Dragonbox: A New Floating-Point Binary-to-Decimal Conversion Algorithm

Junekey Jeon

The Department of Mathematics
University of California, San Diego
USA
j6jeon@ucsd.edu

Abstract

We present a new algorithm for efficiently converting a binary floating-point number into the shortest and correctly rounded decimal representation. The algorithm is based on *Schubfach* algorithm [1] introduced in around 2017-2018, and is also inspired from Grisu [2] and Grisu-Exact [4]. In addition to the core idea of Schubfach, Dragonbox utilizes some Grisu-like ideas to minimize the number of expensive 128-bit $\times$ 64-bit multiplications, at the cost of having more branches and divisions-by-constants. According to our benchmarks, Dragonbox performs better than Ryū, Grisu-Exact, and Schubfach for both IEEE-754 binary32 and binary64 formats.

0. Disclaimer

This paper is not a completely formal writing, and is not intended for publications into peer-reviewed conferences or journals. The paper might contain some alleged claims and/or lack of references.

1. Introduction

Due to recent popularity of JavaScript and JSON, interest on fast and correct algorithm for converting between binary and decimal representations of floating-point numbers has been continuously increasing. As a consequence, many new algorithms have been proposed recently, in spite of the long history of the subject.

We will assume all floating-point numbers are in either IEEE-754 binary32 or binary64 formats, as these are the most common formats used today.[1][2] We will also focus on the binary-to-decimal conversion in this paper and will not discuss how to do decimal-to-binary conversion. Contrary to one might think, in fact decimal-to-binary conversion and binary-to-decimal conversion are largely asymmetric, because of the asymmetric nature of input and output. In general, for the input side, one needs to deal with wide variety of possible input data, but the form of output is usually definitive. On the other hand, for the output side, the input data has a strict format but one needs to choose between various possibilities of outputs. Floating-point I/O is not an exception. When it comes to decimal-to-binary conversion, which corresponds to the input side, the input data can be usually arbitrarily long so we have to somehow deal with that, but any input data can, if not malformed, usually represent a unique floating-point number. On the other hand, in binary-to-decimal conversion, which corresponds to the output side, the input is a single binary floating-point number but the output can be all decimal numbers which any correct parser will read as the original binary floating-point number. To resolve this ambiguity, Steele and White proposed the following criteria in [6]:[3]

1. **Information preservation**: a correct decimal-to-binary converter must return the original binary floating-point number,
2. **Minimum-length output**: the output decimal significand should be as short as possible, and
3. **Correct rounding**: among all possible shortest outputs, the one that is closest to the true value of the given floating-point number should be chosen.

[1] Details of these formats will be reviewed in Section 2

[2] It should be not so difficult to generalize Dragonbox to similar formats, such as IEEE-754 binary16 or binary128.

[3] To be precise, the criteria given by Steele and White were in terms of the character string generated from the decimal representation. However, we can write those criteria in terms of the decimal representation itself as well.

Notable examples of recently proposed binary-to-decimal conversion algorithms include but not limited to Grisu [2], Errol [3], Ryū [5], and Grisu-Exact [4]. Among these, Errol, Ryū, and Grisu-Exact satisfy all of the above criteria. Grisu does not satisfy all of the criteria, but Grisu3, which can detect its failure to satisfy the criteria, with the fallback into Dragon4 [6], proposed by Steele and White and satisfies all the criteria, is still popular.

Schubfach [1] is another example of those algorithms, developed in around 2017-2018, but it seems that, compared to Ryū, it did not get much attention from the public probably because at that time there was no document explaining details of the algorithm. Nevertheless, the underlying idea of Schubfach is theoretically very appealing and its implementation [7] also seems to outperform that of the other algorithms.

Although Schubfach is already a very tight algorithm, there can be ways to improve its performance further. One possible way might be to eliminate the necessity to perform three 128-bit × 64-bit multiplications all the time. The core idea of Dragonbox is to achieve this by applying some Grisu-like ideas to Schubfach.

2. IEEE-754 Specifications[4]

Before diving into the details of Dragonbox, let us review IEEE-754 and fix some related notations. For a real number w, by *(binary) floating-point representation* we mean the representation

$$w = (-1)^{\sigma_w} \cdot F_w \cdot 2^{E_w}$$

where $\sigma_w = 0, 1$, $0 \leq F_w < 2$, and E_w is an integer. We say the above representation is *normal* if $1 \leq F_w < 2$. Of course, there is no normal floating-point representation of 0, while any other real number has a unique normal floating-point representation. If the representation is not normal, we say it is *subnormal*.

IEEE-754 specifications consist of the following rules that define a mapping from the set of fixed-length bit patterns $b_{q-1}b_{n-2} \ \cdots \ b_0$ for some q into the real line augmented with some special values:

1. The most-significant bit b_{q-1} is the sign σ_w.
2. The least-significant p-bits $b_{p-1} \ \cdots \ b_0$ are for storing the significand F_w, while the remaining $(q-p-1)$-bits are for storing the exponent E_w. We call p the *precision* of the representation.[5]
3. If $q-p-1$ exponent bits are not all-zero nor all-one, the representation is normal. In this case, we compute F_w as
$$F_w = 1 + 2^{-p} \cdot \sum_{k=0}^{p-1} b_k \cdot 2^k$$
and E_w as
$$E_w = -(2^{q-p-2} - 1) + \sum_{k=0}^{q-p-2} b_{p+k} \cdot 2^k.$$
The constant term $2^{q-p-2} - 1$ is called the *bias*, and we denote this value as $E_{\max} \coloneqq 2^{q-p-2} - 1$.
4. If $q-p-1$ exponent bits are all-zero, the representation is subnormal. In this case, we compute F_w as
$$F_w = 2^{-p} \cdot \sum_{k=0}^{p-1} b_k \cdot 2^k$$
and let $E_w = -(2^{q-p-2} - 2)$. Let us denote this value of E_w as $E_{\min} \coloneqq -(2^{q-p-2} - 2)$.
5. If $q-p-1$ exponent bits are all-one, the pattern represents either $\pm\infty$ when all of p significand bits are zero, or NaN's (Not-a-Number) otherwise.

When $(q, p) = (32, 23)$, the resulting encoding format is called *binary32*, and when $(q, p) = (64, 52)$, the resulting encoding format is called *binary64*.

For simplicity, let us only consider bit patterns corresponding to positive real numbers from now on. Zeros, infinities, and NaN's should be treated specially, and for negative numbers, we can simply ignore the sign until the final output string is generated. Hence, for example, we do not think of all-zero nor all-one patterns, and especially exponent bits are never all-one. Also, we always assume that the sign bit is 0. With these assumptions, the mapping defined above is one-to-one: each bit pattern corresponds to a unique real number, and no different bit patterns correspond to a same real number.

From now on, by saying $w = F_w \cdot 2^{E_w}$ a *floating-point number* we implicitly assumes that

(1) w is a positive number representable within an IEEE-754 binary format with some q and p, and

(2) F_w and E_w are those obtained from the rules above.

In particular, the representation is normal ($1 \leq F_w < 2$) if $E_w \neq E_{\min}$ and is can be subnormal ($0 \leq F_w < 1$) only if $E_w = E_{\min}$. If the representation is normal, we call w a *normal number*, and for otherwise, we call w a *subnormal number*.

For a floating-point number $w = F_w \cdot 2^{E_w}$, we define w^- as the greatest floating-point number smaller than w. When w is the minimum possible positive floating-number representable within the specified encoding format, that is, $w = 2^{-p} \cdot 2^{E_{\min}}$, then we define $w^- = 0$. Similarly, we define w^+ as the smallest floating-point number greater than w. Again, if w is the largest possible finite number representable within the format, that is, $w = (2 - 2^{-p})2^{E_{\max}}$, then we define $w^+ \coloneqq 2^{E_{\max}+1}$.

[4] This section is mostly copied from [4].

[5] Usually, it is actually $p+1$ that is called the precision of the format in other literatures. However, we call p the precision in this paper for simplicity.

In general, it can be shown that

$$w^- = \begin{cases} (F_w - 2^{-p-1})2^{E_w} & \text{if } F_w = 1 \text{ and } E_w \neq E_{\min} \\ (F_w - 2^{-p})2^{E_w} & \text{otherwise} \end{cases}$$

and

$$w^+ = (F_w + 2^{-p})2^{E_w}.$$

We will also use the notations

$$m_w^- := \frac{w^- + w}{2} = \begin{cases} (F_w - 2^{-p-2})2^{E_w} & \text{if } F_w = 1 \text{ and} \\ & E_w \neq E_{\min} \\ (F_w - 2^{-p-1})2^{E_w} & \text{otherwise} \end{cases},$$

$$m_w^+ := \frac{w + w^+}{2} = (F_w + 2^{-p-1})2^{E_w}$$

to denote the midpoints of the intervals $[w^-, w]$, $[w, w^+]$, respectively.

2.1 Rounding Modes

Floating-point calculations are inherently imprecise as the available precision is limited. Hence, it is necessary to round calculational results to make them fit into the precision limit. Specifying how any rounding should be performed means to define for each real number a corresponding floating-point number in a consistent way. IEEE-754 currently defines five rounding modes. We can describe those rounding modes by specifying the inverse image in the real line of each floating-point number w:

1. *Round to nearest, ties to even*: If the LSB (Least Significant Bit) of the significand bits of w is 0, then the inverse image is the closed interval $[m_w^-, m_w^+]$. Otherwise, it is the open interval (m_w^-, m_w^+). This is the default rounding mode in most of the platforms. In fact, it is required to be the default mode for binary encodings.
2. *Round to nearest, ties away from zero*: The inverse image of w is the half-open interval $[m_w^-, m_w^+)$. This mode is introduced in the 2008 revision of the IEEE-754 standard. Some platforms and languages, such as the recent standards of the C and C++ languages, do not have the corresponding way of representing this rounding mode.
3. *Round toward* 0: The inverse image of w is the half-open interval $[w, w^+)$.
4. *Round toward* $+\infty$: The inverse image of w is the half-open intervals $(w^-, w]$ if w is positive, and $[w, w^+)$ if w is negative.[6]
5. *Round toward* $-\infty$: The inverse image of w is the half-open intervals $[w, w^+)$ if w is positive, and $(w^-, w]$ is w is negative.

[6] We supposed to deal only with positive numbers, so w here is actually a positive number. The phrases "if w is positive" or "if w is negative" simply mean that the original input is positive or negative, respectively.

Though not included in the IEEE-754 standard, we can think of the following additional rounding modes with their obvious meanings:

- *Round to nearest, ties to odd*
- *Round to nearest, ties toward zero*
- *Round to nearest, ties toward* $+\infty$
- *Round to nearest, ties toward* $-\infty$
- *Round away from* 0

Note that if I is the interval given as the inverse image of w according to a given rounding mode, then a correct decimal-to-binary converter must output w from any numbers in I. Therefore, in order to produce a shortest possible decimal representation of w, we need to search for a number inside I that has the least number of decimal significand digits.

2.2 Notations

From now on, we will assume that a floating-point number w and a specific rounding mode is given so the interval I is defined accordingly. Note that for all cases I is an interval contained in the positive real axis and it avoids 0. We will denote the left and the right endpoints of I as w_L and w_R, respectively. For example, when one of the round-to-nearest rounding mode is specified, $w_L = m_w^-$ and $w_R = m_w^+$. We will also denote the length of I as $\Delta := w_R - w_L$. Note that there are only three possible values of Δ:

1. $\Delta = 2^{E_w-p-1}$, if $w_L = w^-$, $w_R = w$, $F_w = 1$, and $E_w \neq E_{\min}$,
2. $\Delta = 3 \cdot 2^{E_w-p-2}$ if $w_L = m_w^-$, $w_R = m_w^+$, $F_w = 1$, and $E_w \neq E_{\min}$, and
3. $\Delta = 2^{E_w-p}$ for all other cases.

We also denote

$$e := E_w - p, \quad f_c := F_w 2^p$$

so that f_c is an integer and

$$\begin{aligned} w &= f_c \cdot 2^e, \\ w^- &= \begin{cases} \left(f_c - \frac{1}{2}\right) \cdot 2^e & \text{if } F_w = 1 \text{ and } E_w \neq E_{\min} \\ (f_c - 1) \cdot 2^e & \text{otherwise} \end{cases}, \\ w^+ &= (f_c + 1) \cdot 2^e, \\ m_w^- &= \begin{cases} \left(f_c - \frac{1}{4}\right) \cdot 2^e & \text{if } F_w = 1 \text{ and } E_w \neq E_{\min} \\ \left(f_c - \frac{1}{2}\right) \cdot 2^e & \text{otherwise} \end{cases}, \\ m_w^+ &= \left(f_c + \frac{1}{2}\right) \cdot 2^e. \end{aligned}$$

With this notation, Δ is one of 2^{e-1}, $3 \cdot 2^{e-2}$, or 2^e.

3. Review of Schubfach

In this section, we will briefly review how Schubfach works. Most of the results are from [1], but we changed the notations and formulations, and also rewrote the proofs to help understanding the rest of our paper.

The beauty of Schubfach is that, not like Ryū or Grisu-Exact, it does not perform an iterative search to find the shortest decimal representation. Rather, Schubfach finds it with just one trial using the following simple fact:[7]

Proposition 3.1.
Let $k_0 := -\lfloor \log_{10} \Delta \rfloor$. *Then*

1. $\left| I \cap 10^{-k_0+1}\mathbb{Z} \right| \leq 1$, *and*
2. $\left| I \cap 10^{-k_0}\mathbb{Z} \right| \geq 1$.[8]

where $|\cdot|$ *denotes the cardinality the set and for any* $a \in \mathbb{R}$ *and* $A \subseteq \mathbb{R}$, aA *denotes the set* $\{av \colon v \in A\}$.

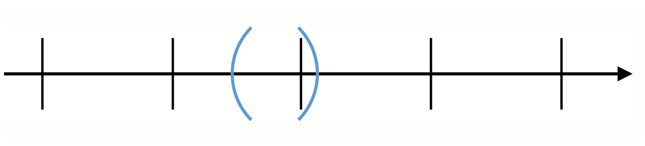

Figure 1. If I is shorter than the unit, then it contains at most one lattice point

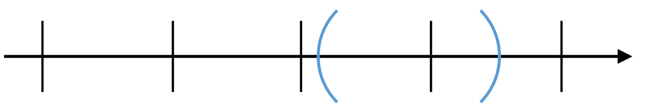

Figure 2. If I is longer than the unit, then it contains at least one lattice point

Proof. By definition of k_0, we have

$$-k_0 \leq \log_{10} \Delta < -k_0 + 1,$$

or equivalently,

$$10^{-k_0} \leq \Delta < 10^{-k_0+1}.$$

If $\left| I \cap 10^{-k_0+1}\mathbb{Z} \right| > 1$, then it means there are at least two distinct points in I which are apart from each other by distance 10^{-k_0+1}. Hence, the length of I should be at least 10^{-k_0+1}, or equivalently,

$$\Delta \geq 10^{-k_0+1},$$

which is a contradiction. This shows the first claim.

On the other hand, pick any point $v \in I$, then we know

$$\lfloor 10^{k_0} v \rfloor \leq 10^{k_0} v < \lfloor 10^{k_0} v \rfloor + 1.$$

We claim that at least one of $\lfloor 10^{k_0} v \rfloor$ and $\lfloor 10^{k_0} v \rfloor + 1$ is in $10^{k_0} I$. Suppose not, then the left endpoint of $10^{k_0} I$ should lie inside $\left[\lfloor 10^{k_0} v \rfloor, 10^{k_0} v \right]$ and the right endpoint of $10^{k_0} I$ should lie inside $\left[10^{k_0} v, \lfloor 10^{k_0} v \rfloor + 1 \right]$. This implies that the length of $10^{k_0} I$ is at most 1, but since $10^{-k_0} \leq \Delta$, it follows that $\Delta = 10^{-k_0}$ and $10^{k_0} I = \left(\lfloor 10^{k_0} v \rfloor, \lfloor 10^{k_0} v \rfloor + 1 \right)$.

Note that $\Delta = 10^{-k_0}$ is only possible for very rare cases; indeed, since 5 does not appear as a prime factor of Δ (as a rational number), the equality $\Delta = 10^{-k_0}$ can hold only when $k_0 = 0$. Hence, we have $\Delta = 1$, which can hold only when $e = 1$ or $e = 0$ because Δ is one of 2^{e-1}, $3 \cdot 2^{e-2}$, or 2^e, depending on how I is given.[9] However, this implies that $w = f_c \cdot 2^e$ is an integer, but since $w \in I$, we get that $I \cap \mathbb{Z} \neq \emptyset$. This is absurd, because I is an open interval between two consecutive integers. □

It should be noted that the shortest decimal numbers in I are the elements of the intersection $I \cap 10^{-k}\mathbb{Z}$ where k is the smallest integer making the intersection nonempty. Although this sounds obvious, let us formally prove it. First, we define the number of decimal significand digits of a nonzero real number v as $\lfloor \log_{10}(v \cdot 10^k) \rfloor + 1$ where k is the smallest integer such that $v \cdot 10^k \in \mathbb{Z}$. For example,

- If $v = 1.23$, then $k = 2$ and $\lfloor \log_{10}(v \cdot 10^k) \rfloor + 1 = 3$,
- If $v = 0.01234$, then $k = 5$ and $\lfloor \log_{10}(v \cdot 10^k) \rfloor + 1 = 5$, and
- If $v = 1200$, then $k = -2$ and $\lfloor \log_{10}(v \cdot 10^k) \rfloor + 1 = 2$.

Proposition 3.2.
The set $I \cap 10^{-k}\mathbb{Z}$, *where* k *is the smallest integer making the intersection nonempty, is precisely the set of elements in* I *with the smallest number of decimal significand digits..*

Proof. By the assumption on k, we know that $I \cap 10^{-k}\mathbb{Z}$ is not empty while $I \cap 10^{-k+1}\mathbb{Z}$ is empty. Equivalently, $10^k I \cap \mathbb{Z}$ is not empty while $10^{k-1} I \cap \mathbb{Z}$ is empty. Since I is an interval, $10^k I \cap \mathbb{Z} = \{m, m+1, \cdots, M-1, M\}$ for some integers $m, M \in \mathbb{Z}$. Since $10^{k-1} I \cap \mathbb{Z}$ is empty, there is no multiple of 10 among $m, \cdots, M$. Hence, we get $\lfloor \log_{10} m \rfloor = \lfloor \log_{10} M \rfloor$; otherwise, we have

$$\begin{aligned} \log_{10} m &< \lfloor \log_{10} m \rfloor + 1 \\ &\leq \lfloor \log_{10} M \rfloor \leq \log_{10} M, \end{aligned}$$

thus

$$m < 10^{\lfloor \log_{10} m \rfloor + 1} \leq M,$$

which contradicts to that there is no multiple of 10 among $m, \cdots, M$. Note that for any v in the set

$$I \cap 10^{-k}\mathbb{Z} = \left\{ 10^{-k} m, \cdots, 10^{-k} M \right\},$$

k is the smallest integer such that $v \cdot 10^k$ is an integer, thus all such v have $\lfloor \log_{10} m \rfloor + 1$ decimal significand digits.

[7] One might regard this proposition as a form of the pigeonhole principle. In fact, the name *Schubfach* is coming from the German name of the pigeonhole principle, *Schubfachprinzip*, meaning "drawer principle".

[8] In fact, we show in the proof that for any $v \in I$, at least one of $\lfloor 10^k v \rfloor$ and $\lfloor 10^k v \rfloor + 1$ should be in $10^k I$.

[9] In fact, since I is an open interval, the first case is impossible, so we have $e = 0$.

Now, let us show that $\lfloor \log_{10} m \rfloor + 1$ is the minimum possible number of decimal significand digits. To show that, we first claim that

$$\lfloor \log_{10}(m-1) \rfloor = \lfloor \log_{10} m \rfloor$$

if $m \neq 1$. Indeed, if not, then we have

$$\begin{aligned}\log_{10}(m-1) &< \lfloor \log_{10}(m-1) \rfloor + 1 \\ &\leq \lfloor \log_{10} m \rfloor \leq \log_{10} m,\end{aligned}$$

thus

$$m - 1 < 10^{\lfloor \log_{10}(m-1) \rfloor + 1} \leq m.$$

Since $10^{\lfloor \log_{10}(m-1) \rfloor} + 1$ is an integer, we must have $m = 10^{\lfloor \log_{10}(m-1) \rfloor + 1}$, which contradicts to that m is not a multiple of 10. This shows the claim.

Next, note that for any $v \in I$ such that there exists $l \in \mathbb{Z}$ with $v \cdot 10^l \in \mathbb{Z}$, we have $l \geq k$ because of how we chose k. If $l = k$, then $v \cdot 10^l$ is one of $m, \cdots, M$, so we may assume $l > k$. Note also that we may assume $m \neq 1$, because if $m = 1$ then the number of decimal significand digits of elements in $I \cap 10^{-k}\mathbb{Z}$ is 1, which is of course a lower bound on the number of decimal significand digits of v. Now, since we have

$$\begin{aligned}\left\lfloor \log_{10}(v \cdot 10^l) \right\rfloor &= \left\lfloor \log_{10}(v \cdot 10^k) \right\rfloor + (l - k) \\ &\geq \left\lfloor \log_{10}(v \cdot 10^k) \right\rfloor + 1,\end{aligned}$$

it suffices to show that $\left\lfloor \log_{10}(v \cdot 10^k) \right\rfloor \geq \lfloor \log_{10} m \rfloor$. This inequality actually follows directly from our previous claim $\lfloor \log_{10}(m-1) \rfloor = \lfloor \log_{10} m \rfloor$; indeed, as $10^{-k}(m-1)$ is not an element of I, we should have $v > 10^{-k}(m-1)$, or equivalently, $v \cdot 10^k > m - 1$, which implies

$$\left\lfloor \log_{10}(v \cdot 10^k) \right\rfloor \geq \lfloor \log_{10}(m-1) \rfloor = \lfloor \log_{10} m \rfloor .$$

□

Since we have the following *chain property*

$$I \cap 10^{-k+1}\mathbb{Z} \subseteq I \cap 10^{-k}\mathbb{Z}$$

for all $k \in \mathbb{Z}$, we get the following:

Corollary 3.3.
Let $k_0 := -\lfloor \log_{10} \Delta \rfloor$. Then:

1. *If $I \cap 10^{-k_0+1}\mathbb{Z}$ is not empty, then the unique element in it has the smallest number of decimal significand digits in I.*
2. *Otherwise, elements in $I \cap 10^{-k_0}\mathbb{Z}$ have the smallest number of decimal significand digits.*

Proof. Suppose first that $I \cap 10^{-k_0+1}\mathbb{Z}$ is not empty. Let $l \in \mathbb{Z}$ be the smallest integer such that $I \cap 10^{-l}\mathbb{Z}$ is not empty, then by the chain property, we know

$$\emptyset \neq I \cap 10^{-l}\mathbb{Z} \subseteq I \cap 10^{-k_0+1}\mathbb{Z},$$

but since $I \cap 10^{-k_0+1}\mathbb{Z}$ can have at most 1 element by Proposition 3.1, it follows that the unique element of $I \cap 10^{-k_0+1}\mathbb{Z}$ is the unique element of $I \cap 10^{-l}\mathbb{Z}$. Hence, that unique element has the smallest number of decimal significand digits in I by Proposition 3.2.

Next, suppose that $I \cap 10^{-k_0+1}\mathbb{Z} = \emptyset$. Then again by the chain property, k_0 must be the smallest integer such that $I \cap 10^{-k_0}\mathbb{Z}$ is not empty, so the result follows from Proposition 3.2. □

Note that, since we always have $w \in I$, so if $I \cap 10^{-k}\mathbb{Z}$ is nonempty for some $k \in \mathbb{Z}$, then at least one of $\left\lfloor w \cdot 10^k \right\rfloor 10^{-k}$ and $\left(\left\lfloor w \cdot 10^k \right\rfloor + 1\right) 10^{-k}$ must be in $I \cap 10^{-k}\mathbb{Z}$. More precisely, pick any $v \in I \cap 10^{-k}\mathbb{Z}$, then if $v \leq w$, then $\left\lfloor w \cdot 10^k \right\rfloor 10^{-k}$ is in $I \cap 10^{-k}\mathbb{Z}$ since $\left\lfloor w \cdot 10^k \right\rfloor$ is the largest integer smaller than or equal to $w \cdot 10^k$, so it should lie in between $v \cdot 10^k$ and $w \cdot 10^k$. Similarly, if $v > w$, then $\left(\left\lfloor w \cdot 10^k \right\rfloor + 1\right) 10^{-k}$ is in $I \cap 10^{-k}\mathbb{Z}$ since $\left\lfloor w \cdot 10^k \right\rfloor + 1$ is the smallest integer strictly greater than $w \cdot 10^k$, so it should lie in between $w \cdot 10^k$ and $v \cdot 10^k$. This leads us to the following strategy of finding the shortest decimal representation of w, which is the basic skeleton of Schubfach:

Algorithm 3.4 (Skeleton of Schubfach)**.**

1. Compute $k_0 := -\lfloor \log_{10} \Delta \rfloor$.
2. Compute $\left\lfloor w \cdot 10^{k_0-1} \right\rfloor$ and $\left\lfloor w \cdot 10^{k_0-1} \right\rfloor + 1$. If one of them (and only one of them) belongs to $I \cdot 10^{k_0-1}$, then call that number s. In this case, $s \cdot 10^{-k_0+1}$ is the unique number in I with the smallest number of decimal significand digits. However, s might contain trailing decimal zeros; that is, it might be a multiple of a power of 10 as $I \cdot 10^{-l}\mathbb{Z}$ might not be empty for some $l < k_0 - 1$. Thus, let d be the greatest integer such that 10^d divides s, then $\frac{s}{10^d} \times 10^{d-k_0+1}$ is the unique shortest decimal representation of w.
3. Otherwise, we compute $\left\lfloor w \cdot 10^{k_0} \right\rfloor$ and $\left\lfloor w \cdot 10^{k_0} \right\rfloor + 1$. Then at least one of them must be in $I \cdot 10^{k_0}$, and if only one of them is inside I, call that number s. In this case, $s \cdot 10^{-k_0}$ is the unique number in I with the smallest number of decimal significand digits. Since we assumed that $I \cap 10^{-k_0+1}\mathbb{Z}$ is empty, s is never divisible by 10 so there is no trailing decimal zeros and $s \times 10^{-k_0}$ is the unique shortest decimal representation of w.
4. If both $\left\lfloor w \cdot 10^{k_0} \right\rfloor$ and $\left\lfloor w \cdot 10^{k_0} \right\rfloor + 1$ are inside $I \cdot 10^{k_0}$, choose the one that is closer to $w \cdot 10^{k_0}$. When the distances from $w \cdot 10^{k_0}$ to those numbers are the same, break the tie according to a given rule.[10] Call the chosen number s, then again s cannot have any trailing decimal zeros and $s \times 10^{-k_0}$ is the correctly rounded shortest decimal representation of w.

[10] The most common rule is to choose the even one, but we can consider other rules as well.

Based on the above strategy, the details of Schubfach include following:

- How to efficiently compute $\lfloor \log_{10} \Delta \rfloor$?
- How to efficiently compute $\lfloor w \cdot 10^{k_0 - 1} \rfloor$, $\lfloor w \cdot 10^{k_0 - 1} \rfloor + 1$, $\lfloor w \cdot 10^{k_0} \rfloor$ and $\lfloor w \cdot 10^{k_0} \rfloor + 1$?
- How to efficiently compare these numbers to the endpoints of $I \cdot 10^{k_0 - 1}$ or $I \cdot 10^{k_0}$?

Similar to Ryū and Grisu-Exact, Schubfach uses a table of precomputed binary digits of powers of 10 in order to accomplish the second item. In addition to that, it uses an ingenious rounding trick to make the third item trivial.[11] More precisely, after computing k_0, Schubfach computes approximations of $w_L \cdot 10^{k_0}$ and $w_R \cdot 10^{k_0}$ along with that of $w \cdot 10^{k_0}$, with the aforementioned rounding rule applied, and the construction of the rounding rule ensures that we can just compare our number to the computed approximations of $w_L \cdot 10^{k_0}$ and $w_R \cdot 10^{k_0}$ in order to deduce if our number is in the interval or not.

However, even with the precomputed cache, computing the approximate multiplications $w_L \times 10^{k_0}$, $w_R \times 10^{k_0}$, and $w \times 10^{k_0}$, is not cheap, because it requires several 64-bit multiplications, which, for typical modern x86 machines, are a lot slower than many other instructions. (We will review how these approximate multiplications can be done in Section 4.2.) The core idea of Dragonbox is, thus, on how we can avoid these multiplications.

4. Dragonbox

For this section, we will assume a round-to-nearest rounding rule, which is the most relevant and at the same time the most difficult case. Algorithms for other rounding rules can be developed in similar ways, and they will be covered in Appendix A and Appendix B.

4.1 Overview

We will describe a brief overview of Dragonbox for the case when $F_w \neq 1$ or $E_w = E_{\min}$ (we call this *normal interval case*), so that $\Delta = 2^e$. The case $F_w = 1$ and $E_w \neq E_{\min}$ (we call this *shorter interval case*) will be covered in Section 5.

Not like Schubfach, consider the following exponent instead of $k_0 := -\lfloor \log_{10} \Delta \rfloor$:

$$k := k_0 + \kappa = -\lfloor \log_{10} \Delta \rfloor + \kappa,$$

where κ is a positive integer constant in a certain range that we will discuss in Section 4.5. We will also discuss on how to compute k efficiently in that section.

[11] To be honest, I did not look at this rounding trick carefully, and do not fully understand how it works. Dragonbox does not rely on this trick, so it should be irrelevant for the rest of the paper. However, it might be that we can still possibly apply the trick also to Dragonbox so that we can make it even faster.

Similarly to [4], let us use the following notations:

$$\begin{aligned} x &:= w_L \cdot 10^k, \\ y &:= w \cdot 10^k, \\ z &:= w_R \cdot 10^k, \\ \delta &:= z - x = \Delta \cdot 10^k, \end{aligned}$$

and for $a \in \mathbb{R}$, we denote $a^{(i)} := \lfloor a \rfloor$, $a^{(f)} := a - \lfloor a \rfloor$.

Using a Grisu-like idea based on the following simple fact, we can mostly avoid computing x and y when doing the second step of Algorithm 3.4:

Proposition 4.1.
Let s, r be the unique integers satisfying

$$z = 10^{\kappa+1} s + r, \quad 0 \le r < 10^{\kappa+1}.$$

Then, $I \cap 10^{-k_0+1}\mathbb{Z}$ is nonempty if and only if

$$s \in 10^{k_0 - 1} I,$$

if and only if:

1. *$r + z^{(f)} \le \delta$, when $I = [w_L, w_R]$,*
2. *$r + z^{(f)} < \delta$, when $I = (w_L, w_R]$.*
3. *$r + z^{(f)} \le \delta$ and $r \neq 0$ or $z^{(f)} \neq 0$, when $I = [w_L, w_R)$, and*
4. *$r + z^{(f)} < \delta$ and $r \neq 0$ or $z^{(f)} \neq 0$, when $I = (w_L, w_R)$.*

Proof. We first show that $I \cap 10^{-k_0+1}\mathbb{Z}$ is nonempty if and only if $s \in 10^{k_0-1} I$. Clearly, $s_\kappa \cdot 10^{-k_0+1}$ is always an element of $10^{-k_0+1}\mathbb{Z}$, so if it belongs to I, then $I \cap 10^{-k_0+1}\mathbb{Z}$ is nonempty.

Conversely, suppose $I \cap 10^{-k_0+1}\mathbb{Z}$ is nonempty. Let v be any element of it. Then, $v \le w_R$, so

$$10^{k-\kappa-1} v \le \frac{z}{10^{\kappa+1}},$$

but since $10^{k-\kappa-1} v = 10^{k_0-1} v \in \mathbb{Z}$, it follows that

$$10^{k-\kappa-1} v \le \left\lfloor \frac{z}{10^{\kappa+1}} \right\rfloor = s.$$

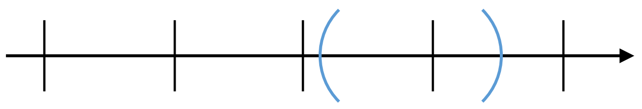

Figure 3. The unique lattice point in I should be the floor of the right endpoint, since I is longer than the unit

Now, since $10^{k_0-1} v$ and s are both integers, if we suppose

$$10^{k_0-1} v \neq s,$$

then

$$10^{k_0-1} v + 1 \le s$$

follows, which implies

$$10^{-k_0+1}s \geq v + 10^{-k_0+1} > v + \Delta \geq w_L + \Delta = w_R$$

by definition of k_0. This is absurd, because

$$10^{-k_0+1}s = 10^{-k} \cdot 10^{\kappa+1}s \leq 10^{-k} \cdot z = w_R.$$

Hence, we deduce $s = 10^{k_0-1}v \in 10^{k_0-1}I$, concluding the first "if and only if".

To show the second "if and only if", let us recall that $10^{-k_0+1}s = 10^{-k} \cdot 10^{\kappa+1}s$ is at most w_R. Hence, when $w_R \in I$, $10^{-k_0+1}s$ is in I if and only if its distance from w_L is less than or equal to Δ, or strictly less than Δ, depending on whether or not if w_L is in I, which are precisely the claims *1* and *2*.

On the other hand, if $w_R \notin I$, then we need to rule out the case $w_R = 10^{-k_0+1}s_\kappa$ in addition, which is precisely the case when $r_\kappa = 0$ and $z^{(f)} = 0$, thus we have the last two claims as well. □

Note that $r + z^{(f)} \leq \delta$ if and only if

1. $r < \delta^{(i)}$, or
2. $r = \delta^{(i)}$ and $z^{(f)} \leq \delta^{(f)}$,

and we have a similar equivalence for $r + z^{(f)} < \delta$. As in [4], we can efficiently perform these comparisons. In particular, since

$$x^{(i)} + x^{(f)} = (z^{(i)} - \delta^{(i)}) + (z^{(f)} - \delta^{(f)}),$$

and $-1 < z^{(f)} - \delta^{(f)} < 1$, we conclude

$$x^{(i)} = \begin{cases} z^{(i)} - \delta^{(i)} & \text{if } z^{(f)} \geq \delta^{(f)} \\ z^{(i)} - \delta^{(i)} - 1 & \text{if } z^{(f)} < \delta^{(f)} \end{cases},$$

so we just need to compare the parity of $x^{(i)}$ and $z^{(i)} - \delta^{(i)}$ to conclude if the inequality $z^{(f)} \geq \delta^{(f)}$ holds or not. Details of how to compute the parity of $x^{(i)}$ is explained in Section 4.3.

Note that we need to compare the fractional parts only when we know $r = \delta^{(i)}$; in this case, note that

$$z^{(i)} - \delta^{(i)} = 10^{\kappa+1}s$$

is always an even number. Thus, we have $z^{(f)} < \delta^{(f)}$ if and only if $x^{(i)}$ is an odd number. When $x^{(i)}$ is an even number, then we have either $z^{(f)} = \delta^{(f)}$ or $z^{(f)} > \delta^{(f)}$. Depending on whether or not w_L is contained in I, we may need to distinguish these two cases. To do that, we check if x is an integer, since $z^{(f)} = \delta^{(f)}$ if and only if $x^{(f)} = 0$ if and only if x is an integer. Details of how to check if x is an integer is explained in Section 4.6.

Let us now more precisely describe how to inspect if $I \cap 10^{-k_0+1}\mathbb{Z}$ is empty:

1. Compute $k = -\lfloor \log_{10} \Delta \rfloor + \kappa$. Since κ is just a fixed constant, it boils down to calculating $\lfloor \log_{10} \Delta \rfloor$. Details will be explained in Section 4.5.
2. Compute $z^{(i)}$. Details will be explained in Section 4.2.
3. Compute s, r by dividing $z^{(i)}$ by $10^{\kappa+1}$. Given that κ is a known constant, this can be done efficiently without actually issuing the notoriously slow integer division instruction, as described in [8]. Compilers these days usually automatically perform this optimization pretty well, but we can sometimes do better than them because of some additional constraints they are usually not aware of. Details will be explained in Section 4.7.
4. Compute $\delta^{(i)}$. Details will be explained in Section 4.4.
5. Check if the inequality $r > \delta^{(i)}$ holds. If that is the case, then we conclude that $I \cap 10^{-k_0+1}\mathbb{Z}$ is empty.
6. Otherwise, check if the inequality $r < \delta^{(i)}$ holds. If that is the case, we need to check if $r = z^{(f)} = 0$ in addition when $w_R \notin I$. We can inspect the equality $z^{(f)} = 0$ by checking if z is an integer; details for that will be explained in Section 4.6.
   - If $w_R \notin I$ and $r = z^{(f)} = 0$, then we conclude that $I \cap 10^{-k_0+1}\mathbb{Z}$ is empty.
   - Otherwise, we conclude that $10^{-k+\kappa+1}s$ is the unique element in $I \cap 10^{-k_0+1}\mathbb{Z}$.
7. Otherwise, we have $r = \delta^{(i)}$. Then, compute the parity of $x^{(i)}$.
   - If $x^{(i)}$ is an odd number, then we have $z^{(f)} < \delta^{(f)}$, so we conclude that $10^{-k+\kappa+1}s$ is the unique element in $I \cap 10^{-k_0+1}\mathbb{Z}$.
   - If $x^{(i)}$ is an even number and $w_L \notin I$, then we conclude that $I \cap 10^{-k_0+1}\mathbb{Z}$ is empty.
   - If $x^{(i)}$ is an even number and $w_L \in I$, then check if x is an integer. If that is the case, then we conclude that $10^{-k+\kappa+1}s$ is the unique element in $I \cap 10^{-k_0+1}\mathbb{Z}$. Otherwise, we conclude that $I \cap 10^{-k_0+1}\mathbb{Z}$ is empty.

Note that in order to compare $z^{(f)}$ and $\delta^{(f)}$, we need to compute (the parity of) $x^{(i)}$ which is what we want to avoid. Hence, we want to minimize the chance of having $r = \delta^{(i)}$. Thus, we want to choose κ as large as possible. However, choosing too big κ will prevent us from computing $z^{(i)}$ and $\delta^{(i)}$ efficiently, so there are in fact not so many choices for κ we have. Details will be explained in Section 4.5.

When we have concluded that $10^{-k+\kappa+1}s$ is the unique element in $I \cap 10^{-k_0+1}\mathbb{Z}$, then since s might contain trailing decimal zeros, find the greatest integer d such that 10^d divides s. Then we conclude that

$$\frac{s}{10^d} \times 10^{-k+\kappa+1+d}$$

is the answer we are looking for.

Next, let us discuss what we do if $I \cap 10^{-k_0+1}\mathbb{Z}$ turns out to be empty. Our procedure in this case is a bit different from the Schubfach's way. Recall that Corollary 3.3 tells us that in this case,

$$I \cap 10^{-k_0}\mathbb{Z} = 10^{-k}\left(10^k I \cap 10^\kappa \mathbb{Z}\right)$$

is not empty and its elements are precisely the elements with the smallest number of significand digits.

We will now compute

$$y^{(ru)} := \left\lfloor \frac{y}{10^\kappa} + \frac{1}{2} \right\rfloor 10^\kappa \quad \text{and}$$
$$y^{(rd)} := \left\lceil \frac{y}{10^\kappa} - \frac{1}{2} \right\rceil 10^\kappa,$$

which are the elements in $10^\kappa\mathbb{Z}$ that are closest to $y \in 10^k I$, using a method similar to that described in [4]. As shown in [4], both of $y^{(ru)}$ and $y^{(rd)}$ should be in $10^k I$ because we have assumed that $F_w \neq 1$ or $E_w = E_{\min}$; we will revisit this and explain in more detail in Section 4.9.

Note that $y^{(ru)} = y^{(rd)} + 1$ if and only if

$$\frac{y}{10^\kappa} - \left\lfloor \frac{y}{10^\kappa} \right\rfloor = \frac{1}{2},$$

and $y^{(ru)} = y^{(rd)}$ otherwise. In other words, $y^{(ru)}$ and $y^{(rd)}$ are same except when there is a tie, so we just need to focus on computing $y^{(ru)}$, detect the tie, and decrease the computed value of $y^{(ru)}$ by one if we prefer to choose $y^{(rd)}$ according to a given rule to break the tie. Let $y^{(r)}$ be the chosen one when we had a tie, or otherwise the common value of $y^{(ru)} = y^{(rd)}$, then the correctly rounded decimal representation of w with the shortest number of digits is thus

$$y^{(r)} \times 10^{-k+\kappa}.$$

To actually compute $y^{(ru)}$, note that

$$\begin{aligned} y^{(ru)} &= \left\lfloor \frac{y + (10^\kappa/2)}{10^\kappa} \right\rfloor \\ &= \left\lfloor \frac{z + (10^\kappa/2) - (z-y)}{10^\kappa} \right\rfloor \\ &= 10s + \left\lfloor \frac{r + (10^\kappa/2) - \epsilon^{(i)} + (z^{(f)} - \epsilon^{(f)})}{10^\kappa} \right\rfloor \end{aligned}$$

where we define

$$\epsilon := z - y.$$

Since we have assumed $F_w \neq 1$ or $E_w = E_{\min}$, w should lie at the exact center of I. Hence in particular, $\epsilon = \frac{\delta}{2}$, so $\epsilon^{(i)} = \left\lfloor \frac{\delta^{(i)}}{2} \right\rfloor$. Also, since κ is a positive integer, $10^\kappa/2$ is an integer. Recall that we already have assumed that $I \cap 10^{-k_0+1}\mathbb{Z}$ is empty; hence, by Proposition 4.1, either $r \geq \delta$ or $r = 0$. Since $\epsilon < \delta$, for the first case we know

$$r + \frac{10^\kappa}{2} - \epsilon^{(i)} > 0.$$

To make the arguments from now on simpler, for the case $r = 0$, let us replace r by $10^{\kappa+1}$ and s by $s-1$ so that we still have the inequality above even for the case $r = 0$. To be precise, let us define

$$\tilde{s} := \begin{cases} s & \text{if } r \neq 0 \\ s-1 & \text{if } r = 0 \end{cases}, \quad \tilde{r} := \begin{cases} r & \text{if } r \neq 0 \\ 10^{\kappa+1} & \text{if } r = 0 \end{cases},$$

so that we have

$$z^{(i)} = 10^{\kappa+1}\tilde{s} + \tilde{r}$$

and

$$y^{(ru)} = 10\tilde{s} + \left\lfloor \frac{\tilde{r} + (10^\kappa/2) - \epsilon^{(i)} - (z^{(f)} - \epsilon^{(f)})}{10^\kappa} \right\rfloor.$$

Now, define

$$D := \tilde{r} + (10^\kappa/2) - \epsilon^{(i)},$$

then by definition it is clear that $D \geq 0$. Next, let t, ρ be the unique integers satisfying

$$D = 10^\kappa t + \rho, \quad 0 \leq \rho < 10^\kappa.$$

Then,

$$y^{(ru)} = (10\tilde{s} + t) + \left\lfloor \frac{\rho + (z^{(f)} - \epsilon^{(f)})}{10^\kappa} \right\rfloor.$$

Note that the residue term

$$\left\lfloor \frac{\rho + (z^{(f)} - \epsilon^{(f)})}{10^\kappa} \right\rfloor$$

is always 0 except when $\rho = 0$ and $z^{(f)} < \epsilon^{(f)}$, and for that case it is equal to -1. Hence, we can just ignore the fractional parts and conclude $y^{(ru)} = 10\tilde{s} + t$ when D is not divisible by 10^κ, which is usually the case especially when κ is large. Of course when D is divisible by 10^κ, we need to compare $z^{(f)}$ and $\epsilon^{(f)}$ but this can be done by computing the parity of $y^{(i)}$ just like the comparison of $z^{(f)}$ and $\delta^{(f)}$. Indeed, note that

$$y^{(i)} + y^{(f)} = (z^{(i)} - \epsilon^{(i)}) + (z^{(f)} - \epsilon^{(f)}),$$

and since $-1 < z^{(f)} - \epsilon^{(f)} < 1$, we conclude

$$y^{(i)} = \begin{cases} z^{(i)} - \epsilon^{(i)} & \text{if } z^{(f)} \geq \epsilon^{(f)} \\ z^{(i)} - \epsilon^{(i)} - 1 & \text{if } z^{(f)} < \epsilon^{(f)} \end{cases},$$

so we just need to compare the parity of $y^{(i)}$ and $z^{(i)} - \epsilon^{(i)}$ to conclude if the inequality $z^{(f)} \geq \epsilon^{(f)}$ holds or not. In fact, since $10^{\kappa+1}$ is even, the parity of $z^{(i)}$ and that of r is same, so we can compare the parity of $y^{(i)}$ with that of $D - (10^\kappa/2)$. If the parities are the same, then we conclude $z^{(f)} \geq \epsilon^{(f)}$ so $y^{(ru)} = 10\tilde{s} + t$, and otherwise, we conclude $z^{(f)} < \epsilon^{(f)}$ so $y^{(ru)} = 10\tilde{s} + t - 1$. Details of how to compute the parity of $y^{(i)}$ will be explained in Section 4.3.

Note that tie happens exactly when $\rho = z^{(f)} - \epsilon^{(f)} = 0$; indeed, tie happens when the fractional part of $\frac{y}{10^\kappa}$ is exactly $1/2$, or equivalently,

$$\frac{y}{10^\kappa} + \frac{1}{2} = (10\tilde{s} + t) + \frac{\rho + (z^{(f)} - \epsilon^{(f)})}{10^\kappa}$$

is an integer. Since

$$-1 < \rho + (z^{(f)} - \epsilon^{(f)}) < 10^\kappa,$$

it follows that $\frac{y}{10^\kappa} + \frac{1}{2}$ is an integer if and only if

$$\rho + (z^{(f)} - \epsilon^{(f)}) = 0,$$

if and only if $\rho = z^{(f)} - \epsilon^{(f)} = 0$. Or equivalently, tie happens if and only if D is divisible by 10^κ and $y = z - \epsilon$ is an integer. If tie happens, then we need to choose between $y^{(ru)} = 10\tilde{s} + t$ and $y^{(rd)} = 10\tilde{s} + t - 1$ according to a given rule. Details of how to check if y is an integer will be explained in Section 4.6.

In summary, when $I \cap 10^{-k_0+1}\mathbb{Z}$ turns out to be empty, then:

1. Compute $D = \tilde{r} + (10^\kappa/2) - \lfloor \delta^{(i)}/2 \rfloor$.
2. Compute t, ρ by dividing D by 10^κ. Again, given that κ is a known constant, this can be done efficiently using the method described in [8]. In fact, since we do not care about the actual value of ρ and we only need to know if ρ is zero or not, we can do even better; details will be explained in Section 4.8.
3. If $\rho \neq 0$, then $(10\tilde{s} + t) \times 10^{-k+\kappa}$ is the answer we are looking for.
4. Otherwise, compare the parity of $y^{(i)}$ with that of $D - (10^\kappa/2)$. If they are different, then we have $z^{(f)} < \epsilon^{(f)}$, so $(10\tilde{s} + t - 1) \times 10^{-k+\kappa}$ is the answer we are looking for.
5. Otherwise, check if y is an integer. If that is the case, then we have a tie; break it according to a given rule, so that we choose one of $(10\tilde{s} + t - 1) \times 10^{-k+\kappa}$ and $(10\tilde{s} + t) \times 10^{-k+\kappa}$ as the answer.
6. Otherwise, $(10\tilde{s} + t) \times 10^{-k+\kappa}$ is the answer we are looking for.

Again, we want to avoid computing (the parity of) $y^{(i)}$, so we prefer to choose κ as big as possible.

4.2 Computing $z^{(i)}$

As in [4], we denote

$$10^k = \varphi_k \cdot 2^{e_k}$$

where e_k is an integer and φ_k is the unique rational number satisfying $2^{Q-1} \leq \varphi_k < 2^Q$. This means that

$$2^{e_k+Q-1} \leq 10^k < 2^{e_k+Q},$$

thus

$$k \log_2 10 - Q < e_k \leq k \log_2 10 - Q + 1,$$

implying

$$e_k = \lfloor k \log_2 10 \rfloor - Q + 1.$$

In Section 6, we will show that if Q is large enough, then

$$\begin{aligned} z^{(i)} &= \lfloor w_R \cdot 10^k \rfloor \\ &= \left\lfloor \left(f_c + \frac{1}{2}\right) \cdot 2^e \cdot 10^k \right\rfloor \\ &= \left\lfloor \left(f_c + \frac{1}{2}\right) \cdot 2^e \cdot \tilde{\varphi}_k \cdot 2^{e_k} \right\rfloor \end{aligned}$$

where $\tilde{\varphi}_k = \lfloor \varphi_k \rfloor$ or $\lfloor \varphi_k \rfloor + 1$, depending on the sign of k. Therefore,

$$z^{(i)} = \left\lfloor \left(\left(f_c + \frac{1}{2}\right) \cdot 2^{e+e_k+Q} \right) \cdot \tilde{\varphi}_k \cdot 2^{-Q} \right\rfloor.$$

Let us define

$$\beta := e + e_k + Q = e + \lfloor k \log_2 10 \rfloor + 1$$

so that

$$z^{(i)} = \left\lfloor \left(\left(f_c + \frac{1}{2}\right) \cdot 2^{\beta} \right) \cdot \tilde{\varphi}_k \cdot 2^{-Q} \right\rfloor.$$

We will impose a condition on κ to make sure that the quantity

$$\left(f_c + \frac{1}{2}\right) \cdot 2^{\beta}$$

is a q-bit integer; see Section 4.5 for details. Then,

$$z^{(i)} = \left\lfloor \left(\left(f_c + \frac{1}{2}\right) \cdot 2^{\beta} \right) \cdot \tilde{\varphi}_k \cdot 2^{-Q} \right\rfloor$$

is nothing but the upper q-bits of the $(q+Q)$-bit result of the multiplication of a q-bit integer $\left(f_c + \frac{1}{2}\right) \cdot 2^\beta$ and a Q-bit integer $\tilde{\varphi}_k$.[12]

Now we will analyze how the computation of $z^{(i)}$ can be done in terms of full/half multiplications. By *P-bit full multiplication*, we mean computing the $2P$-bit result of a multiplication of two P-bit integers. By *P-bit half multiplication*, we mean computing the lower half of the result of P-bit full multiplication. Typical machines today, like modern x86, often provide instructions for full multiplications. Some machines do not provide them, but even for such cases we can emulate full multiplication using several half multiplications. See, for example, [9]. It should be noted that even if the machine provides instructions for full multiplications, it is often the case that they are slower than some half multiplication instructions for the same size of integers. Also, it

[12] To be precise, we need to be aware there might be a possibility that $\tilde{\varphi}_k$ is not a Q-bit integer, if $\lfloor \varphi_k \rfloor = 2^Q - 1$ and $\tilde{\varphi}_k = \lfloor \varphi_k \rfloor + 1$. However, this never happens for any practical values of k and Q.

is worth mentioning that for a typical modern x86 CPU, 64-bit multiplications tend to be significantly slower than 32-bit multiplications.

For the case of binary32 format, we choose $Q = 2q = 64$. Hence, we need to compute the upper 32-bits from the 96-bit multiplication result of a 32-bit integer and a 64-bit integer. On a typical modern x86 CPU, this can be done by one 64-bit full multiplication.[13]

For the case of binary64 format, we choose $Q = 2q = 128$. Hence, we need to compute the upper 64-bits from the 192-bit multiplication result of a 64-bit integer and a 128-bit integer. This can be done by two 64-bit full multiplications, one 64-bit addition, and one 64-bit addition-with-carry. One can see, for example, Section 3.7 of [4] for details.

4.3 Computing the Parities of $x^{(i)}$ and $y^{(i)}$

We need to compute the parities (that is, the *least significant bits*) of $x^{(i)}$ and $y^{(i)}$, when we compare the fractional part of z and that of δ and ϵ, respectively. This can be done faster than the full computation of $x^{(i)}$ and $y^{(i)}$.

First, note that

$$x^{(i)} = \left\lfloor \left(\left(f_c - \frac{1}{2} \right) \cdot 2^{\beta} \right) \cdot \tilde{\varphi}_k \cdot 2^{-Q} \right\rfloor$$

and

$$y^{(i)} = \left\lfloor \left(f_c \cdot 2^{\beta} \right) \cdot \tilde{\varphi}_k \cdot 2^{-Q} \right\rfloor .$$

Here, the trick is to compute the multiplication of $\tilde{\varphi}_k$ with $2f_c - 1$ or $2f_c$, not with $\left(f_c - \frac{1}{2}\right) \cdot 2^{\beta}$ or $f_c \cdot 2^{\beta}$. Note that

$$x^{(i)} = \left\lfloor (2f_c - 1) \cdot \tilde{\varphi}_k \cdot 2^{-Q+\beta-1} \right\rfloor .$$

Here, $2f_c - 1$ is at most $(p + 2)$-bit integer, so assuming $q \geq p+2$ (which is always the case for all relevant formats), we can represent it as a q-bit integer. Note that the least significant bit of $x^{(i)}$ is nothing but the $(\beta - 1)^{\text{th}}$ bit of the second first q-bit block of the $(q + Q)$-bit result of the multiplication $(2f_c - 1) \cdot \tilde{\varphi}_k$, counting from the most significant bit. The same can be said for $y^{(i)}$.

For the case of binary32 format, we choose $Q = 2q = 64$. Hence, we need to compute the middle 32-bits from the 96-bit multiplication result. Since the middle 32-bits are nothing but the upper 32-bits of the lower 64-bits, this can be done by one 64-bit half multiplication. After performing the multiplication, we shift the result to the right by $64-(\beta-1)$ bits, and return the least significant bit of the shifted result.

For the case of binary64 format, we choose $Q = 2q = 128$. Hence, we need to compute the middle 64-bits from the 192-bit multiplication result. This can be done by one 64-bit full multiplication (the upper half of the yellow boxes in the Figure 4) and one 64-bit half multiplication (the lower half of the purple boxes in the Figure 4), and one 64-bit addition. After computing the addition, we shift the result to the right by $64 - (\beta - 1)$ bits, and return the least significant bit of the shifted result.

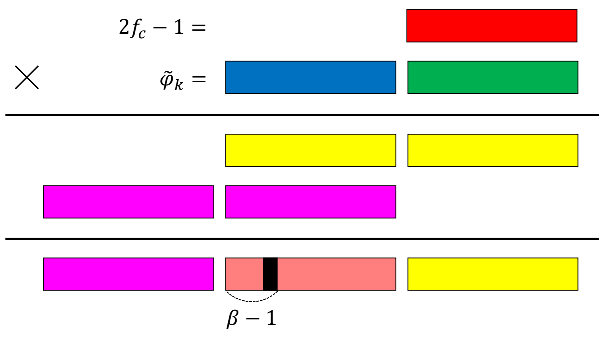

Figure 4. Illustration of the parity computation of $x^{(i)}$

4.4 Computing $\delta^{(i)}$

Since are considering the normal interval case ($F_w \neq 1$ or $E_w = E_{\min}$), computation of $\delta^{(i)}$ is very simple, as $\Delta = 2^e$ is a power of 2. Section 6 shows that if Q is large enough, then

$$\delta^{(i)} = \left\lfloor 2^e \cdot 10^k \right\rfloor = \left\lfloor 2^e \cdot \tilde{\varphi}_k \cdot 2^{e_k} \right\rfloor = \left\lfloor \tilde{\varphi}_k \cdot 2^{\beta - Q} \right\rfloor ,$$

so $\delta^{(i)}$ is nothing but the first β bits of $\tilde{\varphi}_k$, counting from the most significant bit.

4.5 Computing k, β, and κ

Note that

$$k = -\left\lfloor \log_{10} \Delta \right\rfloor + \kappa = -\left\lfloor e \log_{10} 2 \right\rfloor + \kappa$$

as we have assumed the normal interval case. The above can be computed efficiently using the usual trick of multiply-and-shift; see, for example, Section 3.4 of [4]. Similarly, we can compute

$$\beta = e + \left\lfloor k \log_2 10 \right\rfloor + 1$$

once we know the value of k.

As noted several times, we want to choose κ as big as possible, but at the same time we need to guarantee that

$$\left(f_c + \frac{1}{2} \right) \cdot 2^{\beta} = (2f_c + 1) \cdot 2^{\beta - 1}$$

is at most a q-bit integer. Hence, let us compute the possible range of β in terms of κ.

From the definition of k, we know

$$\kappa - k = \left\lfloor e \log_{10} 2 \right\rfloor \leq e \log_{10} 2 < \kappa - k + 1,$$

so

$$\kappa - e \log_{10} 2 \leq k < \kappa + 1 - e \log_{10} 2. \tag{1}$$

[13] To be precise, we only need the upper 64-bits, but generally computing the upper half while ignoring the lower half is not noticeably faster than the full multiplication. Thus, it is fair to consider such a computation as a form of full multiplication.

Hence,

$$\kappa \log_2 10 - e \le k \log_2 10 < (\kappa+1)\log_2 10 - e,$$

so

$$\kappa \log_2 10 + 1 \le e + k\log_2 10 + 1 < (\kappa+1)\log_2 10 + 1.$$

Therefore, taking the floor gives

$$\lfloor \kappa \log_2 10 \rfloor + 1 \le \beta \le \lfloor (\kappa+1)\log_2 10 \rfloor + 1. \tag{2}$$

It is clear from the above that $\beta \ge 1$, so $\left(f_c + \frac{1}{2}\right)\cdot 2^\beta$ is an integer. Also, since $f_c + \frac{1}{2}$ is strictly less than 2^{p+1}, it follows that

$$\left(f_c + \frac{1}{2}\right)\cdot 2^\beta < 2^{\beta+p+1},$$

so it suffices to have

$$\beta + p + 1 \le q.$$

Thus, from (2), we know that it is sufficient to have

$$\lfloor (\kappa+1)\log_2 10 \rfloor + p + 2 \le q. \tag{3}$$

For the case of binary32 format, we have $q = 32$ and $p = 23$, so (3) becomes

$$\lfloor (\kappa+1)\log_2 10 \rfloor \le 7,$$

so $\kappa \le 1$. Since we want κ to be at least 1, the only possible choice is $\kappa = 1$.

For the case of binary64 format, we have $q = 64$ and $p = 52$, so (3) becomes

$$\lfloor (\kappa+1)\log_2 10 \rfloor \le 10,$$

so $\kappa \le 2$. Since we want κ to be at least 1, the only possible choices are $\kappa = 1, 2$. As we want to choose κ as big as possible, we let $\kappa = 2$ in this case.

4.6 Integer Checks

Recall that sometimes we need to know if x, y, z are integers or not. Let us look at the case of z first. Recall that

$$z = \left(f_c + \frac{1}{2}\right)\cdot 2^e \cdot 10^k = (2f_c+1)\cdot 2^{e+k-1}\cdot 5^k,$$

and $2f_c + 1$ is an odd integer. Therefore, z is an integer if and only if:

1. $e + k - 1 \ge 0$, and
2. Either $k \ge 0$ or $k < 0$ and 5^{-k} divides $2f_c + 1$.

Note that

$$k = -\lfloor e \log_{10} 2 \rfloor + \kappa,$$

so $0 \le e + k - 1$ if and only if

$$0 \le e + \kappa - 1 - \lfloor e\log_{10} 2 \rfloor,$$

if and only if

$$\lfloor e \log_{10} 2 \rfloor \le e + \kappa - 1,$$

if and only if

$$e\log_{10} 2 < e + \kappa,$$

if and only if

$$-\kappa < e \log_{10} 5,$$

if and only if

$$-\kappa \log_5 10 < e.$$

Or equivalently,

$$e \ge -\lfloor \kappa \log_5 10 \rfloor = -\kappa - \lfloor \kappa \log_5 2 \rfloor$$

as $\kappa \log_5 10$ is never an integer.

On the other hand, note that we have $k \ge 0$ if and only if

$$\lfloor e \log_{10} 2 \rfloor \le \kappa$$

if and only if

$$e \log_{10} 2 < \kappa + 1$$

if and only if

$$e < (\kappa+1)\log_2 10.$$

Or equivalently,

$$e \le \lfloor (\kappa+1)\log_2 10 \rfloor$$

as $(\kappa+1)\log_2 10$ is never an integer.

Consequently,

1. If $e < -\kappa - \lfloor \kappa \log_5 2 \rfloor$, then $e + k - 1 < 0$, so z is not an integer.
2. Otherwise, if $e \le \lfloor (\kappa+1)\log_2 10 \rfloor$, then $k \ge 0$, so z is an integer.
3. Otherwise, z is an integer if and only if 5^{-k} divides $2f_c + 1$.

Recall that $f_c + \frac{1}{2}$ is strictly smaller than 2^{p+1}, so

$$2f_c + 1 < 2^{p+2}.$$

Hence, $2f_c + 1$ cannot have 5^{-k} as a factor if $5^{-k} \ge 2^{p+2}$, or equivalently,

$$-k \ge (p+2)\log_5 2.$$

Or, in terms of e, we can rewrite the above inequality as

$$\lfloor e \log_{10} 2 \rfloor - \kappa \ge (p+2)\log_5 2,$$

or equivalently,

$$\lfloor e \log_{10} 2 \rfloor - \kappa > \lfloor (p+2)\log_5 2 \rfloor,$$

which is equivalent to

$$e \log_{10} 2 \ge \lfloor (p+2)\log_5 2 \rfloor + \kappa + 1.$$

Thus, we conclude that z is not an integer if

$$e > \lfloor (\lfloor (p+2)\log_5 2 \rfloor + \kappa + 1) \log_2 10 \rfloor .$$

If

$$\lfloor (\kappa+1)\log_2 10 \rfloor < e \\ \leq \lfloor (\lfloor (p+2)\log_5 2 \rfloor + \kappa + 1) \log_2 10 \rfloor ,$$

then we do need to check divisibility of $2f_c + 1$ by 5^{-k}. In this case, we can apply the divisibility test method introduced in [8]. To briefly explain the method, for given a positive integer a in a certain range, we precompute the modular inverse of 5^m in the ring $\mathbb{Z}/2^q$. Since multiplying the modular inverse of 5^m is an automorphism on $\mathbb{Z}/2^q$, and since multiplying the modular inverse of 5^m coincides with dividing by 5^m for numbers divisible by 5^m, it follows that the set of integers $0 \leq n < 2^q$ that is divisible by 5^m should be bijectively mapped onto the set $\{0, 1, \cdots, \lfloor (2^q-1)/5^m \rfloor\}$. Hence, if the result of multiplying the modular inverse of 5^m to the given number is less than or equal to the precomputed $\lfloor (2^q-1)/5^m \rfloor$, then we conclude that the given number is divisible by 5^m, and otherwise, it is not divisible by 5^m. See Section 9 of [8] for details.

Now, for our case, the exponent $-k$ lies in the range $1, 2, \cdots, \lfloor (p+2)\log_5 2 \rfloor$, so it suffices to precompute the modular inverses and the maximum possible quotients for those exponents and store them in a static data table, and then use them to determine if $2f_c + 1$ is divisible by 5^{-k}.

To check if x is an integer, we can apply exactly the same procedure. However, to check if y is an integer, we need a slight modification since we do not know how many times f_c is divisible by 2. To be precise, recall that

$$y = f_c \cdot 2^e \cdot 10^k = f_c \cdot 2^{e+k} \cdot 5^k,$$

so y is an integer if

1. Either $e + k \geq 0$ or $e + k < 0$ and 2^{-e-k} divides f_c, and
2. Either $k \geq 0$ or $k < 0$ and 5^{-k} divides f_c.

Following a similar procedure, we can deduce that $e+k \geq 0$ if and only if

$$e \geq -\lfloor (\kappa+1)\log_5 10 \rfloor = -(\kappa+1) - \lfloor (\kappa+1)\log_5 2 \rfloor .$$

Thus, the strategy of checking if y is an integer is:

1. If $e > \lfloor (\kappa+1)\log_2 10 \rfloor$, then $e + k \geq 0$ and $k < 0$, so y is an integer if and only if 5^{-k} divides f_c.
2. Otherwise, if $e \geq -(\kappa+1) - \lfloor (\kappa+1)\log_5 2 \rfloor$, then $e + k \geq 0$ and $k \geq 0$, so y is an integer.
3. Otherwise, we have $e + k < 0$ and $l \geq 0$, so y is an integer if and only if 2^{-e-k} divides f_c.

Note that f_c is divisible by 2^{-e-k} if and only if there are at least $-e-k$ many trailing zeros in the binary representation of f_c. Many typical modern CPU's provide an instruction returning the number of trailing zeros, so on such machines this is very cheap. Otherwise, we can still check divisibility by, for example, shifting f_c to right by $-e-k$ bits and then to left by $-e-k$ bits, and then comparing the result with the original value of f_c. In this case, we need to be careful that shifting by an excessive amount of bits might not be a valid operation in many CPU's.

4.7 Efficient Division by $10^{\kappa+1}$

As noted earlier, we can replace the notoriously slow integer division by simpler instructions if the divisor is a known constant, as explained in [8]. Usually, compilers these day are smart enough to perform this optimization very well, but still there is a chance that we can do better than them when there are some constraints that compilers may not be aware of.

In this section, we will discuss on how to optimize the computation of the integers s, r satisfying

$$z^{(i)} = 10^{\kappa+1}s + r, \quad 0 \leq r < 10^{\kappa+1}.$$

Note that the usual trick of optimizing divisions-by-constants is to find a binary approximation of the reciprocal of the divisor, multiply it to the dividend, and then shift the result. However, this sometimes does not work because the required precision of the approximation might be too large so that the multiplication can overflow. Therefore, the valuable piece of information here is that the dividend $z^{(i)}$ does not span the full range of q-bit integers, so that the required precision can be smaller than usual. More specifically, recall that

$$z = \left(f_c + \frac{1}{2}\right) \cdot 2^e \cdot 10^k,$$

and since

$$k = -\lfloor e \log_{10} 2 \rfloor + \kappa < -e \log_{10} 2 + \kappa + 1$$

and $f_c + \frac{1}{2} < 2^{p+1}$, it follows that

$$z < 2^{p+1} \cdot 2^e \cdot 2^{-e} \cdot 10^{\kappa+1} = 2^{p+1} \cdot 10^{\kappa+1}.$$

Now, we use the following lemma from [4], originally presented in [5], to find a required precision for dividing by $10^{\kappa+1}$.

Lemma 4.2 (Adams, 2018)**.**
Let k be a nonnegative integer, b an integer, and g a positive integer. Then for any integer u satisfying

$$u > b + \log_2 \frac{5^k g}{5^k - (2^b g \bmod 5^k)},$$

we have

$$\left\lfloor \frac{g \cdot 2^b}{5^k} \right\rfloor = \left\lfloor g \cdot 2^{b-u} \left(\left\lfloor \frac{2^u}{5^k} \right\rfloor + 1 \right) \right\rfloor .$$

In our setting, $g = z^{(i)}$, $b = -\kappa - 1$, and $k = \kappa + 1$, so that

$$s = \left\lfloor \frac{z^{(i)}}{10^{\kappa+1}} \right\rfloor = \left\lfloor \frac{z^{(i)} \cdot 2^{-\kappa-1}}{5^{\kappa+1}} \right\rfloor.$$

Hence, by the lemma,

$$s = \left\lfloor z^{(i)} \cdot \left(\left\lfloor \frac{2^u}{5^{\kappa+1}} \right\rfloor + 1 \right) \cdot 2^{-\kappa-u-1} \right\rfloor$$

if u satisfies the inequality

$$u > -\kappa - 1 + \log_2 \frac{5^{\kappa+1} z^{(i)}}{5^{\kappa+1} - (2^{-\kappa-1} z^{(i)} \bmod 5^{\kappa+1})}. \quad (4)$$

Note that

$$(2^{-\kappa-1} z^{(i)} \bmod 5^{\kappa+1}) \le 5^{\kappa+1} - 2^{-\kappa-1},$$

so the right-hand side of the inequality (4) is upper-bounded by

$$-\kappa - 1 + \log_2 \left(2^{\kappa+1} \cdot 5^{\kappa+1} z^{(i)} \right) = \log_2 \left(5^{\kappa+1} z^{(i)} \right),$$

which is again strictly upper-bounded by

$$\log_2 \left(5^{\kappa+1} \cdot 2^{p+1} \cdot 10^{\kappa+1} \right) = p + \kappa + 2 + (2\kappa + 2) \log_2 5.$$

Therefore, in order to conclude

$$s = \left\lfloor z^{(i)} \cdot \left(\left\lfloor \frac{2^u}{5^{\kappa+1}} \right\rfloor + 1 \right) \cdot 2^{-\kappa-u-1} \right\rfloor,$$

it suffices to have

$$u \ge p + \kappa + 3 + \lfloor (2\kappa + 2) \log_2 5 \rfloor.$$

For the case of binary32 format with $\kappa = 1$, the minimum possible value of u estimated above is

$$23 + 1 + 3 + \lfloor 4 \log_2 5 \rfloor = 36.$$

This actually does not give us a better bound compared to the classical method explained in [8], Theorem 4.2, which gives us $u \ge 35$. Thus, there is little hope that we can do better than the compiler in this case.

On the other hand, for the case of binary64 format with $\kappa = 2$, the minimum possible value of u estimated above is

$$52 + 2 + 3 + \lfloor 6 \log_2 5 \rfloor = 70,$$

which is better than the bound we get from [8], Theorem 4.2, which gives us $u \ge 71$. Although it may seem to be not a big difference, the consequence of saving one more bit here is actually quite big. Indeed, note that the approximation given by [8] is

$$\left\lceil \frac{2^{71}}{125} \right\rceil = \texttt{0x1,0624,dd2f,1a9f,be77},$$

which exceeds 64-bits, while the approximation we derived is

$$\left\lceil \frac{2^{70}}{125} \right\rceil = \texttt{0x8312,6e97,8d4f,df3c},$$

which fits inside 64-bits. Therefore, our approximation enables us to compute s by only one 64-bit full multiplication and one 64-bit shift, but that is not achievable with the classical method. Specifically, according to Lemma 4.2, we know

$$s = \left\lfloor z^{(i)} \cdot \left\lceil \frac{2^{70}}{125} \right\rceil \cdot 2^{-73} \right\rfloor,$$

thus we can compute s by first performing a 64-bit full multiplication of $z^{(i)}$ and $\left\lceil \frac{2^{70}}{125} \right\rceil$, taking the upper 64-bits from the result, and then shifting it to the right by 9 bits.

It is also worth mentioning that since r is strictly smaller than $10^{\kappa+1}$, we do not need q-bits for storing r. For example, for the case of binary64 format, we can store r in a 32-bit register. This enables us to compute r without performing 64-bit operations. Instead, it suffices to perform one 32-bit half multiplication to compute the lower 32-bits of $10^{\kappa+1}s$, and then by subtracting the result from the lower 32-bits of $z^{(i)}$, we get the correct answer for r.

4.8 Efficient Division by 10^κ

Recall that when $I \cap 10^{-k_0+1}\mathbb{Z}$ is not empty, we need to divide

$$D = \tilde{r} + (10^\kappa/2) - \left\lfloor \delta^{(i)}/2 \right\rfloor$$

by 10^κ to compute the integers t, ρ satisfying

$$D = 10^\kappa t + \rho, \quad 0 \le \rho < 10^\kappa.$$

Usually, obtaining both the quotient and the remainder requires two multiplications to be performed. However, since we are only interested in whether or not ρ is zero, rather than the complete value of ρ, we might be able to do better. Indeed, because D and 10^κ are not big, we can reduce the required number of multiplications to 1.

Recall from Section 9 of [8] that an N-bit integer n is divisible by 5^m if and only if the lower N-bits of n times the modular inverse of 5^m is less than or equal to $\lfloor (2^N - 1)/5^m \rfloor$. On the other hand, recall from Section 4 of [8] that we can divide by a constant by multiplying the binary expansion of the reciprocal of the divisor, and then shifting to the right by a certain amount. Now, the trick is to combine two magic numbers of these methods into one. We will explain this trick in more detail for each of the binary32 and the binary64 formats separately.

Before that, let us first observe that D is at most $10^{\kappa+1}$. Indeed, by definition we have $\tilde{r} \le 10^{\kappa+1}$. Also, because of how we choose k, we have $\delta \ge 10^\kappa$. To see why, recall from (1) that

$$\kappa - e \log_{10} 2 \le k < \kappa + 1 - e \log_{10} 2.$$

Thus,

$$10^\kappa \cdot 2^{-e} \leq 10^k < 10^{\kappa+1} \cdot 2^{-e},$$

and since $\Delta = 2^e$ and $\delta = \Delta \cdot 10^k$, it follows that

$$10^\kappa \leq \delta < 10^{\kappa+1}. \tag{5}$$

This shows $D \leq 10^{\kappa+1}$.

Next, note that dividing by 10^κ is not different from dividing first by 2^κ and then by 5^κ. The first division is nothing but shifting to the right by κ bits. Divisibility check by 2^κ is also trivial; we just need to take κ bits counting from the least significant bit and check if they are all zero. If D turns out to be not divisible by 2^κ, we do not need to further check if D is divisible by 5^κ. Hence, in this case, we just divide D by 10^κ directly, as we do not need to care about divisibility anymore. Assuming D is stored as a 32-bit integer which is the most common preferred word size of today's machines, since we know $D \leq 10^{\kappa+1}$ and $10^{\kappa+1}$ fits inside 16-bits for small values of κ we are caring about, division by 10^κ can be performed by a single 32-bit half multiplication and a single shift. This can be theoretically verified, but can be also exhaustively checked for all possible values of $D \leq 10^{\kappa+1}$. Our reference implementation [10] contains a program verifying this.

Hence, we only need to consider how to divide $\frac{D}{2^\kappa}$ by 5^κ and at the same time check the divisibility when D has turned out to be divisible by 2^κ. Thus, we can further reduce the range of dividend to $[0, 2 \cdot 5^{\kappa+1}]$.

Now, let us consider the binary32 format with $\kappa = 1$. In this case, we are dividing $D/2$ by 5. Again assuming D is stored as a 32-bit integer, we wish to compute the quotient and at the same time check if $D/2$ is divisible by 5, by only performing one 32-bit half multiplication. Luckily, a very special fact about 5 is that its modular inverse in any $\mathbb{Z}/2^N$ always coincides with the approximate reciprocal of 5 given by Theorem 4.2 of [8], whenever N is a multiple of 4.[14] Hence, we can indeed perform two operations (computing the quotient and checking the divisibility) by just one multiplication. More concretely, our strategy is the following.

1. Compute the 32-bit half multiplication of $D/2$ and the magic number `0xcccd`. Note that `0xcccd` is the modular inverse of 5 in $\mathbb{Z}/2^{16}$. At the same time it satisfies the condition for approximate reciprocal of 5 given by Theorem 4.2 of [8]. Indeed, since $D/2$ is at most $2 \cdot 25 = 50$, so D is at most a 6-bit integer. And, we have the inequality
$$\left\lceil \frac{2^{6+12}}{5} \right\rceil \leq \texttt{0xcccd} \leq \left\lfloor \frac{2^{6+12} + 2^{12}}{5} \right\rfloor,$$
thus Theorem 4.2 of [8] applies. The multiplication of $D/2$ and `0xcccd` is at most 22-bits, so it cannot overflow as well.
2. The quotient can be obtained by shifting the result to the right by 18 bits.
3. Furthermore, $D/2$ is divisible by 5 if and only if the lower 16-bits of the result of the multiplication is less than or equal to $\lfloor (2^{16} - 1)/5 \rfloor$.

[14] This indeed comes from the fact that 5 is a number of the form $2^n + 1$. Note that the binary expansion of 5 is 101, and multiplying the binary number $1100, 1100, \cdots 1100$ to 101 results in $1111, 1111, \cdots 1111, 00$, so multiplying $1100, 1100, \cdots 1101$ to 101 results in $1, 0000, 0000, \cdots 0000, 01$, regardless of how many 1100's we initially had.

Next, let us consider the binary64 format with $\kappa = 2$. In this case, we are dividing $D/4$ by 25. Again assuming D is stored as a 32-bit integer, we wish to compute the quotient and at the same time check if $D/4$ is divisible by 25, by only performing one 32-bit half multiplication. Unfortunately, 25 is not that good compared to 5 in the sense that the approximate reciprocal and the modular inverse are in general very different. However, since $D/4$ is at most 250, which fits in 8-bits, we can split the magic number into two parts, so that the upper part consists of the approximate reciprocal and the lower part consists of the modular inverse.

To be precise, we choose the magic number μ such that the lower 8-bits of μ is the modular inverse of 25 in $\mathbb{Z}/2^8$ and μ satisfies the inequality

$$\left\lceil \frac{2^{8+\ell}}{25} \right\rceil \leq \mu \leq \left\lfloor \frac{2^{8+\ell} + 2^\ell}{25} \right\rfloor.$$

The smallest ℓ such that such μ exists is 12, and we can choose $\ell = \texttt{0xa429}$. Thus, our strategy is:

1. Compute the 32-bit half multiplication of $D/4$ and the magic number `0xa429`. The result of the multiplication is at most 24-bits, so we do not need to worry about overflow.
2. The quotient can be obtained by shifting the result to the right by 20 bits.
3. Furthermore, $D/4$ is divisible by 25 if and only if the lower 8-bits of the result of the multiplication is less than or equal to $\lfloor (2^8 - 1)/25 \rfloor$.

4.9 Some Facts about Correct Rounding

In this section, we will show that

$$y^{(ru)} := \left\lfloor \frac{y}{10^\kappa} + \frac{1}{2} \right\rfloor 10^\kappa \quad \text{and}$$
$$y^{(rd)} := \left\lceil \frac{y}{10^\kappa} - \frac{1}{2} \right\rceil 10^\kappa$$

are always inside $10^k I$. First, note that $y^{(ru)}$ and $y^{(rd)}$ should be one of $a := \left\lfloor \frac{y}{10^\kappa} \right\rfloor 10^\kappa$ and $b := \left(\left\lfloor \frac{y}{10^\kappa} \right\rfloor + 1 \right) 10^\kappa$. More precisely,

1. $y^{(ru)} = y^{(rd)} = a$ if $\left(\frac{y}{10^\kappa}\right)^{(f)} < \frac{1}{2}$,
2. $y^{(ru)} = b$ and $y^{(rd)} = a$ if $\left(\frac{y}{10^\kappa}\right)^{(f)} = \frac{1}{2}$,
3. $y^{(ru)} = y^{(rd)} = b$ if $\left(\frac{y}{10^\kappa}\right)^{(f)} > \frac{1}{2}$.

Note that $\frac{a}{10^\kappa} = \lfloor w \cdot 10^{k_0} \rfloor$. As shown in the proof of Proposition 3.1, $w \in I$ implies that at least one of $\frac{a}{10^\kappa} \in 10^{k_0} I$ or $\frac{b}{10^\kappa} \in 10^{k_0} I$ holds, thus we have at least one of $a \in 10^k I$ or $b \in 10^k I$.

Suppose first that $a \notin 10^k I$, so $b \in 10^k I$. We claim that in this case the fractional part of $\frac{y}{10^\kappa}$ should be strictly greater than $\frac{1}{2}$, so $y^{(ru)} = y^{(rd)} = b \in 10^k I$. Since y is at the exact center of $10^k I$, $a \notin 10^k I$ and $b \in 10^k I$ together imply that $b - y \geq y - a$. In other words, the fractional part of $\frac{y}{10^\kappa}$ should be at least $\frac{1}{2}$. Now it suffices to show that the fractional part cannot be equal to $\frac{1}{2}$. Suppose on the contrary that $\left(\frac{y}{10^\kappa}\right)^{(f)} = \frac{1}{2}$. Then $b - y = y - a$, but since $a \notin 10^k I$, $b \in 10^k I$, and y is at the center of $10^k I$, it follows that $10^k I = (a, b]$. However, since

$$\begin{aligned} 10^\kappa \mathbb{Z} \ni b = z &= (2f_c + 1) \cdot 2^{e-1} \cdot 10^k \\ &= 2^{e+k-1} \cdot 5^k \cdot (2f_c + 1) \end{aligned}$$

and $2f_c + 1$ is an odd integer, we must have

$$e + k - 1 = \kappa \quad \text{and} \quad 2f_c + 1 = 5^{e-1}.$$

However, by the same reason, $a = x$ implies

$$e + k - 1 = \kappa \quad \text{and} \quad 2f_c - 1 = 5^{e-1},$$

which is a contradiction. This shows the claim.

Next, suppose that $b \notin 10^k I$, so $a \in 10^k I$. We claim that in this case the fractional part of $\frac{y}{10^\kappa}$ should be strictly smaller than $\frac{1}{2}$, so $y^{(ru)} = y^{(rd)} = b \in 10^k I$. Again, similar reasoning shows that the fractional part should be at most $\frac{1}{2}$, and we should have $10^k I = [a, b)$ in order to have $\left(\frac{y}{10^\kappa}\right)^{(f)} = \frac{1}{2}$, which is absurd by the same reason. Therefore, we always have that $y^{(ru)}, y^{(rd)} \in 10^k I$.

5. Shorter Interval Case

So far, we have assumed that either $F_w \neq 1$ or $E_w = E_{\min}$, so that the length of the interval Δ is always equal to 2^e. In this section, we will assume $F_w = 1$ and $E_w \neq E_{\min}$ so that $\Delta = 3 \cdot 2^{e-2}$. Note that presence of this shorter interval case complicates a lot of things we argued in the last section, including but not limited to computation of k and $\delta^{(i)}$, integer checks, and the claim that $y^{(ru)}$ and $y^{(rd)}$ are always in $10^k I$ is no longer true, etc.. Thus, we will follow a completely separate path for the shorter interval case.

We will in fact more closely mimic the original Schubfach algorithm, rather than what is described in Section 4 in this case, because of the following reasons:

1. Shorter interval cases are rare, especially extremely rare for the binary64 format. Thus, whatever we do with them will not affect an average performance very much.[15]

[15] In fact, we have observed that failing to inline the code path for the shorter interval case resulted in a measurably worse performance. Hence, in our reference implementation [10], we enforced the compiler to inline the code path for the shorter interval case.

2. The original Schubfach algorithm is much simpler compared to the algorithm given in Section 4 especially given that lots of the assumptions we made are simply not true for the shorter interval case. And algorithmic simplicity matters when it comes to performance optimization.
3. Because we have $F_w = 1$, computing the approximate multiplications by 10^k is no more a heavy operation. Thus, there is little reason to try very hard to avoid it. We will give some detailed explanation on this in Section 5.2.

5.1 Overview

Following Schubfach [1], we will work with $k_0 = -\lfloor \log_{10} \Delta \rfloor$ rather than $k = k_0 + \kappa$. Let us define

$$\begin{aligned} x &:= w_L \cdot 10^{k_0}, \\ y &:= w \cdot 10^{k_0}, \\ z &:= w_R \cdot 10^{k_0} \end{aligned}$$

as before, where k is replaced by k_0.

5.2 Computing $x^{(i)}$ and $z^{(i)}$

5.3 Computing $y^{(ru)}$

5.4 Computing k and β

5.5 Some Facts about Correct Rounding

6. Sufficiency of Cache Precision

7. Performance

A. Right-Closed Directed Rounding Case

B. Left-Closed Directed Rounding Case

References

[1] R. Giulietti. The Schubfach Way to Render Doubles. 2020. https://drive.google.com/file/d/1KLtG_LaIbK9ETXI290zqCxvBW94dj058/view (Sep. 2020)

[2] F. Loitsch. Printing Floating-Point Numbers Quickly and Accurately with Integers. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation, PLDI 2010*. ACM, New York, NY, USA, 233–243. https://doi.org/10.1145/1806596.1806623

[3] M. Andrysco, R. Jhala, and S. Lerner. Printing Floating-Point Numbers: a Faster, Always Correct Method. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*. ACM, New York, NY, USA, 555–567. https://doi.org/10.1145/2837614.2837654

[4] J. Jeon. Grisu-Exact: A Fast and Exact Floating-Point Printing Algorithm. 2020. https://github.com/jk-jeon/Grisu-Exact/blob/master/other_files/Grisu-Exact.pdf. (Sep. 2020)

[5] U. Adams. Ryū: Fast Float-to-String Conversion In *Proceedings of the ACM SIGPLAN 2018 Conference on Programming Language Design and Implementation, PLDI 2018*. ACM, New York, NY, USA, 270–282. https://doi.org/10.1145/3296979.3192369

[6] G. L. Steel Jr. and J. L. White. How to Print Floating-Point Numbers Accurately. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, PLDI 1990*. ACM, New York, NY, USA, 112–126. https://doi.org/10.1145/93542.93559

[7] https://github.com/abolz/Drachennest. (Sep. 2020)

[8] T. Granlund and P. L. Montgomery. Division by Invariant Integers using Multiplication. In *ACM SIGPLAN Notices, Vol 29, Issue 6, Jun. 1994*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/773473.178249

[9] https://stackoverflow.com/questions/25095741/how-can-i-multiply-64-bit-operands-and-get-128-bit-result-portably. (Jun. 2020)

[10] https://github.com/jk-jeon/dragonbox. (Sep. 2020)

[11] https://github.com/jk-jeon/Grisu-Exact. (Sep. 2020)

[12] https://github.com/ulfjack/ryu. (Jun. 2020)