Printing Floating-Point Numbers Quickly and Accurately with Integers
用整数快速准确地打印浮点数

Florian Loitsch
作者: Florian Loitsch

Inria Sophia Antipolis
在法国索菲亚科技园
2004 Route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex
2004 年法国索菲亚科技园
orian.loitsch@inria.fr
Orian.loitsch@inria.fr

Abstract
摘要

We present algorithms for accurately converting floating-point numbers to decimal representation. They are fast (up to 4 times faster than commonly used algorithms that use high-precision in-tegers) and correct: any printed number will evaluate to the same number, when read again.
我们提出了将浮点数精确转换为小数表示的算法。它们速度快(比使用高精度四进制的常用算法快 4 倍)并且是正确的: 当再次读取时，任何打印的数字都将计算为相同的数字。

Our algorithms are fast, because they require only fixed-size integer arithmetic. The sole requirement for the integer type is that it has at least two more bits than the significand of the floating-point number. Hence, for IEEE 754 double-precision numbers (having a 53-bit significand) an integer type with 55 bits is sufficient. Moreover we show how to exploit additional bits to improve the generated output.
我们的算法很快，因为它们只需要固定大小的整数运算。整数类型的唯一要求是它至少比浮点数的有效值多两位。因此，对于 IEEE 754 双精度数字(具有 53 位有效值)，一个 55 位的整数类型就足够了。此外，我们还展示了如何利用额外的位来提高生成的输出。

We present three algorithms with different properties: the first algorithm is the most basic one, and does not take advantage of any extra bits. It simply shows how to perform the binary-to-decimal transformation with the minimal number of bits. Our second algorithm improves on the first one by using the additional bits to produce a shorter (often the shortest) result.
我们提出了三种不同性质的算法: 第一种算法是最基本的算法，不利用任何额外的比特。它只是简单地展示了如何用最少的比特数执行二进制到十进制的转换。我们的第二个算法比第一个算法有所改进，通过使用额外的比特来产生一个更短的(通常是最短的)结果。

Finally we propose a third version that can be used when the shortest output is a requirement. The last algorithm either produces optimal decimal representations (with respect to shortness and rounding) or rejects its input. For IEEE 754 double-precision numbers and 64-bit integers roughly 99.4% of all numbers can be processed efficiently. The remaining 0.6% are rejected and need to be printed by a slower complete algorithm.
最后，我们提出了第三个版本，可以在需要最短输出时使用。最后一种算法要么产生最佳的十进制表示(关于短和舍入)，要么拒绝它的输入。对于 IEEE 754 双精度数字和 64 位整数，大约 99.4% 的数字可以被有效地处理。剩下的 0.6% 会被拒绝，需要一个更慢的完整算法来打印。

Categories and Subject Descriptors I.m [Computing Methodolo-gies]: Miscellaneous
类别和主题描述符 I.m [计算方法学] : 杂项

General Terms Algorithms
通用术语算法

Keywords floating-point printing, dtoa
浮点打印

1. Introduction
引言

Printing floating-point numbers has always been a challenge. The naive approach is not precise enough and yields incorrect results in many cases. Throughout the 1970s and 1980s many language libraries and in particular the printf function of most C libraries were known to produce wrong decimal representations.
打印浮点数始终是一个挑战。天真的方法不够精确，在很多情况下会产生不正确的结果。在整个 20 世纪 70 年代和 80 年代，许多语言库，特别是大多数 c 库的 printf 函数都被认为会产生错误的十进制表示。

In the early 1980s Coonen published a paper [Coonen(1980)] and a thesis [Coonen(1984)] containing algorithms for accurate yet economical binary-decimal conversions, but his work went largely unnoticed (at least with respect to the printing algorithms).

20 世纪 80 年代初期，Coonen 发表了一篇论文[Coonen (1980)]和一篇论文[Coonen (1984)]，其中包含了精确而经济的二进制-十进制转换算法，但他的工作基本上没有引起人们的注意(至少在打印算法方面)。

Steele and White's paper [Steele Jr. and White(1990)][1] had a much bigger impact. Correct printing become part of the specification of many languages and furthermore all major C libraries (and as a consequence all programs relying on the printf functions) adapted accurate algorithms and print correct results now.

Steele and White 的论文[Steele jr. and White (1990)]1 产生了更大的影响。正确的打印成为许多语言规范的一部分，而且所有主要的 c 语言库(因此所有依赖于 printf 函数的程序)现在都适应了精确的算法并打印正确的结果。

Steele and White's algorithm, "Dragon4", relies on high precision arithmetic (also known as "bignums") and even though two other papers ([Gay(1990)] and [Burger and Dybvig(1996)]) proposed improvements and optimizations to the algorithm this requirement remained. It is natural to wonder if limited-precision arithmetic could suffice. Indeed, according to Steele and White's retrospective of 2003 [Steele Jr. and White(2004)] "[d]uring the 1980s, White investigated the question of whether one could use limited-precision arithmetic [...] rather than bignums. He had ear-lier proved by exhaustive testing that just 7 extra bits suffice for correctly printing 36-bit PDP-10 floating-point numbers, if pow-ers of ten used for prescaling are precomputed using bignums and rounded just once". The document continues by asking whether "[one could] derive, without exhaustive testing, the necessary amount of extra precision solely as a function of the precision and exponent range of a floating-point format".

Steele 和 White 的算法" Dragon4"依赖于高精度算法(也称为" bignums")，尽管其他两篇论文([Gay (1990)]和[Burger and Dybvig (1996)])提出了对算法的改进和优化，但这个要求仍然存在。自然而然地想知道有限精度的算术是否可以满足要求。事实上，根据 Steele and White 对 2003 年的回顾[Steele jr. and White (2004)]"在 20 世纪 80 年代，White 调查了是否可以使用有限精度的算术而不是大数的问题。他早些时候已经通过详尽的测试证明，如果用于预分级的 10 位幂使用 bignums 预先计算并四舍五入一次，那么只需要 7 个额外的位就足以正确地打印 36 位 PDP-10 浮点数。"。该文件继续询问，"(人们)是否可以在不进行详尽测试的情况下，仅仅通过浮点格式的精度和指数范围的函数推导出必要的额外精度"。

In this paper we will present a new algorithm Grisu, which allows us to answer this question. Grisu requires only two extra bits and a cache of precomputed powers-of-ten whose size depends on the exponent range.

在本文中，我们将提出一个新的算法 Grisu，它允许我们回答这个问题。Grisu 只需要两个额外的位和一个预先计算的幂 -10 的缓存，其大小取决于指数范围。

However, Grisu does not supersede Dragon4 and its optimized descendants. While accurate and fast (up to 4 times faster than previous approaches) it produces suboptimal results. For instance the IEEE 754 double-precision number representing 0.3 is printed as 29999999999999998e-17. When read, both numbers will be approximated to the same floating-point number. They are hence both accurate representations of the corresponding floating-point number, but the shorter 0.3 is clearly more desirable.

然而，Grisu 并没有取代 Dragon4 及其优化后代。虽然精确和快速(比以前的方法快 4 倍)，但它产生的结果并不理想。例如，表示 0.3 的 IEEE 754 双精度数字打印为 29999999999999999998e-17。当读取时，这两个数字将被近似为相同的浮点数。因此，它们都是相应浮点数的精确表示，但更短的 0.3 显然更令人满意。

With just two extra bits it is difficult to do better than in our example, but often there exists an integer type with more bits. For IEEE 754 floating-point numbers, which have a significand size of 53, one can use 64 bit integers, providing 11 extra bits. We have developed an algorithm Grisu2 that uses these extra bits to shorten the output. However, even 11 extra bits may not be sufficient in every case. There are still boundary conditions under which Grisu2 will not be able to produce the shortest representation. Since this property is often a requirement (see [Steele Jr. and White(2004)]

只有两个额外的比特，很难做得比我们的例子更好，但通常存在一个整数类型有更多的比特。对于有效大小为 53 的 IEEE 754 浮点数，可以使用 64 位整数，提供 11 个额外的位。我们已经开发了一个 Grisu2 算法，它使用这些额外的比特来缩短输出。然而，即使是 11 个额外的比特也可能不足以满足所有情况。仍然存在一些边界条件，在这些条件下 Grisu2 不能产生最短的表示。因为这个属性通常是一个要求(参见[Steele jr. 和 White (2004)])

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.
PLDI'10,2010 年 6 月 5 日至 10 日，加拿大安大略省多伦多。

1 A draft of this article had existed long before and had already been mentioned in "Knuth Volume 2"[Knuth(1981)] in 1981.
1 这篇文章的一个草稿很久以前就存在了，而且已经存在了1981 年在" Knuth 第二卷"[Knuth (1981)]中提到。

for some examples) we propose a variant, Grisu3, that detects (and aborts) when its output may not be the shortest. As a consequence Grisu3 is incomplete and will fail for some percentage of its input. Given 11 extra bits roughly 99.5% are processed correctly and are thus guaranteed to be optimal (with respect to shortness and rounding). The remaining 0.5% are rejected and need to be printed by another printing algorithm (like Dragon4).

对于一些例子)，我们提出了一个变体 Grisu3，当其输出可能不是最短的时候检测(并中止)。因此 Grisu3 是不完整的，并且会在一定百分比的输入中失败。给定 11 个额外的位，大约 99.5% 被正确处理，因此保证是最佳的(关于短小和舍入)。剩下的 0.5% 会被拒绝，需要另一种打印算法(比如 Dragon4)来打印。

All presented algorithms come with code snippets in C that show how they can be efficiently implemented. We use C99, as this version provides the user with a platform independent means of using 64-bit data types.

所有提出的算法都带有 c 语言的代码片段，这些代码片段展示了如何有效地实现它们。我们使用 C99，因为这个版本为用户提供了一个使用 64 位数据类型的平台独立的方法。

In this paper we will concentrate exclusively on IEEE 754 double-precision floating-point numbers. They are the de facto standard today and while our work applies to other floating-point representations it would unnecessarily complicate the descriptions.

在本文中，我们将专注于 IEEE 754 双精度浮点数。它们是当今事实上的行业标准，虽然我们的工作适用于其他浮点数表示，但它会不必要地使描述复杂化。

We will now discuss some basics in Section 2. In Section 3 we present a custom floating-point data-type which will be used in all remaining sections. Section 4 details the requirements on the cache of powers-of-ten. In Section 5 we introduce Grisu, and in Section 6 we present its evolutions Grisu2 and Grisu3. In Section 7 we interpret experimental results. Section 8 discusses related work, and we finally conclude in Section 9.

我们现在将在第 2 节中讨论一些基础知识。在第三部分中，我们展示了一个定制的浮点数据类型，它将在所有剩下的部分中使用。第 4 节详细描述了幂 -10 的缓存需求。在第 5 节我们介绍 Grisu，在第 6 节我们介绍它的进化 Grisu2 和 Grisu3。在第 7 节中，我们解释实验结果。第 8 部分讨论了相关的工作，我们最后在第 9 部分结束。

2. Floating-Point Numbers
浮点数

In this section we will give a short introduction on floating-point numbers. Interested readers may want to consult [Goldberg(1991)] for a thorough discussion of this subject. For simplicity we will consider only positive floating-point numbers. It is trivial to extend the text to handle signs.

在这一节中，我们将对浮点数作一个简短的介绍。有兴趣的读者可以参考[Goldberg (1991)]对这个主题进行全面的讨论。为了简单起见，我们将只考虑正浮点数。扩展文本来处理符号是很简单的。

Section 2.3 contains examples for all notions we introduce in this section. Readers might want to have a look at this section whenever a definition is unclear.

第 2.3 节包含了我们在这一节中介绍的所有概念的例子。当定义不清楚时，读者可能想看看这一节。

A floating point number, as the name suggests, has a radix point that can “float”. Concretely a floating-point number v in base b (usually 2) with precision p is built out of an integer significand (also known as mantissa or fraction) f_v of at most p digits and an exponent e_v, such that $v = f_v b^{e_v}$.

浮点数，顾名思义，有一个可以“浮动”的基点。具体来说，基数 b (通常为 2)中精度为 p 的浮点数 v 是由最多 p 位的整数有效数(也称为尾数或分数) f v 和指数 ev 构成的，即 v = f v bev。

Unless otherwise stated, we use the convention that the signifi-cand of a floating-point number is named f with the variable’s name as subscript. Similarly the exponent is written as e with the same subscript. For instance a floating-point number w is assumed to be composed of f_w and e_w.

除非另有说明，否则我们使用的约定是浮点数的 signfi-c 被命名为 f，变量的名字作为下标。类似的，指数写成 e，下标也是相同的。例如，一个浮点数 w 被假定为由 f w 和 w 组成。

Any significand f satisfies $f = \sum_{i=0}^{p-1} d_i b^{-i}$, $0 \le d_i < b$ where
任何显著性 f 满足 f = Pp 1 di bi，0 di < b，其中

the integers d_i are called the digits of f . We call a number “normal-ized” if the most-significant digit d_{p-1} is non-zero.

整数 di 称为 f 的数字。如果最有效的数字 dp 1 是非零的，我们称之为“标准化”数字。

If the exponent has unlimited range any non-zero number can be normalized by “shifting” the significand to the left while adjusting the exponent accordingly. When the exponent is size-limited then some numbers can not be normalized. We call non-normalized numbers that have the minimal exponent “denormals”.

如果指数有无限的范围，任何非零的数字都可以通过“移动”有效值到左边，同时相应地调整指数来归一化。当指数是大小有限的时候，一些数字就不能被标准化。我们称具有最小指数的非规范化数字为“非规范数”。

Note. Floating-point numbers may allow different representations for the same value (for example $12\ 10^{1}$ and $1{:}2\ 10^{2}$). The rep-resentation is however unique when all numbers are either normal-ized or denormal.

注意。浮点数可能允许对同一值有不同的表示(例如 12101 和 1:2102)。然而，当所有数字都是标准化或非标准化时，表示法是唯一的。

2.1 Rounding and Errors
2.1 舍入和错误

Floating point numbers have only a limited size and thus a limited precision. Real numbers must hence be rounded in order to fit into this finite representation. In this section we will discuss the rounding mechanisms that are used in this document and introduce a mechanism to quantify the error they introduce.

浮点数只有有限的大小，因此精度有限。因此，为了适应这种有限的表示，实数必须四舍五入。在本节中，我们将讨论本文中使用的舍入机制，并介绍一种机制来量化它们引入的误差。

The most natural way of rounding is to chose the nearest avail-able floating-point number. This rounded-to-nearest approach is

四舍五入最自然的方法是选择最近的可用浮点数。这种四舍五入到最近的方法是

straightforward except for half-way cases (in the decimal system numbers ending with 5).
除了中间情况(以 5 结尾的十进制数字)以外，其他情况都很简单。

In this paper we will use the following strategies for half-way cases:
在本文中，我们将使用以下策略来处理中间情况:

- up: picks the number closer to +infinity. We will use the nota-tion [x]" when rounding x by this strategy.

选择接近于 + 无穷大的数字。我们用这个策略舍入 x 时使用符号[x]"。

- even: picks the number that is even: [x] . For instance, in the decimal system, 1.5 would round to 2, whereas 0.5 would round to 0. This is the default strategy used by IEEE.

偶数: 选择偶数: [x]。例如，在十进制系统中，1.5 会四舍五入到 2，而 0.5 会四舍五入到 0。这是 IEEE 使用的默认策略。

Whenever the half-way rounding strategy has no importance we will use a star to make this fact explicit: $[x]^?$.
当中间舍入策略不重要时，我们将使用一个星号来明确这个事实: [x] 。

We will use the notation $\tilde{x} = [x]^s_p$ to indicate that the floating-point number ~x contains a normalized significand of size p which has been computed by rounding-to-nearest using strategy s (up, other words, by half a unit in the last place (of the significand). Following established conventions we will use the shorthand ulp to describe these units. A ulp needs to be given with respect to a certain floating-point number. In almost all cases the associated floating-point number is clear from context. In the remaining cases we add the associated number as subscript as so: 1 ulpx.
我们将使用符号 ~ x = [x] sp 来表示浮点数 ~ x 包含一个大小 p 的归一化有效值，这个有效值是通过使用策略 s (上，换句话说，在最后一个位置(有效值的半个单位)四舍五入得到的。按照既定的惯例，我们将使用简写 ulp 来描述这些单位。一个 ulp 需要关于一个特定的浮点数。在几乎所有情况下，相关的浮点数从上下文中都是清晰的。在其他情况下，我们添加相关的数字作为下标，如下: 1 ulpx。

even, or any).
甚至，或任何)。
~x ~ form f b
x x 表格 f e 是
≈x x x

We can quantify j ’s error j as follows: is of
我们可以量化误差 j J 如下: 是的 e

, or, in 或者，

and since f has been rounded to nearest j~x xj 0:5 b
因为 f 已经四舍五入到最接近的 j ~ x xj 0:5 b 加入

During the remainder of this document we will use the tilde-notation to indicate that a number has been rounded-to-nearest. In most cases its error will be 0.5 ulp, but this is not always the case.
在本文的其余部分，我们将使用波浪符号来表示一个数字已经四舍五入到最接近。在大多数情况下，它的误差是 0.5 ulp，但并不总是如此。

2.2 Neighbors and Boundaries
2.2 邻里及边界

For floating-point number types where the value of each encoded number is unique we can define predecessors and successors.
对于浮点数类型，其中每个编码数字的值是唯一的，我们可以定义前辈和后继者。

Let $v = f_v b^{e_v}$ be a strictly positive floating-point number. The predecessor v of v is the next smallest number. If v is mini-mal, then we define 0 to be its predecessor. Similarly v^+ desig-nates the successor of v. For the maximal v we define v^+ to be aries can not be expressed in the given floating-point number type, since its value lies between two adjacent floating-point num-bers. Every floating-point number v has two associated boundaries:
设 v = f v bev 是严格正的浮点数。V 的前身 v 是下一个最小的数。如果 v 是极小的，那么我们将 0 定义为它的前身。同样，v + 指定 v 的继承者。对于最大的 v，我们将 v + 定义为不能用给定的浮点数类型表示的变量，因为它的值位于两个相邻的浮点数之间。每个浮点数 v 都有两个相关的边界:

v+ := v + v
V + : = v + v . That is for this particular v the succe
v 这是为了这个特殊的 v
is at the same distance than the predecessor
是相同的
neighbors of v.
V 的邻居。

The boundary between two adjacent numbers v1 an
两个相邻数字 v1 和 v2 之间的

simply their arithmetic mean: m := $\frac{v_1 + v_2}{2}$. By definition
它们的算术平均值: m: = 。根据定义

m := $\frac{v + v}{2}$ and m+ := $\frac{v + v^+}{2}$. Clearly, any real number w, such
M: = 和 m + : = 显然，任何实数 w，

that $m < w < m^+$, will round to v. Should w be equal to one of the boundaries then we assume that w is rounded to even (the IEEE 754 default). That is, the rounding algorithm will chose the floating-point number with an even significand.
如果 w 等于其中一个边界，那么我们假设 w 是四舍五入到偶数(IEEE 754 的默认值)。也就是说，舍入算法将选择具有偶数显着性的浮点数。

We conclude this section with a definition we will use frequently in the remainder of this document.
我们用一个定义来结束这一部分，我们将在本文余下的部分中经常使用这个定义。

Definition 2.1. A printed representation R of a floating-point num-ber v satisfies the internal identity requirement iff R would convert to v when read again.
定义 2.1。浮点数 v 的打印表示形式 r 满足内部身份要求，如果 r 在再次读取时会转换为 v。

For IEEE 754 double-precision numbers (where half-way cases are rounded to even) this implies $[R]_p = v$. In other words:
对于 IEEE 754 双精度数字(其中中间情况四舍五入到偶数)，这意味着[r] p = v。换句话说:

m V m + when f v is even, and
V m 当 f v 是偶数，并且
m < V < m when f v is odd.
< v < m 当 f v 很奇怪。

2.3 Examples

2.3 Examples 2.3 例子

In this section we show some examples for the previously defined notions. For simplicity we will work in a decimal system. The significand's size p is set to 3, and any exponent is in range 0 to 10. All numbers are either normalized or denormals.
在本节中，我们将展示前面定义的概念的一些示例。为了简单起见，我们将使用十进制系统。有效值的大小 p 设置为 3，任何指数的范围为 0 到 10。所有数字都是标准化的或非标准化的。

In this configuration the extreme values are min := 1 10^0 and max := 999 10^{10}. The smallest normalized number equals 100 10^0. Non-normalized representations like 3 10^4 are not valid. The significand must either have three digits or the expo-nent must be zero.

在这种配置中，极值是 min: = 1100 和 max: = 9991010。最小的标准化数字等于 100100。像 3104 这样的非规范化表示是无效的。有效值必须是三位数，否则展示值必须是零。

Let v := 1234 be a real number that should be stored inside the floating-point number type. Since it contains four digits the number will not fit exactly into the representation and it must be rounded. When rounded to the nearest representation then ~v := $[v]^?_3$:= 123 10^1 is the only possible representation. The rounding error is equal to 4 = 0:4 ulp.

让 v: = 1234 成为应该存储在浮点数类型中的实数。因为它包含四个数字，所以这个数字不能完全符合表示形式，必须四舍五入。当四舍五入到最接近的表示形式时 ~ v: = [v] ？3: = 123101 是唯一可能的表示。舍入误差等于 4 = 0:4 ulp。

Contrary to v the real number w := 1245 lies exactly between to possible representations. Indeed, 124 10^1 and 125 10^1 are both at distance 5. The chosen representation depends on the rounding mechanism. If rounded up then the significand 125 is chosen. If rounded to even then 124 is chosen. For w' = 1235 both rounding mechanisms would have chosen 124 as significand.

与 v 相反，实数 w: = 1245 恰好位于可能的表示之间。实际上，124101 和 125101 都在距离 5 处。选择的表示取决于舍入机制。如果四舍五入，则选择意义 125。如果四舍五入到偶数，则选择 124。对于 w'= 1235，两种舍入机制都选择 124 作为显着性。

The neighbors of w are w := 123 10^1 and w^+ := 125 10^1. Its respective boundaries are therefore m := 123:5 10^1 and m^+ := 124:5 10^1. In this case the neighbors were both at the same distance. This is not true for r := 100 10^3, with neighbors r := 999 10^2 and r^+ := 101 10^3. Clearly r is closer to r than is r^+.

W 的邻居是 w: = 123101 和 w + : = 125101。因此，它们各自的边界是 m: = 123:5101 和 m + : = 124:5101。在这种情况下，邻居都在相同的距离。对于 r: = 100103，邻居 r: = 999102 和 r + : = 101103，这是不正确的。显然 r 比 r + 更接近 r。

For the sake of completeness we now show the boundaries for the extreme values and the smallest normalized number. The number min has its lower (resp. upper) boundary at 0:5 10^1 (resp. 1:5 10^1). For max, the boundaries are 998:5 10^{10} and

为了完整起见，我们现在给出了极值和最小归一化数的边界。数字 min 有其较低的(回复。上)边界在 0:5101。1:5101).对 max 来说，界限是 998:51010 和

999:5 10^{10}.

999:51010.

The boundaries for the smallest normalized number are special: even though its significand is equal to 100 the distance to its lower neighbor (99 10^0) is equal to 1 ulp and not just 0.5 ulp. Therefore its boundaries are 99:5 10^0 and 100:5 10^0.

最小归一化数的边界是特殊的: 即使它的有效值等于 100，到它的下邻居(99100)的距离也等于 1 ulp，而不仅仅是 0.5 ulp。因此它的边界是 99:5100 和 100:5100。

2.4 IEEE 754 Double-Precision

2.4 IEEE 754 双精度

An IEEE 754 double-precision floating-point number, or simply “double”, is defined as a base 2 data type consisting of 64 bits. The first bit is the number sign, followed by 11 bits reserved for the exponent eIEEE , and 52 bits for the significand f IEEE . For the purpose of this paper the sign-bit is irrelevant and we will assume to work

IEEE 754 双精度浮点数，或简称“ double”，定义为由 64 位组成的基 2 数据类型。第一位是数字符号，接着是指数 eIEEE 保留的 11 位，以及 IEEE 有效值的 52 位。为了本文的目的，符号位是不相关的，我们将假设工作

with positive numbers.

有正数。

With the exception of some special cases (which will be dis-cussed shortly) all numbers are normalized which in base 2 implies a starting 1 bit. For space-efficiency this initial bit is not included in the encoded significant. IEEE 754 numbers have hence effec-tively a 53 bit significand where the first 1 bit is hidden (with value hidden = 2^{52}). The encoded exponent e_{IEEE} is an unsigned positive integer which is biased by bias = 1075. Decoding an e_{IEEE} con-sist of subtracting 1075. Combining this information, the value v of

除了一些特殊情况(稍后将讨论)，所有的数字都是标准化的，在基数 2 中意味着起始的 1 位。为了节省空间，这个初始位不包括在编码的有效位中。因此，IEEE 754 数字有效地具有 53 位有效值，其中第一个 1 位是隐藏的(隐藏值 = 252)。编码的指数 eIEEE 是一个无符号的正整数，偏差 = 1075。解码 eIEEE 包含减去 1075。结合这些信息，v 的值为

any normalized double can be computed as f_v := hidden + f_{IEEE} , e_v := e_{IEEE} bias and hence v = f_v 2^{ev} .

任何归一化的双精度可以计算为 f v: = hidden + f IEEE，ev: = eIEEE 偏差，因此 v = f v2ev。

Note. This choice of decoding is not unique. Often the significand is decoded as fraction with a decimal separator after the hidden bit.

注意。这种解码方式并不是唯一的。通常，有效值被解码为小数点，在隐藏位之后有一个小数分隔符。

IEEE 754 reserves some configurations for special values: when Ieee754 为特殊值保留了一些配置:

e_{IEEE} = 0x7FF (its maximum) and f_{IEEE} = 0 then the double is in-finity (or minus infinity, if the bit-sign is set). When e_{IEEE} = 0x7FF and f IEEE 6= 0 then the double represents “NaN” (Not a Number).

eIEEE = 0x7FF (它的最大值)，如果 IEEE = 0，那么双精度就是无穷大(或者减去无穷大，如果设置了位符号)。当 eIEEE = 0x7FF，如果 ieee6 = 0，则双精度表示“ NaN”(不是数字)。

The exponent e_{IEEE} = 0 is reserved for denormals and zero.

指数 eIEEE = 0 是为正规值和零保留的。

Denormals do not have a hidden bit. Their value can be computed as follows: f_{IEEE} $2^{1\ bias}$.

正规值没有隐藏位，它们的值可以计算如下:。

Throughout this paper we will assume that positive and nega-tive infinity, positive and negative zero, as well as NaN have al-ready been handled. Developers should be careful when testing for negative zero, though. Following the IEEE 754 specification 0:0 = +0:0 and 0:0 6< +0:0. One should thus use the sign-bit

在本文中，我们将假设正负无穷、正负零以及 NaN 已经处理好了。不过，开发人员在测试负零时应该非常小心。遵循 IEEE 754 规范 0:0 = + 0:0 和 0:06 < + 0:0。因此，我们应该使用符号位

to efficiently determine a number's sign. In the remainder of this paper a "floating-point number" will designate only a non-special number or a strictly positive denormal. It does not include zero, NaN or infinities.
来有效地确定一个数字的符号。在本文的其余部分，"浮点数"将只表示一个非特殊数字或严格正规数。它不包括 0，NaN 或者无穷大。

Note. Any value representable by doubles (except for NaNs) has a unique representation.
任何可用 double 表示的值(NaNs 除外)都有唯一的表示形式。

Not
e.
注 For any non-special strictly positive IEEE double v with
意 谢 任何非特殊的、严格正的 IEEE 双 v
。 谢 = 0 the upper and lower boundaries m+ are at
and m dis-
f = 0 上下界 m + 和 m 在分歧
IE
EE
IE
EE
电
气
工
程
师
协
会
分
段
第
一
阶
6ve
1
6ve . When f IEEE = 0
1 第 then m is still at v t
版 当 IEEE = 0 时，则 + distance 2 e
m 仍然在距离 2 我 1 但
是
the lower boundary only
satisfies v m2ve 2.2
下边界只满足 v M2ve 2.2

3. Handmade Floating-Point
手工制作的浮点

```
1:  typedef struct diy fp f
结构 diy fp f
2:      uint64 t f;
Uint64 t f;
3:      int e;
国际;
4:  g diy fp;
G diy fp;
```

Figure 1: The diy fp type.
图 1: diy fp 类型。

Grisu and its variants only require fixed-size integers, but these integers are used to emulate floating-point numbers. In general reimplementing a floating-point number type is a non-trivial task, but in our context only few operations with severe limitations are needed. In this section we will present our implementation, diy fp, of such a floating-point number type. As can be seen in Figure 1 it consists of a limited precision integer (of higher precision than the input floating-point number), and one integer exponent. For the sake of simplicity we will use the 64 bit long uint64 t in the accompanying code samples. The text itself is, however, size-agnostic and uses q for the significand's precision.

Grisu 及其变体只需要固定大小的整数，但这些整数被用来模拟浮点数。一般来说，重新实现一个浮点数类型是一个非常重要的任务，但是在我们的上下文中只需要几个有严重限制的操作。在本节中，我们将介绍我们的实现，diy fp，这样一个浮点数类型。如图 1 所示，它由一个有限精度的整数(精度高于输入浮点数)和一个整数指数组成。为了简单起见，我们将在附带的代码示例中使用 64 位长的 uint64 t。然而，文本本身是大小不可知的，并且使用 q 表示重要值的精度。

Definition 3.1 (diy fp). A diy fp x is composed of an unsigned q-bit integer f_x (the significand) and a signed integer e_x (the exponent) of unlimited range. The value of x can be computed as
定义 3.1(diy fp)。Diy fp x 由无符号 q 位整数 fx (有效值)和无限范围的有符号整数 ex (指数)组成。X 的值可以计算为

$x = f_x \cdot 2^{e_x}$.
= f x 2 ex.

The "unlimited" range of diy fp's exponent simplifies proofs. In practice the exponent type must only have a slightly greater range than the input exponent. Input numbers are systematically normalized, and a denormal will therefore require more bits than the original data-type. We furthermore need some extra space to avoid overflows. For IEEE doubles which reserves 11 bits for the exponent, a 32-bit signed integer is by far big enough.

Diyfp 指数的"无限"范围简化了证明。在实践中，指数类型必须只比输入指数有一个稍大的范围。输入数字是系统规范化的，因此非正规数字需要比原始数据类型更多的比特。我们还需要一些额外的空间来避免溢出。对于为指数保留 11 位的 IEEE 双精度数，32 位有符号整数已经足够大了。

3.1 Operations
3.1 运算

Grisu extracts the significand of its diy fps in an early stage and diy fps are only used for two operations: subtraction and multiplication. The implementation of the diy fp type is furthermore simplified by restricting the input and by relaxing the output. For instance, both operations are not required to return normalized results (even if the operands were normalized). Figure 2 shows the C implementation of the two operations.
Grisu 在早期阶段就提取了 diy fps 的有效值，diy fps 只用于两种操作: 减法和乘法。通过限制输入和放松输出，diy fp 类型的实现进一步简化。例如，两个操作都不需要返回规范化的结果(即使操作数被规范化了)。图 2 显示了两个操作的 c 实现。

The operands of the subtraction must have the same exponent and the result of subtracting both significands must fit into the significand-type. Under these conditions the operation clearly does not introduce any imprecision. The result might not be normalized.
减法的操作数必须具有相同的指数，并且减去两个有效值的结果必须符合有效值类型。在这些条件下，运算显然不会引入任何不精确性。结果可能不会被标准化。

The multiplication returns a diy fp ~r containing the rounded result of multiplying the two given diy fps x and y. The result might not be normalized. In order to distinguish this imprecise from the precise multiplication we will use the "rounded" symbol for this operation: ~r := x y.
乘法返回一个 diy fp ~ r，其中包含两个给定的 diy fps x 和 y 的四舍五入结果。结果可能不会被标准化。为了区分这个不精确的乘法和精确的乘法，我们将使用"舍入"符号来进行这个操作:。

[2] The inequality is only needed for $e_{IEEE} = 1$ where the predecessor is a denormal.
这个不等式只在 eIEEE = 1 中需要，前者是正规的。

2ex + ey + q

```
1:  diy fp minus(diy fp x, diy fp y) f
Diy fp 减(diy fp x，diy fp y) f
2:      assert(x.e == y.e && x.f >= y.f);
断言(x.e = = y.e & & & x.f > = y.f) ;
3:      diy fp r = f.f = x.f - y.f, .e = x.eg;
Diy fp r = f.f = x.f-y.f，. e = x.eg;
4:      return r;
返回 r;
5:  g
```

(a) Subtraction
减法

```
1:  diy fp multiply(diy fp x, diy fp y) f
(diy fp x，diy fp y) f
2:      uint64 t a,b,c,d,ac,bc,ad,bd,tmp;
Uint64t a，b，c，d，ac，bc，ad，bd，tmp;
3:      diy fp r; uint64 t M32 = 0xFFFFFFFF;
Diy fp r; uint64t M32 = 0xFFFFFFFF;
4:      a = x.f >> 32; b = x.f & M32;
A = x.f > > 32; b = x.f & M32;
5:      c = y.f >> 32; d = y.f & M32;
C = y.f > > 32; d = y.f & M32;
6:      ac = a*c; bc = b*c; ad = a*d; bd = b*d;
Ac = a * c; bc = b * c; ad = a * d; bd = b * d;
7:      tmp = (bd>>32) + (ad&M32) + (bc&M32);
(bd > 32) + (ad & M32) + (bc & M32) ;
8:      tmp += 1U << 31;  // Round
Tmp + = 1U < < 31;//轮
9:      r.f = ac + (ad>>32) + (bc>>32) +
        (tmp>>32);
R.f = ac + (ad > 32) + (bc > 32) + (tmp >
        32) ;
10:     r.e = x.e + y.e + 64;
R.e = x.e + y.e + 64;
11:     return r;
返回 r;
12: g
```

(b) Multiplication
乘法

Figure 2: diy fp operations
图 2: diy fp 操作

Definition 3.2. Let x and y be two diy fps. Then
定义 3.2 设 x 和 y 是两个 diy fps

$$x \otimes y := \frac{f_x \times f_y}{2^q} \cdot 2^{e_x + e_y + q}$$

The C implementation emulates in a portable way a partial 64-bit multiplication. Since the 64 least significant bits of the multi-plication $f_x \times f_y$ are only used for rounding the procedure does not compute the complete 128-bit result. Note that the rounding can be implemented using a simple addition (line 8).

C 实现以可移植的方式模拟部分 64 位乘法运算。由于乘法 $f_x \times f_y$ 的 64 个最小有效位仅用于四舍五入过程，因此不计算完整的 128 位结果。注意四舍五入可以用一个简单的加法来实现(第 8 行)。

Since the result is rounded to 64 bits a diy fp multiplication introduces some error.

由于结果四舍五入到 64 位，diy fp 乘法会引入一些错误。

Lemma 3.3. Let x and y be two diy fps. Then the error of x ⊗ y is less than or equal to .5 ulp:

引理 3.3。设 x 和 y 是两个 diy fps。那么 x ⊗ y 的误差小于等于 0.5 ulp:

$$|x \otimes y - x \cdot y| \le .5 \text{ ulp}$$

$$|x \otimes y - x \cdot y| \le .5 \text{ ulp}$$

Proof. We can write $x \cdot y$ as $\frac{f_x \cdot f_y}{2^q} \cdot 2^{e_x+e_y+q}$. Furthermore, by definition $x \otimes y = \left[\frac{f_x \cdot f_y}{2^q}\right] \cdot 2^{e_x+e_y+q}$ where the rounding only introduces an error of .5. Since, for $x \otimes y$, 1 ulp = $2^{q+e_x+e_y}$ we can conclude that the error is bounded by $.5 \cdot 2^{q+e_x+e_y}$.

证据，作者：我们可以写 x y 为 $\frac{f_x f_y}{2^q} 2^{e_x + e_y + q}$，如果 $x \otimes y = \left[\frac{f_x f_y}{2^q}\right] 2^{e_x+e_y+q}$ 四舍五入而已会引起 0.5 的错误：因为，对于 x y，1 ulp = $2^{q + e_x + e_y}$ 我们可以得出结论，误差范围是 $.5 \cdot 2^{q+e_x+e_y}$。

翻译: $.5 \cdot 2^{q + e_x + e_y}$。 □

Lemma 3.4. Let x and ỹ be two diy fps, and y a real such that $|y - \tilde{y}| \le u_y$ ulp. In other words ỹ is the approximated diy fp of y and has a maximal error of u_y ulp. Then the errors add up and the result is bounded by $(.5 + u_y)$ ulp.

引理 3.4。设 x 和 y 是两个 diy fps，y 是一个真实的 |y-ỹ| ≤ uy ulp。换句话说，y 是 y 的 diy fp 的近似值，并且有 uy ulp 的最大误差。然后误差加起来，得到的结果是(: 5 + uy) ulp。

$$\forall y; |y - \tilde{y}| \le u_y \text{ ulp} \Rightarrow |x \cdot y - x \otimes \tilde{y}| < (u_y + .5) \text{ ulp}$$

Proof. By Lemma 3.3 we have $|x \cdot \tilde{y} - x \otimes \tilde{y}| \le .5 \cdot 2^{q+e_x+e_{\tilde{y}}}$ and by hypothesis $|y - \tilde{y}| \le u_y$ ulp $= 2^{e_y} u_y$. Clearly $|x \cdot y - x \cdot \tilde{y}| \le u_y 2^{q+e_x+e_y}$ and thus, by summing the inequalities $|x \cdot y - x \otimes \tilde{y}| < (.5 + u_y) 2^{q+e_x+e_y}$.

证明: 引理 3.3 我们得到了 jx ỹ − x ⊗ ỹj ≤ .5 · $2^{q+e_x+e_{\tilde{y}}}$ 及通过假设 jy ~ yj uy ulp = $2^{e_y} u_y$。因此，通过显然是 |x·y − x·ỹ| ≤ $u_y 2^{q+e_x+e_y}$。

求不等式的总和 X ~ yj < (: 5 + uy) □

Lemma 3.5. Let x be a normalized diy fp, ~y be a diy fp,
设 x 是一个标准化 diy fp，~ y 是一个 diy fp,

and y a real such that jy ... . If x ... (the minimal significand) then x ~y undershoots ... ulp compared to by at most

y 是一个实数，这样 jy yj uy 如果 Yj uy x (最小 ... 然后 x ~ y 最多下调 ... 相比之下 x y.

x y.

X y.

jy ~yj uy ulp ^ fx = 2 =) x y x ~y uy/2 ulp

Jy ~ yj uy ulp ^ fx = 2 =) x y X ~ y 2 (ulp)

By definition 根据定义

Proof. 证明。 y = ... Y ... 2ex +ey +q Since ... is either ex ... 要么是前任 ... and hence x ~y ~y. Also xh ... x ~y ... 因此 x ~ y x 还有 Xh ... act or a half-way case we have ... 2q + ex + ey 1 and ... 1 和

我们已经做了一半的工作 ... f y 2 两年后

thus x y uy/2 ulp.

因此 x ~y 呃 y X ~ y 2 。 □

4. Cached Powers

缓存的异能

Similar to White's approach (see the introduction) Grisu needs a cache of precomputed powers-of-ten. The cache must be precomputed using high-precision integer arithmetic. It consists of normalized diy fp values c~k := [ck]? where ck := 10k . Note that, since all ck are normalized 8i; 3 ec~i ec~i-1 4.

与 White 的方法类似(参见介绍)，Grisu 需要一个预计算幂 -10 的缓存。缓存必须使用高精度的整数运算来预先计算。它由标准化 diy fp 值 c ~ k: = [ck] ? 其中 ck: = 10k。注意，因为所有 ck 均归一化为 8i; 3ec ~ i Ec ~ i-14.

The size of the cache (k's range) depends on the used algorithm as well as the input's and diy fp's precision. We will see in Sec-tion 5 how to compute the needed range. For IEEE doubles and 64 bit diy fps a typical cache must hold roughly 635 precomputed values. Without further optimizations the cache thus takes about 8KB of memory. In [Coonen(1984)] Coonen discusses efficient ways to reduce the size of this cache.

缓存的大小(k 的范围)取决于使用的算法以及输入和 diy fp 的精度。我们将在第 5 节看到如何计算所需的范围。对于 IEEE double 和 64 位 diȳ fps，一个典型的缓存必须保存大约 635 个预计算值。如果没有进一步的优化，缓存大约需要 8KB 的内存。在[Coonen (1984)]中，Coonen 讨论了减少缓存大小的有效方法。

The corresponding C procedure has the following signature:
相应的 c 过程有以下签名:

```
diy fp cached power(int k);
Diy fp 缓存电源(int k) ;
```

4.1 k Computation

4.1 k 计算

Grisu (and its evolutions) need to find an integer k such that its cached power $c\sim_k = f_{ck} 2^{ec_k} = 10^{k} ?_q$ satisfies e_{ck} + e for a given e, and . We impose + 3, since otherwise a solution is not always possible. We now show how to compute the sought k.

Grisu (及其演化)需要找到一个整数 k，使其缓存的幂 c ~ k = f ck 2eck = 10k？对于给定的 e，q 满足 eck + e，并且。我们强加 + 3，因为否则的话，解决方案并不总是可能的。我们现在展示如何计算所寻找的 k。

All cached powers are normalized and any f_{ck} thus satisfies 2q 1 f ck < 2q. Hence, 2eck +q 1 c~k < 2eck +q.
所有缓存的幂都被标准化，因此任何 f ck 都满足 2q 1 f ck < 2q。因此，2eck + q 1 c ~ k < 2eck + q。

Suppose that all cached powers are exact (i.e. have no rounding errors). Then k (and its associated $c_{\sim k}$) can be found by computing the smallest power of ten 10^k that verifies 10^k 2 $^{e+q\ 1}$.
假设所有缓存的幂都是精确的(即没有舍入误差)。那么 k (及其相关的 c ~ k)可以通过计算 1010k 的最小幂来验证 10k2 e + q1。

$$k := \log_{10}2^{e+q\ 1} = (e + q\ 1)\ \frac{1}{\log_2 10}\ \text{Log2}$$

$$K: = \log_{10}2\ e + q\ 1 = (E + q\ 1)\ \frac{1}{10}$$

```
1:  #define D 1 LOG2 10 0.30102999566398114 // 1/lg(10) int k comp(int e,
    int alpha, int gamma) f
# 定义 d1 log2100.30102999566398114//1/1g (10) int k comp (int e，int
    alpha，int gamma) f
2:          return ceil((alpha-e+63) * D 1 LOG2 10);
返回细胞((alpha-e + 63) * d 1 log210) ;
3:  g
```

Figure 3: k computation C procedure
图 3: k 计算 c 程序

Figure 3 presents a C implementation (specialized for q = 64) of this computation. In theory the result of the procedure could be wrong since $c_{\sim k}$ is rounded, and the computation itself is approxi-mated (using IEEE floating-point operations). In practice, however, this simple function is sufficient. We have exhaustively tested all exponents in the range -10000 to 10000 and the procedure returns the correct result for all values.
图 3 展示了这个计算的一个 c 实现(专门针对 q = 64)。从理论上讲，过程的结果可能是错误的，因为 c ~ k 是四舍五入的，而计算本身是近似的(使用 IEEE 浮点运算)。然而，在实践中，这个简单的函数就足够了。我们已经彻底测试了范围在 -10000 到 10000 之间的所有指数，该过程返回所有值的正确结果。

5. Grisu
Grisu

In this section we will discuss Grisu, a fast intuitive printing algo-rithm. We will first present its idea, followed by a formal descrip-tion of the algorithm. We then prove its correctness, and finally show a C implementation.
本节我们将讨论 Grisu，一种快速直观的打印算法。我们将首先介绍它的想法，然后是对算法的正式描述。然后我们证明了它的正确性，最后展示了一个 c 语言的实现。

Grisu is very similar to Coonen's algorithm (presented in [Coonen(1980)]). By replacing the extended types (floating-point numbers with higher precision) of the latter algorithm with diy fp types, Coonen's algorithm becomes a special case of Grisu.
Grisu 与 Coonen 的算法非常相似(见[Coonen (1980)])。Coonen 算法将后一种算法的扩展类型(精度更高的浮点数)替换为 diy fp 类型，从而成为 Grisu 算法的一个特例。

5.1 Idea
5.1 Idea 5.1 点子

Printing a floating-point number is difficult because its significant and exponent cannot be processed independently. Dragon4 and its variants therefore combine the two components and work with high-precision rationals instead. We will now show how one can print floating-point numbers without bignums.
打印浮点数是困难的，因为它的有效性和指数不能独立处理。Dragon4 及其变体因此结合了这两个组件，用高精度的有理数来代替。现在我们将展示如何在没有大数的情况下打印浮点数。

Assume, without loss of generality, that a floating-point number v has a negative exponent. Then v can be expressed as $_2$ f $^{v}_{ev}$. The
在不失一般性的前提下，假设一个浮点数 v 有一个负指数。那么 v 可以表示为 2 f vev。The

decimal digits of v can be computed by finding a decimal exponent t such that 1 f v_2 10_{ev} ^{t} < 10.
V 的十进制数字可以通过找到一个十进制指数 t 来计算，使得 1fv210evt < 10。

The first digit is the integer result of this fraction. Subsequent digits are generated by repeatedly taking the remainder of the fraction, multiplying the numerator by 10 and by computing the integer result of the newly obtained fraction.
第一个数字是这个分数的整数结果。后续的数字是通过重复取分数的剩余部分，乘以分子的 10 和通过计算新获得的分数的整数结果来生成的。

The idea behind Grisu is to cache approximated values of $\frac{10^t}{2^{e_t}}$.
Grisu 背后的想法是缓存 10t 的近似值。

The expensive bignum operations disappear and are replaced by operations on fixed-size integer types.
代价高昂的 bignum 操作将消失，取而代之的是对固定大小的整数类型的操作。

A cache for all possible values of t and e_t would be expensive and Grisu therefore simplifies its cache requirement by only storing normalized floating-point approximations of all relevant powers of ten: $c_{\sim k} := 10^k$? $_q$ (where q is the precision of the cached numbers). By construction the digit generation process uses a power of ten with an exponent $e_{c\sim t}$ close to e_v. Even though $e_{c\sim t}$ and e_v do not can-cel each other out anymore, the difference between the two expo-nents will be small and can be easily integrated in the computation of v's digits.
为 t 和 et 的所有可能值建立缓存是很昂贵的，因此 Grisu 通过只存储所有相关幂的标准化浮点近似值来简化其缓存需求: c ~ k: = 10k？Q (其中 q 是缓存数字的精度)。通过构造数字生成过程使用指数 ec ~ t 接近 ev 的 10 次方。虽然 ec ~ t 和 ev 不再互相抵消，但两者之间的差异很小，可以很容易地集成在 v 的数字计算中。

In fact, Grisu does not use the power of ten $c_{\sim k}$ that yields the smallest remaining power of two, but selects the power-of-ten so that the difference lies in a certain range. We will later see that different ranges yield different digit-generation routines and that the smallest difference is not always the most efficient choice.
事实上，Grisu 并没有使用产生 2 的最小剩余功率的 10 c ~ k 的功率，而是选择 -10 的功率，使得差值在一定范围内。我们稍后会看到，不同的范围产生不同的数字生成例程，最小的差异并不总是最有效的选择。

5.2 Algorithm
5.2 算法

In this section we present a formalized version of Grisu. As ex-plained in the previous section, Grisu uses a precomputed cache of powers-of-ten to avoid bignum operations. The cached numbers cancel out most of v's exponent so that only a small exponent re-mains. We have also hinted that Grisu chooses its power-of-ten de-pending on the sought remaining exponent. In the following algo-rithm we parametrize the remaining exponent by the variables and . We impose + 3 and later show interesting choices for these parameters. For the initial discussion we assume := 0 and := 3.
在本节中，我们介绍 Grisu 的一个正式版本。如前一节所述，Grisu 使用一个预先计算的幂 -10 的缓存来避免双数操作。缓存的数字抵消了 v 的大部分指数，所以只剩下一个小指数。我们还暗示 Grisu 根据所寻找的剩余指数选择 -10 的幂。在下面的算法中，我们通过变量和。我们加上 + 3，然后对这些参数进行有趣的选择。对于最初的讨论，我们假设: = 0 和: = 3。

Algorithm Grisu
算法 Grisu

Input: positive floating-point number v of precision p
输入: 精度 p 的正浮点数 v

Preconditions: diy fp's precision q satisfies q p + 2, and the powers-of-ten cache contains precomputed normalized rounded diy fp values $c_{\sim k} = 10^k$? $_q$. We will determine k's necessary range shortly.
前提条件: diy fp 的精度 q 满足 qp + 2，且 -10 的幂缓存包含预计算的归一化 diy fp 值 c ~ k = 10k？Q.我们将很快确定 k 的必要范围。

Output: a string representation in base 10 of V such that $[V]_p$ = v. That is, V would round to v when read again.
Output: 以 v 的 10 为基数的字符串表示，使[v] p = v。也就是说，当再次读取时，v 将四舍五入到 v。

Procedure:
程序:

1. Conversion: determine the normalized diy fp w such that w = v.
Conversion: 确定规范化的 diy fp w，使 w = v̄。

2. Cached power: find the normalized $c_{\sim -k}$ = f c 2ec such that ec + ew + q Ec + ew + q
Cached power: 查找归一化的 c ~-k = f c2ec，使得 e D 艾德

3. Product: let D = f D 2 := w $c_{\sim -k}$.
乘积: 设 d = fd2 校对: = w c ~-k。

4. Output: define V := D 10 . Produce the decimal representation of D followed by the string "e" and the decimal representation of k.
Output: 定义 v: 产生十进制表示法 D 后面跟着字符串" e"和 k 的小数表示形式。

Since the significand of the diy fp is bigger than the one of the input number the conversion of step 1 produces an exact result. By definition diy fps have an infinite exponent range and w's exponent is hence big enough for normalization. Note that

由于 diy fp 的有效值大于输入数的有效值，因此步骤 1 的转换产生一个精确的结果。根据定义，diy fps 有一个无穷大的指数范围，因此 w 的指数大到足以进行标准化。注意

the exponent e_w satisfies e_w e_v (q p). With the exception of

指数 ew 满足 ew ev (q p)

denormals we actually have $e_w = e_v$ (q p).

实际上我们有 ew = ev (q p)。

The sought $\tilde{c}_{-k}$ of step 2 must exist. It is easy to show that 8i; 0 < e~ e~ 4 and since the cache is unbounded the re-

第二步中寻找的 c ~-k 必须存在。显示 8i; 0 < e ~ e ~ 4 是很容易的，因为缓存是无界的，所以重复

c_i c_{i-1} Ci ci-1

quired $\tilde{c}_{-k}$ has to be in the cache. This is the reason for the initial requirement + 3.

必需的 c ~-k 必须在缓存中，这就是最初需求 + 3 的原因。

An infinite cache is of course not necessary. k's range depends only on the input floating-point number type (its exponent range), the diy fp's precision q and the pair , . By way of example we will now show how to compute k_{min} and k_{max} for IEEE doubles, q = 64, and = 0, = 3.

无限的缓存当然是不必要的。K 的范围只取决于输入的浮点数类型(它的指数范围)，diy fp 的精度 q 和对。作为例子，我们现在将展示如何计算 IEEE 双精度数 q = 64，和 = 0，= 3 的 kmin 和 kmax。

Once IEEE doubles have been normalized (which requires them to be stored in a different data-type) the exponent is in range 1126 to +971 (this range includes denormals but not 0). Stored as diy fps the double's exponent decreases by the difference in precision (accounting for the normalization), thus yielding a range of 1137 to +960. Invoking k comp from Section 4.1 with these

一旦 IEEE 双精度数被标准化(这要求它们存储在不同的数据类型中)，指数的范围是 1126 到 + 971(这个范围包括非正规值，但不包括 0)。存储为 diy fps 的双精度指数随着精度的差异而减小(考虑到标准化)，因此产生 1137 到 + 960 的范围。用以下命令调用第 4.1 节中的 k comp

values yields:

价值收益率:

- k_{min} := k comp(960 + 64) = 289, and
  Kmin: = k comp (960 + 64) = 289,
- k_{max} := k comp(1137 + 64) = 342.
  Kmax: = k comp (1137 + 64) = 342.

In step 3 w is multiplied with $\tilde{c}_{-k}$. The goal of this operation is to

在第三步中，w 乘以 c ~-k。这个操作的目标是

~

obtain a diy fp D that has an exponent e_D such that e_D .

获得具有指数 eD 的 diy fp d，使得 eD。

Some configurations make the next step (output) easy. Suppose,

一些配置使得下一步(输出)变得容易，

~

for instance, that eD becomes zero. Then D = f D and the decimal

例如，eD 变为 0，然后 d = f d 和小数点后

digits of D can be computed by printing the significand f_D (a q-bit integer). With an exponent eD 6= 0 the digit-generation becomes slightly more difficult, but since eD's value is bounded by the computation is still straightforward.

D 的数字可以通过打印有效值 f d (q 位整数)来计算。在指数 eD 6 = 0 的情况下，数字的生成变得稍微困难一些，但是由于 eD 的值受到计算的限制，所以生成数字还是很简单的。

Grisu's result is a string containing D's decimal representation followed by the character "e" and k's digit. As such it represents

Grisu 的结果是一个包含 d 的小数表示形式的字符串，后面跟着字符" e"和 k 的数字。因此，它表示

the number V := D ~ k. We claim that V yields v when rounded

数字 v: = d10 我们声称 v 在四舍五入时产生 v

to floating-point number of precision p.

到精度 p 的浮点数。

Theorem 5.1. Grisu's result V satisfies the internal identity re-quirement: $[V]_p = v$.

定理 5.1 Grisu 的结果 v 满足内恒等式要求: [v] p = v。

Proof. In the best case V = v and the proof is trivial. Now, suppose V > v. This can only happen if $\tilde{c}_{-k} > c_{-k}$. We will ig-nore V's parity and simply show the stronger strict inequality < m+. Since c-k is positive we can reformulate our requirement

证明。在最好的情况下 v = v，证明是微不足道的。现在，假设 v > v。只有当 c ~-k > c-k。我们将忽略 v 的奇偶性，并简单地证明更强的严格不等式 < m + 。因为 c-k 是正的，我们可以重新制定我们的要求

as (V v) c-k < m+ v c-k .

(v) c-k < m + V c-k.

Using the equalities v = w, V = w $\tilde{c}$ 10 k = 1; and m

利用等式 v = w, V = w C 10 k = 1; 还有 m

this expands to w $\tilde{c}_{-k}$ w c-k < 2 c-k .

这个扩展到 w c ~- w c-k < 2 C-k W c-k < 2 k.

Since, by hypothesis, ev ew + 2 it is hence sufficient to show that

因为，根据假设，Ew + 2 因此足以表明

w c~-k w c-k < 2 ew +1 c-k:

W c ~-k W c-k < 2 Ew + 1 C-k:

We have two cases:

我们有两个案子:

- $f_c > 2^{q-1}$. By hypothesis $\tilde{c}_{-k}$'s error is bounded by .5 ulp and f c = 2q 1. Since the next lower diy fp is only at distance 2ec 1 and c-k is rounded to nearest, c-k 's error is bounded by

F c > 2q1.根据假设 c ~-k 的误差以 0.5 ulp 和 f c = 2q 1 为界。由于下一个较低的 diy fp 只在距离 2ec 1 处，c-k 四舍五入到最近，c-k 的误差由

2(q 1)+ec 2(q 1) + ec thus 因此 It suffices to show that w $\tilde{c}_{-k}$ w c-k < 2 ew + q+ ec 这足以证明 w c ~-k w c-k < 2 ew + q + ec

strictly less than 2 which is guaranteed by Lemma 3.4.

严格小于 2 由引理 3.4 保证。

1 ulp. Clearly c-k 2q 1 2ec 2(q 1)+ec for any

显然 c-k 2q 1 2c 2(q 1) + ec

q 2. The inequality w $\tilde{c}_{-k}$ w c-k < 2(ew +1)+(q

W c-k < 2(ew +1)+(q

不等式 w c ~-k

1)+ec

2(ew + 1) +

(q1) + ec

is (due to the smaller error of $c\sim_{-k}$) guaranteed by Lemma 3.4. We have proved the theorem for V v. The remaining case < v can only happen when c~-k < c-k . Now suppose:

是由 Lemma 3.4 保证的(由于 c ~-k 的较小误差)。我们证明了 v v 的定理。剩下的情形 < v 只能发生在 c ~-k < c-k 时。现在假设:

- $f_v > 2^{p\ 1}$ and therefore v m = $2^{ev\ 1}$. The proof for this case is similar to the previous cases.
- f v > 2p1，因此 v m = 2ev1。这个案例的证据与之前的案例相似。

• f v = 2p 1 and = 2ev 2. Since f v is
therefore v even we
F v = 2p1，因此 v m = 2ev 2 既然 f v 是偶数
V. Using similar steps as
only need to show m before it
只需要证明 m 使用与以前相似的步骤
‡) ec w c~-k w c-k
‡(1) which is
suffices to show that Ec w c ~-k w c-k 也就
2ew 是
足以证明 q
guaranteed by Lemma 3.5.
引理 3.5 保证。

□

5.3 C Implementation
5.3 c 的实施

We can now present a C implementation of Grisu. This implementation uses 64 bit integers, but a proof of concept version, using only 55 bits, can be found on the author's homepage.
我们现在可以展示 Grisu 的 c 实现。这个实现使用了 64 位整数，但是一个概念版本的证明，只使用了 55 位，可以在作者的主页上找到。

```
1:  #define TEN7 10000000
# 定义 ten710000000
2:  void cut(diy fp D, uint32 t parts[3]) f
Void cut (diy fp d，uint32t parts [3]) f
3:      parts[2] = (D.f % (TEN7 >> D.e)) << D.e;
部分[2] = (D.f% (TEN7 > D.e)) < D.e;
4:      uint64 t tmp = D.f / (TEN7 >> D.e);
Uint64 t tmp = D.f/(TEN7 > > D.e) ;
5:      parts[1] = tmp % TEN7;
部分[1] = tmp% TEN7;
6:      parts[0] = tmp / TEN7;
部分[0] = tmp/TEN7;
7:  g
8:  void grisu(double v, char* buffer) f
Void grisu (double v，char * buffer)
9:      diy fp w; uint32 t ps[3];
; uint32 t ps [3] ;
10:     int q = 64, alpha = 0, gamma = 3;
Int q = 64，α = 0，γ = 3;
11:     w = normalize diy fp(double2diy fp(v));
W = 规范化 diy fp (double2diy fp (v)) ;
12:     int mk = k comp(w.e + q, alpha, gamma);
Int mk = k comp (w.e + q，alpha，gamma) ;
13:     diy fp c mk = cached power(mk);
Diy fp c mk = 缓存功率(mk) ;
14:     diy fp D = multiply(w, c mk);
Diy fp d = 乘(w，c mk) ;
15:     cut(D, ps);
切割(d，ps) ;
16:     sprintf(buffer, "%u%07u%07ue%d",
Sprintf (buffer,"% u% 07u% 07ue% d",
17:                 ps[0], ps[1], ps[2], -mk);
ps [0]，ps [1]，ps [2] ,-mk) ;
18: g
```

Figure 4: C implementation of Grisu with ； = 0;3.
图 4: Grisu 的 c 实现; = 0; 3。

In Figure 4, line 8 we show the core grisu procedure specialized for := 0 and := 3. It accepts a non-special positive dou-ble and fills the given buffer with its decimal representation. Up to line 15 the code is a direct translation from the pseudo-algorithm to C. In this line starts step 4 (output).

在图 4 中，第 8 行显示了专门针对: = 0 和: = 3 的核心 grisu 程序。它接受一个非特殊的正双精度数，并用它的小数表示形式填充给定的缓冲区。直到第 15 行，代码是从伪算法直接翻译成 c 的。这一行从第 4 步开始(输出)。

By construction D.e is in the range 0 - 3. With a sufficiently big data-type one could simply shift D.f, the significand, and dump its decimal digits into the given buffer. Lacking such a type (we assume that uint64 t is the biggest native type), Grisu cuts D into three smaller parts (stored in the array ps) such that the concatenation of their decimal digits gives D's decimal digits (line 15).

通过建筑 D.e 在 0-3 的范围内。有了足够大的数据类型，人们可以简单地移动 D.f，有效值，并转储其小数位到给定的缓冲区。由于缺少这样一个类型(我们假设 uint64t 是最大的本机类型)，Grisu 将 d 分成三个较小的部分(存储在数组 ps 中)，这样它们的小数位串联就得到了 d 的小数位(第 15 行)。

Note that 2^{67} = 147573952589676412928 has 21 digit. Three 7-digit integers will therefore always be sufficient to hold all deci-mal digits of D.
注意 267 = 147573952589676412928 有 21 位。因此，三个 7 位整数总是足以容纳 d 的所有小数位。

In line 16 ps' digits and the decimal exponent are dumped into the buffer. For simplicity we have used the stdlib's sprintf procedure. A specialized procedure would be significantly faster, but would unnecessarily complicate the code.
在第 16ps 行中，数字和十进制指数被转储到缓冲区中。为了简单起见，我们使用了 stdlib 的 sprintf 过程。一个专门的过程会快得多，但是会不必要地使代码复杂化。

Another benefit of cutting D's significand into smaller pieces is that the used data-type (uint32 t) can be processed much more efficiently. In our specialized printing procedure (replacing the call to sprintf) we have noticed tremendous speed improvements due to this choice. Indeed, current processors are much faster when dividing uint32 ts than uint64 ts. Furthermore the digits for each part can be computed independently which removes pipeline stalls.
将 d 的重要性切割成小块的另一个好处是，使用的数据类型(uint32t)可以更有效地处理。在我们的专门打印过程中(取代了对 sprintf 的调用)，我们注意到由于这种选择，速度得到了巨大的提高。事实上，当前的处理器在除 uint32 时比除 uint64 时要快得多。此外，每个部分的数字都可以独立计算，这样就可以去除流水线上的停顿。

5.4 Interesting target exponents
5.4 有趣的目标指数

We will now discuss some interesting choices for and . The most obvious choice ； := 0;3 has already been presented in the previous section. Its digit-generation technique (cutting D into three parts of 7 digits each) can be easily extended to work for target exponents in the range := 0 to := 9. One simply has to cut D into three uint32 ts of 9 decimal digits each. As a consequence
现在我们将讨论一些有趣的选择。最明显的选择: : = 0; 3 已经在前面的章节中介绍过了。它的数字生成技术(将 d 分成三部分，每部分 7 个数字)可以很容易地扩展到目标指数的范围: = 0 到: = 9。一个简单的方法是将 d 分割成三个单位 32，每个单位 9 个小数位。因此
D's decimal representation might need up to 27 digits.
D 的小数表示可能最多需要 27 位数。

On the one hand the bigger increases the output size (without increasing its precision), but on the other hand the extended range provides more room to find a suitable cached power-of-ten. The increased clearance can, for instance, be used to reduce the number of cached powers-of-ten. It is possible to remove two thirds of the cache while still being able to find the required c~-k of step 2.

一方面越大，输出大小就越大(但不会提高其精度)，但另一方面，扩展的范围提供了更多的空间，可以找到一个合适的缓存功率 -10。例如，增加的间隙可以用来减少缓存的幂 -10 的次数。可以删除三分之二的缓存，同时仍然能够找到步骤 2 所需的 c ~-k。

Indeed, two cached powers-of-ten c~ and c~ will always satisfy
实际上，两个缓存的 -10 c ~ 和 c ~ 的幂总是可以满足的

i i+3
I i + 3

$e_{c\sim}$ $e_{c\sim}$ 10.
Ec ~ ec ~ 10.
i+3 i
I + 3 i

Another technique uses the increased liberty to choose the “best” cached power-of-ten among all that satisfy the requirement. For example, a heuristic could prefer exact cached numbers over in-exact ones. Without additional changes to the core algorithm there is however little benefit in using such a heuristic.

另一种技术使用增加的自由度，在满足需求的所有方法中选择“最佳”缓存的 -10 次方。例如，一个启发式算法可以选择精确的缓存数字而不是精确的。如果不对核心算法进行额外的修改，那么使用这种启发式算法就没有什么好处。

Despite the added optimization opportunities the basic digit-generation technique still stays the same, though. We therefore move on to the next interesting exponent range: ; := 63; 60.

尽管增加了优化的机会，基本的数字生成技术仍然保持不变。因此，我们进入下一个有趣的指数范围:。

```
1:   int digit gen no div(diy fp D, char* buffer) f
Int digit gen no div (diy fp d，char * buffer) f
2:       int i = 0, q = 64; diy fp one;
Int i = 0，q = 64; diy fp one;
3:       one.f = ((uint64 t) 1) << -D.e; one.e = D.e; buffer[i++] = '0' + (D.f >>
         -one.e); //division
1. f = ((uint64t)1) <-D.e; 1. e = D.e; buffer [ i + + ] ='0'+ (D.f >-one. e) ;//除
         法
4:       uint64 t f = D.f & (one.f - 1); // modulo
Uint64 t f = D.f & (one. f-1) ;//模
5:       buffer[i++] = '.';
缓冲器[ i + + ] ='
6:       while (-one.e > q - 5) f
而(- 1 > q-5) f
7:           uint64 t tmp = (f << 2) & (one.f - 1);
Uint64tmp = (f < 2) & (1. f-1) ;
8:           int d = f >> (-one.e - 3);
Int d = f > > (- 1. e-3) ;
9:           d &= 6;-f = f-+ tmp; d += f >> (-one.e - 1);
D & = 6; f = f + tmp; d + = f > (- 1. e-1)-;
10:          buffer[i++] = '0' + d;-
缓冲液[ i + + ] ='0'+ d;
11:          one.e++; one.f >>= 1;
一个 + + ; 一个 f > > = 1;
12:          f &= one.f - 1;
F & = 1                 -
13:      g
14:      while (i < 19) f
而(i < 19) f
15:          f *= 10;
F * = 10;
16:          buffer[i++] = '0' + (f >> -one.e);
缓冲区[ i + + ] ='0'+ (f >-one. e) ;
17:          f &= one.f - 1;
F & = 1
18:      g
19:      return i;
返回 i;
20:  g
```

Figure 5: Digit generation for = 63 and = 60.
图 5: = 63 和 = 60 的数字生成。

The beauty of this exponent range lies in the fact that the normalized diy fp one, representing the number 1, is composed of $f_{one} = 2^{63}$ and $e_{one} = 63$. Usually expensive operations, such as division and modulo, can be implemented very efficiently for this significand. The C implementation in Figure 5 dispenses of division and modulo operators entirely and uses only inexpensive operations such as shifts and additions. With the exception of the exponent (which has at most 3 digits) Grisu manages to produce a decimal representation of an input IEEE floating-point number without any division at all. The price for this feat is the complicated code of Figure 5. Its complexity is necessary to avoid overflows. For simplicity we will start by describing the algorithm without caring for data-type sizes.

这个指数范围的美妙之处在于，表示数字 1 的规范化 diy fp one 由 fone = 263 和 eone = 63 组成。通常昂贵的操作，如除法和模，可以非常有效地实现这一重要性。图 5 中的 c 实现完全省去了除法和模运算符，只使用廉价的运算，如移位和加法。除了指数(最多有 3 位数字)之外，Grisu 设法生成输入 IEEE 浮点数的十进制表示形式，而根本没有任何除法。这一壮举的代价是图 5 中复杂的代码。它的复杂性对于避免溢出是必要的。为了简单起见，我们将从描述算法开始，而不考虑数据类型的大小。

Algorithm digit-gen-no-div
算法数字-gen-no-div

Input: a diy fp D with exponent 63 e_D 60.
Input: 具有指数 63ed60 的 diy fp d。

Output: a decimal representation of D.
输出: d 的小数表示形式。

Procedure:
程序:

1. One: determine the diy fp one with $f_{one} = 2^{e_D}$ and $e_{one} = e_D$.
1.1: 确定 diy fp One，f1 = 2 eD，eone = eD。

2. Digit0: compute $d_0 := {}_{one}{}^{D}$ and D_1 := D mod one
计算 d0: = oneD 和 D1: = d mod one

2b. Ten: If d_0 10 emit the digit ”1” followed by the character representing d_0 10. Otherwise emit the character represent-ing the digit d_0.
2b.10: 如果 d010 发出数字“1”，后面跟着代表 d010 的字符。否则发出代表数字 d0 的字符。

3. Comma: emit ”.”, the decimal separator.
逗号: 发出“ .”，小数点分隔符。

4. Digits: generate and emit digits d_i as follows
Digits: 生成和发出数字 di 如下

- d_i := 10 D
- Di: = 10 Di
  one
  一个

- emit the character representing the digit d_i
  发出表示数字 di 的字符
- $D_{i+1} := 10\ D_i \bmod one$

  Di + 1: = 10 Di mod one

5. Stop: stop at the smallest positive integer n such that $D_n = 0$.
Stop: 在最小的正整数 n 处停止，使得 Dn = 0。

We will now show that the algorithm computes a decimal rep-resentation of D. Let R_i be the number that is obtained by reading the emitted characters up to and including d_i.

现在，我们将展示该算法计算 d 的十进制表示形式。让 Ri 成为通过读取发出的字符直到并包括 di 而获得的数字。

In step 2b d_0 is printed. Since d_0 consists of at most 4 binary digits it cannot exceed 15, and therefore (after this step) R_0 evalu-ates to d_0. We declare the following invariant for the loop of step 4: $D = R_i + {}^{D}10^{i+1}{}_{-i}$. Clearly the invariant holds for i = 0, and the invari-ant is still valid after the execution of the loop-body. We can hence conclude that $D = R_{n-1}$.

在步骤 2b 中打印 d0。由于 d0 最多由 4 个二进制数组成，它不能超过 15，因此(在这一步之后) R0 的计算结果为 d0。我们声明步骤 4 的循环的以下不变量: d = Ri + D10i + 1-i。显然，这个不变量适用于 i = 0，并且在循环体执行之后，这个不变量仍然有效。因此我们可以得出这样的结论: d = Rn-1。

The C implementation of this algorithm is more involved as it has to deal with overflows. When multiplying D_i by ten (step 4) the result might not fit into a uint64 t. The code has therefore been split into two parts, one that deals with potential overflows, and an-other where the product safely fits in the data-type. The test in line 7 checks if the result fits into a uint64 t. Indeed, D_i < one for any 1 i n and with 4 additional bits the multiplication will not overflow. The easy, fast case is then handled in line 15. This loop corresponds to the loop of step 4. Note that digit gen no div produces at most 18 digits. We will discuss this choice shortly.

这个算法的 c 实现更加复杂，因为它必须处理溢出。当 Di 乘以 10(步骤 4)时，结果可能不适合 uint64t。因此，代码被分成两部分，一部分处理潜在的溢出，另一部分处理产品安全地适合数据类型的情况。第 7 行的测试检查结果是否符合 uint64 t。实际上，Di < 1 表示任何 1 i n，如果有 4 个额外的位，乘法就不会溢出。简单快速的情况在第 15 行处理。这个循环对应于步骤 4 的循环。注意，数字 gen no div 最多产生 18 个数字。我们将很快讨论这个选择。

Should 10 D_i not fit into a uint64 t the more complicated loop of line 7 is used. As to avoid overflows the code combines the multiplication by ten with the division/modulo by one. By construction $e_D = e_{one}$ and $f_{one} = 2^{e_{one}}$. The division by one can

如果 10di 不适合 uint64t，则使用更复杂的第 7 行循环。为了避免溢出，代码将 10 的乘法和 1 的除法/模相结合。通过构造 eD = eone，f one = 2 eone。除以 1 可以

thus be written as 因此可以写成

$$\frac{D_i}{e^{one}} = \frac{D_i \cdot 10}{f_{one}} = \frac{4\,f\,D_i + f\,D_i}{2^{e^{one}}}$$

从这个方程式

. From this equation

it is then only a small step to the implementation in Figure 5.
这只是图 5 中实现的一小步。

In order to escape from this slow case digit gen no div introduces an implicit common denominator. In line 12 one is divided by this denominator. This way one's exponent decreases at each iteration and after at most 5 iterations the procedure switches to the lightweight loop.

为了避免这种慢速的大小写数字，没有 div 引入一个隐式的公分母。在第 12 行，1 除以这个分母。这样，每次迭代的指数都会减小，最多 5 次迭代之后，过程就会切换到轻量级循环。

Our implementation takes some shortcuts compared to the de-scribed algorithm: it skips step 2b and prints at most 18 digits. The first shortcut is only possible when Grisu uses the smallest cached power-of-ten that satisfies the range-requirement, since in that case d_0 < 10. The 18 digit shortcut relies on the high precision (64 bits) used in the implementation. An implementation featuring only two extra-bits (55 bits for IEEE doubles) is forced to continue iterating until $D_i = 0$. Since each iteration clears only one bit one could end up with 55 decimal digits.

与上述算法相比，我们的实现采取了一些捷径: 跳过步骤 2b，最多打印 18 位数字。第一个快捷方式只有在 Grisu 使用满足范围要求的最小缓存幂 -10 时才可能实现，因为在这种情况下 d0 < 10。18 位的快捷方式依赖于实现中使用的高精度(64 位)。一个只有两个额外位的实现(IEEE 双精度版本为 55 位)被迫继续迭代，直到 Di = 0。因为每次迭代只清除一个比特，所以最终的结果可能是 55 个小数位。

```
1:   int digit gen mix(diy fp D, char* buffer) f
Int 数字 gen mix (diy fp d，char * buffer) f
2:       diy fp one;
Diy fp 一;
3:       one.f = ((uint64 t)1)<<-D.e; one.e = D.e;
1. f = ((uint64t)1) < <-D.e;
4:       uint32 t part1 = D.f >> -one.e;
Uint32t 部分 1 = D.f > >-一个;
5:       uint64 t f = D.f & (one.f - 1);
Uint64 t f = D.f & (one. f-1) ;
6:       int i = sprintf(buffer, "%u", part1);
Int i = sprintf (缓冲区，“% u”，第 1 部分) ;
7:       buffer[i++] = '.';
缓冲器[ i + + ] ='
8:       while (i < 19) f
而(i < 19) f
9:           f *= 10;
F * = 10;
10:          buffer[i++] = '0' + (f >> -one.e);
缓冲区[ i + + ] ='0'+ (f >-one. e) ;
11:          f &= one.f - 1;
F & = 1
12:      g
13:      return i;
返回 i;
14:  g
```

Figure 6: Digit generation for = 59 and = 32.
图 6: = 59 和 = 32 的数字生成。

Finally one can mix both digit-generation techniques. The pro-cedure in Figure 6 can be used for ; := 59; 32. It combines the advantages of the previous approaches. It cuts the input num-ber D into two parts: one that fits into a 32 bit integer and one part that can be processed without divisions. By construction it does not need to worry about overflows and therefore features relatively

最后，人们可以混合使用这两种数字生成技术。图 6 中的程序可用于: : = 59; 32。它结合了以前方法的优点。它将输入数字 d 分成两部分: 一部分适合 32 位整数，另一部分不需要除法即可处理。通过构造，它不需要担心溢出，因此具有相对的特性

straightforward code. Among the presented digit-generation proce-dures it also accepts the greatest range of exponents. Compared to the configuration ; = 0;3 this version needs only a ninth of the cached powers. For completeness sake we now present its pseudo-algorithm:
简单明了的代码。在所呈现的数字生成过程中，它也接受最大范围的指数。与配置; = 0; 3 相比，这个版本只需要九分之一的缓存功率。为了完整起见，我们现在给出它的伪算法:

Algorithm digit-gen-mix
算法数字-代混合

Input: a diy fp D with exponent 59 e_D 32.
Input: 具有指数 59ed32 的 diy fp d。

Output: a decimal representation of D.
输出: d 的小数表示形式。

Procedure:
程序:

1. One: determine the diy fp one with $f_{one} = 2^{e_D}$ and $e_{one} = e_D$.
1.1: 确定 diy fp One，f1 = 2 eD，eone = eD。

2. Parts: compute part1 := $_{one}^{D}$ and part2 := D mod one
Part: compute part 1: = oneD and part 2: = d mod one

3. Integral: print the digits of part1.
Integral: 打印第 1 部分的数字。

4. Comma: emit ".", the decimal comma separator.
逗号: 发出" ."，小数逗号分隔符。

5. Fractional: let D_0 := part2. Generate and emit digits d_i (for i 0) as follows
小数: 让 D0: = part2。生成并发出数字 di (i 0)，如下所示

- d_i := $\frac{10\ D_i}{}$
- Di: = 10 Di $_{\text{一个}}^{\text{one}}$
- emit the character representing the digit d_i
  发出表示数字 di 的字符
- D_{i+1} := 10 D_i mod one

Di + 1: = 10 Di mod one

6. Stop: stop at the smallest positive integer n such that $D_n = 0$.
Stop: 在最小的正整数 n 处停止，使得 Dn = 0。

The C implementation takes the same shortcut as for the no-division technique: it stops after 18 digits. The reason is the same as before.
C 实现采用了与无除法相同的快捷方式: 在 18 位后停止。原因和之前一样。

Note that the mixed approach can be easily extended to accept exponents in the range ; := 59;0 by cutting the input number into four (instead of two) parts. This last version would require 64 bit divisions and would therefore execute slower than the shown one. However it would require the least amount of cached powers-of-ten.
注意，混合方法可以很容易地扩展为接受范围内的指数; := 59; 0，方法是将输入数字分成四个(而不是两个)部分。最后一个版本需要 64 位的除法，因此执行速度比图中的慢。然而，它需要最少的缓存功率 -10。

We will base future evolutions of Grisu on digit-get-mix with ; = 59; 32. This configuration contains the core ideas of all presented techniques without the obfuscating overflow-handling operations. All improvements could be easily adapted to other ranges.
我们将把 Grisu 的未来进化建立在数字获取混合; = 59; 32。这个配置包含所有呈现的技术的核心思想，没有混淆溢出处理操作。所有的改进都可以很容易地适应其他范围。

6. Evolutions
进化

In this section we will present evolutions of Grisu: Grisu2 and Grisu3. Both algorithms are designed to produce shorter outputs. Grisu may be fast, but its output is clearly suboptimal. For example, the number 1.0 is printed as 10000000000000000000e-19. The optimal solution (printed by Grisu2 and Grisu3) avoids the trailing '0' digits.
在这一节中，我们将介绍 Grisu 的进化: Grisu2 和 Grisu3。两种算法都被设计为产生较短的输出。Grisu 可能很快，但它的输出显然是次优的。例如，数字 1.0 打印为 10000000000000000000000000e-19。最佳解决方案(由 Grisu2 和 Grisu3 打印)避免了尾随的"0"数字。

Grisu2 and Grisu3 use the extra capacity of the used integer type to shorten the produced output. That is, if the diy fp integer type has more than two extra bits, then these bits can be used to create shorter results. The more bits are available the more often the produced result will be optimal. For 64-bit integers and IEEE doubles (with a 53-bit significand) more than 99% of all input-numbers can be converted to their shortest decimal representation.
Grisu2 和 Grisu3 使用所使用的整数类型的额外容量来缩短生成的输出。也就是说，如果 diy fp 整数类型有超过两个额外的位，那么这些位可以用来创建更短的结果。可用的比特越多，产生的结果就越优化。对于 64 位整数和 IEEE 双精度数(53 位有效值)，超过 99% 的输入数可以转换为它们的最短小数表示形式。

Grisu2 and Grisu3 differ in the way they handle the non-optimal solutions. Grisu2 simply generates the best solution that is possible with the given integer type, whereas Grisu3 rejects numbers for which it cannot prove that the computed solution is optimal.
Grisu2 和 Grisu3 在处理非最优解的方式上有所不同。Grisu2 只是简单地生成给定整数类型所能得到的最佳解，而 Grisu3 拒绝那些它无法证明计算出的解是最优的数。

For demonstration purposes we include rounding as an optimal-ity requirement for Grisu3. It is simple to adapt Grisu2 so it rounds its outputs, too.
为了演示目的，我们将舍入作为 Grisu3 的最佳性能要求。适应 Grisu2 很简单，所以它的输出也是四舍五入的。

Finally we render Grisu2 and Grisu3 more flexible compared to Grisu. There are different ways to format a floating-point num-ber. For instance the number 1.23 could be formatted as 1.23, 123e-2, or 0.123e1. For genericity it is best to leave the for-matting to a specialized procedure. Contrary to Grisu, Grisu2 and
最后，我们使 Grisu2 和 Grisu3 比 Grisu 更灵活。格式化浮点数有不同的方法。例如，数字 1.23 可以被格式化为 1.23,123e-2，或者 0.123e1。为了泛型，最好将 For-matting 留给一个专门的过程。与 Grisu 相反，Grisu2 和

Grisu3 do not produce a complete decimal representation but simply produce its digits (“123”) and the corresponding exponent (-2). The formatting procedure then needs to combine this data to produce a representation in the required format.
Grisu3 不产生一个完整的小数表示，而是简单地产生它的数字(“123”)和相应的指数(- 2)。格式化过程然后需要结合这些数据产生一个所需格式的表示。

6.1 Idea
6.1 点子

We will first present the general idea of Grisu2 and Grisu3, and then discuss each algorithm separately. Both algorithms try to produce optimal output (with respect to shortness) for a given input-number v.
我们将首先介绍 Grisu2 和 Grisu3 的一般概念，然后分别讨论每一种算法。两种算法都试图为给定的输入数 v 产生最佳输出(关于短度)。

The optimal output of input v represents a number V with the smallest leading length that still satisfies the internal identity requirement for v.[3] The “leading length” of V is its digit length once it has been stripped of any unnecessary leading and trailing ‘0’ digits.
输入 v 的最佳输出表示一个数字 v，它的最小前导长度仍然满足 v. 3 的内部恒等式要求。 v 的“前导长度”是指一旦去掉了任何不必要的前导和尾随的“0”位数，它的数字长度。

Definition 6.1. Let v be a positive real number and n, l and s be integers, such that l 1, $10^{l-1} s < 10^{l}$, v = s 10^{n-l} and l as small as possible. Then the l decimal digits of s are v’s leading digits and l is v’s leading length.
定义 6.1。设 v 是一个正实数，n，l 和 s 是整数，使得 l1,101s < 10l，v = s10nl 和 l 尽可能小。那么 s 的小数位 l 是 v 的前导数字，l 是 v 的前导长度。

In the following we demonstrate how the optimal V can be computed. Let v be a floating-point number with precision p and let m^-, m^+ be its boundaries (as described in Section 2.2). Assume, without loss of generality, that its significand f v is even. The optimal output consist of a number V such that m V m+ and such
下面我们将演示如何计算最优 v。设 v 为精度为 p 的浮点数，m，m + 为其边界(如第 2.2 节所述)。假设，在不损失一般性的情况下，其有效的 f v 是偶数。最佳输出由一个数字 v 组成，即 m v m + 等等 that V’s significant length is minimal.
V 的有效长度是最小的。

The current state of art [Burger and Dybvig(1996)] computes V by generating the input number v’s digits from left to right and by stopping once the produced decimal representation would evaluate to v when read again. Basically the algorithm tests for two termination conditions tc1 and tc2 after each generated digit d_i:
目前的技术[Burger and Dybvig (1996)]通过从左到右生成输入数 v 的数字，并且在再次读取时停止一次生成的十进制表示将计算为 v 来计算 v。基本上，算法在每个生成的数字 di 后测试两个终止条件 tc1 和 tc2:

- tc1 is true when the produced number (consisting of digits d_0: : :d_i) is greater than m , and
  当生成的数字(由数字 d0: : : di 组成)大于 m 时，tc1 为 true，并且
- tc2 is true when the rounded up number (consisting of digits d_0: : :(d_i + 1)) is less than m^+.
  当四舍五入的数字(由数字 d0: : (di + 1)组成)小于 m + 时，tc2 为真。

In the first case a rounded down number (of v) would be re-turned, whereas in the second case the result would be rounded up.
在第一种情况下，将返回一个四舍五入的数字(v) ，而在第二种情况下，结果将四舍五入。

Since these two tests are slow and cumbersome to write we have developed another technique that needs only one. The basic approach is similar: one produces the decimal digits from left to right, but instead of using v to compute the digits the faster approach generates the digits of m^+. By construction any rounded up number of the generated digits will be greater than m^+ and thus not satisfy the internal identity requirement anymore. Therefore the second termination condition will always fail and can hence be discarded.
由于这两个测试写起来既慢又麻烦，我们开发了另一种只需要一个测试的技术。基本方法是相似的: 从左到右生成十进制数字，但是更快的方法不是用 v 来计算数字，而是生成 m + 的数字。通过构造，任何生成的四舍五入的数字都将大于 m + ，因此不再满足内部恒等式的要求。因此，第二个终止条件将总是失败，因此可以被丢弃。

We can show that this technique generates the shortest possible number.
我们可以证明这种技术可以产生尽可能短的数。

Theorem 6.2. Let x and y two real numbers, such that x y. Let k be the greatest integer, such that y mod 10^k y x. Then V := $_{10}y_k$ 10^k satisfies x V y. Furthermore V’s leading length l is the smallest of all possible numbers in this interval: any number V’ such that x V’ y has a leading length l’ l.
定理 6.2。设 x 和 y 是两个实数，使得 x y 是最大的整数，使得 y mod 为 10k y x。那么 v: = 10yk 10k 满足 x v y。此外 v 的前导长度 l 是这个区间内所有可能数中最小的: 任意 v’使得 x v’y 有一个前导长度 l’l。

Proof. We start by showing we know that V y. Also Y mod 10 Y x V x.
证明。我们首先展示 x V y : 10k 我们知道 还有 y mod 10 y x since y = V + y mod 10k 因为 y = v + y mod 10k and therefore 因此

X.

For the sake of contradiction assume that there exists a V’ with leading length l’, such that x V’ y and l’ < l.
为了自相矛盾，假设存在一个前导长度为 l’的 v’，比如 x v’y 和 l’< l。

[3] The shortest output may not be unique. There are many numbers that verify the internal identity requirement for a given floating-point number, and several of them might have the same leading length.
最短的输出可能不是唯一的。有很多数字可以验证给定浮点数的内部身份要求，其中几个可能有相同的前导长度。

The number V’ has a leading length of l’ and by definition
根据定义，数字 v’的前导长度为 l’
因此是 s’ n’因此是 10l’ S’ < 10l’ 和
l
’
there exists n’
存在 n
V’ = s’ 10
V = s’10

There are three cases to consider:
有三种情况需要考虑:

1. s’ > 10 n y : impossible since this implies V > y.
第一章 10 n : 不可能，因为这意味着 v 是的。

2. s’ = 10 n y : contradiction, since this implies V = V’.
2. s 10 n : 矛盾，因为这意味着 v = v’。

3. s’ < 10 n y : we first prove the case for n’ > k.
3. s 10 n 我们首先证明 n’> k 的情况。

By hypothesis k is maximal and hence y mod 10n’ > y x.
通过 假设 k 是最大的，因此 y mod 10n’ > y X.
模式 10n = y 10n

Given that y n’ 10 n we can conclude
考虑到这一点 10 n 我们可以得出结论
that > y x y an d “leading length” we
那个 n’ 不 k 及 “前导长度”我们
l’ n’

Suppose now that l .kBy of definitionL’
假设现在我 1 根据定义 of < 10
10 10 < 10 10 . Since < l

know that V l 1+ k and V’ l’ we
知道 v l 1 + k 和 v V’ V V 自从我 我们
have v l 1+k V. Also x l 1+k and therefore
有 v’l 1 + k 也是 x l 1 + k 及 因此
<l y1+kx and thus, using
x < 10 y. Clearly y < 1 y1 + kx，因
X < 10 Y 显然 y 此，使用
same equality as before 10 y mod 10 Y mod x which
the 样平等 作为之前，10 y X 什么
contradicts the minimality property of k.
与 k 的极小性质相矛盾。

□

6.2 Grisu2

6.2 Grisu2

In this section we will present Grisu2. As described above it will use extra bits to produce shorter outputs. As an evolution of Grisu, Grisu2 will not work with exact numbers (requiring bignum representations) either, but compute approximations of m and m^+, instead. In order to avoid wrong results (outputs that do not satisfy the internal identity requirement) we add a safety margin around the approximated boundaries. As a consequence Grisu2 sometimes fails to return the shortest optimal representation which could lie outside the conservative approximation. Also this safety-margin re-quires us to change the precondition. Indeed, using only 2 extra bits, the computation is so imprecise that Grisu2 could end up with an empty interval. In that case Grisu2 could simply fall back to Grisu, but this would unnecessarily complicate the following algorithm. We thus opted to give Grisu2 an extra bit: q p + 3.

在这一节中，我们将介绍 Grisu2。如上所述，它将使用额外的位来产生更短的输出。作为 Grisu 的进化，Grisu2 也不会使用精确的数字(需要用 bignum 表示)，而是计算 m 和 m + 的近似值。为了避免错误的结果(不满足内部身份要求的输出)，我们在近似边界附近增加了一个安全裕度。因此 Grisu2 有时不能返回可能位于保守近似之外的最短最优表示。此外，这种安全边际要求我们改变先决条件。事实上，仅仅使用 2 个额外的位，计算是如此的不精确以至于 Grisu2 可能以一个空的间隔结束。在这种情况下，Grisu2 可以简单地回到 Grisu，但是这会不必要地使下面的算法复杂化。因此我们选择给 Grisu2 一个额外的位: q p + 3。

Algorithm Grisu2
算法 Grisu2

Input: same as for Grisu.
输入: 与 Grisu 相同。

Preconditions: diy fp’s precision q satisfies q p + 3, and the powers-of-ten cache contains precomputed normalized rounded diy fp values c~k = 10k ? .
前提条件: diy fp 的精度 q 满足 q p + 3，并且 -10 的幂缓存包含预计算的归一化四舍五入 Diy fp 值 c ~ k = 10k ? .

Output: decimal digits d_i for i 0 n and an integer K such that the real V := d0: : :dn 10K verifies [V]p = v.
输出: 十进制数字 d 对我来说 0 N 和整数 k 等真正的 v: = d0: : dn 10K 验证[v] p = v。

Procedure:
程序:

1. Boundaries: compute v’s boundaries m and m+.
边界: 计算 v 的边界 m 和 m + 。

2. Conversion: determine the normalized diy fp w^+ such that w+ = m+. Determine the diy fp w such that w = m and that ew = ew+.
转换: 确定标准化的 diy fp w + W + = m + 。确定 diy fp w 这样 w = m 及 ew = ew + .

真恶心

3. Cached Power: find the normalized $\tilde{c}_{-k} = f_c 2^{e_c}$ such that

3. 缓存 Power: 查找标准化的 C ~-k = fc2ec 如此 那个

$e_c + e^+_w + q$ (with and as discussed for Grisu).

Ec + e + w + q (与 Grisu 一起讨论并讨论)。

4. Product: compute $\tilde{M}^-_\uparrow := \tilde{M}^-_w \tilde{c}_{-k}$, $\tilde{M}^+_\uparrow := \tilde{M}^+_w \tilde{c}_{-k}$, and let $M^\uparrow := \tilde{M}^-_\uparrow + 1$ ulp, $M^+_\# := \tilde{M}^+_\uparrow - 1$ ulp.

4. m 产品: 计算 m + W c-k, m w +, 然后放手 M": = m + 1 ulp, m # 译者注: 1 ulp, = m # M"。

5. Digit Length: find the greatest such that $M^+_\#$ mod 10

数字长度: 找到最大的 m + # mod 10

and define $P := \lfloor \frac{M^+_\#}{10} \rfloor$.

定义 p: = M # / 10.

6. Output: define $V := P\,10^{k+}$. The decimal digits d_i and n are obtained by producing the decimal representation of P (an integer). Set K := k + , and return it with the n digits d_i.

Output: define v: = p10k + .十进制数字 di 和 n 是通过产生 p (一个整数)的十进制表示得到的。设置 k: = k + ，并返回 n 位数 di。

We will show efficient implementations combining step 5 and 6 later, but first, we prove Grisu2 correct. As a preparation we start by showing that $M^\uparrow$ $M^+_\#$.

我们将在后面展示结合步骤 5 和步骤 6 的有效实现，但首先，我们证明 Grisu2 是正确的。作为准备，我们首先展示 m”m + # 。

Lemma 6.3. The variables M^{-} and $M^{+}_{\#}$ as described in step 4 verify M^{-} $M^{+}_{\#}$.

引理 6.3。步骤 4 中描述的变量 m”和 m + # 验证 m”m + # 。

Proof. By definition

证据，根据定义

= ~ + 1
= ~ + 1

M" M ulp

M, m, ulp

= w c~-k + 1 ulp

W c ~-k + 1 ulp

= hf w f c i" + 1 2ew +ec +q

= hf w f c i”+ 12 ew + ec + q

2q

2q

f w 2q f c + 1:52 ew +ec +q

F w 2 q f c + 1:52 ew + ec + q

ew +ec +q

f w+ f c

Ew + ec + q

F w + f c

2q 1:5 2

and similarly M# = 2q 1 分 52 秒

和类似的 M # = = 2q + 2q p

Since f w 1 2q p + 2qp2

自从 f w f w it suffices to show

w F w 这足以说明

f wf c (f w +2q p 1+2q p 2) f c

F wf c (f w + 2 q p1 + 2 q p2) f c

+ q p

1:5 f c q p 1 2

+ (2 2q Q p 1:5 or

2q 1:5 F c Qp12 q2 1:5 或

(2 + 2)

2q

3 2q

Using the inequalities f 1 and 3 it is sufficient to

利用不等式 f 2q q 2q 1 和 q p 3 足以

q 1 2 + 1 c

+ 2 + (2 (2 2)

show 3 2 2q

表演 3 2q .

□

Theorem 6.4. Grisu2’s result $V = d_0 : : : d_n\ 10^K$ satisfies the inter-

定理 6.4 Grisu2 的结果 v = d0: : dn 10K 满足

Proof. < 10 M 10 <

证明。 show that m M k + k m+

给我看看 < 10 M 10 < m

m k V + k +

We will 我们会 v’s boundaries).

(with m and m 的 (有 m 和 m V 的界

限)。

The inner inequality, M" 10k V M#+ 10k , is a consequence of Theorem 6.2. Remains to show that m < M" 10k and

内部不等式 m”10k v m # + 10k，是一个 consequence 定理 6.2 的序列，剩下的证明 m M”10k 和

M#+ 10k < m+.

M # + 10k < m + .

By Lemma 3.4 M and M have an error of strictly less than

引理 340 为 和 m 有严格小于... 的误差

ulp 10 k 10 k

, and therefore M" + m and M# 和 m # . As a consequence

因此 m 作为

1 consequence m < V < m

后果 < v < m .

□

Grisu2 does not give any guarantees on the shortness of its result. Its result is the shortest possible number in the interval $M^{-}\ 10^k$ to $M^{+}_{\#}\ 10^k$ (boundaries included), where M^{-} and $M^{+}_{\#}$ are dependent on diy fp’s precision q. The higher q, the closer

Grisu2 不能保证它的结果是短的。它的结果是在 m“10k 到 m + # 10k (包括边界)的区间内的最短可能数，其中 m”和 m + # 取决于 diy fp 的精度 q。Q 越高，距离越近

M^{-} and $M^{+}_{\#}$ are to the actual boundaries m and m+. For q = 64

和 m + 的实际边界 m 和 m + 。对于 q = 64

and p = 53 (as in our code samples) Grisu2 produces the shortest number for approximately 99.9% of its input.

和 p = 53(在我们的代码样本中) Grisu2 产生最短的数字，大约 99.9% 的输入。

The C implementation of Grisu2 is again cut into two parts: a core routine, independent of the chosen / , and a digit-generation procedure that needs to be tuned for the chosen target exponents. The core procedure is straightforward and we will therefore omit its C implementation.

Grisu2 的 c 实现再次被分为两部分：独立于所选/的核心例程和需要针对所选目标指数进行调优的数字生成过程。核心过程很简单，因此我们将省略它的 c 实现。

In Figure 7 we present a version of the digit-generation routine tuned for ; = 59; 32. The input variables Mp, and delta correspond to $M^{+}_{\#}$ and respectively. The len and K are used as return values (with obvious meanings). We assume that K has been initialized with k. We hence only need to add the missing .

在图 7 中，我们展示了为; = 59; 32 调优的数字生成例程的一个版本。输入变量 Mp 和 delta 分别对应于 m + # 和。Len 和 k 被用作返回值(有明显的含义)。我们假设 k 已经用 k 初始化。因此，我们只需要添加缺失的。

The proposed implementation combines step 5 and 6. While trying all possible s (starting from the “top”) it generates the

提出的实现结合了步骤 5 和步骤 6。当尝试所有可能的 s (从“顶部”开始)时，它会生成

M
#
. There are two digit-generation loops.
digits of M One for the
的数字 # 。有两个数字生成循环。一个为
10
Mp
宪兵
most-significant digits one , stored in p1, and one for the least-significant digits Mp mod one, stored in p2. Let R be the number that is obtained by reading the generated digits (R := 0 if no digit has been generated yet). Then the following invariants holds for
最高有效数字 1 存储在 p1 中，最低有效数字 1 存储在 p2 中。让 r 是通过读取生成的数字获得的数字(r: = 0，如果还没有生成数字)。那么下面的不变量适用于
both loops (line 9 and line 宪兵 . For the first
17): R = loop we
两个循环(第 9 行和第 17 10ka 在第一个循环
行) : r = 中
can show that .. The
p1 帕 equation in
++ = 中的等式
one p2
个 P2
line 13 thus tests if mod
M# 第 13 行因此测试 模式
m # 10 .
10

The following invariant holds for the second loop (line 17):
以下不变量适用于第二个循环(第 17 行) :

宪兵

10kappa

mod one.

10kappa mod
p2 =
P2 = 1.

```
1:  #define TEN9 1000000000
定义 ten91000000000
2:  void digit gen(diy fp Mp, diy fp delta,
空位数 gen (diy fp Mp，diy fp delta,
3:                 char* buffer, int* len, int* K) f
char * buffer，int * len，int * k) f
4:      uint32 t div; int d,kappa; diy fp one; one.f = ((uint64 t) 1) << -Mp.e;
        one.e = Mp.e;
一、 f = ((uint64t)1) < <-Mp.e; 一、 e = Mp.e;
5:      uint32 t p1 = Mp.f >> -one.e;
Uint32 t p1 = Mp.f > >-一个;
6:      uint64 t p2 = Mp.f & (one.f - 1);
Uint64tp2 = Mp.f & (one. f-1) ;
7:      *len = 0; kappa = 10; div = TEN9;
* len = 0; kappa = 10; div = TEN9;
8:      while (kappa > 0) f
而(kappa > 0) f
9:          d = p1 / div;
D = p1/div;
10:         if (d || *len) buffer[(*len)++] = '0' + d;
如果(d | | * len)缓冲液[(* len) + + ] ='0'+ d;
11:         p1 %= div; kappa--; div /= 10;
P1% = div; kappa; div/= 10;
12:         if ((((uint64 t)p1)<<-one.e)+p2 <= delta.f) f
如果(((uint64t) p1) <-one. e) + p2 < = delta.f) f
13:             *K += kappa; return;
* k + = kappa; 返回;
14:     g
15:     g
16:     do f
做 f
17:         p2 *= 10;
P2 * = 10;
18:         d = p2 >> -one.e;
D = p2 > >-1
19:         if (d || *len) buffer[(*len)++] = '0' + d;
如果(d | | * len)缓冲液[(* len) + + ] ='0'+ d;
20:         p2 &= one.f - 1; kappa--; delta.f *= 10;
P2 & = 1. f-1 kappa-; delta.f * = 10;
21:     g while (p2 > delta.f);
G 而(p2 > delta.f) ;
22:         *K += kappa;
* k + = kappa;
23:  g
```

Figure 7: Grisu2’s digit generation routine (for ； = 59; 32).
图 7: Grisu2 的数字生成例程(用于; = 59; 32)。

6.3 Grisu3

6.3 Grisu3

Given enough extra precision, Grisu2 computes the best result (still with respect to shortness) for a significant percentage of its input. However there are some numbers where the optimal result lies outside the conservative approximated boundaries. In this section we present Grisu3, an alternative to Grisu2. It will not be able to produce optimal results for these numbers either, but it reports failure when it detects that a shorter number lies in the uncertain region. We denote with “uncertain region” the interval around the approximated boundaries that might, or might not be inside the boundaries. That is, it represents the error introduced by Grisu3’s imprecision.
如果有足够的额外精度，Grisu2 计算其输入的很大一部分的最佳结果(仍然是关于短度)。然而，在一些数字中，最佳结果位于保守的近似边界之外。在这一节中，我们介绍 Grisu3，Grisu2 的替代品。它也不能产生这些数字的最优结果，但当它检测到一个较短的数字位于不确定区域时，它会报告失败。我们用“不确定区域”表示可能在或可能不在边界内的近似边界周围的区间。也就是说，它代表 Grisu3 的不精确性引入的错误。

Until now, optimality was defined with respect to the leading length (and of course accuracy) of the generated number V. For Grisu3 we add “closeness” as additional desired property: whenever there are several different numbers that are optimal with respect to shortness, Grisu3 should chose the one that is closest to
到目前为止，最优性是根据生成的数字 v 的前导长度(当然还有准确度)来定义的。对于 Grisu3，我们添加了“接近性”作为额外的期

望属性: 当有几个不同的数字在短度方面是最优的时候，Grisu3 应该选择最接近的那个

v.

V.

Instead of generating a valid number and then verifying if it is the shortest possible, Grisu3 will produce the shortest number inside the enlarged interval and verify if it is valid. Whereas Grisu2 used a conservative approximation of m and m^+, Grisu3 uses a liberal approximation and then, at the end, verifies if its result lies in the conservative interval, too.

Grisu3 不会生成一个有效数，然后验证它是否是最短的，而是在扩大的区间内生成最短的数，并验证它是否是有效的。Grisu2 使用 m 和 m + 的保守近似，Grisu3 使用自由近似，然后在最后验证它的结果是否也在保守区间。

Algorithm Grisu3

算法 Grisu3

Input and preconditions: same as for Grisu2.

输入和先决条件: 与 Grisu2 相同。

Output: failure, or decimal digits d_i for i 0 n and an integer K such that the integer V := d_0: : :d_n 10^K verifies $[V]_p$ = v. V has the shortest leading length of all numbers verify-ing this property. If more than one number has the shortest leading length, then V is the closest to v.

Output: failure，或者对于 i0n 和整数 k 的十进制数字 di，使得整数 v: = d0: : dn 10K 验证[v] p = v. v 具有验证该属性的所有数字中最短的前导长度。如果超过一个数字有最短的前导长度，那么 v 是最接近 v 的。

Procedure

:

程序:

1-

2.

1- same as for Grisu2.

2. Grisu2 也是如此。

Conversion: determine the normalized diy fp w such that

2b. Conversion: 确定规范化的 diy fp w，以便

w = v.

W = v.

3-

4.

3- same as for Grisu2.

4. Grisu2 也是如此。

Product2: let M# := 1 ulp, M" := M + 1 ulp, and

4b. Product2: 让 m # : = 一百万 : = 米 注: 和

m

:= M+

校对: =

m + M .

" #

5. Digit Length: find the greatest such that M^{+}" mod 10 < .
数字长度: 找到最大的 m + “ mod 10 < 。

Round: compute $\tilde{W}$ let $\tilde{W}$ ~ 1 ulp, and
圆的：计算 让 w 然后 1 分钟，

6. $W^{\sim}_{\#} := w \sim + 1$ ulp. Set $\tilde{k}$, and 10 I 为 I 为 I, 让我们

the greatest integer that verifies P_m 10 > $M_{\#}$.
验证 Pm 10 > m 的最大整数。

Let u, 0 u m the smallest integer such that $|P_u$ 10 W"$|$ is minimal. Similarly let d, 0 d m the largest integer such that $|P_d$ 10 W#$|$ is minimal.
让 u，0 u m，最小的整数，如 jPu 10 w“ j 是最小的。类似地，设 d，0 d 最大的整数 那个 jPd 10 W # j 是最小的。

If u 6= d return failure, else set P := P_u.
如果 u6 = d 返回失败，则设置 p: = Pu。

7. Weed: if not M" P 10$M^{+}_{\#}$ return failure.
Weed: 如果不是 m”p 10M + # 返回失败。

8. Output: define V := P 10^{k+} . The decimal digits d_i and n are obtained by producing the decimal representation of P (an integer). Set K := k + , and return it with the n digits d_i.
Output: define v: = p10k + .十进制数字 di 和 n 是通过产生 p (一个整数)的十进制表示得到的。设置 k: = k + ，并返回 n 位数 di。

Grisu3 uses the liberal boundary approximations ($M_{\#}$ and M^{+}") instead of the conservative ones (M" and $M^{+}_{\#}$). These values are guaranteed to lie outside the actual interval m to m^{+}. The test in step 5 therefore features a strict inequality. This way $M_{\#}$ is excluded from consideration. There is however no mechanism to exclude M^{+}". In the rare event when M^{+}" mod 10 = 0 then Grisu3 will, at this point of the algorithm, wrongly assume that M^{+}" is a potentially valid representation of v. Since M^{+}" lies outside the conservative region Grisu3 would the return failure in step 7. This case is rare and counter-measures are expensive, so we decided to accept this flaw.

Grisu3 使用自由边界近似(m # 和 m +”)而不是保守边界近似(m”和 m + #)。这些值保证位于 m 到 m + 的实际区间之外。因此，步骤 5 中的测试具有一个严格的不等式。这样 m # 就被排除在考虑之外了。然而，没有排除 m + 的机制。”。在极少数情况下，当 m + “ mod 10 = 0 时，Grisu3 将在算法的这一点上错误地假设 m +”是 v 的一个可能有效的表示形式。由于 m + “位于保守区域之外 Grisu3 将在步骤 7 中返回失败。这种情况很少见，反制措施也很昂贵，所以我们决定接受这个缺陷。

Rounding is performed in step 6. Grisu3 simply tries all possible numbers with the same leading length and picks the one that is closest to v. At this stage Grisu3 works with approximated values

四舍五入在步骤 6 中执行。Grisu3 只是简单地尝试所有可能的具有相同前导长度的数字，然后选择最接近 v 的那个。在这个阶段，Grisu3 使用的是近似值

and W, the approximation of v, may have an error of up to 1 ulp.
和 w，v 的近似值，可能具有高达 1 ulp 的误差。

The closest representation P 10 must not only be the closest to
最接近的 p10 表示不仅必须是最接近的

W but to all possible values in W’s margin of error. Grisu3 first finds the closest P_u to the upper boundary, and P_d for the lower boundary. If they are not the same, then the precision is not enough to determine the optimal rounding and Grisu3 aborts.
但对于 w 的误差范围内的所有可能值。Grisu3 首先找到离上边界最近的 Pu，下边界的 Pd。如果它们不一样，则精度不足以确定最佳舍入和 Grisu3 中止。

Finally, just before outputting the computed representation, Grisu3 verifies if the best pick is within the conservative boundaries. If it is not, then the optimal solution lies in the uncertain region and Grisu3 returns failure.

最后，在输出计算表示之前，Grisu3 验证最佳选择是否在保守边界内。如果不是，那么最优解在于不确定区域，Grisu3 返回失败。

```
bool round weed(char* buffer, int len,
bool round weed (char * buffer, int len,
        uint64 t wp W, uint64 t Delta,
        Uint64t wp w uint64t Delta,
        uint64 t rest, uint64 t ten kappa,
        休息 64 分钟, 休息 64 分钟,
        uint64 t ulp) f
        64 t ulp) f
  uint64 t wp Wup = wp W - ulp;
  Wup = wp w-ulp;
  uint64 t wp Wdown = wp W + ulp;
  Wdown = wp w + ulp;
  while (rest < wp Wup &&
  (休息 < wp Wup & & &
      Delta - rest >= ten kappa &&
      三角洲休息 > = 10 卡帕 & &
第九名:
      (rest + ten kappa < wp Wup ||
      (休息 + 10 卡帕 < wp Wup ||
      wp Wup-rest >= rest+ten kappa - wp Wup))
      Wp Wup-rest > = rest + ten kappa-wp Wup)
  f
    buffer[len-1]--; rest += ten kappa;
    Buffer [ len-1]-; rest + = 10 kappa;
  g
  if (rest < wp Wdown &&
  如果(休息 < wp Wdown & & &
     Delta - rest >= ten kappa &&
     三角洲休息 > = 10 卡帕 & &
     (rest + ten kappa < wp Wdown ||
     (休息 + 10 卡帕 < wp Wdown ||
      wp Wdown-rest > rest+ten kappa - wp Wdown))
      Wp Wdown-rest > rest + ten kappa-wp Wdown)
   return false;
返回 false;
   return 2*ulp <= rest && rest <= Delta - 4*ulp;
返回 2 * ulp < = rest & & & rest < = Delta-4 * ulp;
g
g
```

Figure 8: Grisu3’s round-and-weed procedure.
图 8: Grisu3 的圆形除草程序。

The digit-generation routine of Grisu2 has to be modified to take the larger liberal boundary interval into account, but these changes are minor and obvious. The interesting difference can be summarized in the round weed procedure shown in Figure 8 which com-

Grisu2 的数字生成例程必须进行修改，以考虑到更大的自由边界区间，但这些变化是微小和明显的。有趣的差异可以总结在图 8 所示的圆形杂草程序中，其中 com

bines step 6 and 7. The function is invoked with the parameters set to the following values: buffer := d_0: : :d_{len-1} where d_0: : :d_{len-1}
第六步和第七步。使用设置为以下值的参数调用该函数：buffer: = d0: : dlen-1，其中 d0: : dlen-1

10^{w^+}, wp W := W~+ W~, are the decimal digits of $10^{"}$ Delta := ，是。的十进制数字 ¯”三角洲: = ，

rest := $W^{+"}$ mod 10 , ten kappa := 10 , and ulp the value of 1 ulp relative to all passed diy fps.
Rest: = w +”mod 10,10 kappa: = 10，并且 ulp 1 ulp 相对于所有通过的 diy fps 的值。

Let T := d_0: : :d_{len-1} 10 . By construction T lies within the unsafe interval: $W_{\#}$ < T $W^{+"}$. Furthermore, among all possible values in this interval, it has the shortest leading length len. If there are other possible values with the same leading length and in the same interval, then they are smaller than T.

Let t: = d0: : dlen-110.通过构建 t 位于不安全的区间内: w # < t w +”。此外，在这个区间内所有可能的值中，它具有最短的前导长度透镜。如果有其他可能的值具有相同的前导长度和相同的间隔，那么它们小于 t。

The loop in line 7 iteratively tests all possible alternatives to find the closest to W". The first test, rest < wp W - ulp, ensures that T is not already less than W" (in which case the current T must be the closest). Then follows a verification that the next lower number with the same leading length is still in the interval $W_{\#}$

第 7 行中的循环迭代地测试所有可能的替代方案，以找到最接近 w”的方案。第一个测试，rest < wp w-ulp，确保 t 不会小于 w”(在这种情况下，当前 t 必须是最接近的)。然后验证下一个具有相同前导长度的较小数字仍然在 w # 的区间内

to $W^{+"}$. In line 9 the procedure tests if the alternative would be closer to W". If all tests succeed, then the number d_0: : :d_{len-1} 10 is guaranteed to lie inside the interval $W_{\#}$ to $W^{+"}$ and is furthermore closer to W" than the current T. The body of the loop thus replaces 到 w + 。在第 9 行，这个过程测试另一个选择是否更接近 w”。如果所有的测试都成功，那么 d0: : dlen-110 保证位于 w # 到 w + 的区间内，而且比当前的 t 更接近 w”。因此，循环的主体取代了
T (physically modifying the buffer) with its smaller alternative.
(物理修改缓冲区)与其较小的替代品。

The if in line 14 then verifies the chosen T is also closest to # . If this check fails then there are at least two candidates that

第 14 行的 if 验证选择的 t 也是最接近 # 的。如果这个检查失败，那么至少有两个候选者

could be the closest to W and Grisu3 returns failure.
可能是最接近 w 和 Grisu3 返回失败。

Now that the buffer has been correctly rounded a final weeding test in line 19 verifies that W" T $W^{+}_{\#}$. That is, that the chosen

现在，缓冲区已经正确地四舍五入了第 19 行中的最后一个除草测试，该测试验证了 w”t w + # 。也就是说，被选中的
T is inside the safe conservative interval.
在安全保守区间内。

7. Benchmarks
基准

Algorithm 算法	R	R/PP R/PP	S	S/PP 私人 助理
sprintf %c Sprintf% c	2.60	-	-	-
sprintf %g	22.83	-	24.17	-

冲刺% g sprintf %.17e Sprintf% 17 e	36.03	-	36.17	-
burger/dybvi g 汉堡包和 dybvig	61.53	61.49	28.73	28.66
grisu Fig4 Grisu 图 4	9.17	-	9.98	-
grisu 0,3 Grisu 0,3	2.85	-	3.26	-
grisu -63,-59 格里苏 -63,- 59	3.36	-	3.77	-
grisu -35,-32 Grisu-35-32	2.66	-	3.04	-
grisu -59,-56 格里苏 -59,- 56	2.50	-	2.96	-
grisu2 -59,- 56 Grisu2-59-56	3.80	4.88	3.07	4.04
grisu2b -59,- 56 Grisu2b-59,- 56	5.38	6.42	4.40	5.48
grisu3 -59,- 56 Grisu3-59-56	4.47	5.61	3.49	4.55

Figure 9: Speed of sprintf, Burger/Dybvig and Grisu.
图 9: sprintf，Burger/Dybvig 和 Grisu 的速度。

Algorithm 算法	optimal 最优	shortest 最短
grisu2 -59,-56	0	99.92
Grisu2-59,-56	099.92	
grisu2b -59,-56	99.85	99.92
Grisu2b-59,-56	99.8599.92	
grisu3 -59,-56	99.49	99.49
Grisu3-59,-56	99.4999.49	

Figure 10: Optimality of Grisu2 and Grisu3.
图 10: Grisu2 和 Grisu3 的最优性。

In this section we present some experimental results. In Fig-ure 9 we compare different variants of Grisu against sprintf and Burger/Dybvig's algorithm (the code has been taken from their website). In order to measure the parsing-overhead of sprintf we added one sprintf benchmark where the double is converted to a char, and then printed (first row). Also we included the un-optimized algorithm of Figure 4. Grisu2b ("grisu2b -35,-32") is a variant of Grisu2 where the result is rounded.

在这一节中，我们介绍了一些实验结果。在图 9 中，我们比较了 Grisu 与 sprintf 和 Burger/Dybvig 算法的不同变体(代码来自他们的网站)。为了测量 sprintf 的解析开销，我们添加了一个 sprintf 基准，其中 double 被转换为 char，然后打印(第一行)。我们还包含了图 4 中未经优化的算法。Grisu2b (" Grisu2b-35,-32")是 Grisu2 的变体，其结果是四舍五入的。

Input numbers are random IEEE doubles. Speed is measured in “seconds per thousand numbers”. All benchmarks have been executed on a Intel(R) Xeon(R) CPU 5120 @ 1.86GHz quad-core system, Linux 2.6.31, glibc 2.11.

输入数字是 IEEE 的随机双精度数字。速度是以“千分之几秒”来衡量的。所有基准测试都是在 Intel (r) Xeon (r) CPU 5120@1.86 GHz 四核系统、 Linux 2.6.31、 glibc 2.11 上执行的。

The first column (R) gives the speed of processing random doubles. The next column shows the time for the same numbers, but with a pretty-printing pass for the algorithms that only returned the digits and the exponent K.

第一列(r)给出了处理随机双精度数据的速度。下一列显示了相同数字的时间，但是对于只返回数字和指数 k 的算法来说，它有一个漂亮的打印通道。

Column S measures the processing speed for short doubles. That is, doubles that have at most 6 leading digits. Algorithms that stop once the leading digits have been found are clearly faster for these numbers. The next column (S/PP) adds again a pretty-printing pass.

Column s 测量短双精度浮点数的处理速度。也就是说，最多有 6 个前导数字的 double。一旦找到前导数字就会停止的算法对于这些数字来说显然更快。下一栏(s/PP)再次添加了一个漂亮的打印通道。

In Figure 10 we show the percentage of numbers that are optimal (shortest and rounded) or just shortest. We have excluded sprintf (0 in both columns), Burger/Dybvig (100 in both columns) and Grisu (0 in both columns). 99.92% of Grisu2’s pre-sentations are the shortest, and once a rounding-phase has been added (Grisu2b) 99.87% of the numbers are optimal. Grisu3 pro-duces optimal results for 99.49% of its input and rejects the rest.

在图 10 中，我们显示了最佳(最短和四舍五入)或者只是最短的数字的百分比。我们排除了 sprintf (两列中均为 0) ，Burger/Dybvig (两列中均为 100)和 Grisu (两列中均为 0)。99.92% 的 Grisu2 呈现是最短的，一旦加入一个舍入阶段(Grisu2b) ，99.87% 的呈现是最优的。Grisu3 对 99.49% 的输入产生最佳结果，并拒绝其余的。

8. Related Work
相关工作

In 1980 Coonen was the first to publish a binary-decimal conversion algorithm [Coonen(1980)]. A more detailed discussion of conversion algorithms appeared in his thesis [Coonen(1984)] in 1984. Coonen proposes two algorithms: one using high-precision integers (bignums) and one using extended types (a floating-point number type specified in IEEE 754 [P754(1985)]). By replacing the extended types with diy fp’s the latter algorithm can be transformed into a special case of Grisu.

1980 年，Coonen 是第一个发表二进制十进制转换算法的人。1984 年，他的论文[Coonen (1984)]中对转换算法进行了更详细的讨论。Coonen 提出了两种算法: 一种使用高精度整数(bignums) ，另一种使用扩展类型(IEEE 754[P754(1985)]中指定的浮点数类型)。通过用 diy fp 替换扩展类型，后一种算法可以转换成 Grisu 的特殊情况。

In his thesis Coonen furthermore describes space efficient al-gorithms to store the powers-of-ten, and presents a very fast logarithm-estimator for the k-estimation (closely related to the k-computation of Section 4.1).

在他的论文中，Coonen 进一步描述了存储 -10 的幂的空间有效算法，并提出了一个非常快的对数估计的 k 估计(与第 4.1 节的 k 计算密切相关)。

In 1990 Steele and White published their printing-algorithm, Dragon, [Steele Jr. and White(1990)]. A draft of this paper had ex-isted for many years and had already been cited in ”Knuth, Volume II” [Knuth(1981)] (1981). Dragon4 is an exact algorithm and re-quires bignums. Dragon4 generates its digits from left to right and stops once the result lies within a certain precision. This approach differs from Coonen’s bignum algorithm where all relevant digits are produced before the result is rounded. The rounding process might lead to changing a trailing sequence of 9s to 0s thus shorten-ing the generated sequence.

1990 年 Steele and White 发表了他们的打印算法 Dragon，[Steele jr. and White (1990)]。该论文的一个草稿已经存在多年，并已在“ Knuth， Volume II”[Knuth (1981)](1981) 中引用。Dragon4 是一个精确的算法，需要大数字。Dragon4 从左到右生成数字，当结果在一定精度范围内时停止。这种方法不同于 Coonen 的 bignum 算法，后者在结果四舍五入之前生成所有相关数字。四舍五入的过程可能会导致将尾随序列从 9s 改为 0s，从而缩短生成的序列。

Conceptually the simplest form of Grisu2 and Grisu3 (presented in Section 6) can be seen as a combination of Coonen’s floating-point algorithm and Dragon.

从概念上讲，Grisu2 和 Grisu3 的最简单形式(见第 6 节)可以看作是 Coonen 的浮点算法和 Dragon 的结合。

In the same year (1990) Gay improved Dragon, by proposing a faster k-estimator and by indicating some shortcuts [Gay(1990)].

在同一年(1990) ，Gay 通过提出更快的 k 估计量和指示一些捷径[Gay (1990)]改进了 Dragon。

Burger and Dybvig published their improvements in 1996 [Burger and Dybvig(1996)]. This paper features a new output for-mat where insignificant digits are replaced by # marks. They had also rediscovered the fast logarithm-estimator that had been pub-lished in Coonen’s thesis.

Burger 和 Dybvig 在 1996 年发表了他们的改进[Burger 和 Dybvig (1996)]。本文提供了一个新的 -mat 输出，其中不重要的数字被 # 标记替换。他们还重新发现了 Coonen 论文中发表的快速对数估计器。

9. Conclusion
结论

We have presented three new algorithms to print floating-point numbers: Grisu, Grisu2 and Grisu3. Given an integer type with at least two more bits than the input's significand, then Grisu prints its input correctly. Grisu2 and Grisu3 are designed to benefit from integer types that have more than just two extra bits.

我们提出了三种打印浮点数的新算法: Grisu、 Grisu2 和 Grisu3。给定一个至少比输入有效值多两位的整数类型，然后 Grisu 正确地输出它的输入。Grisu2 和 Grisu3 被设计成受益于超过两个额外位的整数类型。

Grisu2 may be used where an optimal decimal representation is desired but not required. Given a 64 bit unsigned integer Grisu2 computes the optimal decimal representation 99.8% of the times for IEEE doubles.

Grisu2 可以用在需要但不需要最佳小数表示的地方。给定一个 64 位无符号整数 Grisu2 计算 IEEE 双精度数的 99.8% 的最佳小数表示。

Grisu3 should be used when the optimal representation is re-quired. In this case Grisu3 will efficiently produce the optimal re-sult for 99.5% of its input (with doubles and 64-bit integers), and reject the remaining numbers. The rejected numbers must then be printed by a more accurate (but slower) algorithm. Since Grisu3 is about 4 times faster than these alternative algorithms the average speed-up is still significant.

当需要最佳表示时，应该使用 Grisu3。在这种情况下，Grisu3 将有效地为其 99.5% 的输入(双精度和 64 位整数)产生最佳结果，并拒绝剩余的数字。被拒绝的数字必须用更精确(但更慢)的算法打印出来。由于 Grisu3 的速度大约是这些替代算法的 4 倍，因此平均速度仍然是显著的。

Grisu (and its evolutions) are furthermore straightforward to implement and do not feature many special cases. This is in stark contrast to efficient printing-algorithms that are based on bignums. Indeed, the major contributions to Dragon4 (one of the first published bignum-algorithms) have been the identification of special cases that could be handled more efficiently. We hope that Grisu3 renders this special cases uneconomic, thus simplifying the complete development of floating-point printing algorithms.

Grisu (及其演变)的实现更加直观，并且没有很多特殊情况。这与基于 bignums 的高效打印算法形成鲜明对比。事实上，Dragon4(最早发表的 bignum 算法之一)的主要贡献在于识别了可以更有效地处理的特殊情况。我们希望 Grisu3 使这种特殊情况不经济，从而简化浮点打印算法的完整开发。

References
参考文献

[Burger and Dybvig(1996)] R. G. Burger and R. K. Dybvig. Printing Floating-Point Numbers Quickly and Accurately. In Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation, PLDI 1996, pages 108–116, New York, NY, USA, June 1996. ACM. doi: 10.1145/249069.231397.

[Burger and Dybvig (1996)] r. g. Burger and r. k. Dybvig.快速准确地打印浮点数。在 ACM SIGPLAN 1996 年关于编程语言设计和实现的会议记录中，PLDI 1996，第 108-116 页，纽约，美国，1996 年 6 月。ACM.Doi: 10.1145/249069.231397.

[Coonen(1980)] J. T. Coonen. An implementation guide to a proposed standard for floating-point arithmetic. Computer, 13(1):68–79, 1980. ISSN 0018-9162. doi: 10.1109/MC.1980.1653344.

J · t · 库恩。浮点运算提议标准的实现指南。计算机，13(1)：68-79,1980。ISSN 0018-9162.Doi: 10.1109/MC. 1980.1653344.

[Coonen(1984)] J. T. Coonen. Contributions to a Proposed Standard for Binary Floating-Point Arithmetic. PhD thesis, University of California, Berkeley, June 1984.

库恩(1984) j · t · 库恩。对提出的二进制浮点运算标准的贡献。博士论文，加州大学伯克利分校，1984 年 6 月。

[Gay(1990)] D. M. Gay. Correctly rounded binary-decimal and decimal-binary conversions. Technical Report 90-10, AT&T Bell Laboraties, Murray Hill, NJ, USA, Nov. 1990.

[Gay (1990)] d. m. Gay.正确四舍五入的二进制-十进制和十进制-二进制转换。技术报告 90-10，at & t 贝尔实验室，Murray Hill，NJ，USA，1990 年 11 月。

[Goldberg(1991)] D. Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys, 23(1): 5–48, 1991. ISSN 0360-0300. doi: 10.1145/103162.103163.

金伯格(1991)。关于浮点运算，每个计算机科学家都应该知道什么。Acm 计算概观，23(1) : 5-48,1991。ISSN 0360-0300.Doi: 10.1145/103162.103163.

[Knuth(1981)] D. E. Knuth. The Art of Computer Programming, Volume

[Knuth (1981)] d. e. Knuth. 计算机编程艺术，卷 II: Seminumerical Algorithms, 2nd Edition. Addison-Wesley, 1981. ISBN 0-201-03822-6.
半数值算法，第 2 版。 Addison-Wesley，1981。 ISBN 0-201-03822-6。

[P754(1985)] I. T. P754. ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic. IEEE, New York, Aug. 12 1985.

[P754(1985)] I.t. P754. ANSI/IEEE 754-1985，二进制浮点运算标准 IEEE，纽约，1985 年 8 月 12 日。

[Steele Jr. and White(1990)] G. L. Steele Jr. and J. L. White. How to print floating-point numbers accurately. In Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI 1994, pages 112–126, New York, NY, USA, 1990. ACM. ISBN 0-89791-364-7. doi: 10.1145/93542.93559.

[Steele J.. and White (1990)] g. l. Steele J.. and j. l. White.如何精确打印浮点数。在 ACM SIGPLAN 1994 年关于编程语言设计和实现的会议记录中，PLDI 1994，第 112-126 页，纽约，美国，1990。ACM.ISBN 0-89791-364-7.Doi: 10.1145/93542.93559.

[Steele Jr. and White(2004)] G. L. Steele Jr. and J. L. White. How to print floating-point numbers accurately (retrospective). In 20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection, pages 372–374. ACM, 2004. ISBN 1-58113-623-4. doi: 10.1145/989393.989431.

[Steele J.. and White (2004)] g. l. Steele J.. and j. l. White.如何准确打印浮点数(回顾性)。1979-1999 年 ACM SIGPLAN 编程语言设计与实现会议 20 年，a Selection，页 372-374。ACM，2004.国际标准书号 1-58113-623-4。Doi: 10.1145/989393.989431.