法想象的事情。

我们这门课如果是数学系开可能会被称为 Numerical analysis，如果是计算机系开，可能会被称为 Numerical computing，我们物理学院开，称之为计算物理。事实上，随着计算机技术的发展，计算物理是作为实验物理、理论物理之外的第三大分支存在的。它的作用是让我们通过数值的方法去解决我们在科研、工程中所遇到的各种物理问题。这与我们学院开的另外一门课，数学物理方法正好相辅相成。数理方法是要通过解析的手段去解决物理问题。但我们知道，很多问题，甚至可以说绝大多数问题，都是没有解析解的。比方说，一个一元五次方程，根据阿贝尔定理，五次及更高次代数方程没有通用的求根公式，只能通过数值算法得到高精度的近似解。

IEEE，也就是电器与电子工程师学会主办的《Computing in science & engineering》杂志曾经选出过 20 世纪的十大算法，也就是对科学和工程发展影响最大的十个算法，按时间顺序排列，分别是

1. 1946 年, 美国 Los Alamos 国家实验室的 John Von Neumann(冯诺伊曼)、S. Ulam、N. Metropolis 开创的 Metropolis 算法，也就是蒙特卡罗方法。

2. 1947 年，美国 RAND 公司的 G. Dantzig 开创的 Simplex 算法 (解线性规划问题)

3. 1950 年, 美国国家标准局数值分析研究所的 M. Hestenes、E. Stiefel 和 C. Lanczos 开创的 Krylov 子空间迭代法 (解稀疏矩阵本征值和本征矢量)

4. 1951 年，美国 Oak Ridge 国家实验室的 A. Householder(豪斯霍尔德) 形式化的矩阵计算的分解方法，也称 Householder 约化

5. 1957 年，美国 IBM 公司的 J. Backus 领导开发的 FORTRAN 最优编译器算法

6. 1959 - 1961 年，英国伦敦 Ferranti 公司的 J. Francis 开创的 QR 算法，是一种计算矩阵特征值的稳定算法 (如果待求的矩阵不是特别巨大 ($n \leq 3000$)，那么 QR 算法实际上是最为合适的算法。)

7. 1962 年, 英国伦敦 Elloit Brothers 公司的 T. Hoare 提出 Quicksort 算法, 也即快速排序算法 (这个算法可能大家比较熟悉，从数列中挑出一个元素，比这个基准元素小的数放在基准前面，大的数放在基准后面，然后对前后两部分数据在进行同样的操作，一直递归操作下去就行了。)

8. 1965 年，美国 IBM 公司的 J. Cooley 和普林斯顿大学及美国贝尔实验室的 J. Tukey 共同提出的快速傅立叶变换算法，即 FFT 算法

9. 1977 年，美国 Brigham Young 大学的 H. Ferguson 和 R. Forcade 提出的整数检测算法 (integer relation detection)

10. 1987 年，美国耶鲁大学的 L. Greengard 和 V. Rokhin 发明的快速多极算法 (fast multipole)

对于这 10 大算法，我们课上讲的内容大概会涵盖 10 大算法的一半，比方蒙特卡罗方法那个章节，会讲到 Metropololis 算法；第六章，讲到本征值和本征矢量数值求解时，会讲到 QR 算法、Householder 约化，以及讲稀疏矩阵本征值问题时，会讲到 Krylov 子空间迭代法。另外在课程的下半节，会讲到快速傅立叶变换。

今天，我们课程的重点是介绍数值计算的一些基础的概念。这包括计算机中的浮点数系统、计算的舍入误差、算法的稳定性等。

1.1 计算机表达的数

由于计算机都只有有限的内存，因此它只能够表达有限多的数。这不仅仅包括实数，也包括整数。目前的经典计算机本质上都是通过记录二进制的数来记录所有的数的。也就是说，最基本的单元是所谓的位 (bit)，也就是比特，它是 binary digit 的缩写，它只有两个可能取值 0 和 1。每 8 个位构成一个字节 (byte)。我们需要在一个字节内表示超过 100 种状态，包括常用数字，大小写字母，和打印机上的其他控制符号等等。一个字节等于 8 比特，能表示 256 种状态，能融入 ASCII 码。所以一字节也就是计算机能处理的最小的一段数据量了。历史上好像是一个德国裔的美国计算机科学家，叫 Werner Buchholz，管一字节叫 1 bite，就是这段信息量刚好能被一口吃掉。但是 bite 如果少写个 e，就很容易和比特单位搞混了，后来就用谐音，改为 byte。

通常的的整数由 4 个 byte 构成，也就是说是 32 位。当然，如果嫌这个不够，也可以用 64 位 (也就是 8 个 byte)。例如在 C 语言之中，int 型的整数就是 4 个字节而所谓的 long int 型的整数则是 8 个字节。对任何一个 4 字节 (32bit) 的非负整数都可以在二进制中表达为,

$$n = \sum_{i=0}^{31} b_i 2^i \tag{1}$$

正整数的范围因此从 0 到 $2^{32} - 1$。如果我们还希望表达负整数，我们只好牺牲一位来标记整数的正负，真正用于记录数字的位数减少到 31 位。整数的最大特点是，只要在其定义域之内，机器对于整数的表达是严格的。而且整数之间的运算也是严格的 (只要结果不超出其定义域)。如果我们需要扩大整数的定义域，我们就需要启用更多位的整数。

对于实数，一般采用所谓的浮点数来表示。显然，由于实数是无穷多数构成的连续集合，因此在计算机中能够表达的实数只能是其中的一个有限的部分。这就是浮点数系统。它保证了在任何一个实数的足够接近的邻域内总能找到一个浮点数来近似代表相应的实数。计算机中可以表达出来的那些数称为可表达的实数，这个集合的大小取决于我们选择用多少位来表达一个实数。所有的那些不可表达的实数必须经过舍入的操作，用它附近的一个可表达的实数来近似代替，相应产生的误差一般被称为舍入误差。特别地，我们可以定义一个量叫机器精度:

$$\epsilon_M = \min\{g \in A | 1 \oplus g > 1,\, g > 0\} \tag{2}$$

其中 A 表示计算机中所有可以表示的实数而 $\oplus$ 表示计算机中实现的"加法"(里面包含了对结果的舍入)。显然，机器精度的数值依赖于我们在一个实数上愿意投入多少内存。例如，对于通常的单精度的实数 (4 个字节) 的，$\epsilon_M \sim 10^{-7}$，而对于双精度的实数 (8 个字节) 来说，$\epsilon_M \sim 10^{-16}$。后面我们会讲到这两个精度是怎么出来的。

上面的描述是对于浮点数系统的一个比较粗略的概述。如果说得更仔细一些的话，我们选取一个正的自然数 $\beta \in N$ 为底来表达的话 (通常 $\beta = 2$)，那么一般实数 x 在以 β 为底的系统中可以用下列整数部分和小数部分分别具有 $(n+1)$ 位和 m 位的小数表达,

$$x_\beta = (-)^s \left[x_n x_{n-1} \cdots x_1 x_0 . x_{-1} x_{-2} \cdots x_{-m}\right], \quad x_n \neq 0 \tag{3}$$

其中 $s = 0, 1$ 称为符号位，用以标记该数是正还是负。这个表达称为实数 x(以 β 为底) 的位置表示。实数的位置表示的另外一个等价写法是

$$x_\beta = (-)^s \left[\sum_{k=-m}^{n} x_k \beta^k\right]. \tag{4}$$

事实上，更为经济与方便的计数方法是将实数提出一个公共的因子,

$$x_\beta = (-)^s \cdot (0.a_1 a_2 \cdots a_t) \cdot \beta^e = (-)^s \cdot m \cdot \beta^{e-t}. \tag{5}$$

这个表示称为实数的浮点数表示，其中的 s 仍然是符号位，整数 $m = a_1 a_2 \cdots a_t$ 是一个 t 位的整数称为尾数，显然 $0 \leq m \leq \beta^t - 1$; 整数 e 则称为指数，它满足 $L \leq e \leq U$，一般选取 $L < 0$ 为一个负整数，$U > 0$ 为一个正整数。于是，对于一个占用 N 位的实数而言，计算机中存储时会让符号位占一位，有效数字 a_i，$i = 1, \cdots, t$ 占 t 位，指数 e 则占据剩下的 $N-t-1$ 位。

常见的情形是单精度的浮点数 (占据 32 位) 和双精度的浮点数 (占据 64 位)，它们除了符号位占一位之外，分别具有 $t = 23$ 和 $t = 52$ 位来存储有效数字，而存储指数的位数则分别是 8 和 11 位。一般来说数字 0 会单独有一个特别的表示。因此，一个一般的浮点数系统 F 可以用它的基底 β、尾数的位数 t、指数 e 的上下限 L 和 U 标记为,

$$F(\beta, t, L, U) = \{0\} \cup \left[x \in R:\, x = (-)^s \beta^e \sum_{k=1}^{t} a_k \beta^{-k}\right]. \tag{6}$$

由于我们总是可以移动小数点并相应地改变指数 e 的数值，因此为了保证数字表示的唯一性，一般假设 $a_1 \neq 0$，否则我们可以向右移动小数点一位并减少 e 一个单位即可。类似的，一般假设 $m \geq \beta^{t-1}$。经过这样约定的浮点数表示称为规范化的浮点数表达。例如，在没有规范化的以 10 为底的浮点数 $F(10, 4, -1, 4)$ 系统中，实数 1 可以有下列四种表达,

$$0.1000 \cdot 10^1, \quad 0.0100 \cdot 10^2, \quad 0.0010 \cdot 10^3, \quad 0.0001 \cdot 10^4 \tag{7}$$

但是经过规范化后，只有第一个是符合要求的。容易证明，如果 $x \in F$，那么我们必定有 $-x \in F$。另外，除了 0 这个特殊字符之外，还有另外两个特殊字符

- Inf: 表示无穷，由有限数字除以 0 产生，比如 1/0
- NaN: 意思是 Not a number, 由没有定义或者不确定的操作产生, 比方说 0/0, 0*Inf 或者 Inf/Inf。

还有一个值得注意的事情，就是我们的数据存放在内存或者数据文件里，是采用一个什么样的存放方式，是顺序存放数据信息还是倒序存放，又被分为 big endian (低地址存放最高有效字节) 和 little endian(低地址存放最低有效字节)。字节序的问题，牵涉到 CPU 的两大派系。那就是 PowerPC 系列的 CPU 采用 big endian 方式存储数据，Intel 的 x86 系列 CPU 则采用 little endian 方式存储数据。有一个简单的方法来测试你的笔记本电脑是那种数据存放方式。

$$\begin{aligned}&\text{int i = 1;}\\&\text{char *p = (char *) \&i;}\\&\text{if (*p == 1) “Little Endian”}\\&\text{else “Big Endian”}\end{aligned} \tag{8}$$

比如我用的 mac 笔记本电脑就是 Little Endian。当我们面对一堆存储好的数据时，我们还是要遵循数据的存储规则，否则我们读出来的数据很有可能是错的。

容易发现，一个规范化表达的浮点数系统 $F(\beta, t, L, U)$ 中的所有可表达的实数的绝对值一定介于 $x_{\min}$ 和 $x_{\max}$ 之间。对于上限来讲

$$e = U, \quad 0.a_1a_2\cdots a_t = 0.(\beta-1)\cdots(\beta-1) = 1-\beta^{-t}, \quad \Rightarrow \quad x_{max} = \beta^U(1-\beta^{-t}) \tag{9}$$

对于下限

$$e = L, \quad 0.a_1a_2\cdots a_t = 0.1, \quad \Rightarrow \quad x_{\min} = \beta^{L-1} \tag{10}$$

总结起来就是

$$x_{\min} \equiv \beta^{L-1} \leq |x| \leq \beta^U(1-\beta^{-t}) \equiv x_{max} \tag{11}$$

为了能够表达更多 (特别是绝对值更小) 的数，我们会在 $e = L$ 时的放弃要求 $a_1 \neq 0$(由于 $e = L$，因此这不会破坏数表示的唯一性，只会加入一系列更小的数) 。这可以产生在 $(-\beta^{L-1}, \beta^{L-1})$ 中的实数。这样的操作叫做次规范化，使得最小绝对值下降为 β^{L-t}。如果比这个还要小的数出现，机器会产生所谓的下溢。虽然次规范化增加了所表示数的范围，但新增加的数的精度要低于其他浮点数，因为它们的有效数字较少。

有了浮点数系统的定义，下面我们来回顾一下机器精度

$$\epsilon_M = \min\{g \in A | 1 \oplus g > 1,\, g > 0\} \tag{12}$$

1 可以写成

$$0.10000 \times \beta^1 \tag{13}$$

最小的 g 应该为

$$\beta^{-t} \times \beta^1 = \beta^{1-t} \tag{14}$$

所以我们得到 $\epsilon_M = \beta^{1-t}$。对于二进制来说，单精度数据，如果取 t=23，那么 $\epsilon_M \sim 10^{-7}$；双精度数据，如果 t=52，那么 $\epsilon_M \sim 10^{-16}$。

浮点数的分布当然不是连续的。事实上也不是均匀的。假定有两个非零的浮点数 $x, y \in F$，两者最近的距离一定介于 $\beta^{-1}\epsilon_M|x|$ 和 $\epsilon_M|x|$ 之间。为了更清楚得看到这一点，我们可以将 x 写为

$$x = (-)^s \cdot (0.a_1a_2\cdots a_t) \cdot \beta^e \tag{15}$$

与 x 最邻近的 y 为

$$y = (-)^s \cdot (0.a_1a_2\cdots (a_t+1)) \cdot \beta^e \tag{16}$$

所以

$$|y - x| = \beta^{e-t} = \epsilon_M \beta^{e-1}, \tag{17}$$

因为 $|x|$ 的范围为 $\beta^{e-1} \le |x| < \beta^e$，所以 β^{e-1} 也可以写成

$$|x|\beta^{-1} < \beta^{e-1} \le |x| \tag{18}$$

因此 x, y 最近的距离一定介于 $\beta^{-1}\epsilon_M|x|$ 和 $\epsilon_M|x|$ 之间。由以上分析我们可以看到，对于浮点数系统 $F(\beta, t, L, U)$，β 代表的是二进制或 N 进制，t 也就是尾数的位数代表数据能达到的精度，L 和 U 代表了这套浮点数系统所表达的数的上限和下限。

对于那些不在系统 F 中的实数怎么办呢? 计算机必须能够处理它们。最常见的操作称为舍入。这个操作是从一个一般的实数 $x \in R$ 到某个浮点数系统 F 的映射，我们记为 $fl(\cdot)$。这个映射必须尽可能能够准确地反映原来的实数的大小，同时还不能改变实数的顺序。最简单的舍入方法就是利用大家熟悉的"四舍五入"原则。我们首先注意到任何的实数实际上都可以用公式 (5) 来表达出来，只要我们允许它的尾数可以有无穷多位。但是对一个给定的计算机浮点数系统，我们固定只能够保留 t 位的尾数。采用舍入的处理之后，虽然我们损失掉了表达一个数的精确性，但以此为代价，我们能够表达的实数范围大大扩大了。绝对值过大的实数仍然无法在计算机中表达。具体来说，如果 $|x| > x_{\max}$，那么我们无法将其舍入到任何一个合理的可表达的实数。这时候计算机会产生一个溢出，程序的运行一般也会终止。类似的，如果某个实数的绝对值过小，$x < x_{\min}$，这时候产生的一般称为下溢，此时往往系统会将其舍入为 0。当然，除了四舍五入之外，还有一种舍入称为截断舍入，就是将 t 位往后的尾数都截去。

以下的分析主要采用四舍五入原则，对舍入造成的误差可以进行量化的估计。对于任意 $x \in R$，且满足 $x_{\min} \le |x| \le x_{\max}$，我们一定有，

$$fl(x) = x(1+\delta), \quad |\delta| \le \frac{1}{2}\epsilon_M = \frac{1}{2}\beta^{1-t} \tag{19}$$

因此我们有时候又称 $\epsilon_M/2$ 为舍入误差单位。这意味着一个实数与它的舍入值之间的误差是完全在控制之中的:

$$E_{\text{rel}}(x) = \frac{|x - fl(x)|}{|x|} \le \frac{1}{2}\epsilon_M, \quad E(x) = |x - fl(x)| \le \frac{1}{2}\beta^{e-t} \tag{20}$$

从这个角度来说，浮点数的表示以及运算都是有误差的。浮点数比整数更为糟糕的是，即使是在其定义域内，浮点数也仅仅是某个实数的近似表示而不是严格表示，而且两个实数之间的运算的结果，即使它不超出定义域，通常也不是一个可表达的实数，需要将其经过舍入过程转换为一个可表达的实数。一般来说，由于舍入问题的存在，计算机中的基本运算比如，加减法、乘法等等，并不满足原先数学中的结合律、分配率等基本规律。例如，对于三个计算机中可表达的数 $a, b, c \in A$，在严格的数学

中，$(a+b)+c=a+(b+c)=a+b+c$，但是在机器的运算中，$(a\oplus b)\oplus c$ 和 $a\oplus(b\oplus c)$ 一般是不相等的。但是，舍入方法的实现应当尽量确保在一次基本运算后造成的舍入误差仅仅在一两个机器精度的范围之内。即使这一点得到满足，由于计算机在复杂的算法中往往需要进行 N 多次的计算，因此我们必须考虑这种误差的传递和累计效应。这使得我们在数值计算中对算法的稳定性的要求提高了。

1.2 误差的种类

首先，我们讲一下数据传递误差与计算误差。考虑一个简单的计算问题：函数求值。假设有函数 $f:R\to R$，x 为函数输入参数的准确值，则准确结果为 $f(x)$。由于数据误差，实际使用的输入值为 $\hat{x}$，同时计算过程也存在近似，使得最后结果为 $\hat{f}(\hat{x})$，其中，$\hat{f}$ 代表计算过程中的近似，因此，总误差为

$$\hat{f}(\hat{x})-f(x)=[\hat{f}(\hat{x})-f(\hat{x})]+[f(\hat{x})-f(x)] \tag{21}$$

其中第一项 $\hat{f}(\hat{x})-f(\hat{x})$ 为输入同为 $\hat{x}$ 时计算过程的误差，它是一个单纯的计算误差。这种误差包括截断误差和我们之前提到过的舍入误差。打个比方说，我们要计算的函数是以 n 阶泰勒级数展开的，我们不可能计算无穷阶级数，如果级数展开收敛性够好，那么在某一处给予截断就好。下面我们来看一个求一阶导数的差分近似的例子，来看一下截断误差和舍入误差分别是如何起作用的。

$$f'(x)\approx\frac{f(x+h)-f(x)}{h} \tag{22}$$

由泰勒展开得到

$$f(x+h)=f(x)+h\cdot f'(x)+\frac{h^2}{2}f''(\xi),\quad \xi\in[x,x+h] \tag{23}$$

假设 M 是 $|f''(\xi)|$ 的上限，那么这个一阶差分的截断误差为 $M\frac{h}{2}$。我们假设，计算函数 $f(x)$ 的过程中，舍入误差为 ε，因为在差分中函数被计算了两次，因此舍入误差最后进入到 $f'(x)$ 中的大小为 $2\varepsilon/h$。所以总的误差为

$$\varepsilon_{\text{tot}}=\frac{Mh}{2}+\frac{2\varepsilon}{h} \tag{24}$$

所以步长 h 并不是越大越好，也不是越小越好；大了，截断误差就会比较大，小了，舍入误差会被 $1/h$ 放大。当 $h=2\sqrt{\varepsilon/M}$ 时总的计算误差达到最小。这就是我们为什么要学习误差分析：所有的算法都是围绕着误差尽可能小、效率尽可能高来设计的。

第二项 $f(\hat{x})-f(x)$ 是由于输入数据误差经过精确的函数求值过程产生的误差，即数据传递误差。数据传递误差是由问题的敏感性决定的。我们定义问题的敏感性，是指输入数据的扰动对问题解的影响程度的大小。敏感性有时候也称问题的病态性。如果输入数据的相对变化引起解的相对变化不是很大，则称这个问题是不敏感的或者良态的 (well-conditioned)；反之，如果解的相对变化远远超过输入数据的变化，则称这个问题是敏感的，或者病态的 (ill-conditioned)。

为了定量地分析问题的敏感性，我们会引入问题的条件数概念。问题的条件数定义为

$$\text{cond}=\frac{||\text{问题的解的相对变化量}||}{||\text{输入数据的相对变化量}||} \tag{25}$$

这里的算符 $||\cdot||$ 表示范数，它用于度量一个量的大小。对于一个实数或复数来讲，范数就是这个数的模。对于其他类型的量，比如说向量、矩阵和函数，范数的具体定义我们在后面会陆续讲到。以函数求值问题为例，其条件数有下面的计算公式

$$\text{cond}=\left|\frac{[f(\hat{x})-f(x)]/f(x)}{(\hat{x}-x)/x}\right| \tag{26}$$

因为这里，我们分析条件数是针对问题而言的，并不涉及实现 $f(x)$ 的具体方法，因此，公式里使用的是准确函数 $f(x)$，而不是实际计算对应的近似函数 $\hat{f}(x)$。

输入数据扰动和引起解的相对误差之间满足一个近似的不等式

$$||\text{数据传递的相对误差}|| \leq \text{cond} \times ||\text{输入数据的相对变化}|| \tag{27}$$

进一步分析函数求值问题的条件数。假设函数 f 可微，则

$$f(\hat{x}) - f(x) \approx f'(x)(\hat{x} - x) \tag{28}$$

所以

$$\text{cond} = \left| \frac{[f(\hat{x}) - f(x)]/f(x)}{(\hat{x} - x)/x} \right| \approx \left| \frac{xf'(x)}{f(x)} \right| \tag{29}$$

这就得到了函数求值问题的条件数的估计式。我们下面来看一个例题，用条件数估计计算 $f(x) = \tan x$ 的问题敏感性。因为 $f'(x) = 1 + \tan^2 x$，有

$$\text{cond} = \left| \frac{xf'(x)}{f(x)} \right| = \left| \frac{x}{\sin x \cos x} \right| \tag{30}$$

当 x 接近于 $\pi/2$ 的时候，比方说 $\pi/2 \approx 1.570796$，如果我们在 $x = 1.57079$ 附近计算，条件数大概是 2.5×10^5。因此在 $x = \pi/2$ 附近计算 $\tan x$ 是个很敏感的问题。

1.3 舍入误差与算法的稳定性

既然计算机中的实数都是利用不精确的浮点数来表达的，我们特别需要讨论算法的稳定性，或者说初始值中可能的误差，随着算法的运行是如何传递到最终的结果中去的。

我们首先来明确一下什么是算法。在计算数学中，从 n 个已知的数据 $x \in R^n$ 出发，最终获得最终 m 个结果 $y \in R^m$ 的一系列有限多次的操作之集合就构成了一个算法。为了获得一个具体问题的解可以有许多种算法。那么什么是一个好的算法呢。首先，一个好的算法首先必须是准确的 (当然，准确只是一个比较模糊的描述词语，具体的精度依赖于具体的问题); 也就是说，它可以给出问题的足够精确的近似解。其次，它必须是在计算资源方面比较快捷和有效的。当然正确性仍然是首要考虑。

既然每次计算机的操作往往都伴随着舍入误差，我们就需要分析这个误差是如何随着算法的发展而传递的。以一个简单的例子来说，假定我们需要计算三个实数 a, b, c 的和，即 $a + b + c$，我们有两个算法来进行这个计算，即 $(a + b) + c$ 和 $a + (b + c)$。这两种算法对应的误差可以分析如下。

对 $(a + b) + c$ 而言，我们有

$$a \oplus b = (a + b)(1 + \varepsilon_1), \quad (a \oplus b) \oplus c = [(a + b)(1 + \varepsilon_1) + c](1 + \varepsilon_2) \tag{31}$$

我们可以忽略掉二阶的无穷小量得到这个算法的结果为,

$$(a \oplus b) \oplus c = (a + b + c)\left[1 + \frac{a + b}{a + b + c}\varepsilon_1 + \varepsilon_2\right] \tag{32}$$

显然，另外一种算法的误差估计就是将 a 与 c 互换，即,

$$(b \oplus c) \oplus a = (a + b + c)\left[1 + \frac{c + b}{a + b + c}\varepsilon_1 + \varepsilon_2\right] \tag{33}$$

因此，两种算法有一个部分是相同的，即 ε_2 的部分；但是另外一个部分的放大因子分别为 $(a+b)/(a+b+c)$ 和 $(b+c)/(a+b+c)$。注意，这两个因子可以相差很远！事实上，两者之比为

$$\left|\frac{a+b}{c+b}\right| \tag{34}$$

这意味着，如果某两个之和恰好特别小，那么先进行这两个数的加法的计算是更好的。

下面，用几个例子来说明如何提高算法稳定性。

- 例子一、避免大数相消现象的出现
  浮点数系统的一个重要问题是存在抵消现象，也就是两个符号相同、值相近的 p 位数相减可能使结果的有效数字远小于 p 位。比如两个十进制的 6 位精度数，$x = 1.92305 \times 10^3$ 和 $y = 1.92137 \times 10^3$，则 $x - y = 1.68$，这一步减法计算过程中未发生舍入，但它的结果却只有 3 位有效数字。舍入是丢弃末尾数位上的数字，而抵消丢失的是高位数字包含的信息，有时候危害更大。我们下面来看一个例子。当 $x < 0$，且 $|x|$ 较大时，利用公式

  $$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{35}$$

  截断前 n 项计算 e^x，会发生严重的抵消，使数值结果误差很大。由于 $|x|$ 较大，则前若干项的求和式中每一项的绝对值都远大于准确结果，当按自然顺序逐项累加，每次计算都是将较大但符号相反的数做加法，等价于符号相同的数相减，因此造成抵消。随着舍入误差的逐渐积累，将可能远大于准确结果，使得结果完全错误。假设 $x = -20$，前 n 项求和的计算值为 $S_n(x)$，随着逐项累加，$S_{96}(x) = 5.62188 \times 10^{-9}$，此时下一步加的项为 $x^{96}/96! = 7.98930 \times 10^{-26}$，它与 $S_{96}(x)$ 的比值已经小于 $\varepsilon_M/2$，因此后续的计算都不会改变部分和的计算值，而 e^{-20} 的准确值为 2.061×10^{-9}，说明计算结果完全错误。

  这里需要说明的是，当 $x > 0$ 时，上述求和式子中每项都大于 0，不会有抵消现象，计算是稳定的。因此，对于 $x < 0$ 的情况，通过公式 $1/e^{-x}$ 来计算 e^x 是有效、可行的算法。

- 例子二、避免出现上溢或下溢
  计算分式

  $$y = \frac{x_1}{x_2 \cdot x_3 \cdot \cdots \cdot x_n} \tag{36}$$

  其中 x_2 比 x_1 小很多，$|x_1/x_2| > 3.404 \times 10^{38}$，那么采用单精度浮点数计算 x_1/x_2 就会发生上溢。一般情况下，假如 y 的准确值不会超过上溢值，为了避免上溢，应先计算分母的值 $z = x_2 \cdot x_3 \cdot \cdots \cdot x_n$，然后再计算 $y = x_1/z$。

- 例子三、避免大数吃掉小数
  我们来看在浮点数系统中计算级数

  $$\sum_{n=1}^{\infty} 1/n \tag{37}$$

  会得到什么样的结果。首先我们知道这个级数是发散的，但在浮点算术系统中却不是这个样子。粗略分析，可能有两种情况，一是部分和非常大以至于发生上溢，二是 $1/n$ 变得越来越小，产生下溢；但事实上，在达到这两种情况之前，计算结果就已经不再变化了，这是因为，一旦增加量

$1/n$ 与部分和 $\sum_{k=1}^{n-1}\frac{1}{k}$ 的值相差悬殊，它们的和就停止变化了，也就是说当条件

$$\frac{1}{n} \leq \frac{\epsilon_M}{2}\sum_{k=1}^{n-1}\frac{1}{k} \tag{38}$$

由于大数吃小数，后面的级数项都不再起作用。

- 例子四、尽量减少运算次数
  比方说我们要进行多项式运算

$$P_n(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n \tag{39}$$

如果直接计算 a_ix^{n-i} 再逐项相加，一共需要 $\sum_{i=1}^{n} = \frac{n(n+1)}{2}$ 次乘法和 n 次加法。一种简单的改进可以是

$$b := a_0 \tag{40}$$
$$\text{For } i = 1, 2, \cdots, n \tag{41}$$
$$b := xb + a_i \tag{42}$$
$$\text{End} \tag{43}$$
$$P_n(x) := b \tag{44}$$

这就可以有效减少运算次数

下面做一下小结：要保证一个数值计算问题结果的准确性，通常要依次考虑三点

- 病态性：这是与待求解的数学问题的性质有关，与具体算法无关，最先考虑
- 稳定性：这是数值算法的性质，应尽量选择稳定性好的算法，减少计算中误差的扩大
- 通过定性分析控制舍入误差，尽量避免例子里出现的各种问题，如果内存允许，可以考虑采用位数较多的高精度浮点数。

要控制舍入误差的影响，除了分配更多的内存，提高浮点数的精度，还有一种方法就是所谓的补偿求和方法，也称 Kahan 求和算法。例如我们需要计算求和：

$$S_n = \sum_{i=1}^{n} x_i \tag{45}$$

其中 $n \gg 1$，例如 $n \sim O(10^7)$ 或更大。一般来说，每一次加法计算机都会进行舍入。舍入过程一定造成误差 ϵ。对于一个一般的 n 项求和，如果每次加法造成的舍入误差为 ϵ，那么通常 S_n 的舍入误差最大可能大约是 $O(n\epsilon)$。也就是说，对于单精度数的求和，当 $n \sim 10^6 - 10^7$，通常的求和造成的舍入误差已经相当可观。一个通常的加法的累加程序大体上如下

$$\begin{aligned} &s = 0.0 \\ &\text{For } i = 1, \cdots, n \\ &\qquad s = s + x_i \\ &\text{End} \end{aligned} \tag{46}$$

当所有数据累加完毕，输出 $S_n = s$。当 n 很大时，这种通常的加法往往会造成明显的误差。其主要原因是，通常求和的中间项 s 会随着求和项数的增加而慢慢变大，而需要加上的 x_i 的低位信息很有可能会被舍入。这就是典型的一个大数加一个小数时造成的对小数的误差。也就是说，如果在求和的中间过程中承载数据的变量 s 没有足够多的位数，那么机器自然就会舍弃那些它认为“不重要”的位数。因此，一个最为直接的解决方法就是让变量 s 的位数足够地多。比如原先是单精度的，现在用双精度。当然，有的时候我们必须采用某种固定的位数。例如，如果我们由于某种原因必须使得 s 是单精度的。可是我们又需要进行大量的数的求和，怎么办呢? 这时可以利用所谓补偿求和 (又称为 Kahan 求和) 的方法。仅仅是稍微修改了一下上述程序，试图在求和中间的每一步都保留待加入的数 x_i 的低位的信息。Kahan 求和算法的伪码给出如下：

$s = 0.0, \quad c = 0.0$

For $i = 1, \cdots, n$

$y = x_i - c$ (一开始的时候，这一步什么都没干。)

$t = s + y$ (我们考虑 s 会比 y 大得多，那么实际上加法过程中 y 的低位信息丢失了。)

$c = (t - s) - y$ (我们先计算 $t - s$，这步相减得到实际上是 y 的高位的信息)

(再减 y 得到是 $-y$ 的低位信息。在循环的过程中 $x_i - c$ 会把丢失掉的 y 低位信息补正)

sum $= t$

End (47)

可以证明的是，这种补偿求和的方法基本上可以使得 S_n 的误差达到 ϵ 的量级，也就是说，它完全不依赖于求和的项数，舍入误差基本上就是每个待加达到数的原始精度。这当然是非常理想的。如果还希望进一步高精度，则只能够通过提高每个数据 x_i 的精度来实现了。