Printing Floating-Point Numbers Quickly and Accurately with Integers

Florian Loitsch

Inria Sophia Antipolis
2004 Route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex
florian.loitsch@inria.fr

Abstract

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 integers) and correct: any printed number will evaluate to the same number, when read again.

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.

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.

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.

Categories and Subject Descriptors I.m [*Computing Methodologies*]: Miscellaneous

General Terms Algorithms

Keywords floating-point printing, dtoa

1. Introduction

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.

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).

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'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 earlier proved by exhaustive testing that just 7 extra bits suffice for correctly printing 36-bit PDP-10 floating-point numbers, if powers 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".

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.

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.

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)]

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019/10/06. . . $5.00.

[1] A draft of this article had existed long before and had already been mentioned in "Knuth Volume 2"[Knuth(1981)] in 1981.

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).

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.

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.

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.

2. Floating-Point Numbers

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.

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.

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 $v = f_v \times b^{e_v}$.

Unless otherwise stated, we use the convention that the significand 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.

Any significand f satisfies $f = \sum_{i=0}^{p-1} d_i \times b^i$, $0 \leq d_i < b$ where the integers d_i are called the digits of f. We call a number “normalized” if the most-significant digit d_{p-1} is non-zero.

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”.

Note. Floating-point numbers may allow different representations for the same value (for example 12×10^1 and 1.2×10^2). The representation is however unique when all numbers are either normalized or denormal.

2.1 Rounding and Errors

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.

The most natural way of rounding is to chose the nearest available floating-point number. This *rounded-to-nearest* approach is straightforward except for half-way cases (in the decimal system numbers ending with 5).

In this paper we will use the following strategies for half-way cases:

- *up*: picks the number closer to +infinity. We will use the notation $[x]^{\uparrow}$ when rounding x by this strategy.
- *even*: picks the number that is even: $[x]^{\square}$. 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.

Whenever the half-way rounding strategy has no importance we will use a star to make this fact explicit: $[x]^{\star}$.

We will use the notation $\tilde{x} = [x]_p^s$ to indicate that the floating-point number $\tilde{x}$ contains a normalized significand of size p which has been computed by rounding-to-nearest using strategy s (up, even, or any).

We can quantify $\tilde{x}$'s error $|\tilde{x} - x|$ as follows: $\tilde{x}$ is of form $f \times b^e$ and since f has been rounded to nearest $|\tilde{x} - x| \leq 0.5 \times b^e$, or, in 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: $1\ \mathtt{ulp}_x$.

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.

2.2 Neighbors and Boundaries

For floating-point number types where the value of each encoded number is unique we can define predecessors and successors.

Let $v = f_v \times b^{e_v}$ be a strictly positive floating-point number. The predecessor v^- of v is the next smallest number. If v is minimal, then we define 0 to be its predecessor. Similarly v^+ designates the successor of v. For the maximal v we define v^+ to be $v^+ := v + (v - v^-)$. That is for this particular v the successor v^+ is at the same distance than the predecessor v^-. We call v^- and v^+ *neighbors* of v.

The *boundary* between two adjacent numbers v_1 and v_2 is simply their arithmetic mean: $m := \frac{v_1 + v_2}{2}$. By definition boundaries can not be expressed in the given floating-point number type, since its value lies between two adjacent floating-point numbers. Every floating-point number v has two associated boundaries: $m^- := \frac{v^- + v}{2}$ and $m^+ := \frac{v + v^+}{2}$. Clearly, any real number w, such 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.

We conclude this section with a definition we will use frequently in the remainder of this document.

Definition 2.1. A printed representation R of a floating-point number v satisfies the *internal identity requirement* iff R would convert to v when read again.

For IEEE 754 double-precision numbers (where half-way cases are rounded to even) this implies $[R]_p^{\square} = v$. In other words:

$$\begin{array}{ll} m^- \leq V \leq m^+ & \text{when } f_v \text{ is even, and} \\ m^- < V < m^+ & \text{when } f_v \text{ is odd.} \end{array}$$

2.3 Examples

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.

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

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 $\tilde{v} := [v]^\star_3 := 123{\times}10^1$ is the only possible representation. The rounding error is equal to $4 = 0.4$ `ulp`.

Contrary to v the real number $w := 1245$ lies exactly between to possible representations. Indeed, $124{\times}10^1$ and $125{\times}10^1$ 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.

The neighbors of w are $w^- := 123{\times}10^1$ and $w^+ := 125{\times}10^1$. Its respective boundaries are therefore $m^- := 123.5{\times}10^1$ and $m^+ := 124.5{\times}10^1$. In this case the neighbors were both at the same distance. This is not true for $r := 100{\times}10^3$, with neighbors $r^- := 999{\times}10^2$ and $r^+ := 101{\times}10^3$. Clearly r^- is closer to r than is 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 $0.5{\times}10^1$ (resp. $1.5{\times}10^1$). For max, the boundaries are $998.5{\times}10^{10}$ and $999.5{\times}10^{10}$.

The boundaries for the smallest normalized number are special: even though its significand is equal to 100 the distance to its lower neighbor ($99{\times}10^0$) is equal to 1 `ulp` and not just 0.5 `ulp`. Therefore its boundaries are $99.5{\times}10^0$ and $100.5{\times}10^0$.

2.4 IEEE 754 Double-Precision

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 e_{IEEE}, 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 with positive numbers.

With the exception of some special cases (which will be discussed 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 effectively 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} consist of subtracting 1075. Combining this information, the value v of any normalized double can be computed as $f_v := hidden + f_{IEEE}$, $e_v := e_{IEEE} - bias$ and hence $v = f_v{\times}2^{e_v}$.

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

IEEE 754 reserves some configurations for special values: when $e_{IEEE} = 0x7FF$ (its maximum) and $f_{IEEE} = 0$ then the double is infinity (or minus infinity, if the bit-sign is set). When $e_{IEEE} = 0x7FF$ and $f_{IEEE} \neq 0$ then the double represents "NaN" (Not a Number).

The exponent $e_{IEEE} = 0$ is reserved for denormals and zero. Denormals do not have a hidden bit. Their value can be computed as follows: $f_{IEEE}{\times}2^{1-bias}$.

Throughout this paper we will assume that positive and negative infinity, positive and negative zero, as well as NaN have already been handled. Developers should be careful when testing for negative zero, though. Following the IEEE 754 specification $-0.0 = +0.0$ and $-0.0 \not< +0.0$. One should thus use the sign-bit 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.

Note. Any value representable by doubles (except for NaNs) has a unique representation.

Note. For any non-special strictly positive IEEE double v with $f_{IEEE} \neq 0$ the upper and lower boundaries m^+ and m^- are at distance 2^{v_e-1}. When $f_{IEEE} = 0$ then m^+ is still at distance 2^{v_e-1} but the lower boundary only satisfies $v - m^- \leq 2^{v_e-2}$.[2]

3. Handmade Floating-Point

```
typedef struct diy_fp {
    uint64_t f;
    int e;
} diy_fp;
```

Figure 1: The `diy_fp` type.

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.

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 $x = f_x{\times}2^{e_x}$.

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.

3.1 Operations

Grisu extracts the significand of its `diy_fp`s in an early stage and `diy_fp`s 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.

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.

The multiplication returns a `diy_fp` $\tilde{r}$ containing the rounded result of multiplying the two given `diy_fp`s 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: $\tilde{r} := x \otimes y$.

[2] The inequality is only needed for $e_{IEEE} = 1$ where the predecessor is a denormal.

```
diy_fp minus(diy_fp x, diy_fp y) {
  assert(x.e == y.e && x.f >= y.f);
  diy_fp r = {.f = x.f - y.f, .e = x.e};
  return r;
}
```

(a) Subtraction

```
 1: diy_fp multiply(diy_fp x, diy_fp y) {
 2:   uint64_t a,b,c,d,ac,bc,ad,bd,tmp;
 3:   diy_fp r; uint64_t M32 = 0xFFFFFFFF;
 4:   a = x.f >> 32; b = x.f & M32;
 5:   c = y.f >> 32; d = y.f & M32;
 6:   ac = a*c; bc = b*c; ad = a*d; bd = b*d;
 7:   tmp = (bd>>32) + (ad&M32) + (bc&M32);
 8:   tmp += 1U << 31;  // Round
 9:   r.f = ac + (ad>>32) + (bc>>32) + (tmp>>32);
10:   r.e = x.e + y.e + 64;
11:   return r;
12: }
```

(b) Multiplication

Figure 2: diy_fp operations

Definition 3.2. Let x and y be two diy_fps. Then

$$x \otimes y := \left[\frac{f_x \times f_y}{2^q}\right]^{\uparrow} \times 2^{e_x+e_y+q}$$

The C implementation emulates in a portable way a partial 64-bit multiplication. Since the 64 least significant bits of the multiplication $f_x \times f_y$ 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*).

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

Lemma 3.3. *Let x and y be two* diy_fps*. Then the error of $x \otimes y$ is less than or equal to .5* ulp*:*

$$|x \times y - x \otimes y| \leq .5\ ulp$$

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

Lemma 3.4. *Let x and $\tilde{y}$ be two* diy_fps*, and y a real such that $|y - \tilde{y}| \leq u_y$* ulp*. In other words $\tilde{y}$ 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*.*

$$\forall y,\ |y - \tilde{y}| \leq u_y\ ulp \Longrightarrow |x \times y - x \otimes \tilde{y}| < (u_y + .5)\ ulp$$

Proof. By Lemma 3.3 we have $|x \times \tilde{y} - x \otimes \tilde{y}| \leq 0.5 \times 2^{q+e_x+e_y}$ and by hypothesis $|y - \tilde{y}| \leq u_y\ \mathtt{ulp} = u_y \times 2^{e_y}$.
Clearly $|x \times y - x \times \tilde{y}| \leq x \times (u_y \times 2^{e_y}) < u_y \times 2^{q+e_x+e_y}$ and thus, by summing the inequalities $|x \times y - x \otimes \tilde{y}| < (.5 + u_y) 2^{q+e_x+e_y}$. □

Lemma 3.5. *Let x be a normalized* diy_fp*, $\tilde{y}$ be a* diy_fp*, and y a real such that $|y - \tilde{y}| \leq u_y$* ulp*. If $x = 2^{q-1}$ (the minimal significand) then $x \otimes \tilde{y}$ undershoots by at most $\frac{u_y}{2}$* ulp *compared to $x \times y$.*

$$|y - \tilde{y}| \leq u_y\ ulp \wedge f_x = 2^{q-1} \Longrightarrow x \times y - x \otimes \tilde{y} \leq \frac{u_y}{2}\ ulp$$

Proof. By definition $x \otimes y = \left[\frac{f_y}{2}\right]^{\uparrow} \times 2^{e_x+e_y+q}$. Since $\frac{f_y}{2}$ is either exact or a half-way case we have $\left[\frac{f_y}{2}\right]^{\uparrow} \times 2^{e_x+e_y+q} \geq f_y \times 2^{e_x+e_y+q-1}$ and hence $x \otimes \tilde{y} \geq x \times \tilde{y}$. Also $|x \times y - x \times \tilde{y}| \leq u_y \times 2^{q+e_x+e_y-1}$ and thus $x \times y - x \otimes \tilde{y} \leq \frac{u_y}{2}$ ulp. □

4. Cached Powers

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 $\tilde{c}_k := [c_k]^{\star}_q$ where $c_k := 10^k$. Note that, since all c_k are normalized $\forall i,\ 3 \leq e_{\tilde{c_i}} - e_{\tilde{c_{i-1}}} \leq 4$.

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 Section 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.

The corresponding C procedure has the following signature:

```
diy_fp cached_power(int k);
```

4.1 k Computation

Grisu (and its evolutions) need to find an integer k such that its cached power $\tilde{c}_k = f_{c_k} \times 2^{e_{c_k}} = \left[10^k\right]^{\star}_q$ satisfies $\alpha \leq e_{c_k} + e \leq \gamma$ for a given e, α and γ. We impose $\gamma \geq \alpha + 3$, since otherwise a solution is not always possible. We now show how to compute the sought k.

All cached powers are normalized and any f_{c_k} thus satisfies $2^{q-1} \leq f_{c_k} < 2^q$. Hence, $2^{e_{c_k}+q-1} \leq \tilde{c}_k < 2^{e_{c_k}+q}$.

Suppose that all cached powers are exact (i.e. have no rounding errors). Then k (and its associated $\tilde{c}_k$) can be found by computing the smallest power of ten 10^k that verifies $10^k \geq 2^{\alpha-e+q-1}$.

$$k := \lceil log_{10} 2^{\alpha-e+q-1} \rceil = \left\lceil (\alpha - e + q - 1) \times \frac{1}{log_2 10} \right\rceil$$

```
#define D_1_LOG2_10 0.30102999566398114 // 1/lg(10)
int k_comp(int e, int alpha, int gamma) {
    return ceil((alpha-e+63) * D_1_LOG2_10);
}
```

Figure 3: k_computation C procedure

Figure 3 presents a C implementation (specialized for $q = 64$) of this computation. In theory the result of the procedure could be wrong since $\tilde{c}_k$ is rounded, and the computation itself is approximated (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.

5. Grisu

In this section we will discuss Grisu, a fast intuitive printing algorithm. We will first present its idea, followed by a formal description of the algorithm. We then prove its correctness, and finally show a C implementation.

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.

5.1 Idea

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.

Assume, without loss of generality, that a floating-point number v has a negative exponent. Then v can be expressed as $\frac{f_v}{2^{-e_v}}$. The decimal digits of v can be computed by finding a decimal exponent t such that $1 \leq \frac{f_v \times 10^t}{2^{-e_v}} < 10$.

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.

The idea behind Grisu is to cache approximated values of $\frac{10^t}{2^{e_t}}$. The expensive bignum operations disappear and are replaced by operations on fixed-size integer types.

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: $\tilde{c}_k := \left[10^k\right]^\star_q$ (where q is the precision of the cached numbers). By construction the digit generation process uses a power of ten with an exponent $e_{\tilde{c}_t}$ close to e_v. Even though $e_{\tilde{c}_t}$ and e_v do not cancel each other out anymore, the difference between the two exponents will be small and can be easily integrated in the computation of v's digits.

In fact, Grisu does not use the power of ten $\tilde{c}_k$ 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.

5.2 Algorithm

In this section we present a formalized version of Grisu. As explained 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 remains. We have also hinted that Grisu chooses its power-of-ten depending on the sought remaining exponent. In the following algorithm we parametrize the remaining exponent by the variables α and γ. We impose $\gamma \geq \alpha + 3$ and later show interesting choices for these parameters. For the initial discussion we assume $\alpha := 0$ and $\gamma := 3$.

Algorithm Grisu

Input: positive floating-point number v of precision p

Preconditions: `diy_fp`'s precision q satisfies $q \geq p + 2$, and the powers-of-ten cache contains precomputed normalized rounded `diy_fp` values $\tilde{c}_k = \left[10^k\right]^\star_q$. We will determine k's necessary range shortly.

Output: a string representation in base 10 of V such that $[V]^\square_p = v$. That is, V would round to v when read again.

Procedure:

1. *Conversion*: determine the normalized `diy_fp` w such that $w = v$.
2. *Cached power*: find the normalized $\tilde{c}_{\text{-}k} = f_c \times 2^{e_c}$ such that $\alpha \leq e_c + e_w + q \leq \gamma$
3. *Product*: let $\tilde{D} = f_D \times 2^{e_D} := w \otimes \tilde{c}_{\text{-}k}$.
4. *Output*: define $V := \tilde{D} \times 10^k$. Produce the decimal presentation of $\tilde{D}$ followed by the string "e" and the decimal representation of k.

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 the exponent e_w satisfies $e_w \leq e_v - (q - p)$. With the exception of denormals we actually have $e_w = e_v - (q - p)$.

The sought $\tilde{c}_{\text{-}k}$ of step 2 must exist. It is easy to show that $\forall i,\ 0 < e_{\tilde{c}_i} - e_{\tilde{c}_{i\text{-}1}} \leq 4$ and since the cache is unbounded the required $\tilde{c}_{\text{-}k}$ has to be in the cache. This is the reason for the initial requirement $\gamma \geq \alpha + 3$.

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 α, γ. By way of example we will now show how to compute k_{min} and k_{max} for IEEE doubles, $q = 64$, and $\alpha = 0$, $\gamma = 3$.

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 values yields:

- $k_{min} :=$ `k_comp`$(960 + 64) = -289$, and
- $k_{max} :=$ `k_comp`$(-1137 + 64) = 342$.

In step 3 w is multiplied with $\tilde{c}_{\text{-}k}$. The goal of this operation is to obtain a `diy_fp` $\tilde{D}$ that has an exponent e_D such that $\alpha \leq e_D \leq \gamma$. Some configurations make the next step (output) easy. Suppose, for instance, that e_D becomes zero. Then $\tilde{D} = f_D$ and the decimal digits of $\tilde{D}$ can be computed by printing the significand f_D (a q-bit integer). With an exponent $e_D \neq 0$ the digit-generation becomes slightly more difficult, but since e_D's value is bounded by γ the computation is still straightforward.

Grisu's result is a string containing $\tilde{D}$'s decimal representation followed by the character "e" and k's digit. As such it represents the number $V := \tilde{D} \times 10^k$. We claim that V yields v when rounded to floating-point number of precision p.

Theorem 5.1. *Grisu's result V satisfies the internal identity requirement: $[V]^\square_p = v$.*

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

Using the equalities $v = w$, $V = w \otimes \tilde{c}_{\text{-}k} \times 10^k$, $10^k \times c_{\text{-}k} = 1$, and $m^+ - v = 2^{e_v - 1}$ this expands to $w \otimes \tilde{c}_{\text{-}k} - w \times c_{\text{-}k} < 2^{e_v - 1} \times c_{\text{-}k}$. Since, by hypothesis, $e_v \geq e_w + 2$ it is hence sufficient to show that $w \otimes \tilde{c}_{\text{-}k} - w \times c_{\text{-}k} < 2^{e_w + 1} \times c_{\text{-}k}$.

We have two cases:

- $f_c > 2^{q-1}$. By hypothesis $\tilde{c}_{\text{-}k}$'s error is bounded by .5 `ulp` and thus $c_{\text{-}k} \geq 2^{(q-1)+e_c}$. It suffices to show that $w \otimes \tilde{c}_{\text{-}k} - w \times c_{\text{-}k}$ is strictly less than $2^{e_w + q + e_c}$ which is guaranteed by Lemma 3.4.
- $f_c = 2^{q-1}$. Since the next lower `diy_fp` is only at distance $2^{e_c - 1}$ and $c_{\text{-}k}$ is rounded to nearest, $c_{\text{-}k}$'s error is bounded by $\frac{1}{4}$`ulp`. Clearly $c_{\text{-}k} \geq \left(2^{q-1} - \frac{1}{4}\right) \times 2^{e_c} \geq \frac{7}{8} \times 2^{(q-1)+e_c}$ for any $q \geq 2$. The inequality $w \otimes \tilde{c}_{\text{-}k} - w \times c_{\text{-}k} < \frac{7}{8} \times 2^{(e_w+1)+(q-1)+e_c}$ is (due to the smaller error of $\tilde{c}_{\text{-}k}$) guaranteed by Lemma 3.4.

We have proved the theorem for $V \geq v$. The remaining case $V < v$ can only happen when $\tilde{c}_{\text{-}k} < c_{\text{-}k}$. Now suppose:

- $f_v > 2^{p-1}$ and therefore $v - m^- = 2^{e_v - 1}$. The proof for this case is similar to the previous cases.

- $f_v = 2^{p-1}$ and therefore $v - m^- = 2^{e_v-2}$. Since f_v is even we only need to show $m^- \leq V$. Using similar steps as before it suffices to show that $2^{e_w+(q-1)+e_c} \leq w \otimes \tilde{c_{\text{-}k}} - w \times c_{\text{-}k}$ which is guaranteed by Lemma 3.5.

□

5.3 C Implementation

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.

```
 1: #define TEN7 10000000
 2: void cut(diy_fp D, uint32_t parts[3]) {
 3:   parts[2] = (D.f % (TEN7 >> D.e)) << D.e;
 4:   uint64_t tmp = D.f / (TEN7 >> D.e);
 5:   parts[1] = tmp % TEN7;
 6:   parts[0] = tmp / TEN7;
 7: }
 8: void grisu(double v, char* buffer) {
 9:   diy_fp w; uint32_t ps[3];
10:   int q = 64, alpha = 0, gamma = 3;
11:   w = normalize_diy_fp(double2diy_fp(v));
12:   int mk = k_comp(w.e + q, alpha, gamma);
13:   diy_fp c_mk = cached_power(mk);
14:   diy_fp D = multiply(w, c_mk);
15:   cut(D, ps);
16:   sprintf(buffer, "%u%07u%07ue%d",
17:           ps[0], ps[1], ps[2], -mk);
18: }
```

Figure 4: C implementation of Grisu with $\alpha,\gamma = 0,3$.

In Figure 4, line *8* we show the core `grisu` procedure specialized for $\alpha := 0$ and $\gamma := 3$. It accepts a non-special positive double 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).

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*).

Note that $2^{67} = 147573952589676412928$ has 21 digit. Three 7-digit integers will therefore always be sufficient to hold all decimal digits of `D`.

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.

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_t`s than `uint64_t`s. Furthermore the digits for each part can be computed independently which removes pipeline stalls.

5.4 Interesting target exponents

We will now discuss some interesting choices for α and γ. The most obvious choice $\alpha,\gamma := 0,3$ 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 $\alpha := 0$ to $\gamma := 9$. One simply has to cut `D` into three `uint32_t`s of 9 decimal digits each. As a consequence $\tilde{D}$'s decimal representation might need up to 27 digits.

On the one hand the bigger γ 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 $\tilde{c_{\text{-}k}}$ of step 2. Indeed, two cached powers-of-ten $\tilde{c_i}$ and $\tilde{c_{i+3}}$ will always satisfy $e_{\tilde{c_{i+3}}} - e_{\tilde{c_i}} \leq 10$.

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 inexact ones. Without additional changes to the core algorithm there is however little benefit in using such a heuristic.

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: $\alpha,\gamma := -63,-60$.

```
int digit_gen_no_div(diy_fp D, char* buffer) {
  int i = 0, q = 64; diy_fp one;
  one.f = ((uint64_t) 1) << -D.e; one.e = D.e;
  buffer[i++] = '0' + (D.f >> -one.e); //division
  uint64_t f = D.f & (one.f - 1); // modulo
  buffer[i++] = '.';
  while (-one.e > q - 5) {
    uint64_t tmp = (f << 2) & (one.f - 1);
    int d = f >> (-one.e - 3);
    d &= 6; f = f + tmp; d += f >> (-one.e - 1);
    buffer[i++] = '0' + d;
    one.e++; one.f >>= 1;
    f &= one.f - 1;
  }
  while (i < 19) {
    f *= 10;
    buffer[i++] = '0' + (f >> -one.e);
    f &= one.f - 1;
  }
  return i;
}
```

Figure 5: Digit generation for $\alpha = -63$ and $\gamma = -60$.

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.

Algorithm digit-gen-no-div
Input: a `diy_fp` D with exponent $-63 \leq e_D \leq -60$.
Output: a decimal representation of D.
Procedure:

1. *One*: determine the `diy_fp` *one* with $f_{one} = 2^{-e_D}$ and $e_{one} = e_D$.

2. *Digit0*: compute $d_0 := \lfloor \frac{D}{one} \rfloor$ and $D_1 := D \bmod one$

2b. *Ten*: If $d_0 \geq 10$ emit the digit "1" followed by the character representing $d_0 - 10$. Otherwise emit the character representing the digit d_0.

3. *Comma*: emit ".", the decimal separator.

4. *Digits*: generate and emit digits d_i as follows

- $d_i := \lfloor \frac{10 \times D_i}{one} \rfloor$

- emit the character representing the digit d_i
- $D_{i+1} := 10{\times}D_i \bmod one$

5. *Stop*: stop at the smallest positive integer *n* such that $D_n = 0$.

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

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 evaluates to d_0. We declare the following invariant for the loop of step 4: $D = R_i + \frac{D_{i+1}}{10^{-i}}$. Clearly the invariant holds for $i = 0$, and the invariant is still valid after the execution of the loop-body. We can hence conclude that $D = R_{n-1}$.

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 another 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 $1 \leq i \leq n$ 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.

Should $10{\times}D_i$ 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 thus be written as $\frac{D_i \times 10}{one} = \frac{f_{D_i} \times 10}{f_{one}} = \frac{4 \times f_{D_i} + f_{D_i}}{2^{-e_{one}-1}}$. From this equation it is then only a small step to the implementation in Figure 5.

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.

Our implementation takes some shortcuts compared to the described 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.

```
int digit_gen_mix(diy_fp D, char* buffer) {
  diy_fp one;
  one.f = ((uint64_t)1)<<-D.e; one.e = D.e;
  uint32_t part1 = D.f >> -one.e;
  uint64_t f = D.f & (one.f - 1);
  int i = sprintf(buffer, "%u", part1);
  buffer[i++] = '.';
  while (i < 19) {
    f *= 10;
    buffer[i++] = '0' + (f >> -one.e);
    f &= one.f - 1;
  }
  return i;
}
```

Figure 6: Digit generation for $\alpha = -59$ and $\gamma = -32$.

Finally one can mix both digit-generation techniques. The procedure in Figure 6 can be used for $\alpha,\gamma := -59,-32$. It combines the advantages of the previous approaches. It cuts the input number `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 straightforward code. Among the presented digit-generation procedures it also accepts the greatest range of exponents. Compared to the configuration $\alpha,\gamma = 0,3$ this version needs only a ninth of the cached powers. For completeness sake we now present its pseudo-algorithm:

Algorithm digit-gen-mix
Input: a `diy_fp` *D* with exponent $-59 \leq e_D \leq -32$.
Output: a decimal representation of *D*.
Procedure:

```
1. One: determine the diy_fp one with $f_{one} = 2^{-e_D}$ and $e_{one} = e_D$.
2. Parts: compute $part1 := \lfloor \frac{D}{one} \rfloor$ and $part2 := D \bmod one$
3. Integral: print the digits of part1.
4. Comma: emit ".", the decimal comma separator.
5. Fractional: let $D_0 := part2$. Generate and emit digits $d_i$ (for $i \geq 0$) as follows
    • $d_i := \lfloor \frac{10 \times D_i}{one} \rfloor$
    • emit the character representing the digit $d_i$
    • $D_{i+1} := 10{\times}D_i \bmod one$
6. Stop: stop at the smallest positive integer n such that $D_n = 0$.
```

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.

Note that the mixed approach can be easily extended to accept exponents in the range $\alpha,\gamma := -59,0$ 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.

We will base future evolutions of Grisu on `digit-get-mix` with $\alpha,\gamma = -59,-32$. 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.

6. Evolutions

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.

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 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.

For demonstration purposes we include *rounding* as an optimality requirement for Grisu3. It is simple to adapt Grisu2 so it rounds its outputs, too.

Finally we render Grisu2 and Grisu3 more flexible compared to Grisu. There are different ways to format a floating-point number. 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 formatting to a specialized procedure. Contrary to Grisu, Grisu2 and

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.

6.1 Idea

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.

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.

Definition 6.1. Let v be a positive real number and n, l and s be integers, such that $l \geq 1$, $10^{l-1} \leq s < 10^{l}$, $v = s \times 10^{n-l}$ 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*.

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 $m^- \leq V \leq m^+$ and such that V's significant length is minimal.

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:

- *tc1* is true when the produced number (consisting of digits $d_0 \ldots d_i$) is greater than m^-, and
- *tc2* is true when the rounded up number (consisting of digits $d_0 \ldots (d_i + 1)$) is less than m^+.

In the first case a rounded down number (of v) would be returned, whereas in the second case the result would be rounded up.

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.

We can show that this technique generates the shortest possible number.

Theorem 6.2. *Let x and y two real numbers, such that $x \leq y$. Let k be the greatest integer, such that $y \bmod 10^k \leq y - x$. Then $V := \lfloor \frac{y}{10^k} \rfloor \times 10^k$ satisfies $x \leq V \leq y$. Furthermore V's leading length l is the the smallest of all possible numbers in this interval: any number V' such that $x \leq V' \leq y$ has a leading length $l' \geq l$.*

Proof. We start by showing $x \leq V \leq y$: since $y = V + y \bmod 10^k$ we know that $V \leq y$. Also $y \bmod 10^k \leq y - x$ and therefore $V \geq x$.

For the sake of contradiction assume that there exists a V' with leading length l', such that $x \leq V' \leq y$ and $l' < l$.

The number V' has a leading length of l' and by definition there exists hence an s', and n' such that $10^{l'-1} \leq s' < 10^{l'}$ and $V' = s' \times 10^{n'-l'}$.

There are three cases to consider:

1. $s' > \lfloor \frac{y}{10^{n'}} \rfloor$: impossible since this implies $V > y$.
2. $s' = \lfloor \frac{y}{10^{n'}} \rfloor$: contradiction, since this implies $V = V'$.
3. $s' < \lfloor \frac{y}{10^{n'}} \rfloor$: we first prove the case for $n' > k$.
   By hypothesis k is maximal and hence $y \bmod 10^{n'} > y - x$. Given that $y \bmod 10^{n'} = y - \lfloor \frac{y}{10^{n'}} \rfloor \times 10^{n'}$ we can conclude that $y - s \times 10^{n'} > y - x$ and thus $V' < x$. Contradiction.
   Suppose now that $n' \leq k$. By definition of "leading length" we know that $V \geq 10^{l-1} \times 10^k$ and $V' < 10^{l'} \times 10^{n'}$. Since $l' < l$ we have $V' < 10^{l-1+k} \leq V$. Also $x \leq V'$ and $V \leq y$ and therefore $x < 10^{l-1+k} \leq y$. Clearly $y - 10^{l-1+k} < y - x$ and thus, using the same equality as before, $y \bmod 10^{l-1+k} \leq y - x$ which contradicts the minimality property of k.

□

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 requires 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: $q \geq p + 3$.

Algorithm Grisu2

Input: same as for Grisu.

Preconditions: `diy_fp`'s precision q satisfies $q \geq p + 3$, and the powers-of-ten cache contains precomputed normalized rounded `diy_fp` values $\tilde{c}_k = [10^k]_q^{\star}$.

Output: decimal digits d_i for $i \leq 0 \leq n$ and an integer K such that the real $V := d_0 \ldots d_n \times 10^K$ verifies $[V]_p^{\square} = v$.

Procedure:

1. *Boundaries*: compute v's boundaries m^- and m^+.
2. *Conversion*: determine the normalized `diy_fp` w^+ such that $w^+ = m^+$. Determine the `diy_fp` w^- such that $w^- = m^-$ and that $e_w^- = e_w^+$.
3. *Cached Power*: find the normalized $\tilde{c}_{-k} = f_c \times 2^{e_c}$ such that $\alpha \leq e_c + e_w^+ + q \leq \gamma$ (with α and γ as discussed for Grisu).
4. *Product*: compute $\tilde{M}^- := w^- \otimes \tilde{c}_{-k}$, $\tilde{M}^+ := w^+ \otimes \tilde{c}_{-k}$, and let $M_{\uparrow}^- := \tilde{M}^- + 1\,\mathtt{ulp}$, $M_{\downarrow}^+ := \tilde{M}^+ - 1\,\mathtt{ulp}$, $\delta := M_{\downarrow}^+ - M_{\uparrow}^-$.
5. *Digit Length*: find the greatest κ such that $M_{\downarrow}^+ \bmod 10^{\kappa} \leq \delta$ and define $P := \left\lfloor \frac{M_{\downarrow}^+}{10^{\kappa}} \right\rfloor$.
6. *Output*: define $V := P \times 10^{k+\kappa}$. The decimal digits d_i and n are obtained by producing the decimal representation of P (an integer). Set $K := k + \kappa$, and return it with the n digits d_i.

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 $M_{\uparrow}^- \leq M_{\downarrow}^+$.

[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.

Lemma 6.3. *The variables $M_{\uparrow}^{-}$ and $M_{\downarrow}^{+}$ as described in step 4 verify $M_{\uparrow}^{-} \leq M_{\downarrow}^{+}$.*

Proof. By definition

$$\begin{aligned} M_{\uparrow}^{-} &= \tilde{M^{-}} + 1\,\mathtt{ulp} \\ &= w^{-} \otimes \tilde{c_{-k}} + 1\,\mathtt{ulp} \\ &= \left(\left[\frac{f_{w^-} \times f_c}{2^q} \right]^{\uparrow} + 1 \right) \times 2^{e_w + e_c + q} \\ &\leq \left(\frac{f_{w^-} \times f_c}{2^q} + 1.5 \right) \times 2^{e_w + e_c + q} \end{aligned}$$

and similarly $M_{\downarrow}^{+} = \left(\frac{f_{w^+} \times f_c}{2^q} - 1.5 \right) \times 2^{e_w + e_c + q}$.

Since $f_{w^+} \geq f_{w^-} + 2^{q-p-1} + 2^{q-p-2}$ it suffices to show

$$\begin{aligned} \frac{f_{w^-} \times f_c}{2^q} + 1.5 &\leq \frac{\left(f_{w^-} + 2^{q-p-1} + 2^{q-p-2}\right) \times f_c}{2^q} - 1.5 \quad \text{or} \\ 3 &\leq \frac{f_c \times \left(2^{q-p-1} + 2^{q-p-2}\right)}{2^q} \end{aligned}$$

Using the inequalities $f_c \geq 2^{q-1}$ and $q - p \geq 3$ it is sufficient to show $3 \leq \frac{2^{q-1} \times \left(2^2 + 2^1\right)}{2^q}$. □

Theorem 6.4. *Grisu2's result $V = d_0 \ldots d_n \times 10^K$ satisfies the internal identity requirement: $[V]_p^{\square} = v$.*

Proof. We will show that $m^- < M_{\uparrow}^{-} \times 10^k \leq V \leq M_{\downarrow}^{+} \times 10^k < m^+$ (with m^- and m^+ v's boundaries).

The inner inequality, $M_{\uparrow}^{-} \times 10^k \leq V \leq M_{\downarrow}^{+} \times 10^k$, is a consequence of Theorem 6.2. Remains to show that $m^- < M_{\uparrow}^{-} \times 10^k$ and $M_{\downarrow}^{+} \times 10^k < m^+$.

By Lemma 3.4 $\tilde{M^-}$ and $\tilde{M^+}$ have an error of strictly less than 1 `ulp`, and therefore $m^- < M_{\uparrow}^{-} \times 10^k$ and $M_{\downarrow}^{+} \times 10^k < m^+$. As a consequence $m^- < V < m^+$. □

Grisu2 does not give any guarantees on the shortness of its result. Its result is the shortest possible number in the interval $M_{\uparrow}^{-} \times 10^k$ to $M_{\downarrow}^{+} \times 10^k$ (boundaries included), where $M_{\uparrow}^{-}$ and $M_{\downarrow}^{+}$ are dependent on `diy_fp`'s precision q. The higher q, the closer $M_{\uparrow}^{-}$ and $M_{\downarrow}^{+}$ are to the actual boundaries m^- and m^+. For $q = 64$ and $p = 53$ (as in our code samples) Grisu2 produces the shortest number for approximately 99.9% of its input.

The C implementation of Grisu2 is again cut into two parts: a core routine, independent of the chosen α/γ, 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.

In Figure 7 we present a version of the digit-generation routine tuned for $\alpha,\gamma = -59,-32$. The input variables `Mp`, and `delta` correspond to $M_{\downarrow}^{+}$ and δ 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 κ.

The proposed implementation combines step 5 and 6. While trying all possible κs (starting from the "top") it generates the digits of $\left\lfloor \frac{M_{\downarrow}^{+}}{10^{\kappa}} \right\rfloor$. There are two digit-generation loops. One for the most-significant digits $\left\lfloor \frac{\mathtt{Mp}}{\mathtt{one}} \right\rfloor$, 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 both loops (line *9* and line *17*): $R = \left\lfloor \frac{\mathtt{Mp}}{10^{\mathtt{kappa}}} \right\rfloor$. For the first loop we can show that $\mathtt{p1} \times \mathtt{one} + \mathtt{p2} = \mathtt{Mp} \bmod 10^{\mathtt{kappa}}$. The equation in line *13* thus tests if $M_{\downarrow}^{+} \bmod 10^{\kappa} \leq \delta$.

The following invariant holds for the second loop (line *17*): $p2 = \frac{\mathtt{Mp}}{10^{\mathtt{kappa}}} \bmod one$.

```
 1: #define TEN9 1000000000
 2: void digit_gen(diy_fp Mp, diy_fp delta,
 3:                char* buffer, int* len, int* K) {
 4:   uint32_t div; int d,kappa; diy_fp one;
 5:   one.f = ((uint64_t) 1) << -Mp.e; one.e = Mp.e;
 6:   uint32_t p1 = Mp.f >> -one.e;
 7:   uint64_t p2 = Mp.f & (one.f - 1);
 8:   *len = 0; kappa = 10; div = TEN9;
 9:   while (kappa > 0) {
10:     d = p1 / div;
11:     if (d || *len) buffer[(*len)++] = '0' + d;
12:     p1 %= div; kappa--; div /= 10;
13:     if ((((uint64_t)p1)<<-one.e)+p2 <= delta.f) {
14:       *K += kappa; return;
15:     }
16:   }
17:   do {
18:     p2 *= 10;
19:     d = p2 >> -one.e;
20:     if (d || *len) buffer[(*len)++] = '0' + d;
21:     p2 &= one.f - 1; kappa--; delta.f *= 10;
22:   } while (p2 > delta.f);
23:   *K += kappa;
24: }
```

Figure 7: Grisu2's digit generation routine (for $\alpha,\gamma = -59,-32$).

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.

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 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.

Algorithm Grisu3

Input and preconditions: same as for Grisu2.

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

Procedure:

1-2. same as for Grisu2.

2b. *Conversion*: determine the normalized `diy_fp` w such that $w = v$.

3-4. same as for Grisu2.

4b. *Product2*: let $M_{\downarrow}^{-} := \tilde{M^{-}} - 1\,\mathtt{ulp}$, $M_{\uparrow}^{+} := \tilde{M^{+}} + 1\,\mathtt{ulp}$, and $\Delta := M_{\uparrow}^{+} - M_{\downarrow}^{-}$.

5. *Digit Length*: find the greatest κ such that $M^{+}_{\uparrow} \bmod 10^{\kappa} < \Delta$.
6. *Round*: compute $\tilde{W} := w \otimes \tilde{c_{-k}}$, and let $W_{\downarrow} := \tilde{W} - 1\,\texttt{ulp}$, and $W_{\uparrow} := \tilde{W} + 1\,\texttt{ulp}$. Set $P_i := \left\lfloor \frac{M^{+}_{\uparrow}}{10^{\kappa}} \right\rfloor - i$ for $i \geq 0$. Let m be the greatest integer that verifies $P_m \times 10^{\kappa} > M^{-}_{\downarrow}$.

   Let u, $0 \leq u \leq m$ the smallest integer such that $|P_u \times 10^{\kappa} - W_{\uparrow}|$ is minimal. Similarly let d, $0 \leq d \leq m$ the largest integer such that $|P_d \times 10^{\kappa} - W_{\downarrow}|$ is minimal.

   If $u \neq d$ return `failure`, else set $P := P_u$.
7. *Weed*: if not $M^{-}_{\uparrow} \leq P \times 10^{\kappa} \leq M^{+}_{\downarrow}$ return `failure`.
8. *Output*: define $V := P \times 10^{k+\kappa}$. The decimal digits d_i and n are obtained by producing the decimal representation of P (an integer). Set $K := k + \kappa$, and return it with the n digits d_i.

Grisu3 uses the liberal boundary approximations ($M^{-}_{\downarrow}$ and $M^{+}_{\uparrow}$) instead of the conservative ones ($M^{-}_{\uparrow}$ and $M^{+}_{\downarrow}$). 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 $M^{-}_{\downarrow}$ is excluded from consideration. There is however no mechanism to exclude $M^{+}_{\uparrow}$. In the rare event when $M^{+}_{\uparrow} \bmod 10^{\kappa} = 0$ then Grisu3 will, at this point of the algorithm, wrongly assume that $M^{+}_{\uparrow}$ is a potentially valid representation of v. Since $M^{+}_{\uparrow}$ 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.

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 and $\tilde{W}$, the approximation of v, may have an error of up to 1 `ulp`. The closest representation $P \times 10^{\kappa}$ must not only be the closest to $\tilde{W}$ but to all possible values in $\tilde{W}$'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.

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.

```
 1: bool round_weed(char* buffer, int len,
 2:                 uint64_t wp_W, uint64_t Delta,
 3:                 uint64_t rest, uint64_t ten_kappa,
 4:                 uint64_t ulp) {
 5:   uint64_t wp_Wup = wp_W - ulp;
 6:   uint64_t wp_Wdown = wp_W + ulp;
 7:   while (rest < wp_Wup &&
 8:          Delta - rest >= ten_kappa &&
 9:          (rest + ten_kappa < wp_Wup ||
10:          wp_Wup-rest >= rest+ten_kappa - wp_Wup))
11:   {
12:     buffer[len-1]--; rest += ten_kappa;
13:   }
14:   if (rest < wp_Wdown &&
15:       Delta - rest >= ten_kappa &&
16:       (rest + ten_kappa < wp_Wdown ||
17:        wp_Wdown-rest > rest+ten_kappa - wp_Wdown))
18:     return false;
19:   return 2*ulp <= rest && rest <= Delta - 4*ulp;
20: }
```

Figure 8: Grisu3's round-and-weed procedure.

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 combines step 6 and 7. The function is invoked with the parameters set to the following values: `buffer` $:= d_0 \ldots d_{len-1}$ where $d_0 \ldots d_{len-1}$ are the decimal digits of $\left\lfloor \frac{W^{+}_{\uparrow}}{10^{\kappa}} \right\rfloor$, `wp_W` $:= \tilde{W^{+}} - \tilde{W}$, `Delta` $:= \Delta$, `rest` $:= W^{+}_{\uparrow} \bmod 10^{\kappa}$, `ten_kappa` $:= 10^{\kappa}$, and `ulp` the value of 1 `ulp` relative to all passed `diy_fps`.

Let $T := d_0 \ldots d_{len-1} \times 10^{\kappa}$. By construction T lies within the unsafe interval: $W^{-}_{\downarrow} < T \leq W^{+}_{\uparrow}$. 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.

The loop in line *7* iteratively tests all possible alternatives to find the closest to $W_{\uparrow}$. The first test, `rest < wp_W - ulp`, ensures that T is not already less than $W_{\uparrow}$ (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 $W^{-}_{\downarrow}$ to $W^{+}_{\uparrow}$. In line *9* the procedure tests if the alternative would be closer to $W_{\uparrow}$. If all tests succeed, then the number $d_0 \ldots d_{len-1} \times 10^{\kappa}$ is guaranteed to lie inside the interval $W^{-}_{\downarrow}$ to $W^{+}_{\uparrow}$ and is furthermore closer to $W_{\uparrow}$ than the current T. The body of the loop thus replaces T (physically modifying the buffer) with its smaller alternative.

The `if` in line *14* then verifies the chosen T is also closest to $W^{-}_{\downarrow}$. If this check fails then there are at least two candidates that could be the closest to $\tilde{W}$ and Grisu3 returns `failure`.

Now that the buffer has been correctly rounded a final weeding test in line *19* verifies that $W^{-}_{\uparrow} \leq T \leq W^{+}_{\downarrow}$. That is, that the chosen T is inside the safe conservative interval.

7. Benchmarks

Algorithm	R	R/PP	S	S/PP
sprintf %c	2.60	-	-	-
sprintf %g	22.83	-	24.17	-
sprintf %.17e	36.03	-	36.17	-
burger/dybvig	61.53	61.49	28.73	28.66
grisu Fig4	9.17	-	9.98	-
grisu 0,3	2.85	-	3.26	-
grisu -63,-59	3.36	-	3.77	-
grisu -35,-32	2.66	-	3.04	-
grisu -59,-56	2.50	-	2.96	-
grisu2 -59,-56	3.80	4.88	3.07	4.04
grisu2b -59,-56	5.38	6.42	4.40	5.48
grisu3 -59,-56	4.47	5.61	3.49	4.55

Figure 9: Speed of `sprintf`, Burger/Dybvig and Grisu.

Algorithm	optimal	shortest
grisu2 -59,-56	0	99.92
grisu2b -59,-56	99.85	99.92
grisu3 -59,-56	99.49	99.49

Figure 10: Optimality of Grisu2 and Grisu3.

In this section we present some experimental results. In Figure 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 unoptimized algorithm of Figure 4. Grisu2b ("grisu2b -35,-32") is a variant of Grisu2 where the result is rounded.

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.

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`.

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.

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 presentations are the shortest, and once a rounding-phase has been added (Grisu2b) 99.87% of the numbers are optimal. Grisu3 produces optimal results for 99.49% of its input and rejects the rest.

8. Related Work

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.

In his thesis Coonen furthermore describes space efficient algorithms 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).

In 1990 Steele and White published their printing-algorithm, Dragon, [Steele Jr. and White(1990)]. A draft of this paper had existed for many years and had already been cited in "Knuth, Volume II" [Knuth(1981)] (1981). Dragon4 is an exact algorithm and requires 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 shortening the generated sequence.

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.

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

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

9. Conclusion

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.

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.

Grisu3 should be used when the optimal representation is required. In this case Grisu3 will efficiently produce the optimal result 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.

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.

References

[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.

[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.

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

[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.

[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.

[Knuth(1981)] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. 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.

[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 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.