附录 D

What Every Computer Scientist Should Know About Floating-Point Arithmetic

注 – 本附录是对论文《What Every Computer Scientist Should Know About Floating-Point Arithmetic》（作者：David Goldberg，发表于 1991 年 3 月号的《Computing Surveys》）进行编辑之后的重印版本。

D.1 摘要

许多人认为浮点运算是一个深奥的主题。这相当令人吃惊，因为浮点在计算机系统中是普遍存在的。几乎每种语言都有浮点数据类型；从 PC 到超级计算机都有浮点加速器；多数编译器可随时进行编译浮点算法；而且实际上，每种操作系统都必须对浮点异常（如溢出）作出响应。本文将为您提供一个教程，涉及的方面包含对计算机系统设计人员产生直接影响的浮点运算信息。它首先介绍有关浮点表示和舍入误差的背景知识，然后讨论 IEEE 浮点标准，最后列举了许多示例来说明计算机生成器如何更好地支持浮点。

类别和主题描述符：（主要） C.0 [计算机系统组织]：概论 —— **指令集设计**； D.3.4 [程序设计语言]：处理器 —— **编译器**，**优化**； G.1.0 [数值分析]：概论 —— **计算机运算**，**错误分析**，**数值算法**（次要）

D.2.1 [软件工程]：要求 / 规范 —— **语言**； D.3.4 程序设计语言]：正式定义和理论 —— **语义**； D.4.1 操作系统]：进程管理 —— **同步**。

一般术语：算法，设计，语言

其他关键字 / 词：非规格化数值，异常，浮点，浮点标准，渐进下溢，保护数位，NaN，溢出，相对误差、舍入误差，舍入模式， ulp，下溢。

D.2 简介

计算机系统的生成器经常需要有关浮点运算的信息。但是，有关这方面的详细信息来源非常少。有关此主题的书目非常少，而且其中的一部《*Floating-Point Computation*》（作者：Pat Sterbenz）现已绝版。本文提供的教程包含与系统构建直接相关的浮点运算（以下简称为**浮点**）信息。它由三节组成（这三节并不完全相关）。第一节第 2 页的“舍入误差”讨论对加、减、乘、除基本运算使用不同舍入策略的含义。它还包含有关衡量舍入误差的两种方法 ulp 和相对误差的背景信息。第二节讨论 IEEE 浮点标准，该标准正被商业硬件制造商迅速接受。IEEE 标准中包括基本运算的舍入方法。对标准的讨论借助了第 2 页的“舍入误差”部分中的内容。第三节讨论浮点与计算机系统各个设计方面之间的关联。主题包括指令集设计、优化编译器和异常处理。

作者已尽力避免在不给出正当理由的情况下来声明浮点，这主要是因为证明将产生较为复杂的基本计算。对于那些不属于文章主旨的说明，已将其归纳到名为“详细信息”的章节中，您可以视实际情况选择跳过此节。另外，此节还包含了许多定理的证明。每个证明的结尾处均标记有符号 ∎。如果未提供证明，∎ 将紧跟在定理声明之后。

D.3 舍入误差

将无穷多位的实数缩略表示为有限位数需要使用近似。即使存在无穷多位整数，多数程序也可将整数计算的结果以 32 位进行存储。相反，如果指定一个任意的固定位数，那么多数实数计算将无法以此指定位数精准表示实际数量。因此，通常必须对浮点计算的结果进行舍入，以便与其有限表示相符。舍入误差是浮点计算所独有的特性。第 4 页的“相对误差和 Ulp”章节说明如何衡量舍入误差。

既然多数浮点计算都具有舍入误差，那么如果基本算术运算产生的舍入误差比实际需要大一些，这有没有关系？该问题是贯穿本章节的核心主题。第 5 页的“保护数位”章节讨论**保护**数位，它是一种减少两个相近的数相减时所产生的误差的方法。 IBM 认为保护数位非常重要，因此在 1968 年它将保护数位添加到 System/360 架构的双精度格式（其时单精度已具有保护数位），并更新了该领域中所有的现有计算机。下面两个示例说明了保护数位的效用。

IEEE 标准不仅仅要求使用保护数位。它提供了用于加、减、乘、除和平方根的算法，并要求实现产生与该算法相同的结果。因此，将程序从一台计算机移至另一台计算机时，如果这两台计算机均支持 IEEE 标准，那么基本运算的结果逐位相同。这大大简化了程序的移植过程。在第 10 页的 “精确舍入的运算”中介绍了此精确规范的其他用途。

D.3.1 浮点格式

已经提议了几种不同的实数表示法，但是到目前为止使用最广的是浮点表示法。[1] 浮点表示法有一个基数 β（始终假定其为偶数）和一个精度 p。如果 β = 10、p = 3，则将数 0.1 表示为 1.00×10^{-1}。如果 β = 2、p = 24，则无法准确表示十进制数 0.1，但是它近似为 $1.10011001100110011001101 \times 2^{-4}$。

通常，将浮点数表示为 $\pm d.dd\ldots d \times \beta^e$，其中 $d.dd\ldots d$ 称为**有效数字**[2]，它具有 p 个数字。更精确地说，$\pm d_0 . d_1 d_2 \ldots d_{p-1} \times \beta^e$ 表示以下数

$$\pm\left(d_0 + d_1\beta^{-1} + \ldots + d_{p-1}\beta^{-(p-1)}\right)\beta^e, (0 \le d_i < \beta)。 \quad (1)$$

术语**浮点数**用于表示一个实数，该实数可以未经全面讨论的格式准确表示。与浮点表示法相关联的其他两个参数是最大允许指数和最小允许指数，即 $e_{\max}$ 和 $e_{\min}$。由于存在 β^p 个可能的有效数字，以及 $e_{\max} - e_{\min} + 1$ 个可能的指数，因此浮点数可以按

$$[\log_2(e_{max} - e_{min} + 1)] + [\log_2(\beta^p)] + 1$$

位编码，其中最后的 +1 用于符号位。此时，精确编码并不重要。

有两个原因导致实数不能准确表示为浮点数。最常见的情况可以用十进制数 0.1 说明。虽然它具有有限的十进制表示，但是在二进制中它具有无限重复的表示。因此，当 β = 2 时，数 0.1 介于两个浮点数之间，而这两个浮点数都不能准确地表示它。一种较不常见的情况是实数超出范围；也就是说，其绝对值大于 $\beta \times \beta^{e_{max}}$ 或小于 $1.0 \times \beta^{e_{min}}$。本文的大部分内容讨论第一种原因导致的问题。然而，超出范围的数将在第 20 页的“无穷”和第 22 页的 “反向规格化的数”章节中进行讨论。

浮点表示不一定是唯一的。例如，0.01×10^1 和 1.00×10^{-1} 都表示 0.1。如果前导数字不是零（在上面的等式 (1) 中，$d_0 \neq 0$），那么该表示称为**规格化**。浮点数 1.00×10^{-1} 是规格化的，而 0.01×10^1 则不是。当 β = 2、p = 3、$e_{\min}$ = -1 且 $e_{\max}$ = 2 时，有 16 个规格化浮点数，如图 D-1 所示。粗体散列标记对应于其有效数字是 1.00 的数。要求浮点表示为规格化，则可以使该表示唯一。遗憾的是，此限制将无法表示零！表示 0 的一种自然方法是使用 $1.0 \times \beta^{e_{min}-1}$，因为 这样做保留了以下事实：非负实数的数值顺序对应于其浮点表示的词汇顺序。[3] 将指数存储在 k 位字段中时，意味着只有 2^k - 1 个值可用作指数，因为必须保留一个值来表示 0。

请注意，浮点数中的 × 是表示法的一部分，这与浮点乘法运算不同。× 符号的含义通过上下文来看应该是明确的。例如，表达式 $(2.5 \times 10^{-3}) \times (4.0 \times 10^2)$ 仅产生单个浮点乘法。

---

1. 其他表示法的示例有**浮动斜线**和**有符号对数** [Matula 和 Kornerup 1985； Swartzlander 和 Alexopoulos 1975]。
2. 此术语由 Forsythe 和 Moler [1967] 提出，现已普遍替代了旧术语 *mantissa*。
3. 这假定采用通常的排列方式，即指数存储在有效数字的左侧。

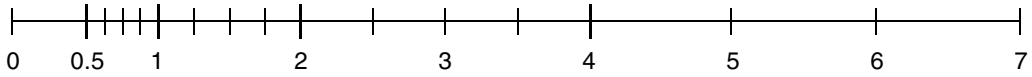


图 D-1 $\beta = 2$、$p = 3$、$e_{min} = -1$、$e_{max} = 2$ 时规格化的数

D.3.2 相对误差和 Ulp

因为舍入误差是浮点计算的固有特性，所以需要有一种方法来衡量此误差，这一点很重要。请考虑使用 $\beta = 10$ 且 $p = 3$ 的浮点格式（此格式在本节中广泛使用）。如果浮点计算的结果是 3.12×10^{-2}，并且以无限精度计算的结果是 .0314，那么您可以清楚地看到最后一位存在 2 单位的误差。同样，如果将实数 .0314159 表示为 3.14×10^{-2}，那么将在最后一位存在 .159 单位的误差。通常，如果浮点数 $d.d\ldots d \times \beta^e$ 用于表示 z，那么将在最后一位存在 $|d.d\ldots d - (z/\beta^e)|\beta^{p-1}$ 单位的误差。[4, 5] 术语 *ulp* 将用作“最后一位上的单位数”的简写。如果某个计算的结果是最接近于正确结果的浮点数，那么它仍然可能存在 .5 ulp 的误差。衡量浮点数与它所近似的实数之间差值的另一种方法是**相对误差**，它只是用两数之差除以实数的商。例如，当 3.14159 近似为 3.14×10^0 时产生的相对误差是 $.00159/3.14159 \approx .0005$。

要计算对应于 .5 ulp 的相对误差，请注意当实数用可能最接近的浮点数 $d.dd...dd \times \beta^e$ 近似表示时，误差可达到 $0.00...00\beta^e \times \beta^e$，其中 β' 是数字 $\beta/2$，在浮点数的有效数字中有 p 单位，在误差的有效数字中有 p 单位个 0。此误差是 $((\beta/2)\beta^{-p}) \times \beta^e$。因为形式为 $d.dd\ldots dd \times \beta^e$ 的数都具有相同的绝对误差，但具有介于 β^e 和 $\beta \times \beta^e$ 之间的值，所以相对误差介于 $((\beta/2)\beta^{-p}) \times \beta^e/\beta^e$ 和 $((\beta/2)\beta^{-p}) \times \beta^e/\beta^{e+1}$ 之间。即，

$$\frac{1}{2}\beta^{-p} \le \frac{1}{2}\mathrm{ulp} \le \frac{\beta}{2}\beta^{-p} \tag{2}$$

特别是，对应于 .5 ulp 的相对误差可能随因子 β 的不同而有所变化。此因子称为**浮动系数**。将 $\varepsilon = (\beta/2)\beta^{-p}$ 设置为上面 (2) 中的上限，表明：将一个实数舍入为最接近的浮点数时，相对误差总是以 ε（称为**机器 ε**）为界限。

在上例中，相对误差是 $.00159/3.14159 \approx .0005$。为了避免此类较小数，通常将相对误差表示为一个因子乘以 ε 的形式，在此例中 $\varepsilon = (\beta/2)\beta^{-p} = 5(10)^{-3} = .005$。因此，相对误差将表示为 $(.00159/3.14159)/.005)\ \varepsilon \approx 0.1\varepsilon$。

以实数 $x = 12.35$ 为例说明 ulp 与相对误差之间的差异。将其近似为 $\tilde{x} = 1.24 \times 10^1$。误差是 0.5 ulp，相对误差是 0.8ε。接下来，将考虑有关数字 8 的计算 $\tilde{x}$。精确值是 $8x = 98.8$，而计算值是 $8\tilde{x} = 9.92 \times 10^1$。误差是 4.0 ulp，相对误差仍然是 0.8ε。尽管相对误差保持不变，但以 ulp 衡量的误差却是原来的 9 倍。通常，当基数为 β 时，以 ulp 表示的固定相对误差可按最大因子 β 进行浮动。反之，如上述公式 (2) 所示，固定误差 .5 ulp 导致相对误差可按 β 进行浮动。

4. 除非数 z 大于 $\beta^{e_{max}}+1$ 或小于 $\beta^{e_{min}}$。否则在另行通知之前，将不会考虑以此方式超出范围的数。

5. 假设 z' 是近似 z 的浮点数。那么，$|d.d\ldots d - (z/\beta^e)|\beta^{p-1}$ 将等价于 $|z'-z|/\mathrm{ulp}(z')$。衡量误差的更精确公式是 $|z'-z|/\mathrm{ulp}(z)$。– 编辑者

衡量舍入误差的最常用方法是以 ulp 表示。例如，舍入为最接近的浮点数相当于一个小于或等于 .5 ulp 的误差。但是，在分析由各种公式导致的舍入误差时，使用相对误差是一种较好的方法。在第 38 页的“证明”章节中的分析很好地说明了这一点。由于浮动因子 β 的关系， ε 可能会过高估计舍入为最接近浮点数的效果，所以在使用较小 β 的计算机上，对公式的误差估计更为精确。

在仅关心舍入误差的大小顺序时，ulp 和 ε 可以互换使用，因为它们只是在因子 β 上有差异。例如，在浮点数的误差为 n ulp 时，这意味着受影响的位数是 $\log_\beta n$。如果某个计算的相对误差是 $n\varepsilon$，那么

受影响的数字是 $\approx \log_\beta n$。 (3)

D.3.3 保护数位

计算两个浮点数之差的一种方法是：精确计算二者之差，然后将其舍入为最接近的浮点数。如果两个操作数的大小差别非常大，那么系统开销将也随之上升。假定 $p = 3$， $2.15 \times 10^{12} - 1.25 \times 10^{-5}$ 将计算为

$x = 2.15 \times 10^{12}$
$y = .000000000000000125 \times 10^{12}$
$x - y = 2.1499999999999999875 \times 10^{12}$

它舍入为 2.15×10^{12}。浮点硬件通常按固定位数执行运算，而不是使用所有这些数字。假定保留位数是 p，并且在右移较小的操作数时，只是舍弃数字（与舍入相对）。那么，$2.15 \times 10^{12} - 1.25 \times 10^{-5}$ 将变为

$x = 2.15 \times 10^{12}$
$y = 0.00 \times 10^{12}$
$x - y = 2.15 \times 10^{12}$

结果是完全相同的，就好像先对差值进行精确计算，然后再进行舍入。请看另一个示例：10.1 - 9.93。它将变成

$x = 1.01 \times 10^{1}$
$y = 0.99 \times 10^{1}$
$x - y = .02 \times 10^{1}$

正确结果是 .17，因此计算差值的误差是 30 ulp，而且每一个数字均不正确：误差可以达到多大程度？

D.3.3.1 定理 1

使用带有参数 β 和 p 的浮点格式，并使用 p 数字计算差值，结果的相对误差可能与 $\beta - 1$ 一样大。

D.3.3.2 证明

当 $x = 1.00\ldots0$ 和 $y = .\rho\rho\ldots\rho$ 时，表达式 $x - y$ 中的相对误差为 $\beta - 1$，其中 $\rho = \beta - 1$。此处，y 具有 p 数字（均等于 ρ）。精确差值是 $x - y = \beta^{-p}$。然而，如果仅使用 p 数字计算结果，那么位于 y 最右侧的数字将被舍弃，因此计算差值是 β^{-p+1}。因此，误差是 $\beta^{-p} - \beta^{-p+1} = \beta^{-p}(\beta - 1)$，相对误差是 $\beta^{-p}(\beta - 1)/\beta^{-p} = \beta - 1$。∎

当 $\beta=2$ 时，相对误差可能与结果一样大；当 $\beta=10$ 时，相对误差可能是结果的 9 倍。换言之，当 $\beta=2$ 时，公式 (3) 显示受影响的位数是 $\log_2(1/\varepsilon) = \log_2(2^p) = p$。即，结果中所有 p 数字都是错误的！假定再添加一个数字（**保护数位**）以避免发生这种情况。即，将较小数截断为 $p + 1$ 数字，然后将相减的结果舍入为 p 数字。使用保护数位后，上例将变为

$x = 1.010 \times 10^1$
$y = 0.993 \times 10^1$
$x - y = .017 \times 10^1$

并且结果是精确的。使用单个保护数位时，结果的相对误差可能大于 ε（例如位于 110 – 8.59 中）。

$x = 1.10 \times 10^2$
$y = .085 \times 10^2$
$x - y = 1.015 \times 10^2$

它舍入为 102，与正确结果 101.41 相比，相对误差为 .006，大于 $\varepsilon = .005$。通常，结果的相对误差只能比 ε 稍大。更为精确的方法是，

D.3.3.3 定理 2

如果 x 和 y 均是使用参数 β 和 p 形式表示的浮点数，并且使用 $p + 1$ 数字（即，一个保护数位）执行减法操作，那么结果中的相对舍入误差将小于 2ε。

此定理将在第 38 页的“舍入误差”中证明。在上述定理中包括了加法运算，因为 x 和 y 可以是正数或负数。

D.3.4 抵消

可以这样总结上一节：在不使用保护数位的情况下，如果在两个相近数之间执行减法，那么产生的相对误差会非常大。换言之，对包含减法运算（或在具有相反符号的数量之间进行加法运算）的任何表达式求值均可能产生较大的相对误差，以至**所有**数字将变为无意义（定理 1）。如果在两个相近数之间执行减法，那么操作数中的最高有效数位将因为相等而彼此抵消。有以下两种抵消：恶性抵消和良性抵消。

恶性抵消发生在操作数受舍入误差的制约时。例如，在二次方程式中出现了表达式 $b^2 - 4ac$。数量 b^2 和 $4ac$ 受舍入误差的制约，因为它们是浮点乘法的结果。假定将它们舍入为最接近的浮点数，因此它们的准确性使误差保持在 .5 ulp 之内。它们相减时，抵消可能会导致许多精确数字消失，而留下受舍入误差影响的数字。因此，差值的误差可能是许多个 ulp。以 $b = 3.34$，$a = 1.22$，$c = 2.28$ 为例。$b^2 - 4ac$ 的精确值是 .0292。但是 b^2 舍入为 11.2，$4ac$ 舍入为 11.1，因此最终结果是 .1，误差是 70 ulp（即使 11.2 - 11.1 的精确结果等于 .1）[6]。减法运算并未引入任何误差，而是显示出先前乘法运算中引入的误差。

良性抵消发生在精确已知数量之间执行减法运算时。如果 x 和 y 没有舍入误差，依据定理 2 可知，若使用保护数位执行减法运算，那么差值 x-y 的相对误差会非常小（小于 2ε）。

有时，表现出恶性抵消的公式可以通过重新整理的方式来消除此问题。我们还是以二次公式为例

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{4}$$

当 $b^2 \gg ac$，则 $b^2 - 4ac$ 不会产生抵消，而且

$$\sqrt{b^2 - 4ac} \approx |b| \text{。}$$

但是公式之一内的另一加法运算（减法运算）将具有恶性抵消。要避免此类情况发生，请将 r_1 的分子和分母乘以

$$-b - \sqrt{b^2 - 4ac}$$

对于 r_2 进行类似处理，得到

$$r_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}}, \quad r_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \tag{5}$$

如果 $b^2 \gg ac$ 且 $b > 0$，则使用公式 (4) 计算 r_1 将产生抵消。因此，使用公式 (5) 计算 r_1，使用 (4) 计算 r_2。另一方面，如果 $b < 0$，则使用 (4) 计算 r_1，使用 (5) 计算 r_2。

表达式 $x^2 - y^2$ 是表示出恶性抵消的另一公式。将它计算为 $(x - y)(x + y)$ 更为精确。[7] 与二次方程式不同，这一改进的形式仍然包含减法运算，但是它是数量的良性抵消（没有舍入误差），而不是恶性抵消。依据定理 2，$x - y$ 中的相对误差最大为 2ε。对于 $x + y$，也同样正确。将两个具有较小相对误差结果的数量相乘，会产生具有较小相对误差的乘积（请参见第 38 页的“舍入误差”章节）。

6. 700，而不是 70。因为 .1 - .0292 = .0708，以 ulp(0.0292) 表示的误差是 708 ulp。 - 编辑者

7. 虽然表达式 $(x-y)(x+y)$ 不产生恶性抵消，但是，如果符合以下两个条件之一，其精确程度要比 x^2 y^2 稍差：$X \gg Y$ 或 $X \ll Y$。在这种情况下，$(x - y)(x + y)$ 具有三个舍入误差，但是 $x^2 - y^2$ 只有两个舍入误差，因为计算 x^2 和 y^2 中的较小者时产生的舍入误差不影响最终的减法运算。

为避免将精确值与计算值相混淆，特采用以下表示法。$x \ominus y$ 表示计算的差值（即具有舍入误差），而 $x - y$ 表示 x 和 y 的精确差值。类似地，$\oplus$、$\otimes$ 和 $\oslash$ 分别表示计算的加法、乘法和除法。全大写函数表示函数的计算值，如 `LN(x)` 或 `SQRT(x)`。小写函数和传统数学符号表示其精确值，如 $\ln(x)$ 和 $\sqrt{x}$ 。

虽然 $(x \ominus y) \otimes (x \oplus y)$ 与 $x^2 - y^2$ 极为近似，但是浮点数 x 和 y 本身可能近似为某些真实数量：$\hat{x}$ 和 $\hat{y}$ 。例如，$\hat{x}$ 和 $\hat{y}$ 可能是精确已知的十进制数，且无法用二进制精确表示。在这种情况下，即使 $x \ominus y$ 与 $x - y$ 极为近似，与真实表达式 $\hat{x} - \hat{y}$ 相比，它也可能具有较大的相对误差，因此 $(x + y)(x - y)$ 相对 $x^2 - y^2$ 的优势就不那么明显了。因为计算 $(x + y)(x - y)$ 与计算 $x^2 - y^2$ 的工作量几乎相同，所以在这种情况下前者明显是首选的形式。然而在一般情况下，如果将恶性抵消替换为良性抵消会产生较大的开销，那么这种做法是不值得的，因为输入通常是（但不总是）一个近似值。但是，即使数据是不精确的，完全消除抵消（如在二次方程式中）也是值得做的。本文始终假定算法的浮点输入是精确的，而且计算结果尽可能精确。

表达式 $x^2 - y^2$ 在重新写为 $(x - y)(x + y)$ 时更为精确，因为恶性抵消被良性抵消所替换。接下来恶性抵消公式示例更为有趣，可以将公式进行改写，以便仅表示出良性抵消。

三角形面积可以直接用其各边 a、b 和 c 的长度表示，如下所示：

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \quad \text{其中 } s = (a+b+c)/2 \tag{6}$$

（假设三角形非常平坦，即 $a \approx b + c$。那么 $s \approx a$，公式 (6) 中的项 $(s - a)$ 减去两个邻近的数，其中一个可能有舍入误差。例如，如果 $a = 9.0$、$b = c = 4.53$、s 的正确值为 9.03，并且 A 为 2.342.... 即使计算出的值 s (9.05) 的误差只有 2 ulps，计算出的值 A 也为 3.04，误差为 70 ulps。

有一种方法可以改写公式 (6)，以便它返回精确结果，甚至对于平坦的三角形也是这样 [Kahan 1986]。即

$$A = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4}, \; a \geq b \geq c \tag{7}$$

如果 a、b 和 c 不满足 $a \geq b \geq c$，那么在应用 (7) 之前请将它们重命名。检查 (6) 和 (7) 右侧是否是代数恒等是一项非常简单的工作。使用上述 a、b 和 c 的值得出的计算面积是 2.35，其误差是 1 ulp，比第一个公式要精确得多。

对于此示例来说，虽然公式 (7) 比 (6) 要精确得多，但是了解 (7) 在通常情况下的执行情况是非常必要的。

D.3.4.1 定理 3

由于减法运算是使用保护数位执行的 *(e ≤ .005)*，且计算的平方根在 *1/2* ulp 内，所以使用 (7) 计算三角形面积时产生的舍入误差最大为 *11*ε。

条件 $e < .005$ 几乎在每个实际的浮点系统中都能得到满足。例如，当 $\beta = 2$ 且 $p \geq 8$ 时，可确保 $e < .005$；当 $\beta = 10$ 时，$p \geq 3$ 已足够。

在命题（如定理 3）中讨论表达式的相对误差时，认为表达式是使用浮点运算计算的。特别是，相对误差实际上是由以下表达式产生的：

$$\texttt{SQRT}((a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))) \oslash 4 \tag{8}$$

由于 (8) 的繁琐性，在定理的陈述中我们通常采用 E 的计算值，而不是写出带有圆形符号的 E。

误差界限通常是非常保守的。在上述数值示例中，(7) 的计算值是 2.35，与真实值 2.34216 相比，相对误差是 0.7ε，这比 11ε 要小得多。计算误差界限的主要原因不是获得精确界限，而是检验公式是否不包含数值问题。

请看最后一个示例：可以改写为使用良性抵消的表达式 $(1 + x)^n$，其中 $x \ll 1$ 。此表达式出现在金融计算中。以每天将 100 美元存入一个银行帐户为例，其年利率为 6%，每天按复利计算。如果 $n = 365$ 且 $i = .06$，那么在年底时累积的金额为

$$100\frac{(1 + i/n)^n - 1}{i/n}$$

美元。如果这是使用 $\beta = 2$ 和 $p = 24$ 进行计算的，那么结果将是 37615.45 美元，与精确结果 37614.05 美元相比，差额为 1.40 美元。导致此问题的原因很明显。表达式 $1 + i/n$ 将导致 1 和 .0001643836 相加，这样会丢失 i/n 的低位。将 $1 + i/n$ 扩大到 n 次幂时，将扩大此舍入误差。

这个令人头痛的表达式 $(1 + i/n)^n$ 可以改写为 $e^{n\ln(1 + i/n)}$，现在的问题是在 x 较小的情况下计算 $\ln(1 + x)$。一种方法是使用近似 $\ln(1 + x) \approx x$，在这种情况下应支付 37617.26 美元，差额为 3.21 美元，甚至还不如使用这个直观的公式精确。但是，有一种方法可以非常精确地计算 $\ln(1 + x)$，如定理 4 所示 [Hewlett-Packard 1982]。使用此公式的计算结果是 37614.07，其精确性使误差保持在两美分之内！

定理 4 假设 `LN(x)` 近似于 $\ln(x)$，其误差在 1/2 ulp 内。它解决的问题是：当 x 很小时，$\texttt{LN}(1 \oplus x)$ 不接近于 $\ln(1 + x)$，因为 $1 \oplus x$ 在 x 的低位中丢失了信息。也就是说，在以下情况下，$\ln(1 + x)$ 的计算值不接近于其实际值：$x \ll 1$ 。

D.3.4.2 定理 4

在使用公式

$$\ln(1 + x) = \begin{cases} x & \text{对于 } 1 \oplus x = 1 \\ \dfrac{x\ln(1 + x)}{(1 + x) - 1} & \text{对于 } 1 \oplus x \neq 1 \end{cases}$$

计算 ln(1 + x) 的情况下，**如果 $0 \le x < 3/4$，减法运算是使用保护数位 $(e < 0.1)$ 执行的，且 ln 的计算值误差在 $1/2$ ulp 之内，则相对误差最大为 5ε**。

此公式适用于 x 的任意值，但仅与 $x \ll 1$ 有关，这就是简易公式 ln(1 + x) 中出现恶性抵消的地方。虽然此公式看起来可能有些神秘，但是一个简单的解释就可以说明它的工作原理。将 ln(1 + x) 写为

$$x\left(\frac{\ln(1+x)}{x}\right) = x\mu(x).$$

可以精确计算左侧因子，但是在将 1 和 x 相加时右侧因子 $\mu(x) = \ln(1+x)/x$ 将产生较大的舍入误差。然而，μ 几乎是恒定的，因为 $\ln(1+x) \approx x$。因此，稍微改变 x 不会引入很大误差。换句话说，如果 $\tilde{x} \approx x$，计算 $x\mu(\tilde{x})$ 将与 $x\mu(x) = \ln(1+x)$ 极为近似。是否存在一个 $\tilde{x}$ 的值，使用它可以精确计算 $\tilde{x}$ 和 $\tilde{x}+1$？是的；该值是 $\tilde{x} = (1 \oplus x) \ominus 1$，因为使用该值后 $1 + \tilde{x}$ 与 $1 \oplus x$ 完全相等。

可以这样总结本节：保护数位在彼此接近且精确已知的数量相减 （良性抵消）时保证了精确性。有时，可以通过使用良性抵消来改写产生不精确结果的公式，以便获得相当高的数值精确性；但是，仅当使用保护数位进行减法运算时此过程才起作用。使用保护数位的开销并不高，因为它仅仅要求将加法器加宽一位。对于 54 位双精度加法器，增加的开销少于 2%。只需付出如此代价，您就能够运行许多算法 （如公式 (6)）来计算三角形的面积和表达式 ln(1 + x)。虽然多数现代计算机具有保护数位，但是也有一些计算机 （如 Cray 系统）没有保护数位。

D.3.5 精确舍入的运算

在使用保护数位进行浮点运算时，它们不如先精确计算再舍入为最接近的浮点数那样精确。以这种方式进行的运算将称为**精确舍入的运算**。[8] 紧邻定理 2 之前的示例显示出单个保护数位并不总是给出精确舍入的结果。上一节给出了需要有保护数位才能正常工作的几个算法示例。本节给出要求精确舍入的算法示例。

到目前为止，尚未给出任何舍入的定义。舍入是很好理解的，只是如何舍入中间数有些令人棘手；例如，应该将 12.5 舍入为 12 还是 13？一个学派将 10 个数字分成两半，使 {0, 1, 2, 3, 4} 向下舍入，使 {5, 6, 7, 8, 9} 向上舍入；因此 12.5 将舍入为 13。这是在 Digital Equipment Corporation VAX 计算机上的舍入方式。另一学派则认为：由于以 5 结尾的数介于两个可能的舍入数中间，向下舍入与向上舍入需视情况而定。实现这一发生概率为 50% 事件的方法之一是要求舍入结果的最低有效数字为偶数。因此 12.5 舍入为 12 而不是 13，因为 2 是一个偶数。其中的哪种方法是最佳的，是向上舍入还是舍入为偶数？Reiser 和 Knuth [1975] 倾向于舍入为偶数，并提供了以下理由。

8. 通常也称为*正确舍入的运算*。– 编辑者

D.3.5.1 定理 5

假设 x 和 y 是浮点数，并定义 $x_0 = x$, $x_1 = (x_0 \ominus y) \oplus y$, …, $x_n = (x_{n-1} \ominus y) \oplus y$。如果使用舍入为偶数精确舍入 $\oplus$ 和 $\ominus$，那么对于所有 n，$x_n = x$；对于所有 $n \geq 1$，$x_n = x_1$。■

为了阐明此结果，请以 $\beta = 10$，$p = 3$ 为例，并令 $x = 1.00$，$y = -.555$。当进行向上舍入时，该序列将变成

$$x_0 \ominus y = 1.56, \quad x_1 = 1.56 \ominus .555 = 1.01, \quad x_1 \ominus y = 1.01 \oplus .555 = 1.57,$$

x_n 的每个连续值都将增加 .01，直至 $x_n = 9.45$ $(n \leq 845)$[9] 为止。在使用舍入为偶数的情况下，x_n 始终是 1.00。此示例说明：在使用向上舍入规则时，计算可能会逐渐向上漂移，而依据此定理，这种情况在使用舍入为偶数时不会发生。本文的其余部分将使用舍入为偶数。

精确舍入的应用之一是执行多种精度运算。有两种基本方法可以获得较高的精度。一种方法使用极大的有效数字表示浮点数，将有效数字存储在字元数组中，并用汇编语言编写用于处理这些数的例程的代码。第二种方法将更高精度的浮点数表示为普通浮点数的数组，在其中增加无限精度的数组元素以便恢复高精度浮点数。此处将讨论第二种方法。使用浮点数的数组的优点是：可以使用高级语言编写其可移植代码，但是它要求精确舍入的运算。

在此系统中乘法的关键是将积 xy 表示为和，其中每个被加数都具有与 x 和 y 相同的精度。通过拆分 x 和 y，可以做到这一点。编写 $x = x_h + x_l$ 和 $y = y_h + y_l$，则精确乘积为

$$xy = x_h y_h + x_h y_l + x_l y_h + x_l y_l \text{ 。}$$

如果 x 和 y 具有 p 位有效数字，那么被加数也将具有 p 位有效数字，但条件是 x_l、x_h、y_h、y_l 可以使用 $\lceil p/2 \rceil$ 位来表示。当 p 是偶数时，很容易找到一种拆分方法。数 $x_0.x_1 \ldots x_{p-1}$ 可以写为 $x_0.x_1 \ldots x_{p/2-1}$ 和 $0.0 \ldots 0x_{p/2} \ldots x_{p-1}$ 之和。当 p 是奇数时，这一简单的拆分方法将不起作用。但是，通过使用负数可以获得额外位。例如，如果 $\beta = 2$，$p = 5$ 且 $x = .10111$，那么 x 可以拆分为 $x_h = .11$ 和 $x_l = -.00001$。拆分数的方法不止一种。一种易于计算的拆分方法是由 Dekker [1971] 提出的，但它需要多个单个保护数位。

D.3.5.2 定理 6

假设 p 是浮点精度，要求在 $\beta > 2$ 时 p 是偶数，并假设浮点运算是精确舍入的。那么，如果 $k = \lceil p/2 \rceil$ 是精度的一半（向上舍入）且 $m = \beta^k + 1$，则可以将 x 拆分为 $x = x_h + x_l$，其中

$$x_h = (m \otimes x) \ominus (m \otimes x \ominus x), \quad x_l = x \ominus x_h,$$

每个 x_i 是可以使用 $\lceil p/2 \rceil$ 位精度表示的。

9. 当 $n = 845$ 时，$x_n = 9.45$，$x_n + 0.555 = 10.0$，$10.0 - 0.555 = 9.45$。因此，当 $n > 845$ 时，$x_n = x_{845}$。

为了在示例中说明此定理的应用，可令 β = 10，$p = 4$，$b = 3.476$，$a = 3.463$，$c = 3.479$。那么，$b^2 - ac$ 在舍入为最接近的浮点数时是 .03480，而 $b \otimes b = 12.08$，$a \otimes c = 12.05$，因此 $b^2 - ac$ 的计算值是 .03。这样，误差就是 480 ulp。使用定理 6 编写 $b = 3.5 - .024$，$a = 3.5 - .037$ 和 $c = 3.5 - .021$，b^2 将变成 $3.5^2 - 2 \times 3.5 \times .024 + .024^2$。每个被加数都是精确的，因此 $b^2 = 12.25 - .168 + .000576$，此时未计算其中的和。类似地，$ac = 3.5^2 - (3.5 \times .037 + 3.5 \times .021) + .037 \times .021 = 12.25 - .2030 + .000777$。最后，将这两个数列逐项相减将为 $0 \oplus .0350 \ominus .000201 = .03480$ 的 $b^2 - ac$ 得出估计值，它与精确舍入的结果完全相同。为了说明定理 6 确实需要精确舍入，请假设 $p = 3$，β = 2，$x = 7$。那么，$m = 5$，$mx = 35$，$m \otimes x = 32$。如果使用单个保护数位进行减法运算，那么 $(m \otimes x) \ominus x = 28$。因此，$x_h = 4$，$x_l = 3$，由此得出 x_l 不能用 $[p/2] = 1$ 位表示。

作为精确舍入的最后一个示例，假设用 10 除 m。结果是一个浮点数，一般情况下将不等于 $m/10$。当 β = 2 时，用 10 乘 $m/10$ 将复原 m，条件是使用了精确舍入。实际上，一个更为常规的事实（由 Kahan 提出）是正确的。其证明非常巧妙，但对此细节不感兴趣的读者可以直接跳至第 14 页的 “IEEE 标准”章节。

D.3.5.3 定理 7

当 β = 2 时，如果 m 和 n 是整数，且 $|m| < 2^{p-1}$，n 具有特殊形式 $n = 2^i + 2^j$，那么 $(m \oslash n) \otimes n = m$，条件是浮点运算是精确舍入的。

D.3.5.4 证明

换算为 2 的幂次方不会产生不良影响，因为这样仅改变指数，而不改变有效数字。如果 $q = m/n$，那么按比例缩放为 n 以便 $2^{p-1} \le n < 2^p$，并按比例缩放为 m 以便 $1/2 < q < 1$。因此，$2^{p-2} < m < 2^p$。因为 m 具有 p 个有效位，所以在二进制点右侧它最多有一位。改变 m 的符号不会产生不良影响，因此假设 $q > 0$。

如果 $\bar{q} = m \oslash n$，那么要证明此定理需要得到

$$|n\bar{q} - m| \le \frac{1}{4} \tag{9}$$

这是因为在二进制点右侧 m 最多具有 1 位，因此 $n\bar{q}$ 将舍入为 m。为了处理 $|n\bar{q} - m| = 1/4$ 时的中间情况，请注意，由于初始的未缩放的 m 具有 $|m| < 2^{p-1}$，其低位是 0，因此已缩放的 m 的低位也是 0。因此，中间情况将舍入为 m。

假定 $q = .q_1q_2 \ldots$，并假设 $\hat{q} = .q_1q_2 \ldots q_p1$。要估算 $|n\bar{q} - m|$，请首先计算

$$|\hat{q} - q| = |N/2^{p+1} - m/n|,$$

其中 N 是一个奇数整数。因为 $n = 2^i + 2^j$ 且 $2^{p-1} \le n < 2^p$，所以对于某些 $k \le p - 2$ 一定存在 $n = 2^{p-1} + 2^k$，因此

$$|\hat{q} - q| = \left| \frac{nN - 2^{p+1}m}{n2^{p+1}} \right| = \left| \frac{(2^{p-1-k} + 1)N - 2^{p+1-k}m}{n2^{p+1-k}} \right| \quad 。$$

分子是一个整数，并且因为 N 是一个奇数，所以实际上它是一个奇数整数。因此，

$|\hat{q} - q| \ge 1/(n2^{p+1-k})$。

假设 $q < \hat{q}$（与 $q > \hat{q}$ 的情况是类似的）。[10] 那么， n

$\bar{q} < m$，

$$|m - n\bar{q}| = m - n\bar{q} = n(q - \bar{q}) = n(q - (\hat{q} - 2^{-p-1})) \le n\left(2^{-p-1} - \frac{1}{n2^{p+1-k}}\right)$$

$$= (2^{p-1} + 2^k)2^{-p-1} - 2^{-p-1+k} = \frac{1}{4}$$

这将得到 (9) 并证明此定理。[11] ∎

只要将 $2^i + 2^j$ 替换为 $\beta^i + \beta^j$，此定理对于任何基数 β 都是成立的。但是，随着 β 的增大，$\beta^i + \beta^j$ 形式的分母差值将越来越大。

我们现在可以回答以下问题：如果基本算术运算引入比所需稍大的舍入误差，这有没有关系？答案是确实有关系，因为通过精确的基本运算我们可以证明公式在具有较小相对误差的情况下是“正确的”。在此意义上，第 6 页的“抵消”章节讨论了几种算法，它们需要保护数位以便产生正确结果。但是，如果这些公式的输入是用不精确的方式表示的数，那么定理 3 和定理 4 的限制就不需要太注意了。原因是：如果 x 和 y 仅仅是某些衡量数量的近似，那么良性抵消 $x - y$ 可能变为恶性抵消。但是，即使对于不精确的数据，精确运算也是有用的；因为通过精确运算我们可以建立精确关系（如定理 6 和定理 7 中所讨论的那些关系）。即使每个浮点变量只是某个实际值的近似，精确运算也是有用的。

10.请注意，在二进制 q 不能等于 $\hat{q}$ 。 – 编辑者

11.作为练习留给读者：将证明扩展到不是 2 的基数。 - 编辑者

D.4 IEEE 标准

浮点计算有两种不同的 IEEE 标准。IEEE 754 是一个二进制标准。对于单精度，它要求 β = 2、p = 24；对于双精度 [IEEE 1987]，它要求 p = 53。它还指定单精度和双精度中位的精确布局。 IEEE 854 允许 β = 2 或 β = 10。与 754 不同，它不指定如何将浮点数编码到位中 [Cody 等 1984]。它不需要 p 的特定值，而是为单精度和双精度指定对 p 的允许值的约束。在讨论这两个标准的公共属性时，将使用术语 IEEE **标准**。

本节简要介绍 IEEE 标准。每个小节讨论的都是标准的一个方面，以及将这方面制定为标准的原因。本文的目的不是论述 IEEE 标准是可能的最佳浮点标准，而是将其接受为指定标准，并介绍它的用途。有关详细信息，请查阅这些标准 [IEEE 1987； Cody 等 1984]。

D.4.1 格式与运算

D.4.1.1 基数

IEEE 854 允许 β = 10 的原因是显而易见的。人们就是使用基数 10 来交换和考虑数字的。β = 10 尤其适合于计算器，每个运算的结果都由计算器以十进制显示。

IEEE 854 要求：如果基数不是 10，就必须是 2。它这样要求有几个原因。第 4 页的“相对误差和 Ulp”节提到了其中的一个原因：当 β 是 2 时误差分析的结果更精确，因为进行相对误差计算时 .5 ulp 的舍入误差按 β 的因子发生变化，而且基于相对误差的误差分析几乎始终是比较简单的。其中的原因必然与大基数的有效精度有关。以比较 β = 16、p = 1 与 β = 2、 p = 4 为例。这两个系统的有效数字都是 4 位。判断 15/8 的计算结果。当 β = 2 时，15 被表示为 1.111×2^3，15/8 被表示为 1.111×2^0。因此 15/8 是精确的。但是，当 β = 16 时， 15 被表示为 $F \times 16^0$，其中 F 是表示 15 的十六进制数字。但是 15/8 被表示为 1×16^0，它只有一个正确位。通常，基数 16 最多可丢失 3 位，因此精度为 p 的十六进制数字具有的有效精度低至 $4p$ - 3，而不是 $4p$ 个二进制位。既然大的 β 值存在这些问题，为什么 IBM 为其 system/370 选择了 β = 16 呢？只有 IBM 才知道确切原因，但是有两个可能的原因。第一个是增大了指数范围。 system/370 上的单精度具有 β = 16、p = 6。因此有效数字需要 24 位。因为必须凑够 32 位，所以会为指数保留 7 位，为符号位保留 1 位。这样，可表示数值的大小范围是从 16^{-2^6} 到 $16^{2^6} = 2^{2^8}$ 。为了在 β = 2 时获得类似的指数范围，需要将 9 位用于指数部分，仅为有效数字部分保留 22 位。但是，上面刚刚指出：当 β = 16 时，有效精度可以低至 $4p$ - 3 = 21 位。更糟的是，当 β = 2 时，有可能获得额外的精度位（正如本节后面所说明的情况），因此 β = 2 的计算机具有 23 个精度位，而 β = 16 的计算机则具有 21 - 24 位的精度范围。

选择 β = 16 的另一种可能原因必然与移位有关。两个浮点数进行加法运算时，如果它们的指数不同，其中一个有效数字必须进行移位才能使小数点对齐，这将降低运算速度。在 β = 16，p = 1 的系统中，1 到 15 之间的所有数字都具有相同的指数，所以这组数字中任意两个不同的数进行加法运算，即有 $\binom{15}{2}$ = 105 可能对时，不需要进行移位。但是，在 β = 2、p = 4 的系统中，这些数的指数范围是从 0 到 3，105 对中有 70 对需要进行移位。

对于现今大多数硬件来讲，通过避免操作数子集进行移位而获得的性能可以忽略，因此 β = 2 的微小波动使其成为更可取的基数。使用 β = 2 的另一个优点是有办法获得额外的有效位。[12] 因为浮点数始终是规格化的，所以有效数字的最高有效位始终是 1，这样就没有必要浪费一个表示它的存储位。使用此技巧的格式具有**隐藏**位。在第 3 页的“浮点格式”中已经指出：这需要对 0 进行特殊约定。那里给出的方法是 $e_{\min} - 1$ 的指数，全零的有效数字不表示

$1.0 \times 2^{e_{min}-1}$，而是表示 0。

IEEE 754 单精度是用 32 位编码的，将 1 位用于符号， 8 位用于指数， 23 位用于有效数字。但是，它使用一个隐藏位，因此有效数字是 24 位 (p = 24)，即使它仅使用 23 位进行编码也是如此。

D.4.1.2 精度

IEEE 标准定义了四种不同的精度：单精度、双精度、单精度扩展和双精度扩展。在 IEEE 754 中，单精度和双精度大致对应于大多数浮点硬件所提供的精度。单精度占用一个 32 位字，双精度占用两个连续的 32 位字。扩展精度是一种至少提供一点额外精度和指数范围的格式 (表 D-1)。

表 D-1 IEEE 754 格式参数

	格式			
参数	**单精度**	**单精度扩展**	**双精度**	**双精度扩展**
p	24	≥ 32	53	≥ 64
$e_{\max}$	+127	≥ 1023	+1023	> 16383
$e_{\min}$	-126	≤ -1022	-1022	≤ -16382
指数宽度 （位）	8	≤ 11	11	≥ 15
格式宽度 （位）	32	≥ 43	64	≥ 79

IEEE 标准仅指定扩展精度提供多少个额外位的下界。允许的最小双精度扩展格式有时称为 80 **位格式**，尽管上表显示它可以使用 79 位。原因是，扩展精度的硬件实现通常不使用隐藏位，因此将使用 80 位而不是 79 位。[13]

12.这似乎是首先由 Goldberg [1967] 发表的，尽管 Knuth （[1981]，第 211 页）将此方法归功于 Konrad Zuse。

13.Kahan 认为，扩展精度具有 64 位的有效数字，因为这是可以在 Intel 8087 上不增加循环时间的情况下进行进位传送的最宽精度 [Kahan 1988]。

该标准将扩展精度作为重中之重，没有做出有关双精度的任何建议，但是强烈建议**实现应该支持对应于所支持的最宽基本格式的扩展格式，** …

使用扩展精度的一个动因来自计算器。通常计算器将显示 10 位，但在内部使用 13 位。通过仅显示 13 位中的 10 位，计算器在用户看来就像一个按 10 位精度计算指数、余弦等的“黑箱”。计算器要在 10 位精度内计算诸如 exp、log 和 cos 之类的函数，且达到合理的效率，它需要使用几个额外位。不难找到这样一个简单的有理数表达式，它求对数近似值的误差为 500 ulp。这样，使用 13 位进行计算就可以得出正确的 10 位结果。通过将额外的 3 位隐藏，计算器为操作者提供了一个简单的模型。

IEEE 标准中的扩展精度起着类似的作用。它使库可以高效地计算数量，且单（或双）精度的误差在约 .5 ulp 内，为使用那些库的用户提供了一个简单模型，即：每个基本操作（不管是简单的乘法运算还是对对数的调用）都会返回精确度大约在 .5 ulp 内的值。但是，当使用扩展精度时，确保它的使用对用户透明是很重要的。例如，在计算器上，如果显示值的内部表示没有舍入到与显示值相同的精度，则进一步运算的结果将取决于隐藏位数，并且对用户来说似乎是不可预知的。

以 IEEE 754 单精度与十进制之间的转换问题为例来进一步说明扩展精度。理想情况下，将用足够位数显示单精度数，以便在将十进制数读回时可以再现单精度数。结果是 9 个十进制位就足以再现单精度二进制数（请参见第 46 页的“二进制到十进制的转换”节）。将十进制数再转换回它的唯一二进制表示时，小至 1 ulp 的舍入误差就是致命的，因为这将给出错误的结果。这是一种扩展精度对于高效算法至关重要的情况。在单精度扩展可用时，存在一种非常简单的方法，它可以将十进制数转换为单精度二进制数。首先将 9 个十进制数字作为整数 N 读入，忽略小数点。在表 D-1, $p \geq 32$，因为 $10^9 < 2^{32} \approx 4.3 \times 10^9$, N 可以用单精度扩展精确表示。接下来，找出提升 N 所需的适当幂 10^P。这将是十进制数的指数以及（到目前为止）被忽略小数点的位置的组合。计算 $10^{|P|}$。如果 $|P| \leq 13$，则这也是精确表示的，因为 $10^{13} = 2^{13}5^{13}$ 且 $5^{13} < 2^{32}$。最后，将 N 和 $10^{|P|}$ 相乘（如果 $p < 0$，则将它们相除）。如果最后的这个运算是精确进行的，则将再现最接近的二进制数。第 46 页的“二进制到十进制的转换”节说明如何精确地进行最后的相乘（或相除）。因此，对于 $|P| \leq 13$，使用单精度扩展格式，可以将 9 位十进制数转换为最接近的二进制数（即精确舍入的数）。如果 $|P| > 13$，则使用单精度扩展不足以使上述算法始终计算出精确舍入的等价的二进制数，但是 Coonen [1984] 证明这足以保证先将二进制数转换为十进制数而后再转换回来将再现原始的二进制数。

如果支持双精度，则上述算法将以双精度而不是单精度扩展运行，但是将双精度转换为 17 位十进制数再转换回来将需要双精度扩展格式。

D.4.1.3 指数

因为指数可以是正数或负数，所以必须选择某个方法来表示其符号。表示有符号数的两种常用方法是符号 / 大小和 2 的补码。符号 / 大小是 IEEE 格式中用于有效数字符号的系统：一个位用于容纳符号，其余位用来表示数的大小。2 的补码表示通常在整数运算中使用。在此方案中，范围 $[-2^{p-1}, 2^{p-1} - 1]$ 内的数由等于它以 2^p 为模的结果的最小非负数表示。

IEEE 二进制标准不使用其中的任一方法来表示指数，而是改用**偏置**的方法表示。在使用单精度数的情况下，指数以 8 位存储，偏差是 127（对于双精度，它是 1023）。这意味着，如果 $\bar{k}$ 是被解释为无符号整数的指数位的值，则浮点数的指数是 $\bar{k}$ – 127。这通常称为**无偏指数**，以区别于偏离指数 $\bar{k}$.

从表 D-1 可以知道，单精度有 e_{max} = 127，e_{min} = -126。其 $|e_{min}| < e_{max}$ 的原因是最小数 $(1/2^{e_{min}})$ 的倒数将不上溢。虽然最大数的倒数确实会发生下溢，但是下溢通常不如上溢严重。第 14 页的 “基数”节解释了 e_{min} - 1 用于表示 0，第 18 页的 “特殊数量”将介绍 e_{max} + 1 的一个用途。在 IEEE 单精度中，这意味着偏离指数介于 e_{min} – 1 = -127 和 e_{max} + 1 = 128 之间，而无偏指数介于 0 和 255 之间 （这正好是可以使用 8 位表示的非负整数）。

D.4.1.4 运算

IEEE 标准要求精确舍入加、减、乘、除的结果。也就是说，必须先精确计算结果，然后舍入为最接近的浮点数 （通过舍入为偶数）。第 5 页的 “保护数位”节指出：当两个浮点数的指数有很大差异时，计算这两个浮点数的精确差或和的开销会非常大。该节介绍了保护数位，它提供了一种在保证相对误差很小的同时计算差值的实用方法。但是，使用单个保护数位进行计算不会总是给出与计算精确结果再进行舍入相同的结果。通过引入第二个保护数位和第三个**粘滞**位，就能够以比单个保护数位稍高的成本计算差值，但是结果是相同的，就好像是先精确计算差值再进行舍入 [Goldberg 1990]。这样，就可以高效实施该标准。

完全指定算术运算结果的一个原因是为了改进软件的可移植性。当一个程序在两台计算机之间移动且这两台计算机都支持 IEEE 运算时，如果任意的中间结果出现不同，则一定是由于软件错误，而不是运算差异。精确指定的另一优点是它使有关浮点的推理更容易进行。有关浮点的证明已足够困难，无须处理多种运算所产生的多种情况。正如整数程序的正确性是可以证明的，浮点程序的正确性也是可以证明的，不过在这种情况下要证明的是结果的舍入误差符合某些界限。定理 4 就是这样的一个证明示例。当精确指定被推理的运算时，就可以使这些证明容易得多。一旦某种算法经证明对于 IEEE 运算正确，那么它将在支持 IEEE 标准的任何计算机上正确工作。

Brown [1981] 曾建议了有关浮点的公理，这些公理包括大多数的现有浮点硬件。但是，在此系统中的证明无法检验第 6 页的 “抵消”和第 10 页的 “精确舍入的运算”节中的算法，它们需要的功能并非存在于所有硬件上。此外，与简单地定义精确执行再进行舍入的运算相比，Brown 的公理更为复杂。因此，从 Brown 的公理证明定理通常比先假定运算是精确舍入的再证明它们要困难。

在浮点标准应该涉及哪些运算这一问题上，没有完全一致的意见。除了基本运算 +、-、× 和 / 之外， IEEE 标准还指定正确舍入平方根、余数以及整数和浮点数之间的转换。它还要求正确舍入内部格式与十进制之间的转换 （非常大的数除外） 。 Kulisch 和 Miranker [1986] 曾建议将内积增加到精确指定的运算列表。他们注意到，在 IEEE 运算中计算内积时最终结果与正确结果之差可能相当大。例如，和是内积的一种特殊情况，和 $((2 \times 10^{-30} + 10^{30}) - 10^{30}) - 10^{-30}$ 正好等于 10^{-30}，但是在使用 IEEE 运算的计算机上计算结果将是 -10^{-30}。使用比实现快速乘数所用更少的硬件计算误差在 1 ulp 内的内积，这是可能的 [Kirchner 和 Kulish 1987]。[14] [15]

在标准中提到的所有运算都需要精确舍入，但十进制和二进制之间的转换除外。原因是精确舍入所有运算的高效算法是已知的（转换除外）。对于转换，已知的最好的有效算法所产生的结果比精确舍入的结果稍差 [Coonen 1984]。

IEEE 标准因**制表人的难题**而不要求精确舍入超越函数。为了说明这一点，假定您正在制作 4 位指数函数表。那么，exp(1.626) = 5.0835。应该将它舍入为 5.083 还是 5.084？如果更仔细地计算，则结果变成 5.08350。然后是 5.083500。然后是 5.0835000。因为 exp 是超越函数，所以在辨别 exp(1.626) 是 5.083500…0*ddd* 还是 5.0834999…9*ddd* 之前该过程会继续任意长的时间。因此，指定超越函数的精度与按无限精度计算它再进行舍入所得的精度相同是不实用的。另一种方法将是从算法上指定超越函数。但是，在所有硬件架构上都工作良好的单个算法似乎是不存在的。有理逼近、CORDIC、[16] 和大表是用于当代计算机上计算超越函数的三种不同方法。每种方法适用于不同种类的硬件，目前还没有一种算法可以被当前广泛的硬件所接受。

D.4.2 特殊数量

在一些浮点硬件上，每种位模式都表示一个有效的浮点数。IBM System/370 就是这样的一个示例。另一方面，VAX™ 保留一些位模式以表示称为**保留操作数**的特殊数。这一思想可追溯到 CDC 6600，它具有表示特殊数量 `INDEFINITE` 和 `INFINITY` 的位模式。

IEEE 标准继承了这一传统，它具有 NaN（**不是数**）和无穷大。如果不使用任何特殊数量，则不存在处理异常情况（例如接受负数的平方根，而不是终止计算）的好方法。在 IBM System/370 FORTRAN 中，响应计算负数（如 -4）的平方根的默认操作导致错误消息的打印。因为每种位模式都表示一个有效数，所以平方根的返回值一定是某个浮点数。对于 System/370 FORTRAN，返回的是 $\sqrt{-4} = 2$。在 IEEE 运算中，此情况下返回 NaN。

IEEE 标准指定了以下特殊值（请参见表 D-2）：± 0、反向规格化的数、±∞ 和 NaN（如下一节中所述，存在多个 NaN）。这些特殊值都是使用 $e_{max} + 1$ 或 $e_{min} - 1$ 的指数进行编码的（已经指出 0 的指数是 $e_{min} - 1$）。

表 D-2 IEEE 754 特殊值

指数	尾数部分	表示
$e = e_{min} - 1$	$f = 0$	±0
$e = e_{min} - 1$	$f \neq 0$	$0.f \times 2^{e_{min}}$

14.一些反对将内积作为一种基本运算包括在内的论点由 Kahan 和 LeBlanc [1985] 提出。

15.Kirchner 写到：在每时钟周期的一个部分乘积中，在硬件中计算误差在 1 ulp 内的内积是可能的。另外需要的硬件可以与为获得该速度所需的乘数数组进行比较。

16.CORDIC 是 Coordinate Rotation Digital Computer（坐标旋转数字计算机）的缩写，它是计算超越函数的一种方法，主要使用移位和加法（即非常少的乘法和除法）[Walther 1971]。在此方法中，另外需要的硬件可以与获得在 Intel 8087 和 Motorola 68881 上使用的速度所需要的乘数数组进行比较。

表 D-2 IEEE 754 特殊值 (续)

指数	尾数部分	表示
$e_{min} \le e \le e_{max}$	—	$1.f \times 2^e$
$e = e_{max} + 1$	$f = 0$	±∞
$e = e_{max} + 1$	$f \neq 0$	NaN

D.4.3 NaN

按照传统，将 0/0 或 $\sqrt{-1}$ 视为导致计算终止的不可恢复错误。但是，存在一些示例，表明在这样的情况下继续进行计算是有意义的。以一个子例程为例，该子例程用于查找函数 f 所包含的零，假设此子例程为 `zero(f)`。按照传统，零查找器要求用户输入一个区间 $[a, b]$，函数是在其上定义的，零查找器将按其进行搜索。也就是说，子例程是以 `zero(f, a, b)` 的形式调用的。更有效的零查找器不要求用户输入此额外信息。这种较普遍的零查找器尤其适合于计算器，通常只是键入一个函数，然而不方便的是必须指定定义域。但是，可以很容易地了解到大多数零查找器需要定义域的原因。零查找器是通过在各个值上探查函数 `f` 来执行工作的。如果探查到 `f` 的定义域之外的值，则 `f` 的代码可能也计算 0/0 或 $\sqrt{-1}$，计算将停止，但未必终止零查找过程。

通过引入称为 NaN 的特殊值，并指定诸如 0/0 和 $\sqrt{-1}$ 之类的表达式计算来生成 NaN 而不是停止计算，就可以避免此问题。表 D-3 中列出了一些可以导致 NaN 的情况。这样，当 `zero(f)` 在 `f` 的定义域之外探查时，`f` 的代码将返回 NaN，并且零查找器可以继续执行。也就是说，`zero(f)` 未因作出错误推测而受到“惩罚”。记住此示例，就可以很容易地了解到将 NaN 与普通浮点数结合在一起的结果应该是什么。假定 `f` 的最后一条语句是 `return(-b + sqrt(d))/(2*a)`。如果 $d < 0$，则 `f` 应该返回 NaN。因为 $d < 0$，所以 `sqrt(d)` 是一个 NaN，而且 `-b + sqrt(d)` 将是一个 NaN（如果 NaN 和任何其他数的和是 NaN）。类似地，如果除法运算的一个操作数是 NaN，则商应该是 NaN。通常，每当 NaN 参与浮点运算时，结果都是另一 NaN。

表 D-3 产生 NaN 的运算

操作	产生 NaN 的表达式
+	$\infty + (-\infty)$
×	$0 \times \infty$
/	$0/0, \infty/\infty$
REM	x REM 0, ∞ REM y
$\sqrt{}$	$\sqrt{x}$（当 $x < 0$ 时）

编写不要求用户输入定义域的零解算程序的另一方法是使用信号。零查找器可以为浮点异常安装一个信号处理程序。这样，如果 `f` 在其定义域之外进行计算且引发了一个异常，则将控制权返回给零解算程序。此方法的问题是每种语言都具有不同的信号处理方法（如果它有一个方法的话），因而不具备可移植性。

在 IEEE 754 中，通常将 NaN 表示为带有指数 $e_{max} + 1$ 和非零有效数字的浮点数。实现可以自由地将系统相关的信息置入有效位。这样，就不存在唯一的 NaN，而是整个系列的 NaN。将 NaN 和普通浮点数结合在一起时，结果应该与 NaN 操作数相同。因此，如果长计算的结果是 NaN，则有效数字中的系统相关信息将是生成计算中的第一个 NaN 时产生的信息。实际上，对于最后一条语句，有一个防止误解的说明。如果两个操作数都是 NaN，则结果将是其中的一个 NaN，但它可能不是首先生成的 NaN。

D.4.3.1 无穷

正如 NaN 提供了一种在遇到诸如 0/0 或 $\sqrt{-1}$ 之类的表达式时继续进行计算的方法一样，无穷大提供了一种在发生上溢时继续执行的方法。这比仅返回可表示的最大数要安全得多。以计算 $\sqrt{x^2 + y^2}$ 为例，此时 $\beta = 10$，$p = 3$ 且 $e_{max} = 98$。如果 $x = 3 \times 10^{70}$ 并且 $y = 4 \times 10^{70}$，则 x^2 将发生上溢，并被替换为 9.99×10^{98}。类似地，y^2 和 $x^2 + y^2$ 都将依次上溢，并由 9.99×10^{98} 替换。因此，最终结果将是 $\sqrt{9.99 \times 10^{98}} = 3.16 \times 10^{49}$，该结果绝对是错误的：正确结果应该是 5×10^{70}。在 IEEE 运算中，x^2 的结果是 ∞，y^2、$x^2 + y^2$ 和 $\sqrt{x^2 + y^2}$ 也是这样。因此最终结果是 ∞，这比返回根本不接近于正确结果的普通浮点数要安全。[17]

用 0 除 0 会产生一个 NaN。但是，用一个非零数去除以零则会返回无穷大：1/0 = ∞，-1/0 = -∞。这一区别的原因是：如果在 x 接近某个极限时 $f(x) \to 0$ 且 $g(x) \to 0$，则 $f(x)/g(x)$ 可以具有任意值。例如，如果 $f(x) = \sin x$ 且 $g(x) = x$ 时，则当 $x \to 0$ 时 $f(x)/g(x) \to 1$。但是，如果 $f(x) = 1 - \cos x$，则 $f(x)/g(x) \to 0$。将 0/0 视为两个非常小的数的商的极限情况时，0/0 可以表示任意数。因此，在 IEEE 标准中，0/0 产生一个 NaN。但是，如果 $c > 0$，$f(x) \to c$，且 $g(x) \to 0$，则对于任意分析函数 f 和 g，$f(x)/g(x) \to \pm\infty$。如果对于小的 x，存在 $g(x) < 0$，则 $f(x)/g(x) \to -\infty$，否则极限是 +∞。因此，IEEE 标准定义 $c/0 = \pm\infty$（只要 $c \neq 0$）。通常情况下，∞ 的符号取决于 c 的符号和 0 的符号，这样 -10/0 = -∞，而 -10/-0 = +∞。您可以区分因上溢而得到 ∞ 和因除以零而得到的 ∞，方法是检查状态标志（将在第 26 页的“标志”节中详细讨论）。在第一种情况下将设置上溢标志，而在第二种情况下设置除以零标志。

对于将无穷大作为操作数的运算，确定其结果的规则比较简单：将无穷大替换为有限数 x 并将极限视为 $x \to \infty$。因此，$3/\infty = 0$，因为

$$\lim_{x \to \infty} 3/x = 0 。$$

类似地，$4 - \infty = -\infty$，$\sqrt{\infty} = \infty$。当极限不存在时，结果是 NaN，因此 ∞/∞ 将是一个 NaN（表 D-3 包含其他示例）。这与用于得出 0/0 应为 NaN 的结论的推理是一致的。

17.细微的要点：虽然默认情况下 IEEE 运算会将发生上溢的数据舍入为 ∞，但是更改这一默认设置是可能的（请参见第 26 页的“舍入模式”）

当子表达式计算为 NaN 时，整个表达式的值也是 NaN。但是，对于 ±∞ 的情况，表达式的值可能是一个普通的浮点数，因为存在类似 $1/\infty = 0$ 的规则。下面是利用无穷大运算规则的实际示例。以计算函数 $x/(x^2+1)$ 为例。这个公式并不好，因为不仅它在 x 大于 $\sqrt{\beta}\beta^{e_{max}/2}$，但是无穷大运算规则将给出错误的答案，因为生成的结果为 0，而不是一个接近 $1/x$ 的数。但是，$x/(x^2+1)$ 可被重写为 $1/(x+x^{-1})$。这一改进的表达式将不会过早上溢，并且由于运用无穷大运算规则，当 $x = 0$: $1/(0+0^{-1}) = 1/(0+\infty) = 1/\infty = 0$ 时将生成正确的值。如果不运用无穷大运算规则，表达式 $1/(x+x^{-1})$ 需要对 $x = 0$ 进行测试，这不仅会增加额外的指令，而且也会中断流水线作业。此示例证明一种常规情况，即无穷大运算规则通常无需进行特殊情况检查；但是，需要对公式进行仔细检查以确保它们在无穷大时不会执行伪行为 （如 $x/(x^2+1)$ 时那样）。

D.4.3.2 有符号零

零是由指数 $e_{min} - 1$ 和零有效位数表示的。由于符号位可以具有两个不同的值，所以有两个零：+0 和 -0。如果在比较 +0 和 -0 时进行了区分，则类似 `if (x = 0)` 的简单测试将具有非常不可预知的行为，具体取决于 x 的符号。因此，IEEE 标准定义比较，以便 +0 = -0，而不是 -0 < +0。虽然忽略零的符号始终是可能的，但是 IEEE 标准没有这样做。当乘法或除法涉及有符号的零时，在计算结果符号时就可以应用通常的符号规则。因此，$3\cdot(+0) = +0$，$+0/-3 = -0$。如果零没有符号，则在 $x = \pm\infty$ 时关系 $1/(1/x) = x$ 将无法成立。原因是 $1/-\infty$ 和 $1/+\infty$ 的结果都是 0，而 $1/0$ 的结果是 $+\infty$，符号信息已经丢失。恢复恒等式 $1/(1/x) = x$ 的一种方法是仅具有一种无穷大，但是那将导致丢失上溢数量的符号的灾难性结果。

使用有符号零的另一示例涉及下溢和在 0 处具有不连续性的函数 （如 log）。在 IEEE 运算中，当 $x < 0$ 时将 $\log 0 = -\infty$ 和 $\log x$ 定义为 NaN 是自然的。假设 x 表示一个已经下溢到零的小负数。由于使用了有符号零，因此 x 将是负的，这样 log 可以返回 NaN。但是，如果不存在有符号零，则 log 函数无法区分下溢的负数与 0，因此将不得不返回 $-\infty$。在 0 处具有不连续性的函数的另一示例是 signum 函数，它返回数的符号。

有符号零的最值得关注的用途很可能出现在复杂运算中。举一个简单的例子，假设等式 $\sqrt{1/z} = 1/(\sqrt{z})$ 。当 $z \geq 0$ 时，这肯定是正确的。如果 $z = -1$，很明显计算结果给出 $\sqrt{1/(-1)} = \sqrt{-1} = i$ 和$1/(\sqrt{-1}) = 1/i = -i$ 。因此会出现：$\sqrt{1/z} \neq 1/(\sqrt{z})$ ！可以将此问题归因于以下事实：平方根是多值的，没有方法来选择值，因此它在整个复平面中是连续的。但是，如果不考虑包括所有负实数的**分支切割**，则平方根是连续的。这样，就留下了如何处理负实数的问题，负实数的形式为 $-x + i0$，其中 $x > 0$。有符号零提供了解决此问题的完美方法。形式为 $x + i(+0)$ 的数具有一个符号 ($i\sqrt{x}$)，分支切割另一侧上形式为 $x + i(-0)$ 的数字具有另一符号 ($-i\sqrt{x}$) 。事实上，计算 $\sqrt{\ }$ 的自然公式将给出这些结果。

返回到$\sqrt{1/z} = 1/(\sqrt{z})$ 。如果 $z = 1 = -1 + i0$，那么

$$1/z = 1/(-1 + i0) = [(-1 - i0)]/[(-1 + i0)(-1 - i0)] = (-1 - i0)/((-1)^2 - 0^2) = -1 + i(-0),$$

并且所以 $\sqrt{1/z} = \sqrt{-1 + i(-0)} = -i$，而 $1/(\sqrt{z}) = 1/i = -i$。这样，IEEE 运算为所有 z 保持了此恒等式。一些更复杂的示例由 Kahan [1987] 给出。虽然区分 +0 和 -0 具有优点，但是有时候会令人迷惑。例如，有符号零破坏了关系 $x = y \Leftrightarrow 1/x = 1/y$，在 $x = +0$ 且 $y = -0$ 时它是错误的。但是， IEEE 委员会断定利用零的符号的优点大于其缺点。

D.4.3.3 反向规格化的数

以 $\beta = 10$、 $p = 3$ 且 $e_{min} = -98$ 的正规化浮点数为例。数 $x = 6.87 \times 10^{-97}$ 和 $y = 6.81 \times 10^{-97}$ 看起来完全是普通的浮点数，它们是最小浮点数 1.00×10^{-98} 的 10 倍多。但是，它们具有一个奇怪的属性：$x \ominus y = 0$ （即使 $x \neq y$! 原因是：$x - y = .06 \times 10^{-97} = 6.0 \times 10^{-99}$ 太小以致于无法表示为规格化数，因此它必须清零。保留

$x = y \Leftrightarrow x - y = 0$ 属性有怎样的重要性呢？ (10)

很容易设想到编写以下代码段：`if (x ≠y) then z = 1/(x-y)`，程序会由于伪除以零而在运行了较长的时间后失败。跟踪与此类似的错误既费力又耗时。在更有哲理的级别上，计算机科学教科书通常指出：即使当前证明大程序正确是不切实际的，但利用证明程序的思想来设计程序，这通常会产生更好的代码。例如，即使不变量将不被用作证明的一部分，引入它也是相当有用的。浮点代码就像任何其他代码一样：它有助于具有所依赖的可证明事实。例如，当分析公式 (6) 时，了解 $x/2 < y < 2x \Rightarrow x \ominus y = x - y$ 是非常有帮助的。类似地，如果知道 (10) 是正确的，就可以更轻松地编写可靠的浮点代码。如果它仅对于大部分的数字是正确的，那么不能使用它来证明一切。

IEEE 标准使用反向规格化的[18]数，这保证了 (10) 以及其他有用的关系。它们是该标准中最有争议的部分，这可能就是 754 经过很长时间的延迟才得以批准的原因。声称符合 IEEE 标准的大多数高性能硬件并不直接支持反向规格化的数，而是在使用或产生非正规数时支持陷阱，并将其留给软件以模拟 IEEE 标准。[19] 支持反向规格化的数的思想可追溯到 Goldberg [1967]，这种思想非常简单。当指数是 e_{min} 时，有效数字不必进行规格化，这样在 $\beta = 10$、$p = 3$ 且 $e_{min} = -98$ 时 1.00×10^{-98} 不再是最小的浮点数，因为 0.98×10^{-98} 也是一个浮点数。

当 $\beta = 2$ 且使用隐藏位时，存在一个小的意外困难，因为指数为 e_{min} 的数将始终具有大于或等于 1.0 的有效位数（由于存在隐式前导位）。解决方法与表示 0 所使用的方法类似，并汇总在表 D-2 中。指数 e_{min} 用于表示反向规格的数。更正式的表述是：如果有效数字域中的位是 b_1， b_2， …， b_{p-1}，且指数的值是 e，那么当 $e > e_{min} - 1$ 时，所表示的数是 $1.b_1b_2\ldots b_{p-1} \times 2^e$，而在 $e = e_{min} - 1$ 时，所表示的数是 $0.b_1b_2\ldots b_{p-1} \times 2^{e+1}$。指数中的 +1 是必需的，因为反向规格数的指数是 e_{min}，而不是 $e_{min} - 1$。

请回想一下在本节开头给出的示例：$\beta = 10$， $p = 3$， $e_{min} = -98$， $x = 6.87 \times 10^{-97}$ 且 $y = 6.81 \times 10^{-97}$。对于反向规格的数，$x - y$ 不清零，而改为由反向规格化的数 $.6 \times 10^{-98}$ 来表示。这种行为称为渐进**下溢**。证实使用渐进下溢时 (10) 始终成立是很容易的。

18.在 854 中，它们被称为 *subnormal* “次正规的数”；在 754 中，称为 *denormal* “反向正规的数”。

19.这是导致标准的最棘手方面之一的原因。在使用软件陷阱的硬件上，频繁下溢的程序通常运行得非常慢。

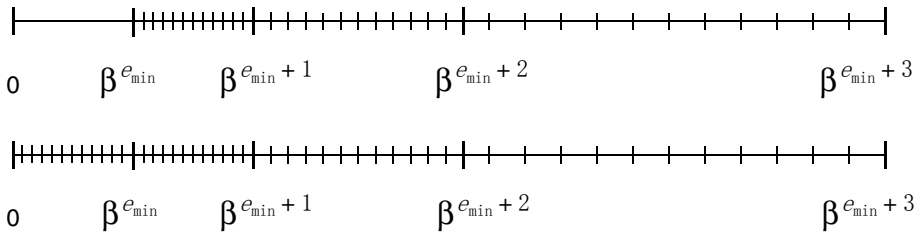


图 D-2 清零与渐进下溢的比较

图 D-2 说明反向规格化的数。图中的上实数直线显示规格化的浮点数。请注意 0 和最小规格化数 $1.0 \times \beta^{e_{min}}$ 之间的间隔。如果浮点计算的结果位于此间隔内，则将其清零。下实数直线显示将反向规格数增加到一组浮点数时的情况。“间隔”被填充，当计算的结果小于 $1.0 \times \beta^{e_{min}}$ 时，将用最接近的反向规格数来表示。将反向规格化的数增加到实数直线时，邻近浮点数之间的间距的变化是有规律的：邻近间距要么长度相同，要么按 β 的因子变化。如果不使用反向规格数，则间距突然从 $\beta^{-p+1}\beta^{e_{\min}}$ 变化到 $\beta^{e_{\min}}$（是 β^{p-1} 的因子），而不是按 β 的因子有规律的变化。由于这一原因，当使用渐进下溢时，对于接近下溢阈值的规格化数会具有较大相对误差的许多算法来说，在此范围内表现良好。

如果不使用渐进下溢，则对于规格化的输入来说，简单表达式 $x - y$ 会具有非常大的相对误差，正如上面所看到的 $x = 6.87 \times 10^{-97}$ 和 $y = 6.81 \times 10^{-97}$ 的相对误差。即使不存在消去，大的相对误差也会发生，如下例所示 [Demmel 1984]。以两个复数 $a + ib$ 和 $c + id$ 相除为例。很明显，公式

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2} \cdot i$$

受到以下问题的影响：如果分母 $c + id$ 的任一部分大于 $\sqrt{\beta}\beta^{e_{max}/2}$，则公式将发生上溢，即使最终结果可能完全处于范围之内也是如此。计算商的一种较好的方法是使用 Smith 的公式：

$$\frac{a+ib}{c+id} = \begin{cases} \dfrac{a+b(d/c)}{c+d(d/c)} + i\dfrac{b-a(d/c)}{c+d(d/c)} & \text{如果}(|d| < |c|) \\ \dfrac{b+a(c/d)}{d+c(c/d)} + i\dfrac{-a+b(c/d)}{d+c(c/d)} & \text{如果}(|d| \geq |c|) \end{cases} \tag{11}$$

将 Smith 的公式应用于 $(2 \cdot 10^{-98} + i10^{-98})/(4 \cdot 10^{-98} + i(2 \cdot 10^{-98}))$ 将得出具有渐进下溢的正确结果 0.5。在清零时它产生 0.4，此时误差为 100 ulp。反向规格化的数用于保证参数的误差边界一直下降到 $1.0 \times \beta^{e_{\min}}$。

D.4.4 异常、标志和陷阱处理程序

在 IEEE 运算中出现类似除以零或上溢等异常情况时，默认设置是传送结果并继续执行。典型的默认结果如下：对于 0/0 和 $\sqrt{-1}$，结果是 NaN；对于 1/0 和上溢，结果是 ∞。前面的几节给出了从具有这些默认值的异常情况中继续执行属于合理操作的示例。当出现任何异常时，还会设置状态标志。要为用户提供一种读取和写入状态标志的方法，需要实施 IEEE 标准。这些标志具有“粘滞性”，因为一旦设置，那么在被显式清除以前它们都将一直保持这种设置。测试标志是区分 1/0（它是来自上溢的真正无穷大）的唯一方法。

有时，在出现异常情况时继续执行是不正确的。第 20 页的“无穷”节给出了示例 $x/(x^2 + 1)$。当 $x > \sqrt{\beta}\beta^{e_{max}/2}$ 时，分母是无穷大，产生最终结果 0，这完全是错误的。虽然此公式可以通过改写为 $1/(x + x^{-1})$ 来解决这个问题，但是改写并不总是能够解决问题。IEEE 标准强烈建议实施要允许安装陷阱处理程序。这样，当出现异常时，将调用陷阱处理程序，而不是设置标志。陷阱处理程序返回的值将用作运算的结果。清除或设置状态标志是陷阱处理程序的责任；否则，允许标志的值是未定义的。

IEEE 标准将异常分成 5 类：上溢、下溢、除以零、无效运算和不精确。每类异常都有单独的状态标志。前三类异常的含义是不言而喻的。无效运算包括表 D-3 中列出的情况，以及产生 NaN 的任何比较。导致无效异常的运算的默认结果是返回一个 NaN，但反过来是不正确的。当某个运算的其中一个操作数是 NaN 时，结果是 NaN，但不引发无效异常，除非该运算还符合表 D-3 中列出的条件之一。[20]

表 D-4 IEEE 754* 中的异常

异常	禁止陷阱时的结果	陷阱处理程序的参数
上溢	$\pm\infty$ 或 $\pm x_{max}$	round($x2^{-\alpha}$)
下溢	0, $\pm 2^{e_{min}}$ 或反向规格数	round($x2^{\alpha}$)
除以零	$\pm\infty$	操作数
无效	NaN	操作数
不精确	round(x)	round(x)

*x 是运算的准确结果，对于单精度有 $\alpha = 192$；对于双精度有 1536，而且 $x_{max} = 1.11\ldots11 \times 2^{e_{max}}$。

当浮点运算的结果不精确时，将引发不精确异常。在 $\beta = 10$、$p = 3$ 系统中，$3.5 \otimes 4.2 = 14.7$ 是精确的，但是 $3.5 \otimes 4.3 = 15.0$ 是不精确的（因为 $3.5 \cdot 4.3 = 15.05$），这将引发不精确异常。第 46 页的“二进制到十进制的转换”讨论了使用不精确异常的算法。表 D-4 中列出了上述五类异常行为的摘要。

20.除非运算中产生“捕获”NaN，否则不引发无效异常。请参见 IEEE 标准 754-1985 的 6.2 节。–编辑者

有一个与以下事实有关的实施问题：不精确异常频繁引发。如果浮点硬件没有自己的标志，而是中断操作系统以发出存在浮点异常的信号，则不精确异常的成本可能会非常高。通过由软件来维护状态标志，可以消除此成本。首次引发异常时，设置相应类别的软件标志，并通知浮点硬件屏蔽该类异常。此后，所有后继异常都将在不中断操作系统的情况下运行。当用户重置这个状态标志时，将重新启用硬件屏蔽。

D.4.4.1 陷阱处理程序

陷阱处理程序的一个明显用途是用于向后兼容。在过去，代码会由于异常的出现而被中止，如今可通过安装陷阱处理程序来中止这样的进程。对于带有类似 `do S until (x >= 100)` 循环的代码，它尤其有用。由于将 NaN 与数值进行 <、≤、>、≥ 或 =（但不包括 ≠）比较操作时，始终会返回 “False”，因此，只要 x 成为 NaN，此代码就将进入无限循环。

在计算诸如 $\Pi_{i=1}^{n} x_i$ 等有可能导致上溢的乘积时，陷阱处理程序有一种更值得关注的用途。一种解决方案是使用对数并改为计算 $\exp(\Sigma \log x_i)$。这种方法存在的问题是：准确性较低，且计算成本高于简单表达式 Πx_i，即使没有出现上溢也是如此。另一种使用陷阱处理程序的解决方案称作**上溢 / 下溢计数**，它可以避免上述两个问题 [Sterbenz 1974]。

该方案的基本思想是：设置一个全局计数器并将其初始化为零。只要 $p_k = \Pi_{i=1}^{k} x_i$ 分乘积对于某个 k 值发生上溢，陷阱处理程序就会将计数器加 1，并将上溢量的指数部分折返后返回。在 IEEE 754 单精度中，$e_{max} = 127$，因此如果 $p_k = 1.45 \times 2^{130}$，则会发生上溢并导致调用陷阱处理程序，该程序将指数限制在范围内，p_k 被更改为 1.45×2^{-62}（请参见下面内容）。类似地，如果 p_k 发生下溢，则将计数器减 1，并将负指数折返为正指数。在完成所有乘法时，如果计数器是零，则最终乘积为 p_n。如果计数器是正数，则说明乘积上溢；如果计数器是负数，则表明乘积下溢。如果乘积的任一部分都没有超出范围，则永远不会调用陷阱处理程序，而且计算也不会引起额外成本。即使存在上溢 / 下溢，计算结果也比使用对数计算的结果更为准确，因为每个 p_k 都是使用完全精度乘法从 p_{k-1} 计算的。Barnett [1987] 讨论了一个公式，上溢 / 下溢计数的完全准确度在该公式的早期表中产生了一个误差。

IEEE 754 规定：当调用上溢或下溢陷阱处理程序时，折返结果将作为参数传递。上溢折返的定义是：计算结果时就好像先计算到无限精度，再除以 2α，然后舍入到相关的精度。对于下溢，将结果乘以 2α。指数 α 是 192（对于单精度）或 1536（对于双精度）。这就是在上面的示例中为什么将 1.45×2^{130} 转换为 1.45×2^{-62} 的原因。

D.4.4.2 舍入模式

在 IEEE 标准中，每当运算结果不精确时都将出现舍入操作，因为每个运算都是先精确计算然后再进行舍入（二进制与十进制转换除外）。默认情况下，舍入是指向最相近的数据进行舍入。该标准要求提供其他三种舍入模式，即向 0 舍入、向 +∞ 舍入和向 -∞ 舍入。在用于转换为整数运算时，向 -∞ 舍入使转换变成“取最小”函数，而向 +∞ 舍入使转换变成“取上限”函数。舍入模式影响上溢，因为当向 0 舍入或向 -∞ 舍入生效时，正量的上溢使得默认结果成为最大的可表示正数，而不是 +∞。类似地，在向 +∞ 舍入或向 0 舍入生效时，负量的上溢将产生最大的可表示负数。

舍入模式的一个应用发生在区间运算中（另一个应用将在第 46 页的“二进制到十进制的转换”中提及）。当使用区间运算时，两个数 x 和 y 的和是一个区间$[\underline{z}, \overline{z}]$，其中 $\underline{z}$ 是向 -∞ 舍入的 $x \oplus y$，$\overline{z}$ 是向 +∞ 舍入的 $x \oplus y$。加法运算的精确结果包含在区间 $[\underline{z}, \overline{z}]$ 中。如果不使用舍入模式，则区间运算常常通过计算$\underline{z} = (x \oplus y)(1 - \varepsilon)$ 和 $\overline{z} = (x \oplus y)(1 + \varepsilon)$，其中$\varepsilon$ 是计算机厄普西隆。[21] 这导致过高估计区间的大小。因为区间运算中的运算结果是一个区间，所以一般情况下运算输入也将是一个区间。如果增加了两个区间 $[\underline{x}, \overline{x}]$ 和$[\underline{y}, \overline{y}]$，结果是$[\underline{z}, \overline{z}]$，其中 $\underline{z}$ 是 $\underline{x} \oplus \underline{y}$（其舍入模式设置为向 -∞ 舍入），$\overline{z}$ 是 $\underline{z} \oplus \underline{y}$（其舍入模式设置为向 +∞ 舍入）。

当使用区间运算执行浮点计算时，最终结果是一个包含精确计算结果的区间。如果结果区间很大（它通常是这样），则这不是非常有帮助。因为正确结果可以处于该区间内的任意位置。当与多精度浮点包联合使用时，区间运算更有意义。首先通过某个精度 p 执行计算。如果区间运算指出最终结果可能不准确，则使用越来越高的精度重新计算，直到最终区间达到合理大小。

D.4.4.3 标志

IEEE 标准有许多标志和模式。如前所述，以下五类异常中的每一类都具有一个状态标志：下溢、上溢、除以零、无效运算和不精确。舍入模式有以下四种：向最相近的数据舍入、向 +∞ 舍入、向 0 舍入和向 -∞ 舍入。强烈建议对于五类异常中的每一类都设置一个启用模式位。此节给出了一些简单的示例，说明如何利用这些模式和标志。更复杂的示例将在第 46 页的“二进制到十进制的转换”节中进行讨论。

21. $\underline{Z}$ 可能大于 $\overline{Z}$（如果 x 和 y 都是负数）。– 编辑者

我们来考虑编写一个计算 x^n 的子例程，其中 n 是一个整数。当 $n > 0$ 时，简单例程类似于下面的程序段

```
PositivePower(x,n) {
 while (n is even) {
     x = x*x
     n = n/2
 }
 u = x
 while (true) {
     n = n/2
     if (n==0) return u
     x = x*x
     if (n is odd) u = u*x
 }
```

如果 $n < 0$，则计算 x^n 的更精确方法不是调用 `PositivePower(1/x, -n)`，而是调用 `1/PositivePower(x, -n)`，因为第一个表达式乘以 n 个数量，其中每个数量都具有来自除法运算（即 $1/x$）的舍入误差。在第二个表达式中它们是精确的（即 x），而且最后的除法只产生一个附加的舍入误差。可惜的是，此策略有一个微小的缺陷。如果 `PositivePower(x, -n)` 出现下溢，那么要么调用下溢陷阱处理程序，要么将设置下溢状态标志。这是错误的，因为如果 x^{-n} 出现下溢，则 x^n 将上溢或处于范围内。[22] 但是，由于 IEEE 标准允许用户访问所有标志，因此子例程可以轻松地为此进行更正。它只需关闭上溢和下溢陷阱启用位，并保存上溢和下溢状态位。然后计算 `1/PositivePower(x, -n)` 即可。如果既没有设置上溢状态位也没有设置下溢状态位，则子例程将它们与陷阱启用位一起恢复。如果设置了其中一种状态位，则子例程恢复标志并使用 `PositivePower(1/x, -n)`（它导致出现正确的异常）重新计算。

另一个使用标志的示例发生在通过以下公式计算余弦时

$$\arccos x = 2 \arctan \sqrt{\frac{1-x}{1+x}}。$$

如果 arctan(∞) 计算为 $\pi/2$，则 arccos(-1) 将正确计算为 $2 \cdot \arctan(\infty) = \pi$（因为是无限运算）。但是，有一个小缺陷，因为 $(1 - x)/(1 + x)$ 的计算结果将导致除以零异常标志的设置，即使 arccos(-1) 不属于异常也是如此。此问题的解决方法很简单。只需在计算反余弦之前保存除以零标志的值，然后在计算之后恢复其旧值即可。

22. 它可以在范围内，因为如果 $x < 1$，$n < 0$ 且 x^{-n} 只是稍微小于下溢阈值 $2^{e_{\min}}$，则 $x^n \approx 2^{-e_{\min}} < 2^{e_{\max}}$，因此可能不上溢，因为在所有 IEEE 精度中，$-e_{\min} < e_{\max}$。

D.5 系统方面

对于计算机系统来说，几乎每个方面的设计都需要有关浮点的知识。计算机架构通常具有浮点指令，编译器必须生成那些浮点指令，而且操作系统必须决定当出现引发那些浮点指令的异常情况时所要执行的操作。计算机系统设计人员很少从数值分析文本中获得指导，这些文本通常针对软件的用户和编写人员，而不是针对计算机设计人员。作为似乎合理的设计决策如何导致未预料行为的示例，请参考下面的 BASIC 程序。

```
q = 3.0/7.0
if q = 3.0/7.0 then print "Equal":
    else print "Not Equal"
```

在 IBM PC 上使用 Borland 的 Turbo Basic 进行编译和运行时，该程序打印 `Not Equal!` 此示例将在下一节中进行分析

顺便说一下，有些人认为此类异常的解决方法决不是比较浮点数以确定是否相等，而是当它们处于某个误差界限 E 内时将其视为相等。这根本不是一个解决一切问题的“灵药”，因为它提出的问题与它解决的问题一样多。E 的值应该是多少？如果 $x < 0$ 和 $y > 0$ 都在 E 内，那么是否确实应该将它们视为相等，即使它们具有不同的符号？此外，此规则定义的关系 $a \sim b \Leftrightarrow |a - b| < E$ 不是等价关系，因为 $a \sim b$ 且 $b \sim c$ 不能得出 $a \sim c$。

D.5.1 指令集

某种算法为了产生精确结果而需要更高精度的短暂成组传送，这是相当常见的。在二次公式中有这样的示例 $(-b \pm \sqrt{b^2 - 4ac})/2a$。如第 43 页的“定理 4 的证明”节中所述，当 $b^2 \approx 4ac$ 时，舍入误差最多可以影响使用二次公式计算的根中的一半数位。通过以双精度执行子计算 $b^2 - 4ac$ 时，会丢失根的双精度位的一半，这意味着将保留所有单精度位。

如果有一条对两个单精度数进行操作并产生双精度结果的乘法指令，则当 a、b 和 c 都是单精度数时按双精度计算 $b^2 - 4ac$ 是很容易的。为了产生两个 p 位数的精确舍入乘积，乘法器需要生成整个 $2p$ 位的乘积，尽管它可能在继续执行时抛弃位。这样，从单精度操作数计算双精度乘积的硬件通常只比单精度乘法器的成本略高一些，而且比双精度乘法器的成本低很多。尽管如此，目前的指令集往往仅提供生成结果与操作数精度相同的指令。[23]

如果作用两个单精度操作数以产生双精度乘积的指令仅适用于二次公式，那么将该指令增加到指令集中是不值得的。但是，此指令有许多其他用途。以解决线性方程组的问题为例，

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

23.这可能是因为设计人员喜欢“正交”指令集，在这样的指令集中浮点指令的精度与实际运算无关。乘法的特殊情况会破坏这一正交性。

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

$$\cdots$$

$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

可以按矩阵形式将它写为 $Ax = b$，其中

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2(1)n} \\ & & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

假定按某种方法（或许是高斯消元法）求出了一个解 $x^{(1)}$。那么有一种简单的方法可改进结果的准确度，这种方法称为**迭代改进**。首先计算

$$\xi = Ax^{(1)} - b \tag{12}$$

然后对方程组进行求解

$$Ay = \xi \tag{13}$$

请注意，如果 $x^{(1)}$ 是一个精确解，则 ξ 是零向量，y 也是零向量。通常，ξ 和 y 的计算会引起舍入误差，因此 $Ay \approx \xi \approx Ax^{(1)} - b = A(x^{(1)} - x)$，其中 x 是 （未知的）正确解。那么，$y \approx x^{(1)} - x$，因此解的改进估计是

$$x^{(2)} = x^{(1)} - y \tag{14}$$

重复执行三个步骤 (12)、 (13) 和 (14)，用 $x^{(2)}$ 替换 $x^{(1)}$，用 $x^{(3)}$ 替换 $x^{(2)}$。这里关于 $x^{(i+1)}$ 比 $x^{(i)}$ 更精确的论述仅是非正式的。有关详细信息，请参见 [Golub 和 Van Loan 1989]。

在执行迭代改进时， ξ 是一个向量，其元素是邻近不精确浮点数的差，因此会受到恶性抵消的影响。这样，迭代改进不是非常有用，除非 $\xi = Ax^{(1)} - b$ 是按双精度计算的。同样，这是计算两个单精度数 （A 和 $x^{(1)}$）的乘积的示例，它需要完全双精度结果。

总而言之，如果存在这样的指令：对两个浮点数做相乘操作并返回两倍于被操作数精度的乘积，那么将该指令增加到指令集中是很有用的。对于编译器而言，这其中的一些蕴涵将在下一节中进行讨论。

D.5.2 语言和编译器

编译器和浮点运算理论的相互影响在 Farnum [1988] 中有所讨论，本节中的讨论大都出自该论文。

D.5.2.1 含混性

理想情况下，一种语言的定义可以阐述该语言的语义，这种阐述应该足够准确，以便可以证明有关程序的声明。尽管对于语言整数节的定义通常可以满足上述要求，但涉及到浮点节时，语言定义通常具有很大的模糊性。其中的原因可能出自于以下事实：许多语言设计人员认为无法证明有关浮点的任何陈述，因为它会引起舍入误差。如果是这样，那么前面几节已经证明这一推理乃是谬论。此部分讨论语言定义中一些常见的模糊性，包括有关如何处理它们的建议。

很明显，某些语言没有清楚地指定如果 `x` 是浮点变量（例如，具有值 `3.0/10.0`），则（例如） `10.0*x` 的每次出现都必须具有相同值。以基于布朗模型的 Ada 为例，它似乎隐含着浮点运算只需满足布朗公理，这样表达式就可以具有多个可能值之一。以这种模糊的方式考虑浮点与 IEEE 模型形成了鲜明的对比，对于后者，每个浮点运算的结果都是精确定义的。在 IEEE 模型中，我们可以证明 `(3.0/10.0)*10.0` 计算为 `3`（定理 7）。在布朗模型中，我们却做不到这一点。

大多数语言定义中的另一种含混性与出现上溢、下溢以及其他异常时所发生的情况有关。IEEE 标准可以准确地指定异常的行为，因此将该标准作为模型的语言可以避免有关这一点的任何含混性。

另一模糊性与对圆括弧的解释有关。由于存在舍入误差，代数学的结合律不一定适用于浮点数。例如，当 $x = 10^{30}$，$y = -10^{30}$ 且 $z = 1$ 时，表达式 `(x+y)+z` 的结果与 `x+(y+z)` 的结果完全不同（在前一种情况下结果为 1，在后一种情况下结果是 0）。应该说，对于保留圆括弧的重要性怎么强调也不过分。定理 3、4 和 6 中提供的算法都取决于它。例如，在定理 6 中，如果不使用圆括号，则公式 $x_h = mx - (mx - x)$ 将简化为 $x_h = x$，因此破坏了整个算法。对于浮点运算，不要求先计算圆括弧中内容的语言定义是无用的。

子表达式计算在许多语言中没有精确定义。假定 ds 是双精度，但是 x 和 y 是单精度。那么，在表达式 ds + x*y 中，乘积是按单精度还是按双精度计算？另一个示例：在 x + m/n （其中 m 和 n 是整数）中，除法是整数运算还是浮点运算？有两种方法可以处理这类问题，其中的任意一种都不完全令人满意。第一种方法是，要求表达式中的所有变量具有相同的类型。这是最简单的解决方法，但是它有一些缺点。首先，具有子范围类型的语言 （如 Pascal）允许混合子范围变量和整型变量，因此禁止混合单精度变量和双精度变量就令人觉得有点奇怪了。另一个问题与常数有关。在表达式 0.1*x 中，大多数语言将 0.1 解释为单精度常数。现在，假定编程人员决定将所有浮点变量的声明从单精度更改为双精度。如果仍然将 0.1 视为单精度常数，则将出现编译时错误。编程人员将必须搜寻到每个浮点常数并对其进行更改。

第二种方法是，允许使用混合表达式，在这种情况下必须提供用于子表达式计算的规则。有许多指导性示例。 C 的最初定义要求按双精度计算每个浮点表达式 [Kernighan 和 Ritchie 1978]。这将导致出现与此节开始处提供的示例类似的异常。表达式 3.0/7.0 是按双精度计算的，但是，如果 q 是一个单精度变量，则将商舍入为单精度以便于存储。由于 3/7 是一个循环二进制分数，因此它的按双精度计算的值与它以单精度存储的值是不同的。这样，比较 $q = 3/7$ 将失败。这表明按最高可用精度计算每个表达式的值并不是一个好规则。

另一个指导性示例是内积。如果内积有数千项，则总和中的舍入误差可能会变得相当大。减少此舍入误差的一种方法是按双精度求和 （这一点将在第 34 页的 “优化器” 节中进行更详细的讨论）。如果 d 是一个双精度变量，且 x[] 和 y[] 是单精度数组，则内积循环将看起来就类似于 d = d + x[i]*y[i]。如果乘法是按单精度进行的，那么会失去双精度累积的许多优点，因为就在增加到双精度变量之前乘积被按照单精度进行截断。

涉及前面两个示例的规则是，按照出现在表达式中的任何变量的最高精度计算该表达式。那么，q = 3.0/7.0 将完全按单精度计算，[24]并具有布尔值 true，而 d = d + x[i]*y[i] 将按双精度计算，获得了双精度累积的全部优点。但是，此规则过于简单，无法完全包括所有情况。如果 dx 和 dy 是双精度变量，则表达式 y = x + single(dx-dy) 包含一个双精度变量，但是按双精度进行求和将是无意义的，因为这两个操作数都是单精度数，结果也是单精度数。

更复杂的子表达式计算规则如下。首先，为每个运算指定试验性精度，该精度是其操作数的最高精度。这种指定必须按表达式树中从叶到根的顺序执行。然后从根到叶执行第二遍。这一次，为每个运算指定最高试验性精度和它的父运算所需要的精度。在 q = 3.0/7.0 情况下，每个叶都是单精度的，因此所有运算都是按单精度执行的。在 d = d + x[i]*y[i] 情况下，乘法运算的试验性精度是单精度，但是在第二遍中它被提升到双精度，因为其父运算需要双精度操作数。而且，在 y = x + single(dx-dy) 中，加法运算是按单精度执行的。 Farnum [1988] 提供了实现此算法并不困难的证据。

24.这假定有以下常用约定：3.0 是一个单精度常数，而 3.0D0 是一个双精度常数。

此规则的缺点是，子表达式的计算取决于它被嵌入的表达式。这会产生一些令人生厌的结果。例如，假定您正在调试程序并希望知道子表达式的值。您不能简单地将子表达式键入调试器，并要求计算它，因为在程序中子表达式的值取决于它被嵌入的表达式。关于子表达式的最后一个注释是：由于将十进制常数转换为二进制是一个运算，因此计算规则还影响对十进制常数的解释。对于不能以二进制精确表示的常数（如 `0.1`），这一点尤其重要。

当一种语言将求幂作为其内置运算之一包括在内时，将出现另一个潜在模糊性。与基本算术运算不同，求幂运算的值并不总是显而易见的 [Kahan 和 Coonen 1982]。如果 `**` 是求幂运算符，则 `(-3)**3` 一定具有值 -27。但是，`(-3.0)**3.0` 是有疑问的。如果 `**` 运算符检查整数幂，那么它将 `(-3.0)**3.0` 计算为 $-3.0^3 = -27$。另一方面，如果公式 $x^y = e^{y \log x}$ 用于为实参定义 `**`，则结果可以是 NaN（当 $x < 0$ 时，使用 $\log(x) =$ `NaN` 的自然定义），这取决于 log 函数。但是，如果使用 FORTRAN `CLOG` 函数，则结果将是 -27，因为 ANSI FORTRAN 标准将 `CLOG(-3.0)` 定义为 $i\pi + \log 3$ [ANSI 1978]。程序设计语言 Ada 通过仅为整数幂定义求幂运算避免了这一问题，而 ANSI FORTRAN 禁止对负数进行实数幂操作。

事实上， FORTRAN 标准规定

> 禁止任何其结果在数学上没有定义的算术运算……

可惜的是，随着 IEEE 标准引入 ±∞，**在数学上没有定义**的含义不再是完全明确的。一种定义可能是使用第 20 页的 “无穷” 节中所示的方法。例如，要确定 a^b 的值，请考虑非常值解析函数 f 和 g，它们具有以下属性：当 $x \to 0$ 时，存在 $f(x) \to a$ 和 $g(x) \to b$。如果 $f(x)^{g(x)}$ 总是接近同一极限，则这应该是 a^b 的值。此定义将设置 $2^\infty = \infty$，这似乎是相当合理的。在 1.0^∞ 的情况下，当 $f(x) = 1$ 且 $g(x) = 1/x$ 时，极限接近 1，但是当 $f(x) = 1 - x$ 且 $g(x) = 1/x$ 时，极限是 e^{-1}。因此 1.0^∞ 应该是一个 NaN。在 0^0 的情况下，$f(x)^{g(x)} = e^{g(x) \log f(x)}$。因为 f 和 g 都是可解析的并在 0 处具有值 0，所以 $f(x) = a_1x^1 + a_2x^2 + \ldots$ 且 $g(x) = b_1x^1 + b_2x^2 + \ldots$。这样，
$\lim_{x \to 0} g(x) \log f(x) = \lim_{x \to 0} x \log(x(a_1 + a_2x + \ldots)) = \lim_{x \to 0} x \log(a_1x) = 0$。因此，对于 f 和 g，$f(x)^{g(x)} \to e^0 = 1$，这意味着 $0^0 = 1$.[25] [26] 使用此定义将明确地定义所有参数的指数函数，尤其是将 `(-3.0)**3.0` 定义为 -27。

25.结论 $0^0 = 1$ 取决于以下限制：f 是非常值函数。如果去掉此限制，那么允许 f 是恒等于 0 的函数就会将 0 作为 $\lim_{x \to 0} f(x)^{g(x)}$ 的可能值，因此必须将 0^0 定义为一个 NaN。

26.在 0^0 的情况下，可以提出似乎合理的论点，但令人信服的论点则包括在 “Concrete Mathematics”（作者：Graham、 Knuth 和 Patashnik）中，该论点是：要使二项式定理成立，必须满足 $0^0 = 1$。—编辑者

D.5.2.2 IEEE 标准

第 14 页的“IEEE 标准”节讨论了 IEEE 标准的许多特性。但是，IEEE 标准没有说明如何通过程序设计语言获得这些特性。因而，在支持该标准的浮点硬件与程序设计语言（如 C、Pascal 或 FORTRAN）之间常常存在着不匹配。某些 IEEE 功能可以通过子例程调用库获得。例如，IEEE 标准要求精确舍入平方根，而平方根函数通常是直接在硬件中实现的。通过平方根例程库可以轻松地获得此功能。但是，该标准的其他方面不能通过子例程来轻松实现。例如，大多数计算机语言最多指定两种浮点类型，而 IEEE 标准具有四种不同精度（尽管建议的配置是单精度加上单精度扩展或者单精度、双精度和双精度扩展）。另一示例是无穷大。表示 $\pm\infty$ 的常量可以由子例程提供。但是这样一来，它们可能无法用在需要常量表达式的位置上，例如用来初始化一个常量。

更微妙的情况是操纵与计算关联的状态，该状态由舍入模式、陷阱启用位、陷阱处理程序和异常标志组成。一种方法是为读取和写入状态提供子例程。此外，能够以原子方式设置新值并返回旧值的单次调用通常很有用。如第 26 页的“标志”节中的示例所示，修改 IEEE 状态的一种常见模式为仅在块或子例程的作用域内更改它。这样，查找块的每个出口并确保状态已恢复是编程人员的责任。如果语言支持在块的作用域 内精确设置状态，那将是很有用的。 Modula-3 是一种为陷阱处理程序实现此思想的语言 [Nelson 1991]。

在一种语言中实施 IEEE 标准时，许多次要事项还需考虑。因为对于所有 x，$x - x = +0$，所以 [27] $(+0) - (+0) = +0$。但是，$-(+0) = -0$，因此不应该将 $-x$ 定义为 $0 - x$。NaN 的引入可能是令人迷惑的，因为 NaN 永不等于任何其他数（包括另一个 NaN），因此 $x = x$ 不再总是正确的。事实上，如果没有提供 IEEE 推荐的函数 `Isnan`，则表达式 $x \neq x$ 是测试 NaN 的最简单方法。此外，NaN 不与所有其他数一起排序，因此不能将 $x \leq y$ 定义为**不是** $x > y$。由于 NaN 的引入导致浮点数变成部分排序的，因此使用返回 <、 =、 > 或 *unordered* 之一的 `compare` 函数会让编程人员可以更轻松地处理比较操作。

虽然 IEEE 标准规定如果任一操作数是 NaN 则基本浮点运算返回 NaN，但是这不可能始终是复合运算的最佳定义。例如，当计算绘制曲线图要使用的适当比例因子时，必须计算一组值的最大值。在这种情况下，最大值运算简单地忽略 NaN 是有意义的。

最后，舍入可以是一个问题。IEEE 标准非常精确地定义了舍入，而且它依赖于舍入模式的当前值。有时，这与类型转换中隐式舍入的定义或语言中的显式 `round` 函数存在冲突。也就是说，希望使用 IEEE 舍入的程序无法使用自然语言的基本要素，反过来，语言基本要素无法在数量不断增加的 IEEE 计算机上进行实施。

27.除非舍入模式是向 $-\infty$ 舍入（在这种情况下，$x - x = -0$）。

D.5.2.3 优化器

编译器教科书往往忽略浮点这一主题。例如， Aho 等 [1986] 提到将 `x/2.0` 替换为 `x*0.5`，引导读者假定 `x/10.0` 应该替换为 `0.1*x`。但是，在二进制计算机上这两个表达式并不具有相同的语义，因为 0.1 无法以二进制精确表示。此教科书还建议将 `x*y-x*z` 替换为 `x*(y-z)`，尽管我们已经看到在 $y \approx z$ 时这两个表达式可以具有完全不同的值。尽管其中指出优化器不应违反语言定义，从而对 “可以使用任意代数恒等式优化代码”这种说法做了限制，但是它给人们留下了浮点语义并不太重要的印象。不管语言标准是否指定必须先计算圆括弧中的内容， `(x+y)+z` 都可以具有与 `x+(y+z)` 完全不同的结果，如上所述。存在一个与保留圆括弧密切相关的问题，如以下代码所示：

```
eps = 1;
do eps = 0.5*eps; while (eps + 1 > 1);
```

此代码旨在估算出计算机厄普西隆。如果进行优化的编译器注意到 $eps + 1 > 1 \Leftrightarrow eps > 0$，将完全更改程序。它将不计算最小数 x （以使 $1 \oplus x$ 仍然大于 1 ($x \approx e \approx \beta^{-p}$)，而是计算最大数 x （对于它， $x/2$ 舍入为 0 ($x \approx \beta^{e_{\min}}$)。避免这种 “优化”是如此重要以致于值得提供另一被其完全毁坏的非常有用的算法。

许多问题 （如数值积分和微分方程的数值解）涉及计算多个项的和。因为每个加法运算都有可能引入大至 .5 ulp 的误差，所以涉及数千项的求和会具有相当大的舍入误差。纠正这一点的简单方法是将部分被加数存储在双精度变量中，并使用双精度执行每个加法运算。如果计算是使用单精度进行的，则在大多数计算机系统上按双精度执行求和是很容易的。但是，如果已经按双精度进行了计算，则使精度加倍就不那么简单了。有时建议的一种方法是对数值进行排序，然后按照从最小到最大的顺序将其相加。但是，有一种大大提高求和准确度，而且效率高得多的方法，即

D.5.2.4 定理 8 （Kahan 求和公式）

假设 $\Sigma_{j=1}^{N} x_j$ 是使用以下算法计算的

```
S = X[1];
C = 0;
for j = 2 to N {
    Y = X[j] - C;
    T = S + Y;
    C = (T - S) - Y;
    S = T;
}
```

那么，计算的和 S *等于* $\Sigma x_j(1+\delta_j) + O(N\varepsilon^2)\Sigma|x_j|$，*其中* ($|\delta_j| \le 2\varepsilon$) 。

使用简单公式 Σx_j，计算的和等于 $\Sigma x_j(1+\delta_j)$，其中 $|\delta_j| < (n-j)e$。将此与 Kahan 求和公式中的误差进行比较，可以看出有显著的改进。每个被加数受到只有 $2e$ 的扰动，而不是简单公式中多至 ne 的扰动。有关详细信息，请参见第 47 页的 “求和中的误差”。

认为浮点运算遵循代数法则的优化器将得出以下结论：$C = [T-S] - Y = [(S+Y)-S] - Y = 0$，从而认为该算法毫无用处。可以这样概括这些示例：在涉及浮点变量的表达式中应用代数恒等式进行优化时，应该极其小心，尽管这些恒等式在数学上对所有实数成立。

优化器可以更改浮点代码语义的另一种方法涉及到常数。在表达式 `1.0E-40*x` 中，隐含了一个十进制到二进制的转换操作，该操作将十进制数转换成了二进制常数。由于此常数无法以二进制精确表示，因此会引发不精确异常。此外，如果表达式是按单精度计算的，则下溢标志将被设置。由于常数是不精确的，因此其到二进制的精确转换取决于 IEEE 舍入模式的当前值。这样，在编译时将 `1.0E-40` 转换为二进制数的优化器将更改程序的语义。但是，可以按最低可用精度精确表示的常数 （如 27.5）能够在编译时安全地进行转换，因为它们始终是精确的，不会引发任何异常且不受舍入模式影响。希望在编译时就进行转换的常数应该使用常量声明，例如 `const pi = 3.14159265`。

关于优化可以更改浮点语义的另一示例是公共子表达式的消除，如以下代码所示

```
C = A*B;
RndMode = Up
D = A*B;
```

虽然 `A*B` 可能看上去是一个公共子表达式，但实际上并不是，因为两个计算位置的舍入模式有所不同。最后三个示例：$x = x$ 不能由布尔常量 `true` 替换，因为当 x 是一个 NaN 时前者将失败；当 $x = +0$ 时，$-x = 0 - x$ 将失败；而且， $x < y$ 并不与 $x \geq y$ 相反，因为 NaN 既不大于也不小于普通浮点数。

尽管存在这些示例，但是仍然存在着一些可以对浮点代码进行的有用优化。首先，存在对浮点数有效的代数恒等式。 IEEE 运算中的一些示例是：$x + y = y + x$、$2 \times x = x + x$、$1 \times x = x$ 和 $0.5 \times x = x/2$。但是，甚至这些简单的恒等式也会在一些计算机（如 CDC 和 Cray 超级计算机）上失败。指令调度和内联过程替换是其他两种可能有用的优化。[28]

作为最后一个示例，请考虑表达式 `dx = x*y`，其中 `x` 和 `y` 是单精度变量，`dx` 是双精度变量。在具有将两个单精度数相乘以产生双精度数的指令的计算机上，可以将 `dx = x*y` 映射到该指令，而不是编译为将操作数转换为双精度数再执行双精度数与双精度数相乘的一系列指令。

28.VAX 上的 VMS 数学库使用弱形式的内联过程替换，因为它们使用转向子例程调用的廉价跳转，而不是使用慢速的 CALLS 和 CALLG 指令。

一些编译器编写人员认为禁止将 $(x + y) + z$ 转换为 $x + (y + z)$ 的限制与他们无关，只有使用不可移植方法的编程人员才关心此限制。也许他们认为浮点数与实数非常类似，应该遵循与实数相同的定律。实数语义的问题是它们的实现成本极高。每次将两个 n 位数相乘时，积将具有 $2n$ 位。每次将两个指数相差很大的 n 位数相加时，和中的位数是 n + 指数之间的差值。和最多可以具有 $(e^{max} - e^{min})$ + n 位，或大约 $2 \cdot e^{max}$ + n 位。产生数千运算的算法 （例如对线性方程组求解）不久将对具有许多有效位的数进行运算，且运算速度慢得令人觉得无望。库函数（例如 sin 和 cos）的实现甚至更困难，因为这些超越函数的值不是有理数。精确的整数运算通常由 lisp 系统提供，对于解决某些问题是很方便的。但是，精确的浮点运算几乎是没有用的。

事实是，存在利用 $(x + y) + z \neq x + (y + z)$ 事实的有用算法 （如 Kahan 求和公式），而且只要界限

$$a \oplus b = (a + b)(1 + \delta)$$

成立 （以及 小 -、× 和 / 的类似界限），这些算法就是适用的。由于这些界限对于几乎所有商业硬件都是成立的，因此数值编程人员忽略这样的算法将是不明智的，编译器编写人员通过伪称浮点变量具有实数语义来破坏这些算法将是不负责任的。

D.5.3 异常处理

到目前为止所讨论的主题主要涉及准确性和精度的系统含意。陷阱处理程序也引发了一些值得关注的系统问题。IEEE 标准强烈建议用户应该能够为五类异常中的每一类都指定陷阱处理程序，第 25 页的“陷阱处理程序”节给出了用户定义的陷阱处理程序的一些应用。在出现无效运算异常和除以零异常的情况下，应该为处理程序提供操作数，否则应该提供精确舍入的结果。根据所使用的程序设计语言，陷阱处理程序也许还能够访问程序中的其他变量。对于所有异常，异常处理程序必须能够识别正在执行什么运算以及该运算的目标精度是什么。

IEEE 标准假定运算在概念上是串行的，当发生中断时，识别出运算及其操作数是可能的。在采用流水线技术或具有多个运算单元的计算机上，当出现异常时，只是让陷阱处理程序检查程序计数器可能是不够的。可能需要能够精确识别出捕获哪个运算的硬件支持。

另一问题如以下程序片段所示。

```
x = y*z;
z = x*w;
a = b + c;
d = a/x;
```

假定第二个乘法运算引发一个异常，陷阱处理程序希望使用 a 的值。在可以并行执行加法运算和乘法运算的硬件上，优化器有可能将加法运算提到第二个乘法运算之前，以便加法运算可以与第一个乘法运算并行进行。这样，当第二个乘法运算被捕获时，a = b + c 已经执行，潜在地更改了 a 的结果。编译器避开此类优化将不是合理的，因为每个浮点运算都有可能被捕获，从而几乎所有指令调度优化都将被消除。通过禁止陷阱处理程序直接访问程序的任何变量，可以避免这一问题。取而代之，可以将操作数或结果作为参数提供给处理程序。

但是，仍然存在问题。在以下片段中

```
x = y*z;
z = a + b;
```

这两个指令也许能够并行执行。如果乘法运算被捕获，则其参数 z 可能已经被加法运算覆盖，尤其是因为加法运算的速度通常比乘法运算快。支持 IEEE 标准的计算机系统必须提供保存 z 值的某种方法，要么在硬件中，要么让编译器首先避免这样的情况。

W. Kahan 曾提议使用**预替换**代替陷阱处理程序以避免这些问题。在此方法中，用户指定一个异常以及希望在出现该异常时用作结果的值。作为一个示例，假定在用于计算 (sin x)/x 的代码中，用户断定 $x = 0$ 如此少见以致于避免测试 $x = 0$ 将提高性能，因此改为在出现 0/0 陷阱时处理这种情况。当使用 IEEE 陷阱处理程序时，用户将编写返回值 1 的处理程序并在计算 sin x/x 之前安装它。当使用预替换时，用户将指定发生无效运算时应该使用值 1。Kahan 将此称为预替换，因为要使用的值必须在出现异常之前指定。而当使用陷阱处理程序时，要返回的值可以在出现陷阱时进行计算。

预替换的优点是它具有简单的硬件实现。[29] 一旦确定异常的类型，就可以使用它为包含所需运算结果的表建立索引。虽然预替换具有一些吸引人的属性，但是 IEEE 标准的广泛认可使其不大可能被硬件制造商广泛执行。

D.6 详细资料

在本文中已经作出了许多有关浮点运算属性的断言。现在，我们继续说明浮点并非是神秘的魔术，而是一个可以在数学上检验其断言的简单易懂的主题。说明分为三个部分。第一部分介绍误差分析，并提供第 2 页的 “舍入误差” 节的详细资料。第二部分探讨二进制到十进制的转换，对第 14 页的 “IEEE 标准” 节进行一些补充。第三部分讨论 Kahan 求和公式，它曾在第 28 页的 “系统方面” 节中用作一个示例。

29.预替换的困难在于它需要直接硬件实现或可持续的浮点陷阱 （如果在软件中实现）。– 编辑者

D.6.1 舍入误差

在讨论舍入误差时，已经指出单个保护数位足以保证加法运算和减法运算将始终是准确的（定理 2）。现在我们继续检验这一事实。定理 2 分为两个部分，一个部分用于减法运算，另一部分用于加法运算。用于减法运算的部分是

D.6.1.1 定理 9

如果 x 和 y 是正的浮点数，其格式为具有参数 β 和 p，而且减法运算是使用 $p + 1$ 位（也就是使用一个保护数位）进行的，则结果中的相对舍入误差小于

$$\left(\frac{\beta}{2} + 1\right)\beta^{-p} = \left(1 + \frac{2}{\beta}\right)e \leq 2e \text{ 。}$$

D.6.1.2 证明

有必要的话，x 和 y 可以互换，因此不妨认为 $x > y$。同样，不妨将 x 和 y 按比例缩放，以便 x 可以表示成 $x_0.x_1 \ldots x_{p-1} \times \beta^0$。如果将 y 表示为 $y_0.y_1 \ y_{p-1}$，则差值是精确的。如果将 y 表示为 $0.y_1 \ldots y_p$，则保护数位确保计算的差值将是舍入为浮点数的精确差值，因此舍入误差最大为 e。对于一般情形，令 $y = 0.0 \ldots 0y_{k+1} \ldots y_{k+p}$ 且 $\bar{y}$ 是截断到 $p + 1$ 位的 y。那么

$$y - \bar{y} < (\beta - 1)(\beta^{-p-1} + \beta^{-p-2} + \ldots + \beta^{-p-k})。 \tag{15}$$

按照保护数位的定义，$x - y$ 的计算值是舍入为浮点数的 $x - \bar{y}$，即 $(x - \bar{y}) + \delta$，其中舍入误差 δ 满足以下条件

$$|\delta| \leq (\beta/2)\beta^{-p}。 \tag{16}$$

精确差值是 $x - y$，因此误差是 $(x - y) - (x - \bar{y} + \delta) = \bar{y} - y + \delta$。存在三种情况。如果 $x - y \geq 1$，则相对误差的界限是

$$\frac{y - \bar{y} + \delta}{1} \leq \beta^{-p}\left[(\beta - 1)(\beta^{-1} + \ldots + \beta^{-k}) + \beta/2\right] < \beta^{-p}(1 + \beta/2)。 \tag{17}$$

第二，如果 $x - \bar{y} < 1$，则 $\delta = 0$。因为 $x - y$ 最小可以是

$1.0 - 0.\underbrace{0\ldots0}_{k}\underbrace{\rho\ldots\rho}_{p} > (\beta - 1)(\beta^{-1} + \ldots + \beta^{-k})$，其中 $\rho = \beta - 1$，

在这种情况下，相对误差的界限是

$$\frac{y-\bar{y}+\delta}{(\beta-1)(\beta^{-1}+\ldots+\beta^{-k})} < \frac{(\beta-1)\beta^{-p}(\beta^{-1}+\ldots+\beta^{-k})}{(\beta-1)(\beta^{-1}+\ldots+\beta^{-k})} = \beta^{-p}。 \tag{18}$$

最后一种情况是：$x - y < 1$ 但 $x - \bar{y} \geq 1$。发生这种情况的唯一条件是 $x - \bar{y} = 1$，在这种情况下 $\delta = 0$。但是，如果 $\delta = 0$，则 (18) 是适用的，因此相对误差的界限同样是 $\beta^{-p} < \beta^{-p}(1 + \beta/2)$。■

当 $\beta = 2$ 时，界限正好是 $2e$，此界限是在 $p \to \infty$ 时在极限中为 $x = 1 + 2^{2-p}$ 和 $y = 2^{1-p} - 2^{1-2p}$ 实现的。将符号相同的数相加时，不使用保护数位也能得到良好的精确度，如以下结果所示。

D.6.1.3 定理 10

如果 $x \geq 0$ 且 $y \geq 0$，则计算 $x + y$ 时的相对误差最大为 2ε，即使未使用保护数位也是如此。

D.6.1.4 证明

使用 k 个保护数位的加法运算的算法与用于减法运算的算法类似。如果 $x \geq y$，则对 y 进行右移位，直到 x 和 y 的小数点对齐。舍弃移过 $p + k$ 位置的任何位。精确计算这两个 $p + k$ 位数的和。然后舍入为 p 位。

我们将检验不使用保护数位时该定理是否成立；一般的情况是类似的。不失一般性，假设 $x \geq y \geq 0$ 并将 x 乘以某一因子使其具有 $d.dd\ldots d \times \beta^0$ 的形式。首先，假定没有进位。那么，从 y 的末尾移掉的位所具有的值小于 β^{-p+1}，且和至少为 1，因此相对误差小于 $\beta^{-p+1}/1 = 2e$。如果有进位，则必须将移位误差增加到

$$\frac{1}{2}\beta^{-p+2}。$$

和至少为 β，因此相对误差小于

$$\left(\beta^{-p+1} + \frac{1}{2}\beta^{-p+2}\right)/\beta = (1 + \beta/2)\beta^{-p} \leq 2\varepsilon \quad 。■$$

显而易见，结合这两个定理可以得出定理 2。定理 2 给出了执行一个运算的相对误差。比较 $x^2 - y^2$ 和 $(x + y)(x - y)$ 的舍入误差要求知道乘法运算的相对误差。$x \ominus y$ 的相对误差是 $\delta_1 = [(x \ominus y) - (x - y)] / (x - y)$，它满足 $|\delta_1| \leq 2e$。或者，以另一种方式书写

$$x \ominus y = (x - y)(1 + \delta_1), \quad |\delta_1| \leq 2e \tag{19}$$

类似地

$$x \oplus y = (x + y)\,(1 + \delta_2), \quad |\delta_2| \le 2e \tag{20}$$

假定乘法是通过计算精确乘积再进行舍入来执行的，相对误差最大为 .5 ulp，那么，对于任何浮点数 u 和 v

$$u \otimes v = uv\,(1 + \delta_3), \qquad |\delta_3| \le e \tag{21}$$

将这三个等式放在一起 （假设 $u = x \ominus y$ 且 $v = x \oplus y$），会得出

$$(x \ominus y) \otimes (x \oplus y) = (x - y)\,(1 + \delta_1)\,(x + y)\,(1 + \delta_2)\,(1 + \delta_3) \tag{22}$$

因此，在计算 $(x - y)\,(x + y)$ 时引起的相对误差是

$$\frac{(x-y) \ominus (x+y) - (x^2 - y^2)}{(x^2 - y^2)} = (1 + \delta_1)(1 + \delta_2)(1 + \delta_3) - 1 \tag{23}$$

此相对误差等于 $\delta_1 + \delta_2 + \delta_3 + \delta_1\delta_2 + \delta_1\delta_3 + \delta_2\delta_3 + \delta_1\delta_2\delta_3$，其边界是 $5\varepsilon + 8\varepsilon^2$。换句话说，最大相对误差大约是舍入误差的 5 倍 （因为 e 是一个很小的数， e^2 几乎可以忽略）。

对 $(x \otimes x) \ominus (y \otimes y)$ 的类似分析无法得出很小的相对误差值，因为将 x 和 y 的两个接近值代入 $x^2 - y^2$ 时，相对误差通常是相当大的。查看它的另一方法是尝试并重复对 $(x \ominus y) \otimes (x \oplus y)$ 进行的分析，将会得出

$$\begin{aligned}(x \otimes x) \ominus (y \otimes y) &= [x^2(1 + \delta_1) - y^2(1 + \delta_2)]\,(1 + \delta_3) \\ &= ((x^2 - y^2)\,(1 + \delta_1) + (\delta_1 - \delta_2)y^2)\,(1 + \delta_3)\end{aligned}$$

当 x 和 y 接近时，误差项 $(\delta_1 - \delta_2)y^2$ 可以与结果 $x^2 - y^2$ 一样大。这些计算证明我们的以下断言是正确的：$(x - y)\,(x + y)$ 比 $x^2 - y^2$ 更精确。

接下来，我们分析三角形面积的公式。为了估算在使用 (7) 计算时出现的最大误差，将需要以下定理。

D.6.1.5 定理 11

如果减法运算使用了保护数位且操作数满足 $y/2 \le x \le 2y$，则 $x - y$ 可以精确算出。

D.6.1.6 证明

请注意，如果 x 和 y 的指数相同，则 $x \ominus y$ 一定是精确的。否则，根据定理中的条件，指数最多可以相差 1。如有必要，x 和 y 可以按比例缩放并互换，从而使 $0 \le y \le x$，并将 x 表示为 $x_0.x_1 \ldots x_{p-1}$，将 y 表示为 $0.y_1 \ldots y_p$。那么，用于计算 $x \ominus y$ 的算法将精确计

算 $x - y$ 并舍入到一个浮点数。如果差的形式为 $0.d_1 \ldots d_p$，则差值的长度将已经是 p 位，因此不必进行舍入。由于 $x \le 2y$, $x - y \le y$，且 y 的形式是 $0.d_1 \ldots d_p$，因此 $x - y$ 也满足这样的形式。▮

当 $\beta > 2$ 时，定理 11 的假设不能由 $y/\beta \le x \le \beta y$ 替换；更严格的条件 $y/2 \le x \le 2y$ 仍然是必需的。在定理 10 被证明之后，随即对 $(x - y)(x + y)$ 中的误差进行了分析，其中利用了加法和减法两种基本运算的相对误差很小（即等式 (19) 和 (20)）这一事实。这是一种最常见的误差分析。但是，就如同以下证明过程所显示的那样，分析公式 (7) 还需要其他内容，即定理 11。

D.6.1.7 定理 12

如果减法运算使用保护数位，而且 a、b 和 c 是三角形的边 $(a \ge b \ge c)$，则计算 $(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))$ 时的相对误差最大为 16ε，条件是 $e < .005$。

D.6.1.8 证明

让我们逐一检查各因子。依据定理 10，$b \oplus c = (b + c)(1 + \delta_1)$，其中 δ_1 是相对误差，$|\delta_1| \le 2\varepsilon$。所以，第一个因子的值是

$$(a \oplus (b \oplus c)) = (a + (b \oplus c))(1 + \delta_2) = (a + (b + c)(1 + \delta_1))(1 + \delta_2),$$

因此

$$\begin{aligned}(a + b + c)(1 - 2\varepsilon)^2 &\le [a + (b + c)(1 - 2\varepsilon)] \cdot (1 - 2\varepsilon) \\ &\le a \oplus (b \oplus c) \\ &\le [a + (b + c)(1 + 2\varepsilon)](1 + 2\varepsilon) \\ &\le (a + b + c)(1 + 2\varepsilon)^2\end{aligned}$$

这意味着存在一个 η_1 满足

$$(a \oplus (b \oplus c)) = (a + b + c)(1 + \eta_1)^2, \ |\eta_1| \le 2\varepsilon。 \tag{24}$$

下一项涉及 c 和 $a \ominus b$ 的有可能是恶性的相减，因为 $a \ominus b$ 可能具有舍入误差。因为 a、b 和 c 是三角形的边，所以 $a \le b + c$，将此与排序 $c \le b \le a$ 结合在一起就可以得出 $a \le b + c \le 2b \le 2a$。因此，$a - b$ 满足定理 11 的条件。这意味着 $a - b = a \ominus b$ 是精确的，因此 $c \ominus (a - b)$ 是一个无害相减，可以依据定理 9 进行估算，估算结果为

$$(c \ominus (a \ominus b)) = (c - (a - b))(1 + \eta_2), \ |\eta_2| \le 2\varepsilon \tag{25}$$

第三项是两个精确正量的和，因此有

$$(c \oplus (a \ominus b)) = (c + (a - b))\,(1 + \eta_3),\ |\eta_3| \le 2\varepsilon \tag{26}$$

最后一项是

$$(a \oplus (b \ominus c)) = (a + (b - c))\,(1 + \eta_4)^2,\ |\eta_4| \le 2\varepsilon, \tag{27}$$

其中同时使用了定理 9 和定理 10。如果假定乘法运算精确舍入，使得 $x \otimes y = xy(1 + \zeta)$ 且 $|\zeta| \le \varepsilon$，那么将 (24)、(25)、(26) 和 (27) 结合在一起就可以得出

$$\begin{aligned}&(a \oplus (b \oplus c))\,(c \ominus (a \ominus b))\,(c \oplus (a \ominus b))\,(a \oplus (b \ominus c))\\ &\quad \le (a + (b + c))\,(c - (a - b))\,(c + (a - b))\,(a + (b - c))\,E\end{aligned}$$

其中

$$E = (1 + \eta_1)^2\,(1 + \eta_2)\,(1 + \eta_3)\,(1 + \eta_4)^2\,(1 + \zeta_1)(1 + \zeta_2)\,(1 + \zeta_3)$$

E 的一个上界是 $(1 + 2\varepsilon)^6(1 + \varepsilon)^3$，其展开结果是 $1 + 15\varepsilon + O(\varepsilon^2)$。有些作者干脆忽略了 $O(e^2)$ 项，但其实说明这一项是很容易的。令 $(1 + 2\varepsilon)^6(1 + \varepsilon)^3 = 1 + 15\varepsilon + \varepsilon R(\varepsilon)$，其中 $R(\varepsilon)$ 是一个具有正系数的、e 的多项式，由此可知它是 ε 的递增函数。因为 $R(.005) = .505$，所以对于所有的 $\varepsilon < .005$，$R(\varepsilon) < 1$，由此可得 $E \le (1 + 2\varepsilon)^6(1 + \varepsilon)^3 < 1 + 16\varepsilon$。求 E 的一个下界时，我们注意 $1 - 15\varepsilon - \varepsilon R(\varepsilon) < E$，因此当 $\varepsilon < .005$ 时，$1 - 16\varepsilon < (1 - 2\varepsilon)^6(1 - \varepsilon)^3$。将这两个界限结合在一起可得 $1 - 16\varepsilon < E < 1 + 16\varepsilon$。因此，相对误差最大为 16ε。∎

定理 12 明白地显示在公式 (7) 中不存在恶性抵消。因此，虽然不必显示公式 (7) 在数值上是稳定的，但是具有整个公式的界限是令人满意的，这正是第 6 页的 “抵消” 中的定理 3 给出的。

D.6.1.9 定理 3 的证明

假设

$$q = (a + (b + c))\,(c - (a - b))\,(c + (a - b))\,(a + (b - c))$$

且

$$Q = (a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))。$$

然后，由定理 12 得 $Q = q(1 + \delta)$，且 $\delta \le 16\varepsilon$。可以很容易地得到

$$1 - 0.52|\delta| \le \sqrt{1 - |\delta|} \le \sqrt{1 + |\delta|} \le 1 + 0.52|\delta| \tag{28}$$

条件是 $\delta \le .04/(.52)^2 \approx .15$，因为 $|\delta| \le 16\varepsilon \le 16(.005) = .08$，所以 δ 确实满足此条件。因此

$$\sqrt{Q} = \sqrt{q(1+\delta)} = \sqrt{q}(1+\delta_1),$$

且 $|\delta_1| \le .52|\delta| \le 8.5\varepsilon$。如果将计算平方根时的误差控制在 .5 ulp 内，则计算 $\sqrt{Q}$ 时的误差是 $(1+\delta_1)(1+\delta_2)$，其中 $|\delta_2| \le \varepsilon$。如果 $\beta = 2$，则在除以 4 时不会产生进一步的误差。否则，在做除法时还需要另一因子 $1+\delta_3$（$|\delta_3| \le \varepsilon$），通过使用证明定理 12 时的方法，$(1+\delta_1)(1+\delta_2)(1+\delta_3)$ 的最终误差界限取决于 $1+\delta_4$（$|\delta_4| \le 11\varepsilon$）。∎

要使紧跟定理 4 陈述之后的启发式说明变得精确，下一个定理正好说明 $\mu(x)$ 近似一个常数时的接近程度。

D.6.1.10 定理 13

如果 $\mu(x) = \ln(1+x)/x$，*则对于* $0 \le x \le \frac{3}{4}$，$\frac{1}{2} \le \mu(x) \le 1$ *且导数满足* $|\mu'(x)| \le \frac{1}{2}$。

D.6.1.11 证明

请注意，$\mu(x) = 1 - x/2 + x^2/3 - \ldots$ 是具有递减项的交错级数，因此对于 $x \le 1$，$\mu(x) \ge 1 - x/2 \ge 1/2$。可以更容易地看出：因为 μ 的级数是交错的，所以 $\mu(x) \le 1$。$\mu'(x)$ 的泰勒级数也是交错的，而且如果 $x \le \frac{3}{4}$ 具有递减项，那么 $-\frac{1}{2} \le \mu'(x) \le -\frac{1}{2} + 2x/3$，或者 $-\frac{1}{2} \le \mu'(x) \le 0$，因此 $|\mu'(x)| \le \frac{1}{2}$。∎

D.6.1.12 定理 4 的证明

因为 ln 的泰勒级数

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots$$

是一个交错级数，所以 $0 < x - \ln(1+x) < x^2/2$，按 x 近似 $\ln(1+x)$ 时产生的相对误差的界限是 $x/2$。如果 $1 \oplus x = 1$，则 $|x| < \varepsilon$，因此相对误差的界限是 $\varepsilon/2$。

当 $1 \oplus x \neq 1$ 时，定义 $\hat{x}$ 满足 $1 \oplus x = 1 + \hat{x}$ 。又因为 $0 \leq x < 1$，所以 $(1 \oplus x) \ominus 1 = \hat{x}$ 。如果要将计算除法和对数时的误差控制在 $\frac{1}{2}$ ulp 内，则表达式 $\ln(1 + x)/((1 + x) - 1)$ 的计算值是

$$\frac{\ln(1 \oplus x)}{(1 \oplus x) \ominus 1}(1 + \delta_1)(1 + \delta_2) = \frac{\ln(1 + \hat{x})}{\hat{x}}(1 + \delta_1)(1 + \delta_2) = \mu(\hat{x})(1 + \delta_1)(1 + \delta_2) \tag{29}$$

其中 $|\delta_1| \leq \varepsilon$ 且 $|\delta_2| \leq \varepsilon$。要估算 $\mu(\hat{x})$，请使用中值定理，该定理指出

$$\mu(\hat{x}) - \mu(x) = (\hat{x} - x)\mu'(\xi) \tag{30}$$

其中 ξ 介于 x 和 $\hat{x}$ 之间。根据 $\hat{x}$ 的定义，可以得出 $|\hat{x} - x| \leq \varepsilon$，将此与定理 13 结合在一起可以得出 $|\mu(\hat{x}) - \mu(x)| \leq \varepsilon/2$，或 $|\mu(\hat{x})/\mu(x) - 1| \leq \varepsilon/(2|\mu(x)|) \leq \varepsilon$，这意味着 $\mu(\hat{x}) = \mu(x)(1 + \delta_3)$，$|\delta_3| \leq \varepsilon$。最后，乘以 x 会引入 δ_4，因此

$x \cdot \ln(1 \oplus x)/((1 \oplus x) \ominus 1)$ 的计算值

是

$$\frac{\ln(1 + x)}{1 + x) - 1}(1 + \delta_1)(1 + \delta_2)(1 + \delta_3)(1 + \delta_4), \qquad \left|\delta_I\right| \leq$$

很容易得出：如果 $\varepsilon < 0.1$，则

$$(1 + \delta_1)(1 + \delta_2)(1 + \delta_3)(1 + \delta_4) = 1 + \delta,$$

其中 $|\delta| \leq 5\varepsilon$。∎

使用公式 (19)、(20) 和 (21) 进行误差分析的一个有趣示例出现在二次方程求根公式 $(-b \pm \sqrt{b^2 - 4ac})/2a$ 中。第 6 页的“抵消”节解释改写公式将如何消除 ± 运算引起的潜在抵消。但是，在计算 $d = b_2 - 4ac$ 时还可能出现另一个抵消。通过对公式进行简单的重新排列不能消除该潜在抵消。大致说来，当 $b^2 \approx 4ac$ 时，舍入误差最多可以影响使用二次方程求根公式计算的根中数位的一半。下面是一个非正式证明（估算二次方程求根公式误差的另一种方法见于 Kahan [1972]）。

如果 $b^2 \approx 4ac$，则舍入误差最多可以影响使用二次方程求根公式 $(-b \pm \sqrt{b^2 - 4ac})/2a$。

证明：记 $(b \otimes b) \ominus (4a \otimes c) = (b^2(1 + \delta_1) - 4ac(1 + \delta_2))(1 + \delta_3)$，其中 $|\delta_i| \leq \varepsilon$。[30] 令 $d = b^2 - 4ac$，上式可改写为 $(d(1 + \delta_1) - 4ac(\delta_2 - \delta_1))(1 + \delta_3)$。为获得此误差的估计大小，忽略 δ_i 中的二次项，此时绝对误差为 $d(\delta_1 + \delta_3) - 4ac\delta_4$，其中 $|\delta_4| = |\delta_1 - \delta_2| \leq 2\varepsilon$。因为

30.在此非正式证明中，假定 $\beta = 2$，使得乘以 4 的结果是精确的，且不需要 δ_i。

$d \ll 4ac$，所以可以忽略第一项 $d(\delta_1 + \delta_3)$。要估算第二项，可令 $ax^2 + bx + c = a(x - r_1)(x - r_2)$，因此 $ar_1r_2 = c$。因为 $b^2 \approx 4ac$，所以 $r_1 \approx r_2$，可得第二误差项为 $4ac\delta_4 \approx 4a^2r_1\delta_4^2$。这样，$\sqrt{d}$ 的计算值是

$$\sqrt{d + 4a^2r_1^2\delta_4}$$

不等式

$$-q \leq \sqrt{p^2 - q^2} \leq \sqrt{p^2 + q^2} \leq p + q, \quad p \geq q > 0$$

显示

$$\sqrt{d + 4a^2r_1^2\delta_4} = \sqrt{d} + E$$

其中

$$|E| \leq \sqrt{4a^2r_1^2|\delta_4|}$$

因此 $\sqrt{d}/2a$ 中的绝对误差大约为 $r_1\sqrt{\delta_4}$。因为 $\delta_4 \approx \beta^{-p}$，所以 $\sqrt{\delta_4} \approx \beta^{-p/2}$，这样 $r_1\sqrt{\delta_4}$ 的绝对误差就破坏了根 $r_1 \approx r_2$ 的下半部分数位。换句话说，由于根的计算涉及计算 $(\sqrt{d})/(2a)$，并且此表达式在对应于 r_i 的一半低位的位置中没有有效位，所以 r_i 的低位不是有效的。▮

最后，我们转到定理 6 的证明。它基于以下事实（将在第 51 页的“定理 14 和定理 8”节中证明）。

D.6.1.13 定理 14

假定 $0 < k < p$，设 $m = \beta^k + 1$，并假定浮点运算是精确舍入的。那么，$(m \otimes x) \ominus (m \otimes x \ominus x)$ 与舍入到 $p - k$ 有效位的 x 完全相等。更精确的说法是，舍入 x 的方法是取 x 的有效位，可以设想为小数点正好在 k 个最低有效位的左侧并舍入为一个整数。

D.6.1.14 定理 6 的证明

依据定理 14，x_h 是舍入到 $p - k = \lfloor p/2 \rfloor$ 位的 x。如果没有进位，则 x_h 当然可以用 $\lfloor p/2 \rfloor$ 个有效位表示。假定存在进位。如果 $x = x_0.x_1 \ldots x_{p-1} \times \beta^e$，则舍入将 x_{p-k-1} 加 1。可能有进位的唯一情况是 $x_{p-k-1} = \beta - 1$，但此时 x_h 的低位数字是 $1 + x_{p-k-1} = 0$，因此 x_h 同样可用 $\lfloor p/2 \rfloor$。

要处理 x_1，请按比例增减 x 使其成为满足 $\beta^{p-1} \le x \le \beta^p - 1$ 的整数。假设 $x = \bar{x}_h + \bar{x}_l$，其中 $\bar{x}_h$ 是 x 的 $p - k$ 个高位数字，$\bar{x}_l$ 是 k 个低位数字。有三种情况需要考虑。如果 $\bar{x}_l < (\beta/2)\beta^{k-1}$，则将 x 舍入到 $p - k$ 位的结果与截断相同，而且 $x_h = \bar{x}_h$ 和 $x_l = \bar{x}_l$。因为 $\bar{x}_l$ 最多有 k 位，如果 β 是偶数，则 $\bar{x}_l$ 的位数最多为 $k = \lceil p/2 \rceil = \lfloor p/2 \rfloor$。否则，$\beta = 2$ 且 $\bar{x}_1 < 2^{k-1}$ 可以使用 $k - 1 \le \lfloor p/2 \rfloor$ 个有效位表示。第二种情况是在 $\bar{x} > (\beta/2)\beta^{k-1}$ 时，计算 x_h 涉及上舍入，因此 $x_h = \bar{x}_h + \beta^k$，且 $x_l = x - x_h = x - \bar{x}_h - \beta^k = \bar{x}_l - \beta^k$。同样，$\bar{x}_l$ 最多有 k 位，因此可以使用 $\lfloor p/2 \rfloor$ 位表示。最后，如果 $\bar{x}_l = (\beta/2)\beta^{k-1}$，则 $x_h = \bar{x}_h$ 或 $\bar{x}_h + \beta^k$，这取决于是否存在上舍入。因此 x_l 是 $(\beta/2)\beta^{k-1}$ 或 $(\beta/2)\beta^{k-1} - \beta^k = -\beta^k/2$，二者都是用 1 位表示的。▮

定理 6 提供了一种将两个工作精度数的乘积精确表示为一个和的方式。有一个用于精确表示和的相应公式。如果 $|x| \ge |y|$，则 $x + y = (x \oplus y) + (x \ominus (x \oplus y)) \oplus y$ [Dekker 1971；Knuth 1981，4.2.2 节中的定理 C]。但是，在使用精确舍入运算时，此公式仅对 $\beta = 2$ 是正确的，而对于 $\beta = 10$ 是不正确的，如示例 $x = .99998$、$y = .99997$ 所示。

D.6.2 二进制到十进制的转换

由于单精度有 $p = 24$，且 $2^{24} < 10^8$，所以您可能会认为将二进制数转换为 8 个十进制位将足以恢复原始二进制数。但是，这是不正确的。

D.6.2.1 定理 15

当将二进制 IEEE 单精度数转换为最接近的八位十进制数时，从十进制数唯一地恢复二进制数并不总是可能的。但是，如果使用九个十进制位，则将十进制数转换为最接近的二进制数将恢复原始浮点数。

D.6.2.2 证明

半开区间 $[10^3, 2^{10}) = [1000, 1024)$ 中的二进制单精度数有 10 位在二进制小数点的左侧，有 14 位在二进制小数点的右侧。因此，在该区间中有 $(2^{10} - 10^3)2^{14} = 393{,}216$ 个不同的二进制数。如果十进制数是使用 8 位表示的，则在同一区间中有 $(2^{10} - 10^3)10^4 = 240{,}000$ 个十进制数。240,000 个十进制数无法表示 393,216 个不同的二进制数。因此，8 个十进制位不足以唯一表示每个单精度二进制数。

要说明 9 位是足够的，只需说明二进制数之间的间隔始终大于十进制数之间的间隔就足够了。这将确保对于每个十进制数 N，区间

$$[N - \tfrac{1}{2}\text{ulp},\ N + \tfrac{1}{2}\text{ulp}]$$

最多包含一个二进制数。这样，每个二进制数都舍入为唯一的十进制数，而十进制数又舍入为唯一的二进制数。

以区间 $[10^n, 10^{n+1}]$ 为例来说明二进制数之间的间隔始终大于十进制数之间的间隔。在此区间上，连续二进制数之间的间隔是 $10^{(n+1)-9}$。在 $[10^n, 2^m]$ 上（其中 m 是满足 $10^n < 2^m$ 的最小整数），二进制数的间隔是 2^{m-24}，并且在该区间中此间隔变得越来越大。因此，只需证明 $10^{(n+1)-9} < 2^{m-24}$。而事实上，由于 $10^n < 2^m$，所以 $10^{(n+1)-9} = 10^n 10^{-8} < 2^m 10^{-8} < 2^m 2^{-24}$。 ▮

应用于双精度的同一参数显示恢复双精度数需要 17 个十进制位。

二进制 - 十进制转换还提供了使用标志的另一示例。回想一下第 15 页的“精度”节中的内容：要从十进制扩展恢复二进制数，必须精确计算十进制到二进制的转换。该转换的方法是按单扩展精度将量 N 和 $10^{|P|}$（如果 $p < 13$，则二者都是精确的）相乘，然后舍入到单精度（如果 $p < 0$，则将它们相除；这两种情况是类似的）。当然，$N \cdot 10^{|P|}$ 的计算结果不可能是精确的；必须保持精确的是组合运算 round($N \cdot 10^{|P|}$)，其中舍入是从单扩展精度到单精度。要了解它为什么可能不是精确的，请举一个简单的例子：$\beta = 10$，$p = 2$（对于单精度）或 $p = 3$（对于单精度扩展）。如果乘积是 12.51，则这将作为单精度扩展乘法运算的一部分舍入为 12.5。舍入到单精度的结果为 12。但是该结果是不正确的，因为将乘积舍入到单精度的结果应该是 13。误差是双舍入导致的。

通过使用 IEEE 标志，可以避免双舍入，如下所示。保存不精确标志的当前值，然后将其重置。将舍入模式设置为“舍入为零”。然后，执行乘法运算 $N \cdot 10^{|P|}$。将不精确标志的新值存储在 `ixflag` 中，并恢复舍入模式和不精确标志。如果 `ixflag` 是 0，则 $N \cdot 10^{|P|}$ 是精确的，因此 round($N \cdot 10^{|P|}$) 直到最后一位都是正确的。如果 `ixflag` 是 1，则某些数字被截去了，因为舍入为零时总是截去数字。乘积的有效位类似于 $1.b_1 \ldots b_{22} b_{23} \ldots b_{31}$。如果 $b_{23} \ldots b_{31} = 10 \ldots 0$，则可能出现双舍入误差。解决这两种情况的简单方法是执行 `ixflag` 和 b_{31} 的逻辑 `OR` 运算。这样，在所有情况下 round($N \cdot 10^{|P|}$) 都能正确计算。

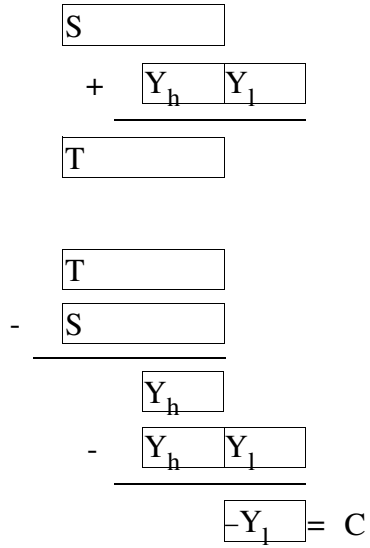
D.6.3 求和中的误差

第 34 页的“优化器”节提到了精确计算非常长的和的问题。改进精度的最简单方法是使精度加倍。为大致估计出精度加倍时和的精度的改进程度，假设 $s_1 = x_1$, $s_2 = s_1 \oplus x_2 \ldots$, $s_i = s_{i-1} \oplus x_i$。由此可得 $s_i = (1 + \delta_i)(s_{i-1} + x_i)$，其中 $|\delta_i| \le \varepsilon$。忽略 δ_i 中的二次项将得出

$$s_n = \sum_{j=1}^{n} x_j \left(1 + \sum_{k=j}^{n} \delta_k\right) = \sum_{j=1}^{n} x_j + \sum_{j=1}^{n} x_j \left(\sum_{k=j}^{n} \delta_k\right) \tag{31}$$

(31) 的第一个等式显示 Σx_j 的计算值与对 x_i 的扰动值执行精确求和时的结果是相同的。第一项 x_1 的扰动范围是 $n\varepsilon$，最后一项 x_n 的扰动范围是 ε。(31) 中的第二个等式显示误差项的界限是 $n\varepsilon\Sigma|x_j|$。使精度加倍具有对 ε 取平方的效果。如果求和是以 IEEE 双精度格式执行的，那么 $1/\varepsilon \approx 10^{16}$，因此对于 n 的任何合理值，$n\varepsilon \ll 1$。这样，使精度加倍采用了最大扰动 $n\varepsilon$，并将其更改为 $n\varepsilon^2 \ll \varepsilon$。因此，Kahan 求和公式（定理 8）的 2ε 误差界限不如使用双精度好，但比使用单精度要好得多。

有关 Kahan 求和公式为何成立的直观解释，请参考下面的过程图。



每次增加被加数时，都有一个校正因子 *C*，该因子将在下一循环中得以应用。因此，首先从 X_j 中减去在上一循环中计算的校正因子 *C*，得出校正后的被加数 *Y*。然后，将此被加数增加到连续和 *S* 中。在和中， *Y* 的低位 （即 Y_l）已丢失。接下来，通过计算 *T - S* 来计算 *Y* 的高位。当从中减去 *Y* 时，*Y* 的低位将恢复。这些是图中第一次求和时丢失的位。它们将成为下一循环的校正因子。定理 8 的正式证明包括在第 51 页的 “定理 14 和定理 8”节中 （摘自 Knuth [1981] 第 572 页）。

D.7 小结

计算机系统设计人员往往忽略系统中与浮点有关的部分。这很可能是由于计算机科学课程中很少注意浮点问题。而这又导致一种流行的看法：浮点不是一个可量化的主题，因此详细讨论处理浮点的硬件和软件没有多大意义。

本文证明了进行有关浮点的严谨推理是可能的。例如，可以证明如果基础硬件具有保护数位，则涉及抵消的浮点算法具有较小的相对误差；再如，存在一种用于二进制 - 十进制转换的高效算法，可以证明在支持扩展精度的条件下它是可逆的。当作为基础的计算机系统支持浮点时，执行构造可靠浮点软件的任务将容易得多。除了刚才提到的两个示例 （保护数位和扩展精度）外，本文的第 28 页的 “系统方面”节还包含许多说明如何更好地支持浮点的示例，内容涉及从指令集设计到编译器优化等各方面。

随着 IEEE 浮点标准得到越来越广泛的应用，利用该标准的各种特性的代码也将变得更具可移植性。第 14 页的 “IEEE 标准”节提供了很多示例，说明在编写实际的浮点代码时如何使用 IEEE 标准的各种特性。

D.8 致谢

本文受到 W. Kahan 于 1988 年 5 月至 7 月在 Sun Microsystems 讲授的课程的启发，该课程由 Sun 的 David Hough 精心组织。我希望大家可以籍此了解浮点与计算机系统之间的相互作用，而不必起床去听上午 8 点的课。在这里，我要感谢 Kahan 和 Xerox PARC 的许多同事（尤其是 John Gilbert），他们阅读了本文的草稿并提出了许多有用的意见。另外，此讲稿的改进还要归功于 Paul Hilfinger 和一位不知名人士的审阅。

D.9 参考书目

Aho, Alfred V., Sethi, R., and Ullman J. D. 1986. 《Compilers:Principles, Techniques and Tools》, Addison-Wesley, Reading, MA.

ANSI 1978. 《American National Standard Programming Language FORTRAN》, ANSI Standard X3.9-1978, American National Standards Institute, New York, NY.

Barnett, David 1987. 《A Portable Floating-Point Environment》, unpublished manuscript.

Brown, W. S. 1981. 《A Simple but Realistic Model of Floating-Point Computation》, ACM Trans. on Math.Software 7(4), pp. 445-480.

Cody, W. J et. al. 1984. 《A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic》, IEEE Micro 4(4), pp. 86-100.

Cody, W. J. 1988. 《Floating-Point Standards - Theory and Practice》, in "Reliability in Computing:the role of interval methods in scientific computing", ed. by Ramon E. Moore, pp.99-107, Academic Press, Boston, MA.

Coonen, Jerome 1984. 《Contributions to a Proposed Standard for Binary Floating-Point Arithmetic》, PhD Thesis, Univ. of California, Berkeley.

Dekker, T. J. 1971.《A Floating-Point Technique for Extending the Available Precision》, Numer. Math. 18(3), pp. 224-242.

Demmel, James 1984. 《Underflow and the Reliability of Numerical Software》, SIAM J. Sci. Stat. Comput. 5(4), pp. 887-919.

Farnum, Charles 1988. 《Compiler Support for Floating-point Computation》, Software-Practice and Experience, 18(7), pp. 701-709.

Forsythe, G. E. and Moler, C. B. 1967. 《Computer Solution of Linear Algebraic Systems》, Prentice-Hall, Englewood Cliffs, NJ.

Goldberg, I. Bennett 1967.《27 Bits Are Not Enough for 8-Digit Accuracy》, Comm. of the ACM.10(2), pp 105-106.

Goldberg, David 1990. 《Computer Arithmetic》, in "Computer Architecture:A Quantitative Approach", by David Patterson and John L. Hennessy, Appendix A, Morgan Kaufmann, Los Altos, CA.

Golub, Gene H. and Van Loan, Charles F. 1989.《Matrix Computations》, 2nd edition, The Johns Hopkins University Press, Baltimore Maryland.

Graham, Ronald L., Knuth, Donald E. and Patashnik, Oren.1989. 《*Concrete Mathematics*》, Addison-Wesley, Reading, MA, p.162.

Hewlett Packard 1982. HP-15C 《Advanced Functions Handbook》.

IEEE 1987. 《IEEE Standard 754-1985 for Binary Floating-point Arithmetic》, IEEE, (1985).Reprinted in SIGPLAN 22(2) pp. 9-25.

Kahan, W. 1972. 《A Survey Of Error Analysis》, in Information Processing 71, Vol 2, pp.1214 - 1239 (Ljubljana, Yugoslavia), North Holland, Amsterdam.

Kahan, W. 1986.《Calculating Area and Angle of a Needle-like Triangle》, unpublished manuscript.

Kahan, W. 1987. 《Branch Cuts for Complex Elementary Functions》, in "The State of the Art in Numerical Analysis", ed. by M.J.D. Powell and A. Iserles (Univ of Birmingham, England), Chapter 7, Oxford University Press, New York.

Kahan, W. 1988. Unpublished lectures given at Sun Microsystems, Mountain View, CA.

Kahan, W. and Coonen, Jerome T. 1982. 《The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments》, in "The Relationship Between Numerical Computation And Programming Languages", ed. by J. K. Reid, pp.103-115, North-Holland, Amsterdam.

Kahan, W. and LeBlanc, E. 1985. 《Anomalies in the IBM Acrith Package》, Proc.7th IEEE Symposium on Computer Arithmetic (Urbana, Illinois), pp. 322-331.

Kernighan, Brian W. and Ritchie, Dennis M. 1978. 《The C Programming Language》, Prentice-Hall, Englewood Cliffs, NJ.

Kirchner, R. and Kulisch, U. 1987. 《Arithmetic for Vector Processors》, Proc.8th IEEE Symposium on Computer Arithmetic (Como, Italy), pp. 256-269.

Knuth, Donald E., 1981. 《The Art of Computer Programming, Volume II》, Second Edition, Addison-Wesley, Reading, MA.

Kulisch, U. W., and Miranker, W. L. 1986. 《The Arithmetic of the Digital Computer:A New Approach》, SIAM Review 28(1), pp 1-36.

Matula, D. W. and Kornerup, P. 1985. 《Finite Precision Rational Arithmetic:Slash Number Systems》 , IEEE Trans. on Comput.C-34(1), pp 3-18.

Nelson, G. 1991. 《Systems Programming With Modula-3》, Prentice-Hall, Englewood Cliffs, NJ.

Reiser, John F. and Knuth, Donald E. 1975. 《Evading the Drift in Floating-point Addition》 , Information Processing Letters 3(3), pp 84-87.

Sterbenz, Pat H. 1974. 《Floating-Point Computation》 , Prentice-Hall, Englewood Cliffs, NJ.

Swartzlander, Earl E. and Alexopoulos, Aristides G. 1975. 《The Sign/Logarithm Number System》 , IEEE Trans. Comput. C-24(12), pp. 1238-1242.

Walther, J. S., 1971. 《A unified algorithm for elementary functions》 , Proceedings of the AFIP Spring Joint Computer Conf.38, pp. 379-385.

D.10 定理 14 和定理 8

此节包含正文中省略的两个技术性更强的证明。

D.10.1 定理 14

假定 $0 < k < p$，*设* $m = \beta^k + 1$，*并假定浮点运算是精确舍入的。那么，*$(m \otimes x) \ominus (m \otimes x \ominus x)$ *与舍入到* $p - k$ *个有效位的* x *完全相等。更精确的说法是，舍入* x *的方法是取* x *的有效位，可以设想为小数点正好在* k *个最低有效位的左侧并舍入为一个整数。*

D.10.2 证明

证明分为两种情况，这取决于 $mx = \beta^k x + x$ 的计算是否具有进位。

假定没有进位。按比例增减 x 使其成为一个整数是无害的。那么，$mx = x + \beta^k x$ 的计算类似于：

```
  aa...aabb...bb
  +aa...aabb...bb
  zz...zzbb...bb
```

其中 x 已经被分为两个部分。处于低位的 k 个数字被标记为 b，处于高位 $p - k$ 个数字被标记为 a。从 mx 计算 $m \otimes x$ 涉及舍弃处于低位的 k 个数字 （标记为 b 的数字），因此

$$m \otimes x = mx - x \bmod(\beta^k) + r\beta^k \tag{32}$$

如果 .bb...b 大于$\frac{1}{2}$，则 r 的值是 1，否则是 0。更准确地说，

如果 a.bb...b 舍入为 $a + 1$，则 $r = 1$，否则 $r = 0$。 (33)

接下来，计算 $m \otimes x - x = mx - x \bmod(\beta^k) + r\beta^k - x = \beta^k(x + r) - x \bmod(\beta^k)$。下图显示如何计算舍入的 $m \otimes x - x$，即 $(m \otimes x) \ominus x$。顶行是 $\beta^k(x + r)$，其中 B 是将 r 与最低位数字 b 相加而产生的数字。

```
aa...aabb...bB00...00
-bb...bb
zz...       zzZ00...00
```

如果 .bb...b < $\frac{1}{2}$，则 $r = 0$，减法运算导致从标记为 B 的数字借位，但是差值将进行上舍入，因此最终的结果是经过舍入的差等于顶行，即 $\beta^k x$。如果 .bb...b > $\frac{1}{2}$ 则 $r = 1$，由于借位而从 B 中减 1，因此结果是 $\beta^k x$。最后，以 .bb...b = $\frac{1}{2}$ 为例。如果 $r = 0$，则 B 是偶数，Z 是奇数，将差进行上舍入会得出 $\beta^k x$。类似地，当 $r = 1$ 时，B 是奇数，Z 是偶数，而差将进行下舍入，因此差同样是 $\beta^k x$。综上所述，

$$(m \otimes x) \ominus x = \beta^k x \tag{34}$$

结合等式 (32) 和 (34) 可以得出 $(m \otimes x) - (m \otimes x \ominus x) = x - x \bmod(\beta^k) + \rho \cdot \beta^k$。执行此计算的结果是

```
r00...00
         +  aa...aabb...bb
         -  .......bb...bb
            aa...aA00...00
```

计算 r 的规则 （即等式 (33)）与将 a... ab...b 舍入到 $p - k$ 位的规则是相同的。因此，在 $x + \beta^k x$ 没有进位的情况下，按浮点运算精度计算 $mx - (mx - x)$ 的结果与将 x 舍入到 $p - k$ 位完全相等。

当 $x + \beta^k x$ 确实有进位时， $mx = \beta^k x + x$ 类似于：

```
aa...aabb...bb
+aa...aabb...bb
zz...zZbb...bb
```

因此，$m \otimes x = mx - x \bmod(\beta^k) + w\beta^k$，其中 $w = -Z$（条件是 $Z < \beta/2$），但是 w 的精确值是不重要的。接下来， $m \otimes x - x = \beta^k x - x \bmod(\beta^k) + w\beta^k$。在图中

```
aa...aabb...bb00...00
-   bb...  bb
+   w
 zz     ...  zZbb  ...bb^31
```

经过舍入得出 $(m \otimes x) \ominus x = \beta^k x + w\beta^k - r\beta^k$，其中 $r = 1$（条件是 .bb...b > $\frac{1}{2}$ 或者 .bb...b = $\frac{1}{2}$ 且 $b_0 = 1$）。[32] 最后，

$$(m \otimes x) - (m \otimes x \ominus x) = mx - x \bmod (\beta^k) + w\beta^k - (\beta^k x + w\beta^k - r\beta^k)$$
$$= x - x \bmod (\beta^k) + r\beta^k。$$

同样，当 a...ab...b 舍入到 $p - k$ 位涉及上舍入时，$r = 1$ 完全成立。这样就在所有情况下，证明了定理 14。∎

D.10.2.1 定理 8（Kahan 求和公式）

假设 $\Sigma_{j=1}^{N} x_j$ 是使用以下算法计算的

```
S = X[1];
C = 0;
for j = 2 to N {
Y = X[j] - C;
   T = S + Y;
   C = (T - S) - Y;
   S = T;
}
```

那么，计算的和 S 等于 $S = \Sigma x_j (1 + \delta_j) + O(N\varepsilon^2) \Sigma |x_j|$，*其中* $|\delta_j| \le 2\varepsilon$。

D.10.2.2 证明

首先，回想一下简单公式 Σx_i 的误差是如何估算的。引入 $s_1 = x_1$，$s_i = (1 + \delta_i)(s_{i-1} + x_i)$。可知计算和为 s_n，它是各项的总和，其中的每一项都是 x_i 乘以包含 δ_j 的表达式。x_1 的精确系数是 $(1 + \delta_2)(1 + \delta_3)\ (1 + \delta_n)$，因此通过重新编号，$x_2$ 的系数一定是 $(1 + \delta_3)(1 + \delta_4) \ldots (1 + \delta n)$，依此类推。定理 8 的证明过程完全相同，只是 x_1 的系数更复杂。具体讲，就是 $s_0 = c_0 = 0$ 且

$$y_k = x_k \ominus c_{k-1} = (x_k - c_{k-1})(1 + \eta_k)$$
$$s_k = s_{k-1} \oplus y_k = (s_{k-1} + y_k)(1 + \sigma_k)$$
$$c_k = (s_k \ominus s_{k-1}) \ominus y_k = [(s_k - s_{k-1})(1 + \gamma_k) - y_k](1 + \delta_k)$$

其中的所有希腊字母的界限都是 ε。虽然 s_k 中 x_1 的系数是所需的最终表达式，但计算 $s_k - c_k$ 和 c_k 中 x_1 的系数更容易。

当 $k = 1$ 时，

31.如果加上 w 时不产生进位，那么这就是和。对于加上 w 产生进位的特殊情况，需要另外的参数。 – 编辑者

32.仅当 $(\beta^k x + w\beta^k)$ 保留 $\beta^k x$ 的形式时，经过舍入才得出 $\beta^k x + w\beta^k - r\beta^k$。 – 编辑者

$$c_1 = (s_1(1 + \gamma_1) - y_1)\ (1 + d_1)$$
$$= y_1((1 + s_1)\ (1 + \gamma_1) - 1)\ (1 + d_1)$$
$$= x_1(s_1 + \gamma_1 + s_1 g_1)\ (1 + d_1)\ (1 + h_1)$$
$$s_1 - c_1 = x_1[(1 + s_1) - (s_1 + g_1 + s_1 g_1)\ (1 + d_1)](1 + h_1)$$
$$= x_1[1 - g_1 - s_1 d_1 - s_1 g_1 - d_1 g_1 - s_1 g_1 d_1](1 + h_1)$$

分别调用这两个表达式 C_k 和 S_k 中 x_1 的系数，那么

$$C_1 = 2\varepsilon + O(\varepsilon^2)$$

$$S_1 = + \eta_1 - \gamma_1 + 4\varepsilon^2 + O(\varepsilon^3)$$

要得出 S_k 和 C_k 的通用公式，请展开 s_k 和 c_k 的定义，忽略所有涉及 x_i $(i > 1)$ 的项，得到

$$s_k = (s_{k-1} + y_k)(1 + \sigma_k)$$
$$= [s_{k-1} + (x_k - c_{k-1})\ (1 + \eta_k)](1 + \sigma_k)$$
$$= [(s_{k-1} - c_{k-1}) - \eta_k c_{k-1}](1+\sigma_k)$$
$$c_k = [\{s_k - s_{k-1}\}(1 + \gamma_k) - y_k](1 + \delta_k)$$
$$= [\{((s_{k-1} - c_{k-1}) - \eta_k c_{k-1})(1 + \sigma_k) - s_{k-1}\}(1 + \gamma_k) + c_{k-1}(1 + \eta_k)](1 + \delta_k)$$
$$= [\{(s_{k-1} - c_{k-1})\sigma_k - \eta_k c_{k-1}(1 + \sigma_k) - c_{k-1}\}(1 + \gamma_k) + c_{k-1}(1 + \eta_k)](1 + \delta_k)$$
$$= [(s_{k-1} - c_{k-1})\sigma_k(1 + \gamma_k) - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))](1 + \delta_k),$$
$$s_k - c_k = ((s_{k-1} - c_{k-1}) - \eta_k c_{k-1})\ (1 + \sigma_k)$$
$$- [(s_{k-1} - c_{k-1})\sigma_k(1 + \gamma_k) - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k)](1 + \delta_k)$$
$$= (s_{k-1} - c_{k-1})((1 + \sigma_k) - \sigma_k(1 + \gamma_k)(1 + \delta_k))$$
$$+ c_{k-1}(-\eta_k(1 + \sigma_k) + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))\ (1 + \delta_k))$$
$$= (s_{-1} - c_{k-1})\ (1 - \sigma_k(\gamma_k + \delta_k + \gamma_k\delta_k))$$
$$+ c_{k-1} - [\eta_k + \gamma_k + \eta_k(\gamma_k + \sigma_k\gamma_k) + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))\delta_k]$$

由于 S_k 和 C_k 最多仅计算到次数 ε^2，因此这些公式可以简化为

$$C_k = (\sigma_k + O(\varepsilon^2))S_{k-1} + (-\gamma_k + O(\varepsilon^2))C_{k-1}$$
$$S_k = ((1 + 2\varepsilon^2 + O(\varepsilon^3))S_{k-1} + (2\varepsilon + O(\varepsilon^2))C_{k-1}$$

使用这些公式可以得出

$$C_2 = \sigma_2 + O(\varepsilon^2)$$

$$S_2 = 1 + \eta_1 - \gamma_1 + 10\varepsilon^2 + O(\varepsilon^3)$$

一般情况下通过归纳可以很容易地得到

$$C_k = \sigma_k + O(\varepsilon^2)$$

$$S_k = 1 + \eta_1 - \gamma_1 + (4_k+2)\varepsilon^2 + O(\varepsilon^3)$$

最后，所需的是 s_k 中 x_1 的系数。要获得此值，令 $x_{n+1} = 0$，假设下标为 $n + 1$ 的所有希腊字母都等于 0，计算 s_{n+1}。可知，$s_{n+1} = s_n - c_n$，且 s_n 中 x_1 的系数小于它在 s_{n+1} 中的系数，后者即 $S_n = 1 + \eta_1 - \gamma_1 + (4n + 2)\varepsilon^2 = (1 + 2\varepsilon + O(n\varepsilon^2))$。 ∎

D.11 各种 IEEE 754 实现的差别

注 – 此节不是已发表的论文的一部分。增加此节是为了澄清某些观点，并纠正读者可能通过论文推知的某些有关 IEEE 标准的误解。此材料不是由 David Goldberg 编写的，但却是经过了他的许可才在此处公布。

前面的论文已经表明：执行浮点运算时必须小心谨慎，因为编程人员可能要依靠其属性来实现程序的正确性和准确性。特别是在实现 IEEE 标准时需小心谨慎，只有在符合标准的系统上才能编写出能够正常工作并给出准确结果的有用程序。读者可能会据此得出结论：这样的程序应该可以移植到所有 IEEE 系统上。事实上，如果 “当一个程序在两台支持 IEEE 运算的计算机之间迁移时，如果任何中间结果是不同的，则一定是由于软件错误，而不是算法差异。”这一条件成立，那么编写可移植软件会变得更容易。

遗憾的是，IEEE 标准并不保证同一程序在所有符合该标准的系统上都将提供完全相同的结果。实际上，由于种种原因，大多数程序都会在不同的系统上产生不同的结果。其中一个原因是，大多数程序都涉及十进制格式和二进制格式之间的数字转换，而 IEEE 标准没有完全指定执行这样的转换必须使用的准确度。另一个原因是，许多程序使用由系统库提供的初等函数，而该标准并没有详细说明这些函数。当然，大多数编程人员都知道这些功能已经超出了 IEEE 标准的范围。

许多编程人员可能没有意识到，甚至是仅使用 IEEE 标准规定的数字格式和操作数的程序也可能在不同的系统上计算出不同的结果。实际上，该标准制定者的初衷就是允许不同的实现获得不同的结果。在 IEEE 754 标准中术语**目标**的定义里清晰地表达了这个意思：“目标可以被用户显式指定，也可以由系统隐式提供（例如，子表达式或过程参数中的中间结果）。某些语言会将中间计算的结果放在用户无法控制的目标中。但是，此标准根据该目标的格式和操作数的值定义运算的结果。”（IEEE 754-1985，第 7 页）换句话说，IEEE 标准要求将每个结果都正确舍入到将放置它的目标的精度，但是标准不要求由用户程序确定的该目标精度。因此，不同的系统可能将其结果提供给不同精度的目标，使同一程序产生不同的结果（有时差异很大），即使那些系统都符合标准亦如此。

前面论文中的几个实例需要有关舍入浮点运算方式的某些知识。为了灵活使用诸如这些例子的实例，编程人员必须能够预知将如何解释程序，尤其是，在 IEEE 系统上，每个算术运算的目标的精度可能是什么。但是，IEEE 标准中*目标*定义的漏洞削弱了编程人员知道将如何解释程序的能力。因此，在高级语言中作为明显可移植程序实现时，上面给出的几个实例可能无法在 IEEE 系统上正常工作，通常将结果提供给其精度与编程人员所预期不同的目标。其他实例也可能正常工作，但是证明它们是否正常工作可能超出了一般编程人员的能力。

在此节中，我们根据 IEEE 754 运算的现有实现通常使用的目标格式的精度对它们进行分类。然后，回顾论文中的一些实例，来说明以比程序预期精度更宽的精度提供结果会导致它计算出错误的结果，即使在使用期望的精度时它被证明是正确的。我们还可以再查看论文中其中一个证明，以阐明处理未预料的精度所需的脑力工作，即使该精度还没有使程序无效。这些实例说明，IEEE 标准允许在不同实现之间存在差异会阻止我们编写出可以准确预知其行为的可移植、高效数值软件，而与它规定的所有内容无关。要开发这样的软件，则首先必须创建限制 IEEE 标准允许的可变性的程序设计语言和环境，并允许编程人员表示其程序所依赖的浮点语义。

D.11.1 当前的 IEEE 754 实现

IEEE 754 运算的当前实现可以分为两组，它们是按支持硬件中不同浮点格式的程度区分的。*基于扩展的*系统，如 Intel x86 系列处理器，完全支持扩展的双精度格式，但是仅部分支持单精度和双精度：它们提供按单精度和双精度装入或存储数据的指令，飞速地将数据在单精度和双精度与扩展双精度格式之间来回转换，它们还提供特殊模式（不是默认模式），在这样的模式下按单精度或双精度舍入算术运算的结果，即使这些结果以扩展双精度格式保存在寄存器中。（Motorola 68000 系列处理器在这些模式下按单精度或双精度格式的精度和范围舍入结果。 Intel x86 及兼容处理器按单精度或双精度格式的精度舍入结果，但保留与扩展双精度格式相同的范围。）*单精度 / 双精度*系统，包括大多数 RISC 处理器，完全支持单精度格式和双精度格式，但不支持符合 IEEE 的扩展双精度格式。（IBM POWER 架构仅部分支持单精度，但出于本节组织的目的，我们将其归入单精度 / 双精度系统。）

要查看计算的行为在基于扩展的系统上与在单精度 / 双精度系统上有何不同，请参考第 28 页的 “系统方面”中实例的 C 版本：

```
int main() {
    double  q;

    q = 3.0/7.0;
    if (q == 3.0/7.0) printf("Equal\n");
    else printf("Not Equal\n");
    return 0;
}
```

在这里，将常数 3.0 和 7.0 解释为双精度浮点数，而且表达式 3.0/7.0 继承双精度数据类型。在单精度 / 精度双系统上，将按双精度计算表达式的值，因为那是可以使用的最有效的格式。因此，将按双精度精确舍入 3.0/7.0 的值赋予 q。在下一行中，将再次按双精度计算表达式 3.0/7.0 的值，当然结果将等于刚赋给 q 的值，于是程序将像期望的那样打印 “Equal”。

在基于扩展的系统上，即使表达式 3.0/7.0 的类型为双精度，也将以扩展双精度格式在寄存器中计算商，因此在默认模式下，将按扩展双精度舍入它。但是，当将计算的值赋予变量 q 时，之后可能将它存储在内存中，因为 q 被声明为双精度，所以将按双精度舍入该值。在下一行中，可能再次按扩展精度计算表达式 3.0/7.0 的值，产生的结果与存储在 q 中的双精度值不同，使程序打印 “Not equal”。当然，其他结果也是可能的：编译器可以确定在将表达式 3.0/7.0 的值与 q 进行比较之前在第二行中存储该值并舍入它，或者可以按扩展精度将 q 保存在寄存器中而不存储它。优化的编译器可能在编译时 （也许是按双精度，也许是按扩展双精度）计算表达式 3.0/7.0 的值。（使用同一个 x86 编译器，在为优化进行编译时程序将打印 “Equal”，在为调试进行编译时将打印 “Not Equal”。）最后，基于扩展的系统的某些编译器自动更改舍入精度模式，以使在寄存器中产生结果的运算按单精度或双精度舍入那些结果，尽管使用更宽的范围是可能的。这样，在这些系统上，我们无法仅通过读取其源代码并运用 IEEE 754 运算的基本知识来预知程序的行为。我们也不能将未能提供符合 IEEE 754 的环境归因于硬件或编译器；硬件已经将正确舍入的结果提供给每个目标（按要求它做的那样），而且编译器已经将某些中间结果赋予超出用户控制的目标 （按允许它做的那样）。

D.11.2 在基于扩展的系统上计算的缺陷

按一般的思维，基于扩展的系统必须产生至少是精确的结果，即便不比在单精度 / 双精度系统上提供的更精确，因为前者始终提供尽可能高的精度，而且通常比后者高。通常的实例 （如上面的 C 程序）以及基于下面讨论的实例的更细致程序说明这种认识至少有一些天真：一些明显可移植的程序确实是可以跨单精度 / 双精度系统移植的，在基于扩展的系统上会提供错误的结果，完全是由于编译器和硬件协同工作，提供的精度有时候比程序所需的高。

当前的程序设计语言使程序很难指定它所需的精度。如第 30 页的“语言和编译器”上的“语言和编译器”节所述，许多程序设计语言不指定在同一上下文中某个表达式（如 `10.0*x`）每次运行应该计算出相同的值。在这方面，某些语言（如 Ada）就受到了 IEEE 标准之前的不同运算之间差异的影响。最近，类似 ANSI C 的语言受到了符合标准的、基于扩展的系统的影响。事实上，ANSI C 标准明确允许编译器按比通常与其类型关联的精度更宽的精度计算浮点表达式的值。因此，表达式 `10.0*x` 的值可能以取决于以下各种因素的方式变化：表达式是立即赋予变量还是作为子表达式出现在更大的表达式中；表达式是否参与比较；表达式是否作为参数传递给函数，如果是这样，该参数是按值还是按引用传递；当前的精度模式；编译程序时的优化级别；在编译程序时编译器所使用的精度模式和表达式计算方法；等等。

不能将表达式计算不完全一致全部归因于语言标准。可能的时候，在扩展精度寄存器中计算表达式，基于扩展的系统的运行效率最高，但是必须存储的值是按所需的最窄精度存储的。约束一种语言使其要求 `10.0*x` 在任何位置都计算为同一值将降低那些系统的性能。可惜的是，允许那些系统在语义上等价的上下文中将 `10.0*x` 计算为不同的值会对开发精确数值软件的编程人员产生它自己的负面影响，它阻止编程人员依赖其程序语法来表示预定语义。

真正的程序是否依赖给定表达式始终计算为同一值的假设？回想在定理 4 中提供的用于计算 ln(1 + *x*) 的算法，在此处用 Fortran 编写：

```
real function log1p(x)
real x
if (1.0 + x .eq.1.0) then
   log1p = x
else
   log1p = log(1.0 + x) * x / ((1.0 + x) - 1.0)
endif
return
```

在基于扩展的系统上，编译器可能在第三行中按扩展精度计算表达式 `1.0 + x` 的值并将结果与 `1.0` 进行比较。但是，将同一表达式传递给第六行中的 log 函数时，编译器可能将其值存储在内存中，并按单精度对其进行舍入。这样，如果对于按扩展精度不能将 `1.0 + x` 舍入为 `1.0`，`x` 不够小，但对于按单精度可以将 `1.0 + x` 舍入为 `1.0`，对于单精度已经足够小，那么 `log1p(x)` 返回的值将是零，而不是 `x`，相对误差将是一 -远远大于 5ε。同样，假定第六行中表达式的其余部分（包括子表达式 `1.0 + x` 的重新出现）是按扩展精度计算的。在这种情况下，如果 `x` 较小但没有小到足以按单精度将 `1.0 + x` 舍入为 `1.0`，则 `log1p(x)` 返回的值可以超过正确值接近 `x`，同样相对误差可以接近一。下面给出一个具体的实例，假设 `x` 等于 $2^{-24} + 2^{-47}$，因此 `x` 是最小的单精度数，所以 `1.0 + x` 上舍入到下一个较大的数 $1 + 2^{-23}$。那么，`log(1.0 + x)` 大约等于 2^{-23}。因为第六行中的表达式的分母是按扩展精度计算的，所以它是精确计算的并提供 `x`，这样 `log1p(x)` 返回的值大约等于 2^{-23}，该值几乎是精确值的两倍。（至少一种编译器实际出现过这种情况。当用于 x86 系统的 Sun WorkShop Compilers 4.2.1 Fortran 77 编译器使用 `-O` 优化标志编译前面的代码时，生成的代码完全像说明的那样计算 `1.0 + x`。因此，函数为 `log1p(1.0e-10)` 提供零，为 `log1p(5.97e-8)` 提供 `1.19209E-07`。）

为使定理 4 的算法正常工作，在表达式 `1.0 + x`每次出现时，必须以相同的方式对其进行计算；在基于扩展的系统上，只有当 `1.0 + x` 的计算结果在一个实例中为扩展双精度表示，在另一个实例中为单精度或双精度表示时，该算法才可能无法正常工作。当然，由于 `log` 是 Fortran 中的一个通用内函数，编译器可能从始至终采用扩展精度来计算表达式 `1.0 + x`的值，以相同的精度计算它的对数，但事实上我们不能假设编辑器将这样做。（还可以设想一个涉及用户定义函数的类似示例。在该情况下，即使用户定义的函数返回的是单精度结果，编译器可能仍以扩展精度保存参数，但是即使任何现有的 Fortran 编译器这样做，也很少。）因此，我们可以尝试确保通过将 `1.0 + x` 指定给某个变量，以便统一计算结果。遗憾的是，如果我们声明变量 `real`，由于编译器替换该变量一种外观表示以扩展精度保存在寄存器中的值，以及该变量另一种外观表示以单精度存储在内存中的值，我们的想法依然无法实现。相反，我们需要用与扩展精度格式对应的某种类型声明该变量。标准 FORTRAN 77 不支持这样做，而 Fortran 95 则提供了 `SELECTED_REAL_KIND` 机制来描述各种格式，它并不明确要求以扩展精度计算表达式结果的实现，以允许采用扩展精度声明变量。简而言之，在标准 Fortran 中没有便捷的方式来编写该程序，以保证始终以我们预想的方式计算表达式 `1.0 + x` 的值。

在基于扩展的系统上，有其他一些示例，即使每个子表达式都已存储并因此用相同的精度舍入，还是不能正常工作。原因就是*双舍入*。在默认精度模式下，基于扩展的系统最初会将每个结果舍入为扩展双精度。如果再将该结果以双精度存储，会对它再次进行舍入操作。这两次舍入的组合可能使生成的值与通过将第一次的结果正确舍入为双精度所得的结果不同。当舍入为扩展双精度是结果是“中间数”时，即，该结果恰好在两个双精度数的中间时，可能出现上述情况，所以第二次舍入由舍入为偶数规则决定。如果第二次舍入和第一次舍入都是向上舍入或向下舍入，那么净舍入误差将超过最后一位中的一半单位。（注意，尽管双舍入只影响双精度计算。但还是可以证明只要 $q \geq 2p + 2$，两个 p- 位数的加、减、乘、除，或者一个 p 位数的平方根运算，第一次舍入为 q 位然后再舍入为 p 位与只舍入一次到 p 位的结果相同。因此，扩展的双精度的宽度已足够，单精度计算无需进行双舍入。）

依赖正确舍入的一些算法会因双舍入而失败。事实上，甚至是不要求正确舍入且在不符合 IEEE 754 的各种计算机上正确工作的一些算法也会因双舍入而失败。其中最有用的算法是用于执行在第 11 页的“定理 5”节中提到的模拟多精度运算的可移植算法。例如，定理 6 中所述的用于将浮点数拆分为高位和低位部分的过程在双舍入运算中不能正确工作：试图将双精度数 $2^{52} + 3 \times 2^{26} - 1$ 拆分为两部分，各部分最多为 26 位。按双精度正确舍入每个运算时，高位部分是 $2^{52} + 2^{27}$，低位部分是 $2^{26} - 1$，但先按扩展双精度舍入每个运算再按双精度舍入时，该过程产生的高位部分是 $2^{52} + 2^{28}$，低位部分是 $-2^{26} - 1$。后一个数占用 27 位，因此无法按双精度精确计算其平方。当然，仍有可能按扩展双精度计算此数的平方，但是得出的对数将不再能够移植到单精度 / 双精度系统。此外，多精度乘法算法中的后续步骤假定按双精度计算了所有的部分乘积。正确处理双精度变量和扩展双精度变量的混合将大大提高实现的成本。

同样，用于将表示为双精度数的数组的多精度数相加的可移植算法会在双舍入运算中失败。这些算法通常依赖类似于 Kahan 求和公式的方法。正如第 47 页的“求和中的误差”节中给出的求和公式非正式解释所提出的，如果 `s` 和 `y` 是浮点变量并且 $|s| \geq |y|$，计算：

```
t = s + y;
e = (s - t) + y;
```

那么，在大多数运算中， e 精确恢复计算 t 时出现的舍入误差。但是，此方法在双舍入运算中不能正常工作：如果 s = $2^{52} + 1$ 且 y = $1/2 - 2^{-54}$，那么， s + y 先按扩展双精度舍入为 $2^{52} + 3/2$，然后依据“在中途情况下舍入为偶数”规则按双精度将此值舍入为 $2^{52} + 2$；这样，计算 t 时的最终舍入误差是 $1/2 + 2^{-54}$，它无法按双精度精确表示，因此它无法用上面所示的表达式精确计算。同样，通过按扩展双精度计算和来恢复舍入误差将是可能的，但是这样程序将必须进行额外的工作才能将最终输出约简回双精度，双舍入也会影响此过程。由于这一原因，虽然通过这些方法模拟多精度运算的可移植程序在各种各样的计算机上正确而高效地运行，但是它们在基于扩展的系统上并不像所声称的那样运行。

最后，起初看上去似乎依赖正确舍入的一些算法事实上可能与双舍入一起正确工作。在这些情况下，处理双舍入的成本不在于实现，而在于对算法是否像声称的那样工作进行的检验。为了说明这一点，我们证明定理 7 的以下变形：

D.11.2.1 定理 7′

如果 m 和 n 是可以按 *IEEE 754* 双精度表示的整数，$|m| < 2^{52}$，n 具有特殊形式 $n = 2^i + 2^j$，那么 $(m \oslash n) \otimes n = m$，条件是两个浮点运算都是按双精度正确舍入的或先按扩展双精度舍入再按双精度舍入的。

D.11.2.2 证明

假定 $m > 0$。那么 $q = m \oslash n$。将二的幂作为因子进行调整，我们可以考虑一个等价设置，其中 $2^{52} \le m < 2^{53}$，对于 q 是类似的，以便 m 和 q 都是这样的整数：其最低有效位占用单位位置（即 $\mathrm{ulp}(m) = \mathrm{ulp}(q) = 1$）。在调整之前，假定 $m < 2^{52}$，因此在调整之后， m 是一个偶数整数。此外，因为 m 和 q 的调整值满足 $m/2 < q < 2m$，所以 n 的对应值必须具有两种形式中的一种，具体取决于 m 和 q 哪个较大：如果 $q < m$，显然 $1 < n < 2$，因为 n 是两个二的幂的和，所以对于一些 k，$n = 1 + 2^{-k}$；同样，如果 $q > m$，那么 $1/2 < n < 1$，因此 $n = 1/2 + 2^{-(k+1)}$。（因为 n 是两个二的幂的和，所以 n 的最接近一的可能值是 $n = 1 + 2^{-52}$。因为 $m/(1 + 2^{-52})$ 不比小于 m 的下一个较小双精度数大，所以不能使 $q = m$。）

假设 e 表示计算 q 时的舍入误差，以便 $q = m/n + e$，计算值 $q \otimes n$ 将是 $m + ne$ 的（一次或两次）舍入值。请首先考虑按双精度正确舍入每个浮点运算的情况。在这种情况下，$|e| < 1/2$。如果 n 具有形式 $1/2 + 2^{-(k+1)}$，则 $ne = nq - m$ 是 $2^{-(k+1)}$ 的整数倍，且 $|ne| < 1/4 + 2^{-(k+2)}$。这意味着 $|ne| \le 1/4$。请回想一下 m 与下一个较大的可表示数之间的差是 1，m 和下一个较小的可表示数之间的差是 1（如果 $m > 2^{52}$）或 1/2（如果 $m = 2^{52}$）。这样，因为 $|ne| \le 1/4$，$m + ne$ 将舍入为 m。（即使 $m = 2^{52}$ 且 $ne = -1/4$，乘积也将按“在中途情况下舍入为偶数”规则舍入为 m。）同样，如果 n 具有形式 $1 + 2^{-k}$，则 ne 是 2^{-k} 的整数倍，且 $|ne| < 1/2 + 2^{-(k+1)}$；这隐含 $|ne| \le 1/2$。在这种情况下，不能使 $m = 2^{52}$，因为 m 严格大于 q，因此 m 与其最接近的可表示数相差 ±1。这样，因为 $|ne| \le 1/2$，$m + ne$ 同样将舍入为 m。（即使 $|ne| = 1/2$，乘积也将按“在中途情况下舍入为偶数”规则舍入为 m，因为 m 是偶数。）这样就完成了正确舍入运算的证明。

在双舍入运算中，仍可能发生以下情况：q 是正确舍入的商（即使它实际上舍入了两次），因此像上面那样 $|e| < 1/2$。在这种情况下，我们可以求助于上一段中的论点，条件是考虑到 $q \otimes n$ 将被舍入两次的事实。为了解释这一点，请注意 IEEE 标准要求扩展双精度格式至少有 64 个有效位，以便数 $m \pm 1/2$ 和 $m \pm 1/4$ 可以按扩展双精度精确表示。这样，如果 n 具有形式 $1/2 + 2^{-(k+1)}$，以便 $|ne| \le 1/4$，则按扩展双精度舍入 $m + ne$ 必须产生与 m 最多相差 1/4 的结果，如上所述，此值将按双精度舍入为 m。同样，如果 n 具有形式 $1 + 2^{-k}$，以便 $|ne| \le 1/2$，则按扩展双精度舍入 $m + ne$ 必须产生与 m 最多相差 1/2 的结果，而且此结果将按双精度舍入为 m。（请回想，在这种情况下 $m > 2^{52}$。）

最后，要考虑具有以下特点的其余情况：因双舍入而导致 q 不是正确舍入的商。在这些情况下，最差具有 $|e| < 1/2 + 2^{-(d+1)}$，其中 d 是扩展双精度格式中的额外位数。（所有基于扩展的现有系统都支持正好具有 64 个有效位的扩展双精度格式；对于此格式，$d = 64 - 53 = 11$。）因为在第二次舍入由“在中途情况下舍入为偶数”规则确定时双舍入仅产生不正确的舍入结果，所以 q 必须是偶数整数。这样，如果 n 具有形式 $1/2 + 2^{-(k+1)}$，则 $ne = nq - m$ 是 2^{-k} 的整数倍，且

$$|ne| < (1/2 + 2^{-(k+1)})(1/2 + 2^{-(d+1)}) = 1/4 + 2^{-(k+2)} + 2^{-(d+2)} + 2^{-(k+d+2)}。$$

如果 $k \le d$，则意味着 $|ne| \le 1/4$。如果 $k > d$，则我们具有 $|ne| \le 1/4 + 2^{-(d+2)}$。在任一情况下，乘积的第一次舍入都将提供与 m 最多相差 1/4 的结果，依据上述论点，第二次舍入将舍入为 m。同样，如果 n 具有形式 $1 + 2^{-k}$，则 ne 是 $2^{-(k-1)}$ 的整数倍，且

$$|ne| < 1/2 + 2^{-(k+1)} + 2^{-(d+1)} + 2^{-(k+d+1)}。$$

如果 $k \le d$，则意味着 $|ne| \le 1/2$。如果 $k > d$，则我们具有 $|ne| \le 1/2 + 2^{-(d+1)}$。在任一情况下，乘积的第一次舍入都将提供与 m 最多相差 1/2 的结果，同样依据上述论点，第二次舍入将舍入为 m。▮

前面的证明表明：仅当商引起双舍入时，乘积才会引起双舍入，甚至在此时乘积也舍入为正确的结果。该证明还表明：即使是对于只有两个浮点运算的程序来说，扩展我们的推理以包括存在双舍入的可能也可以是有挑战性的。对于更复杂的程序，系统地解释双舍入的影响也许是不可能的，更不必说双精度计算和扩展双精度计算的更一般组合。

D.11.3 扩展精度的程序设计语言支持

不应该由前面的实例得出扩展精度*本身*是无益的这一结论。当编程人员能够有选择地使用扩展精度时，许多程序可以从扩展精度受益。遗憾的是，当前的程序设计语言没有提供足够的方式，供编程人员用来指定应该使用扩展精度的时间和方式。为了指出所需的支持，我们考虑可能希望管理扩展精度的使用的方式。

在将双精度用作其标称工作精度的可移植程序中，我们可能希望控制更宽精度的使用的方式有以下五种：

1. 在基于扩展的系统上尽可能使用扩展精度进行编译，以产生速度最快的代码。显然，大多数数值软件对运算的要求仅仅是每个运算中的相对误差以“计算机厄普西隆”为界限。当内存中的数据按双精度存储时，通常认为计算机厄普西隆是该精度中的最大相对舍入误差，因为（正确或错误地）假定在输入数据被输入时已经过舍入，在存储结果时将对其进行类似舍入。这样，尽管按扩展精度计算一些中间结果可能产生更精确的结果，但是扩展精度不是必需的。在这种情况下，我们可能更希望仅当扩展精度不会明显减慢程序速度时编译器才使用扩展精度，否则使用双精度。

2. 如果比双精度宽的一种格式相当快且足够宽，则使用该格式，否则使用其他格式。当扩展精度可用时，一些计算可以更容易地执行，它们也可以按双精度执行，但是会更加复杂。以计算双精度数向量的欧几里得范数为例。通过计算元素的平方并以 IEEE 754 扩展双精度格式（使用其较宽的指数范围）累加其和，可以为实用长度的向量很普通地避免过早下溢或上溢。在基于扩展的系统上，这是计算范数的最快方式。在单精度/双精度系统上，扩展双精度格式将必须在软件中仿真（如果支持一种格式的话），而且这样的仿真将比仅使用双精度慢得多，测试异常标志以确定是否发生下溢或上溢，如果是这样，使用显式调整重复该计算。请注意，要支持扩展精度的这一使用，语言必须同时提供相当快的最宽可用格式的指示（以便程序可以选择使用哪种方法）和指示每种格式的精度及范围的环境参数（以便程序可以检验最宽的快速格式是否足够宽，例如，检验它是否具有比双精度更宽的范围）。

3. 即使比双精度更宽的格式必须在软件中仿真，也要使用该格式。对于比欧几里得范数实例更复杂的程序，编程人员可能仅希望消除编写程序的两个版本的需要；相反，即使扩展精度的速度慢也依赖它。同样，语言必须提供环境参数，程序才能确定最宽可用格式的范围和精度。

4. 请勿使用更宽的精度；即使采用扩展的范围，也将结果正确舍入为双精度格式。对于需要依赖正确舍入的双精度运算轻松编写的程序（包括上面提到的部分示例），语言必须为编程人员提供指出不得使用扩展精度的方式，即使可以在寄存器中使用比双精度更宽的指数范围来计算中间结果也不能使用扩展精度。（以此方式计算出的中间结果在存储到内存时，如果出现下溢，仍可能执行双舍入：如果算术运算的结果第一次被舍入为 53 个有效位，然后在必须对该结果进行非规格化操作时，再次将它舍入位更小的有效位数，那么最终结果可能与只舍入一次，成为非规格化数所得的结果不同。当然，这种双舍入的形式对任何实际的程序产生不良影响的可能性非常低。）

5. 按双精度格式的精度和范围正确舍入结果。对于测试数值软件或接近双精度格式的范围及精度的极限的运算本身的程序，进行双精度严格强制将是极其有用的。以可移植方式编写这样细致的测试程序往往是很困难的；当它们必须利用伪子例程和其他技巧强制按特定格式舍入结果时，编写它们更加困难（且容易出错）。因此，使用基于扩展的系统开发必须可以移植到所有 IEEE 754 实现的可靠软件的编程人员，将很快发现可以仿真单精度 / 双精度系统的运算而无需特别的努力。

当前语言都不支持所有这五种选择。事实上，几乎没有语言尝试为编程人员提供控制扩展精度的使用的能力。一个值得注意的例外是 ISO/IEC 9899:1999 程序设计语言 - C 标准，它是 C 语言的最新修订，现在处于标准化的最终阶段。

C99 标准允许实现以比通常与其类型关联的格式更宽的格式计算表达式的值，但是 C99 标准建议使用仅仅三种表达式计算方法之一。建议使用的三种方法以将表达式“提升”到更宽格式的程度为特征，鼓励实现通过定义预处理程序宏 `FLT_EVAL_METHOD` 来识别它使用哪种方法：如果 `FLT_EVAL_METHOD` 是 0，则以对应于其类型的格式计算每个表达式；如果 `FLT_EVAL_METHOD` 是 1，则将浮点表达式提升到对应于双精度的格式；如果 `FLT_EVAL_METHOD` 是 2，则将浮点和双精度表达式提升到对应于长双精度的格式。（允许实现将 `FLT_EVAL_METHOD` 设置为 -1 以指示表达式计算方法是不能确定的。）C99 标准还要求 `<math.h>` 头文件定义类型 `float_t` 和 `double_t`，它们分别至少与浮点和双精度一样宽，旨在与用于计算浮点和双精度表达式的类型匹配。例如，如果 `FLT_EVAL_METHOD` 是 2，则 `float_t` 和 `double_t` 都是长双精度。最后，C99 标准要求 `<float.h>` 头文件定义指定对应于每种浮点类型的格式的范围及精度的预处理程序宏。

C99 标准要求或建议的功能组合支持上面列出的五种选择中的一些选择，但是不支持所有选择。例如，如果实现将长双精度类型映射到扩展双精度格式，并将 `FLT_EVAL_METHOD` 定义为 2，则编程人员可以合理地假定扩展精度的速度较快，因此类似欧几里得范数实例的程序只需使用长双精度（或 `double_t`）类型的中间变量即可。另一方面，同一实现必须按扩展精度保存匿名表达式，即使它们存储在内存中（例如，当编译器必须使浮点寄存器溢出时），并且它必须存储赋予声明为双精度的变量的表达式的结果，以便将其转换为双精度，即使它们可以保存在寄存器中。这样，双精度或 `double_t` 类型都不能进行编译以便在基于扩展的当前硬件上产生速度最快的代码。

同样，C99 标准提供了此部分中实例所说明的一些问题（但不是所有问题）的解决方法。如果将表达式 `1.0 + x` 赋予一个变量（为任何类型）且该变量是自始至终使用的，则可以保证 `log1p` 函数的 C99 标准版本正确工作。但是，用于将双精度数拆分为高位部分和低位部分的可移植的、高效的 C99 标准程序更为困难：在无法保证按双精度正确舍入双精度表达式的情况下，如何在正确的位置进行拆分并避免双舍入？一种解决方法是使用 `double_t` 类型在单精度 / 双精度系统上按双精度执行拆分，在基于扩展的系统上按扩展精度进行拆分，以便在任一情况下都将正确舍入运算。定理 14 告诉我们可以在任意位的位置进行拆分，条件是知道基础运算的精度，`FLT_EVAL_METHOD` 和环境参数宏应该提供此信息。

下面的程序片段显示一种可能的实现：

```
#include <math.h>
#include <float.h>

#if (FLT_EVAL_METHOD==2)
#define PWR2  LDBL_MANT_DIG - (DBL_MANT_DIG/2)
#elif ((FLT_EVAL_METHOD==1) || (FLT_EVAL_METHOD==0))
#define PWR2  DBL_MANT_DIG - (DBL_MANT_DIG/2)
#else
#error FLT_EVAL_METHOD unknown!
#endif

...
    double   x, xh, xl;
    double_t m;

    m = scalbn(1.0, PWR2) + 1.0;  // 2**PWR2 + 1
    xh = (m * x) - ((m * x) - x);
    xl = x - xh;
```

当然，要求出此解，编程人员必须知道：双精度表达式可能是按扩展精度计算的；随之出现的双舍入问题可以导致算法出错；依据定理 14 可能改用扩展精度。更显而易见的解决方法是只需指定按双精度正确舍入每个表达式即可。在基于扩展的系统上，这只要求更改舍入精度模式，但可惜的是， C99 标准没有提供做到这一点的可移植方法。（Floating-Point C Edits （指定为支持浮点而对 C90 标准进行的更改的工作文档）的早期草案建议具有舍入精度模式的系统上的实现提供 `fegetprec` 和 `fesetprec` 函数以获取和设置舍入精度，这两个函数与获取和设置舍入方向的 `fegetround` 和 `fesetround` 函数类似。在对 C99 标准进行更改之前删除了此建议案。）

巧合的是， C99 标准支持具有不同整数运算功能的系统之间的可移植性的途径是建议了一种支持不同浮点架构的更好方法。每个 C99 标准实现都提供一个 `<stdint.h>` 头文件，该文件定义实现所支持的那些整数类型，它们按大小和效率命名：例如， `int32_t` 是 32 位宽的整数类型， `int_fast16_t` 是实现的速度最快且至少 16 位宽的整数类型， `intmax_t` 是支持的最宽整数类型。可以为浮点类型设想一个类似的方案：例如， `float53_t` 可用于命名具有正好 53 位精度但可能具有更宽范围的一种浮点类型， `float_fast24_t` 可用于命名实现的速度最快且具有至少 24 位精度的类型， `floatmax_t` 可用于命名支持的最宽且速度相当快的类型。快速类型可能允许基于扩展的系统上的编译器生成可能最快的代码，这仅受到以下约束的影响：指定变量的值不能因寄存器溢出而改变。精确宽度类型将导致基于扩展的系统上的编译器将舍入精度模式设置为按指定精度舍入，从而允许更宽的范围受到同一约束的影响。最后， `double_t` 可用于命名一种同时具有 IEEE 754 双精度格式的精度和范围的类型，条件是使用严格的双精度计算。与相应命名的环境参数宏一起，这样的方案就可以支持上述的所有五种选择，并允许编程人员轻松而明确地指出其程序所需的浮点语义。

扩展精度的语言支持必须这样复杂吗？在单精度 / 双精度系统上，上面列出的五种选择中的四种是一致的，因此不必区分快速和精确宽度类型。但是，基于扩展的系统会给选择带来困难：它们既不支持纯双精度计算的效率也不支持纯扩展精度计算的效率与这两种运算的混合一样高，而且不同的程序要求不同的混合程度。此外，选择何时使用扩展精度不应该留给编译器编写人员，他们通常受到测试基准的影响 （有时得到数值分析员的直接通知），将浮点运算视为“本身不精确”，因此不值得也不能够预知整数运算。相反，必须由编程人员进行选择，他们将需要能够表达其选择的语言。

D.11.4 结束语

上述评论并非旨在贬低基于扩展的系统，而是要揭示几个谬论，第一个谬论是所有 IEEE 754 系统都必须为同一程序提供完全相同的结果。我们着重说明了基于扩展的系统与单精度 / 双精度系统之间的不同，但是在其中每个系列内的系统之间存在进一步的不同。例如，一些单精度 / 双精度系统提供单个指令将两个数相乘并与第三个数相加，只进行一次最终舍入。此运算称为 *合并的乘 - 加*，会导致同一程序在不同的单精度 / 双精度系统上产生不同的结果；与扩展精度一样，它甚至会导致同一程序在同一系统上产生不同的结果，这取决于是否使用它和何时使用它。（合并的乘 - 加也可以阻止定理 6 的溢出过程，尽管能够以不可移植的方式使用它来执行多精度乘法运算，而不必进行拆分。）尽管 IEEE 标准没有预期到这样的操作，但是它与此一致：中间乘积被提供给超出用户控制的“目标”，它的宽度足以精确容纳该乘积，最终的和被正确舍入以适合其单精度或双精度目标。

但是，以下意见是吸引人的：IEEE 754 精确规定给定的程序必须提供的结果。许多编程人员喜欢相信他们可以理解程序的行为，并证明它将正确运行，而不涉及编译它的编译器或运行它的计算机。在许多方面，对于计算机系统和程序设计语言的设计人员来说，支持此意见是一个需要花费精力的目标。可惜的是，当涉及到浮点运算时，该目标几乎是不可能实现的。IEEE 标准的制定者知道他们不试图实现该目标。因此，尽管在整个计算机行业几乎都符合 IEEE 754 标准 （的大部分内容），但是开发可移植软件的编程人员必须继续处理不可预知的浮点计算。

如果编程人员利用 IEEE 754 的各种特性，则他们将需要使浮点运算可预知的程序设计语言。C99 标准在一定程度上改进了可预知性，代价是要求编程人员编写其程序的多个版本，对于每个 `FLT_EVAL_METHOD` 都编写一个版本。现在还不知道：将来的语言是否将改为选择允许编程人员使用明确表达程序对 IEEE 754 语义的依赖程度的语法，来编写单个程序。基于扩展的现有系统对该前景构成了威胁，因为它们使我们很容易假定编译器和硬件对应该如何在给定系统上执行计算的了解比编程人员更好。该假定是第二个谬论：计算的结果所需的准确度不取决于产生它的计算机，而是仅取决于从它得出的结论；在编程人员、编译器和硬件中，最多只有编程人员才能知道那些结论可能是什么。