Grisu-Exact: A Fast and Exact Floating-Point Printing Algorithm

Junekey Jeon

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

Abstract

We present a new algorithm for accurately and efficiently converting floating-point numbers to decimal representation. The algorithm is a variant of Grisu2 [1], algorithm presented in 2010. Our algorithm uses 128-bit integer arithmetic to produce shortest and correctly rounded representation, whose correctness can be verified using methods introduced for Ryū [2], another algorithm for floating-point printing presented in 2018. Our proposed algorithm has a better performance than Ryū for numbers with small number of digits and similar performance to Ryū for numbers with large number of digits. For example, our algorithm is about 84% faster for numbers with 2 digits, about 52% faster for numbers with 6 digits for IEEE-754 binary64 encoded floating-point numbers.

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

Converting between binary and decimal representations of floating-point numbers is not a trivial problem. The majority of the existing computing platforms internally uses binary representations for floating-point numbers, because it enables much faster computations compared to the decimal counterpart. Obviously, binary representations are not human-friendly, thus most of inputs and outputs of floating-point numbers might require conversions. Many languages, such as JavaScript, even mandate decimal representation as the only way for input and output of floating-point numbers. JavaScript even does not have built-in integer types; floating-point numbers (which are internally stored according to IEEE-754 binary64 format) are the only native way of dealing with numbers. Due to recent continuing increase of JavaScript's usage, demand for fast conversion algorithm had been re-arisen recently in spite of the topic's long history.

Although the input-side (decimal-to-binary conversion) and the output-side (binary-to-decimal conversion) are both equally important, arguably the output-side often has more degree of freedom. Uncertain formatting specifications might be one source of this freedom, but more fundamental is the way we interpret a binary representation stored in memory: it often represents an *interval*, not a single real number. For example, there is no way to exactly represent 0.3 in a finite-precision binary expansion, but usually such an input is not treated as an error; rather, we compute the closest possible binary expansion of 0.3 and treat as if 0.3 and the resulting expansion were the same number. As a consequence, for each binary representation there are infinitely many decimal representations that round to that binary representation, while every valid decimal representation has a unique corresponding binary representation if we fix the rounding rule. One way to resolve this ambiguity of converting binary representation into decimal string is to apply the following set of criteria by Steele and White in 1990 [3]: [1]

1. **Information preservation**: a correct parser must return the original floating-point value from the output string,
2. **Minimum-length output**: the output string must be as short as possible, and

[1] Actually, there is one more criterion they have given: left-to-right generation of digits. However, we will not consider this criterion because the end output of the algorithm will be integers representing the decimal signifcand and the exponent, rather than a character array. This is arguably advantageous for higher level formatting.

3. **Correct rounding**: the output string must be as close to the input binary representation as possible.

The first criterion is perhaps the minimum requirement to be a "correct" algorithm. The second criterion also certainly offers some practical advantages. For example, the resulting output will be less bewildering to users because they would almost absolutely prefer 0.3 rather than 0.29999999999999997 although these numbers correspond to the same binary representation. The third criterion seems to be less important than the other two, but whenever there are multiple decimal representations with the minimum number of digits, it would be more probable that the one closest to the given input (interpreted as a single real number) could be closer to the "true" answer.

Note that common printing functions like `printf` from C standard library cannot satisfy these criteria because of its specification: using too low precision breaks the information preservation criterion, while using too high precision breaks the minimum-length output criterion.

The algorithm proposed in [3] satisfies these criteria, but its performance is not very good, mainly due to its heavy computations involving high (or variable) precision numbers, commonly called "big integers." In 2010, Florian Loitsch [1] proposed a new series of algorithms called Grisu, Grisu2, and Grisu3, which completely avoid on-the-fly big integer arithmetic and constrain themselves into 64-bits integer arithmetic. However, these algorithms do not satisfy the above criteria. To be more precise, Grisu only satisfies the first criterion and completely ignored the others. Grisu2 addresses the second criterion and produces shortest outputs for most of possible inputs, but not all; 1E+23 is a well-known failure case. Grisu3 also does not always produce shortest outputs for all possible inputs, but it detects its failure so that one can resort to slower but more precise algorithms. Grisu3 also deals with the third criterion for most of possible inputs and is able to detect its failure. Although not perfect, Grisu2 and Grisu3 proved their practical value and many higher-level open-source software components have adopted and implemented these algorithms.

In 2018, finally an algorithm both satisfying all the criteria above and utilizing only fixed-precision integer arithmetic is proposed [2]. This algorithm, Ryū, outperforms many implementations of Grisu2 for numbers with large number of digits. However, it has worse performance than some well-tailored implementations of Grisu2 for numbers with small number of digits, which is arguably more important use-case.

With the inspiration from Ryū, we could develop a new variant of the Grisu algorithm, *Grisu-Exact*, which performs better than Ryū for this small digits case but at the same time satisfies all the criteria. It also has performance comparable to Ryū for large digits case as well. Mathematics behind the correctness proof of Ryū again plays a critical role here.

2. IEEE-754 Specifications

Before diving into the details of the algorithm, 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 $s = 0, 1$, $0 \leq F_w < 2$, 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.
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
$$E_w = -(2^{n-p-2} - 1) + \sum_{k=0}^{n-p-2} b_{p+k} \cdot 2^k.$$
The constant term $2^{q-p-2} - 1$ is called the *bias*, and we denote this value as $E_{\max} := 2^{q-p-2} - 1$.
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 $E_{\min} := -(2^{q-p-2} - 2)$.
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 pattern 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 n and p, and (2) F_w and E_w are those obtained from the rules above. In particular, the representation is normal ($1 \le F_w < 2$) if $E_w \neq E_{\min}$ and is normal or subnormal ($0 \le F_w < 1$) 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 $w^+ := 2^{E_{\max}+1}$.

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 calculation 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 for 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++ language, 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.[2]
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.

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 parser must output w from any string representations of numbers in I. Therefore, in order to produce a shortest possible output string that is interpreted as w by a correct parser, we need to search for a number inside I that has the least number of decimal significand digits. This is the basic frame of all of algorithms by Steele and White, Grisu family, and Ryū.

3. Flow of Grisu-Exact

3.1 Overview

The main idea of Grisu is that:

(a) the number of decimal significand digits is invariant under multiplications by 10^k's, and

(b) we can map any floating-point number into a prescribed interval by multiplying some appropriate 10^k, so that

(c) we only need to find a way of searching for a number in that interval.

[2] 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.

The idea behind Grisu for (b) is that if we just give up performing perfectly exact calculation and afford some amount of error, then we can do the calculation very efficiently by using cached approximations of 10^k's. However, exact calculation up to the required precision is indeed possible if we use more but still finite amount of caches, and we can prove it using the method introduced in the correctness proof of Ryū. Details will be given in Section 4.

On the other hand, Grisu's solution to (c) can be described as follows. It starts with the right endpoint, say z, of the given interval. Then it successively cuts off least significant digits from z, until the resulting number goes out of the interval. Then the last number that stayed inside the interval should be a candidate for the shortest output.

One of the reasons why this procedure for (c) works for Grisu is that Grisu is an approximate algorithm, but we are demanding an absolutely exact algorithm in this paper so we need a slightly stronger guarantee. More specifically, the procedure above might output the interval's one of the two endpoints, but depending on the rounding mode those endpoints might not belong to the interval.[3] In order to overcome this difficulty, we need to adjust the procedure a little bit. Also, by changing successive search into binary search, we can get a lot of performance improvement. Details for this binary search will be given in Section 3.8.

3.2 Promotion of Significand to Wider Integers

Let q, p be the total number of bits and precision, respectively, for a given IEEE-754 encoding format. For example, $(q,p) = (32, 23)$ for binary32 format and $(q,p) = (64, 52)$ for binary64 format. For a given floating-point number $w = F_w \cdot 2^{E_w}$ with the given encoding format, we first transform it so that its significand is promoted to an integer type of width q.[4] Since $q \geq p+3$, all of $w, w^-, w^+, m_w^-, m_w^+$ can be promoted without any loss of precision; explicitly, let $e := E_w - q + 1$, and write

$$w = f_c \cdot 2^e$$

where $f_c := F_w 2^{q-1}$[5] is an integer, and accordingly,

$$w^- = f^- \cdot 2^e, \quad w^+ = f^+ \cdot 2^e,$$
$$m_w^- = f_m^- \cdot 2^e, \quad m_w^+ = f_m^+ \cdot 2^e$$

where

$$f^- = \begin{cases} f_c - 2^{q-p-2} & \text{if } F_w = 1 \text{ and } E_w \neq E_{\min} \\ f_c - 2^{q-p-1} & \text{otherwise} \end{cases},$$
$$f^+ = f + 2^{q-p-1},$$
$$f_m^- = \begin{cases} f_c - 2^{q-p-3} & \text{if } F_w = 1 \text{ and } E_w \neq E_{\min} \\ f_c - 2^{q-p-2} & \text{otherwise} \end{cases},$$
$$f_m^+ = f_c + 2^{q-p-2}.$$

Note that f_c is an integer in the interval $[2^{q-1}, 2^q - 2^{q-p-1}]$ if w is normal, or in the range $[2^{q-p-1}, 2^{q-1} - 2^{q-p-1}]$ if w is subnormal.[6] Other f's are also all in the interval $[0, 2^q)$. Note also that

$$f_c - f^- = \begin{cases} 2^{q-p-2} & \text{if } F_w = 1 \text{ and } E_w \neq E_{\min} \\ 2^{q-p-1} & \text{otherwise} \end{cases},$$
$$f^+ - f_c = 2^{q-p-1},$$
$$f_m^+ - f_m^- = \begin{cases} 3 \cdot 2^{q-p-3} & \text{if } F_w = 1 \text{ and } E_w \neq E_{\min} \\ 2^{q-p-1} & \text{otherwise} \end{cases}$$

are the possible lengths of the interval corresponding to w after the transform.

One must be careful that computation of f^+ can overflow when f_c is the maximum possible value. This does not cause any problem for one of nearest rounding modes, but for other rounding modes this special case must be treated carefully. See Section 3.7 for details.

3.3 Grisu Multiplier

Next we describe how to map the exponent e into a prescribed range by multiplying some appropriate 10^k to the promoted integers. We may call this 10^k the *Grisu multiplier* for w.

Let Q be a positive integer greater than q; here, Q will be the precision of the caches; we will show in Section 4 that it suffices to choose $Q = 64$ for binary32 format and $Q = 128$ for binary64 format. For a given integer k, let us write

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

where $e_k \in \mathbb{Z}$ and $2^{Q-1} \leq \varphi_k < 2^Q$. Note that in the above φ_k is not an integer; it is just a rational number in the interval $[2^{Q-1}, 2^Q)$.

By taking the logarithm, we get

$$k \log_2 10 = e_k + \log_2 \varphi_k,$$

so

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

[3] The Grisu2's solution to this problem is to search inside a conservatively reduced interval whose endpoints are always inside the actual interval. Thus, Grisu2 never output a wrong result, just suboptimal results.

[4] This is actually not strictly necessary but rather for convenience. For example, it is totally possible to use 64-bit integers instead of 32-bit integers for binary32 format. However, the width of the integral type should be at least $p + 3$ anyway.

[5] c stands for "center."

[6] Grisu uses a slightly different transform that ensures $f_c \in [2^{q-1}, 2^q)$ for both normal and subnormal numbers. In Grisu such a choice is necessary in order to more easily derive an error bound. Since Grisu-Exact is an exact algorithm, this is not necessary; therefore, we use this arguably simpler transform here.

concluding

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

We can write

$$w \cdot 10^k = (f_c \varphi_k 2^{-Q}) 2^{e+e_k+Q},$$

where $f_c \varphi_k 2^{-Q} < 2^q$, and we want to choose k satisfying

$$\alpha \leq e + e_k + Q \leq \gamma \tag{1}$$

for some predetermined integers α, γ such that $\gamma \geq \alpha + 3$.

We will choose $k := \lceil (\alpha - e - 1) \log_{10} 2 \rceil$[7] and define $\beta := e + e_k + Q$ with this k.

Proposition 3.1.
With the above choice of k, we have the inequality

$$\alpha \leq e + e_k + Q \leq \gamma.$$

Proof. From $(\alpha - e - 1) \log_{10} 2 \leq k$, we get

$$2^{\alpha - e - 1} \leq 10^k = \varphi_k \cdot 2^{e_k},$$

thus

$$\alpha - e - 1 \leq e_k + \log_2 \varphi_k, \quad \alpha \leq e + e_k + (\log_2 \varphi_k + 1).$$

Since $\log_2 \varphi_k$ is in the interval $[Q-1, Q)$, in order to have $\alpha \leq e + e_k + (\log_2 \varphi_k + 1)$ we should in fact have

$$\alpha \leq e + e_k + Q$$

because e, e_k, Q, α are all integers. On the other hand, note that

$$\begin{aligned} k &< (\alpha - e - 1) \log_{10} 2 + 2 \\ &< (\alpha - e - 1) \log_{10} 2 + 4 \log_{10} 2 \\ &\leq (\gamma - e) \log_{10} 2 \end{aligned}$$

since $\gamma \geq \alpha + 3$. Thus,

$$2^{\gamma - e} > 10^k = \varphi_k \cdot 2^{e_k},$$

and taking the logarithm gives

$$\gamma - e > e_k + \log_2 \varphi_k, \quad \gamma > e + e_k + \log_2 \varphi_k.$$

Again, since $\log_2 \varphi_k \in [Q-1, Q)$ and all of e, e_k, Q, γ are integers, we conclude

$$\gamma \geq e + e_k + Q.$$

This completes the proof. □

In [1], the author's favored choice of α is -63. However, in Grisu-Exact, we will set α to be not that small. It is this choice of α that allows us to use binary search rather than linear search, which is perhaps the most crucial factor of performance improvement.

3.4 Calculating k and β

In [1], k is computed using floating-point $\log$ function directly. However, we can do it better. Consider the following hexadecimal expansion of $\log_{10} 2$:

$$\log_{10} 2 = \texttt{0x0.4d104d427de7fbcc}\cdots$$

Hence, we can write

$$\begin{aligned} \log_{10} 2 = 2^{-32}&(\texttt{0x4d104d42}) \\ &+ 2^{-32}(\texttt{0x0.7de7fbcc}\cdots). \end{aligned}$$

Therefore, for an integer n,

$$\begin{aligned} n \log_{10} 2 = 2^{-32}&(n \times \texttt{0x4d104d42}) \\ &+ 2^{-32}(n \times \texttt{0x0.7de7fbcc}\cdots). \end{aligned}$$

We claim that when $n \in [-2^{16}, 2^{16}]$, $\lfloor n \log_{10} 2 \rfloor$ can be computed as

$$\lfloor n \log_{10} 2 \rfloor = (n \times \texttt{0x4d104d42}) \texttt{>>}_{\text{ar}} 32,$$

where $\texttt{>>}_{\text{ar}}$ is the arithmetic shift. [8]

We consider the case when n is nonnegative first. Since $0 \leq n \leq 2^{16}$, we have

$$n \times \texttt{0x0.7de7fbcc}\cdots < \texttt{0x7de8}.$$

Hence, whenever

$$\left((n \times \texttt{0x4d104d42}) \bmod 2^{32} \right) + \texttt{0x7de8} < 2^{32}, \tag{2}$$

we can safely drop the term

$$2^{-32}(n \times \texttt{0x0.7de7fbcc}\cdots).$$

One can check by direct computation that (2) is indeed true for all $n \in [0, 2^{16}]$. Our reference implementation [4] contains a program verifying this.

On the other hand, suppose $n < 0$. Note that

$$\lfloor n \log_{10} 2 \rfloor = - \lceil -n \log_{10} 2 \rceil,$$

and

$$\begin{aligned} -n \log_{10} 2 = 2^{-32}&((-n) \times \texttt{0x4d104d42}) \\ &+ 2^{-32}((-n) \times \texttt{0x0.7de7fbcc}\cdots). \end{aligned}$$

Clearly, $-n \log_{10} 2$ is not an integer, so

$$\lceil -n \log_{10} 2 \rceil = \left\lfloor 2^{-32}((-n) \times \texttt{0x4d104d42}) \right\rfloor + 1$$

by (2). On the other hand, the fractional part of $(-n) \times$ `0x4d104d42`, which is nothing but the last 32 bits of $(-n) \times$ `0x4d104d42`, is not zero, thus we have

$$\lceil -n \log_{10} 2 \rceil = \left\lceil 2^{-32}((-n) \times \texttt{0x4d104d42}) \right\rceil,$$

[7] Our choice of k is almost identical to that of [1]. However, many popular implementations of Grisu use a slightly different choice of k that allows them to use much fewer amount of caches compared to that offered by the original paper. This is possible due to the choice of α and γ that are far apart from each other. In this paper, we will take advantage of the fact that α and γ are chosen closely, so such a reduction is much harder.

[8] In modern Intel CPU's, there is no direct instruction for multiplying a 64-bit integer variable with a 64-bit constant, while there is an instruction for multiplying a 64-bit integer variable with a 32-bit constant. Hence, using a 32-bit constant instead of a longer one results in a much better performance.

so

$$\lfloor n \log_{10} 2 \rfloor = -\left\lceil 2^{-32}((-n) \times \texttt{0x4d104d42}) \right\rceil,$$

where the right-hand side is exactly the arithmetic right-shift of $n \times \texttt{0x4d104d42}$ by 32. Thus, the claim is proved.

Using the claim, we can compute k:

$$k = -\lfloor -(\alpha - e - 1) \log_{10} 2 \rfloor.$$

Clearly, $-(\alpha - e - 1)$ for all reasonable values of α and e for binary32 and binary64 formats are inside the range $[-2^{16}, 2^{16}]$.

In the original Grisu,, e_k's are pre-computed with 10^k's and stored as cache, but we can use the same idea to compute

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

on runtime without sacrificing performance a lot. Here, we use the following hexadecimal expansion of $\log_2 10$:

$$\begin{aligned} \log_2 10 &= 3 + \log_2 \frac{5}{4} \\ &= \texttt{0x3.5269e12f346e2bf9}\cdots \end{aligned}$$

Using the similar trick, one can show that for any integer $n \in [-2^{16}, 2^{16}]$, we have

$$\lfloor n \log_2 10 \rfloor = 3n + \Big((n \times \texttt{0x5269e12f}) \gg_{\text{ar}} 32\Big).$$

Again, our reference implementation [4] contains a program verifying this.

3.5 The Greatest Number with the Smallest Number of Digits

Let w_L and w_R be the left and the right endpoints of the interval associated to w, and f_L, f_R be the corresponding promoted significands so that

$$w_L = f_L \cdot 2^e, \quad w_R = f_R \cdot 2^e.$$

Now we multiply the Grisu multiplier:

$$\begin{aligned} x &:= w_L \cdot 10^k = f_L \varphi_k 2^{-Q} 2^{\beta}, \\ y &:= w \cdot 10^k = f_c \varphi_k 2^{-Q} 2^{\beta}, \\ z &:= w_R \cdot 10^k = f_R \varphi_k 2^{-Q} 2^{\beta}. \end{aligned}$$

Then the resulting interval I from x to z is where we find a minimum digit number. Let $\delta := z - x$, the length of the interval. Note that I might or might not contain its endpoints.

As explained in Section 3.1, our basic strategy is to start from z and then cutting off least significant digits successively until the resulting number goes outside of I. This procedure can be reformulated as the procedure of finding the greatest integer κ and corresponding $\left\lfloor \frac{z}{10^\kappa} \right\rfloor$ such that:

- $z \bmod 10^\kappa \le \delta$ if $x \in I$, or
- $z \bmod 10^\kappa < \delta$ if $x \notin I$.

The actual algorithm for finding κ and $\left\lfloor \frac{z}{10^\kappa} \right\rfloor$ will be explained in Section 3.8; here, let us just focus on why this procedure is, if possibly done efficiently, able to give an answer satisfying the minimum-length output criterion. (We will also ignore the correct rounding criterion for a moment.) The following proposition is a slight extension of Theorem 6.2 of [1], accounting for the left boundary.

Proposition 3.2.
Assuming $z \in I$, the number $\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa$ is the greatest number in I having the smallest number of decimal digits.

Consequently, assuming $z \in I$, $\left\lfloor \frac{z}{10^\kappa} \right\rfloor \times 10^{\kappa - k}$ is a decimal representation of the floating-point number w with the smallest number of digits.

To be precise, by *number of decimal digits* of a nonzero real number z we mean

$$\inf \left\{ \lfloor \log_{10} |s| \rfloor + 1 : z = s \cdot 10^l \text{ for some } s, l \in \mathbb{Z} \right\}.$$[9]

The number of decimal digits of 0 might be defined to be either 0 or 1, depending on the usage; however, in our case, the interval I will always avoid 0, so we do not need to care about this special case.

Proof. We will not write down a formal proof of this proposition; rather, we will sketch a higher level description of the insight behind it. Translation of this description into a formal proof should be not difficult. Let the decimal expansion of z be something like

$$z = 12345.6789\cdots$$

and κ is say, -2, so that

$$\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa = 12345.67.$$

By the assumption on κ, we know that the number

$$12345.6$$

is too small to be inside I. Now, if there is a number in I with only 6 decimal digits, then that number should look like

$$\text{xxxxx.x}$$

But since 12345.6 is outside of I, we should have

$$\text{xxxxx.x} > 12345.6.$$

Hence, xxxxx.x should be something like 12345.7; however this immediately implies that xxxxx.x is strictly larger than z, which is of course impossible. Therefore, 12345.67 should be indeed a number with the smallest number of digits. Of course, it is the greatest among those numbers; the next number with the same number of digits is 12345.68, and it is already strictly greater than z. ☐

[9] As usual, we define $\inf \emptyset = \infty$.

So far, so good. But what happens if $z \notin I$? If our $\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa$ is strictly smaller than z, or equivalently, if

$$z \bmod 10^\kappa > 0,$$

then there is no problem; $\left\lfloor \frac{z}{10^\kappa} \right\rfloor \times 10^{\kappa-k}$ is a shortest representation of the input. However, if $z \bmod 10^\kappa = 0$, then the resulting output is a representation of w_R, which is not valid. To resolve this problem, we find the greatest $\kappa' \leq \kappa$ such that:

- $10^{\kappa'} \leq \delta$ if $x \in I$, or[10]
- $10^{\kappa'} < \delta$ if $x \notin I$.

Then, we can use $z - 10^{\kappa'}$ instead of z.

First of all, $z - 10^{\kappa'}$ is clearly inside I. Also, it can be seen that the number of decimal digits of $z - 10^{\kappa'}$ is that of z plus $\kappa - \kappa'$. For example, if $z = 12345.67$, $\kappa = -2$, and $\kappa' = -4$ then

$$z - 10^{\kappa'} = 12345.6699.$$

In other words, subtracting $10^{\kappa'}$ decreases the last nonzero decimal digit of z by one, and then adding $\kappa - \kappa'$ number of 9's at the end. By the condition on κ', we know that

$$z - 10^{\kappa'+1} = 12345.669$$

should not be in I, so any number in I should look like

$$1234.669\text{xxx}\cdots$$

where the first x is nonzero. Hence, no number in I can have less decimal digits than 12345.6699. With this intuition, it is not hard to write down a formal proof of the following proposition:

Proposition 3.3.
If every number in I is strictly smaller than $\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa = z$ and strictly greater than $\left\lfloor \frac{z}{10^{\kappa+1}} \right\rfloor 10^{\kappa+1}$, then

$$z - 10^{\kappa'} = \left(\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^{\kappa-\kappa'} - 1 \right) 10^{\kappa'}$$

is the greatest number in I having the smallest number of decimal digits.

Consequently, assuming $z \notin I$ and $\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa = z$, $\left(\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^{\kappa-\kappa'} - 1 \right) \times 10^{\kappa'-k}$ is a decimal representation of the floating-point number w with the smallest number of digits.

3.6 Search Range of κ and Conditions on (α, γ)

Recall that we want to find the greatest integer κ such that

$$z \bmod 10^\kappa \leq \delta \quad \text{or} \quad z \bmod 10^\kappa < \delta, \tag{3}$$

depending on the boundary condition. Note that

$$\begin{aligned} \delta &= (f_R - f_L)\varphi_k 2^{-Q} 2^\beta \\ &\geq 2^{q-p-2} \cdot 2^{Q-1} \cdot 2^{-Q} \cdot 2^\beta \\ &= 2^{q-p-3} \cdot 2^\beta \geq 2^{q-p-3} \cdot 2^\alpha. \end{aligned} \tag{4}$$

Therefore, the inequality (3) trivially follows if we have

$$10^\kappa < 2^{q-p-3} \cdot 2^\alpha,$$

or equivalently,

$$\kappa < (q - p - 3 + \alpha) \log_{10} 2.$$

Therefore, the maximum κ satisfying the inequality (3) should satisfy

$$\kappa \geq \lceil (q - p - 3 + \alpha) \log_{10} 2 \rceil - 1.$$

On the other hand, note that δ is always strictly less than z; hence, the inequality (3) cannot hold when $10^\kappa > z$.[11] Note that

$$z = f_R \varphi_k 2^{-Q} 2^\beta < 2^q \cdot 2^Q \cdot 2^{-Q} \cdot 2^\gamma = 2^{q+\gamma}, \tag{5}$$

so we always have $10^\kappa > z$ if $\kappa \geq (q + \gamma) \log_{10} 2$. Hence, the maximum κ satisfying the inequality (3) should satisfy

$$\kappa \leq \lceil (q + \gamma) \log_{10} 2 \rceil - 1.$$

Consequently, we only need to inspect the inequality (3) for κ's in the range

$$\begin{aligned} \lceil (q - p - 3 + \alpha) \log_{10} 2 \rceil - 1 &\leq \kappa \\ &\leq \lceil (q + \gamma) \log_{10} 2 \rceil - 1. \end{aligned} \tag{6}$$

Note that when $\kappa \geq 0$, arguably the inequality (3) is mostly determined only by integer parts of z and δ. Since we wish to stick to bounded precision arithmetics, this is really valuable information. Hence, in order to utilize this property, we assume that our choice of α satisfies

$$\lceil (q - p - 3 + \alpha) \log_{10} 2 \rceil - 1 \geq 0,$$

or equivalently,

$$\alpha \geq -(q - p - 4).$$

In Section 3.7, we will see how to actually compute the integer parts of z and δ.

Also, we put additional restriction $\gamma \leq 0$. By the upper bound

$$z < 2^{q+\gamma}$$

[10] Though rare, the case $10^{\kappa'} = \delta$ actually happens, when $w_R - w_L$ is exactly 1.

[11] To be precise, δ is equal to z when $I = (0, 2^{q-p-1}] \cdot 10^k$. This is the only exceptional case, and for this case we are anyway excluding the left boundary, so we should not allow $z \bmod 10^\kappa = \delta$. Hence, there is no problem in eliminating $10^\kappa > z$ even in this case.

given in (5), this enables us to handle integer parts of z and δ with q-bit integers.[12] In summary, we impose following conditions on α and γ:

1. $\alpha \geq -(q-p-4)$,
2. $\gamma \leq 0$, and
3. $\alpha + 3 \leq \gamma$.

There are not so many possible choices of (α, γ) satisfying the above conditions, and in the reference implementation [4], empirically chosen values $(\alpha, \gamma) = (-5, -2)$ are used. With this specific choice, the search range of κ is $0 \leq \kappa \leq 9$ for binary32 format, and $1 \leq \kappa \leq 18$ for binary64 format.

3.7 Calculating Integer Parts of z and δ

In this section, a method of computing the integer parts of z and δ is explained. Here, we need an assumption that if the precision Q for the cache is large enough, then there is an integer $\tilde{\varphi}_k$ in the interval $[2^{Q-1}, 2^Q)$ so that

$$\begin{aligned}\lfloor z \rfloor &= \left\lfloor f_R \tilde{\varphi}_k 2^{-Q} 2^\beta \right\rfloor, \\ \lfloor \delta \rfloor &= \left\lfloor (f_R - f_L) \tilde{\varphi}_k 2^{-Q} 2^\beta \right\rfloor.\end{aligned}$$

Not like φ_k, $\tilde{\varphi}_k$ is just a Q-bit integer and does not have (possibly infinitely expanding) fractional part. Hence, we can store pre-computed caches of all $\tilde{\varphi}_k$'s and use them anytime. We will show in Section 4, using the method introduced in [2], that $Q = 2q$ is sufficient to guarantee the existence of $\tilde{\varphi}_k$'s for both binary32 and binary64 formats.

Note that modern computing platforms can indeed efficiently compute $\lfloor z \rfloor$ using the above formula. In particular, when $q = 64$ and $Q = 128$, the integer part $\lfloor z \rfloor$ of z is nothing but the upper $64 + \beta$ bits of the 192-bit integer $f_R \tilde{\varphi}_k$. Computing platforms like modern x64 machines provide an instruction for computing 128-bit result of multiplying two 64-bit integers, and an instruction for adding two 64-bit integers together with the carry bit resulting from the preceding addition. In such a platform, this computation of $\lfloor z \rfloor$ can be done like the following:

1. First, divide $\tilde{\varphi}_k$ into two 64-bit parts, $\mathrm{UPPER}(\tilde{\varphi}_k)$ and $\mathrm{LOWER}(\tilde{\varphi}_k)$.
2. Next, compute the 64-bit integers
$$\begin{aligned}a &:= \mathrm{UPPER}(f_R \cdot \mathrm{UPPER}(\tilde{\varphi}_k)), \\ b &:= \mathrm{LOWER}(f_R \cdot \mathrm{UPPER}(\tilde{\varphi}_k)), \quad \text{and} \\ c &:= \mathrm{UPPER}(f_R \cdot \mathrm{LOWER}(\tilde{\varphi}_k)).\end{aligned}$$
3. Compute a plus the carry of $b + c$, then the upper $64+\beta$-bits of the result is our desired $64 + \beta$ bits. (Recall $\beta \leq \gamma \leq 0$)

Even when the platform does not provide such fancy instructions, it is totally possible to do this computation only using standard 64-bit modular arithmetics. For example, it is possible to efficiently calculate the 128-bit result of 64-bit integer multiplication by splitting the multiplicands into 32-bit integers, multiplying those pieces, and then putting them together. See, for example, [5].

As noted before, when $f_R = f^+$, we need a special treatment of the case when f_c is the maximum possible value because in that case the proper value of f^+, which is 2^q, cannot be stored in a q-bit integer type. This is not a serious problem though, because if $f^+ = 2^q$, then obviously the upper $q + \beta$ bits of $f_R \tilde{\varphi}_k$ is nothing but the upper $q + \beta$ bits of $\tilde{\varphi}_k$.

In fact, computation of $\lfloor \delta \rfloor$ can be done without resorting to 128-bit arithmetics. Recall that δ is one of $3 \cdot 2^{q-p-3}$, 2^{q-p-2}, or 2^{q-p-1} times 10^k. Thus, except for the case $\delta = 3 \cdot 2^{q-p-3} \cdot 10^k$, computing $\lfloor \delta \rfloor$ is nothing but extracting upper $(q-p-j) + \beta$, $j = 1, 2, 3$ bits of $\tilde{\varphi}_k$. Fortunately, even for the case $\delta = 3 \cdot 2^{q-p-3} \cdot 10^k$, $\lfloor \delta \rfloor$ can be similarly computed:

1. Extract upper 63 bits, and upper 62 bits of $\tilde{\varphi}_k$, respectively, and call them a, b.
2. Add a and b.
3. Extract upper $(q-p-1) + \beta$ bits of $a + b$.

This procedure is based on an intuition coming from the identity $\delta = 2^{q-p-1}(2^{-1} \cdot 10^k + 2^{-2} \cdot 10^k)$, and it can be computationally verified that for all possible values of k, this computation agrees with the direct computation of $\lfloor \delta \rfloor$ using the same method of computing $\lfloor z \rfloor$. See our reference implementation [4] for a program verifying this.

3.8 Search Procedure

In this section we describe how we proceed to find κ inside the range given in (6):

$$\begin{aligned}0 \leq \lceil (q-p-3+\alpha) \log_{10} 2 \rceil - 1 &\leq \kappa \\ &\leq \lceil (q+\gamma) \log_{10} 2 \rceil - 1.\end{aligned}$$

First, let us denote the integer and fractional parts of z, δ as:

$$\begin{aligned}z^{(i)} &:= \lfloor z \rfloor, \quad z^{(f)} := z - z^{(i)}, \\ \delta^{(i)} &:= \lfloor \delta \rfloor, \quad \delta^{(f)} := \delta - \delta^{(i)}\end{aligned}$$

and write

$$z^{(i)} = 10^\kappa s_\kappa + r_\kappa \tag{7}$$

for given κ, where s_κ, r_κ are nonnegative integers with $r_\kappa < 10^\kappa$. Then we know

$$(z \mod 10^\kappa) = r_\kappa + z^{(f)}$$

and

$$\left\lfloor \frac{z}{10^\kappa} \right\rfloor = s_\kappa.$$

[12] In fact, the original proposal of the Grisu-Exact algorithm did not have this restriction. Instead, it splitted z and δ into two q-bit integers and handled them separately. It turned out that removing this splitting by imposing $\gamma \leq 0$ did not result in a dramatic speedup, but it certainly simplified the whole algorithm a lot.

Note that if $r_\kappa > \delta^{(i)}$, then we immediately conclude $z \mod 10^\kappa > \delta$, and if $r_\kappa < \delta^{(i)}$, then we immediately conclude $z \mod 10^\kappa < \delta$. In these cases, we can just safely forget about the fractional parts $z^{(f)}$ and $\delta^{(f)}$. When $r_\kappa = \delta^{(i)}$, then we do need to compare $z^{(f)}$ and $\delta^{(f)}$, but we can do that without actually computing them. See Section 3.9 for details for this comparison; in this section, let us just assume that we can compare $z^{(f)}$ and $\delta^{(f)}$ whenever needed.

Now, the very first thing to do is to compute $z^{(i)}$ and $\delta^{(i)}$ according to the procedure explained in the previous section. And then, we take an initial guess κ_0 of κ. In this paper, we will just take a predetermined value for κ_0 (in the reference implementataion [4], we chose $\kappa_0 = 2$ for binary32 and $\kappa_0 = 3$ for binary64), but there can be a better strategy for picking the initial guess.

Next, we compute the corresponding s_{κ_0} and r_{κ_0}. To do that, we perform integer division of $z^{(i)}$ by 10^{κ_0}. Integer division is a notoriously slow operation to actually perform, but since the divisor is a known constant 10^{κ_0}, it can be replaced by a series of simpler operations; this technique is known as *Barrett reduction*, and most modern compilers perform this optimization automatically.

Next, we can compare $z \mod 10^{\kappa_0}$ to δ:

- if $r_{\kappa_0} > \delta^{(i)}$, then $z \mod 10^{\kappa_0} > \delta$, and
- if $r_{\kappa_0} < \delta^{(i)}$, then $z \mod 10^{\kappa_0} < \delta$, and
- if $r_{\kappa_0} = \delta^{(i)}$, then compare $z^{(f)}$ to $\delta^{(f)}$ using the method explained Section 3.9:
  - if $z^{(f)} > \delta^{(f)}$, then $z \mod 10^{\kappa_0} > \delta$, and
  - if $z^{(f)} < \delta^{(f)}$, then $z \mod 10^{\kappa_0} < \delta$, and
  - if $z^{(f)} = \delta^{(f)}$, then $z \mod 10^{\kappa_0} = \delta$.

Now, depending on the boundary condition, we can conclude whether or not if the inequality (3) is satisfied for $\kappa = \kappa_0$. If it is not satisfied (that is, $z \mod 10^{\kappa_0}$ is larger), then that means our choice $\kappa = \kappa_0$ was too large, and if it is satisfied (that is, $z \mod 10^{\kappa_0}$ is smaller), then maybe we can find a bigger κ that still satisfies (3). We deal with each case separately.

3.8.1 Case I: Decreasing Search (When $10^{\kappa_0} s_{\kappa_0} \notin I$)

First, consider the case when $z \mod 10^{\kappa_0}$ is larger, so $10^{\kappa_0} s_{\kappa_0} \notin I$. This means our choice $\kappa = \kappa_0$ was too large, so for a fixed positive integer λ, we try to compute $s_{\kappa-\lambda}$ and $r_{\kappa-\lambda}$ to see if $z \mod 10^{\kappa-\lambda}$ is still bigger than δ. Note that if we write

$$s_{\kappa-\lambda} = 10^\lambda s + \eta$$

for nonnegative integers s, η with $\eta < 10^\lambda$ and plug it into (7), we get

$$z^{(i)} = 10^\kappa s + (10^{\kappa-\lambda}\eta + r_{\kappa-\lambda}),$$

so we conclude

$$s_\kappa = s, \quad r_\kappa = 10^{\kappa-\lambda}\eta + r_{\kappa-\lambda}.$$

Hence, we can first compute η and $r_{\kappa-\lambda}$ by dividing r_κ by $10^{\kappa-\lambda}$, and then compute $s_{\kappa-\lambda}$ as

$$s_{\kappa-\lambda} = 10^\lambda s_\kappa + \eta. \tag{8}$$

However, this is not a good idea, because we wish to iterate this procedure with varying κ; here, the divisor $10^{\kappa-\lambda}$ is therefore not a fixed constant, which means we cannot utilize Barrett reduction here. To resolve this issue, note that

$$10^{\kappa_0-\kappa} r_\kappa = 10^{\kappa_0-\lambda}\eta + 10^{\kappa_0-\kappa} r_{\kappa-\lambda}. \tag{9}$$

Here, if we know $10^{\kappa_0-\kappa} r_\kappa$ instead of r_κ, then we can still compute η and $10^{\kappa_0-\kappa} r_{\kappa-\lambda}$ with Barrett reduction. And if we know $10^{\kappa_0-\kappa}\delta^{(i)}$ in addition, then we can still proceed the comparison procedure:

- if $10^{\kappa_0-\kappa} r_{\kappa-\lambda} > 10^{\kappa_0-\kappa}\delta^{(i)}$, then $z \mod 10^{\kappa-\lambda} > \delta$, and
- if $10^{\kappa_0-\kappa} r_{\kappa-\lambda} < 10^{\kappa_0-\kappa}\delta^{(i)}$, then $z \mod 10^{\kappa-\lambda} < \delta$, and
- if $10^{\kappa_0-\kappa} r_{\kappa-\lambda} = 10^{\kappa_0-\kappa}\delta^{(i)}$, then compare $z^{(f)}$ to $\delta^{(f)}$ using the method explained in a later section:
  - if $z^{(f)} > \delta^{(f)}$, then we conclude $z \mod 10^{\kappa-\lambda} > \delta$, and
  - if $z^{(f)} < \delta^{(f)}$, then we conclude $z \mod 10^{\kappa-\lambda} < \delta$, and
  - if $z^{(f)} = \delta^{(f)}$, then we conclude $z \mod 10^{\kappa-\lambda} = \delta$.

If what we conclude in the above is that the inequality (3) is not satisfied, then that means our choice of $\kappa - \lambda$ is still too large. In this case, we compute

$$10^{\kappa_0-(\kappa-\lambda)} r_{\kappa-\lambda} = 10^\lambda \left(10^{\kappa_0-\kappa} r_{\kappa-\lambda}\right) \quad \text{and}$$
$$10^{\kappa_0-(\kappa-\lambda)}\delta^{(i)} = 10^\lambda \left(10^{\kappa_0-\kappa}\delta^{(i)}\right),$$

replace κ by $\kappa - \lambda$, and do the procedure again with a different λ. Here, we also do not need to care about overflow, because from the equation (9) we know that

$$0 \le 10^{\kappa_0-\kappa} r_{\kappa-\lambda} < 10^{\kappa_0-\lambda},$$

so

$$0 \le 10^{\kappa_0-(\kappa-\lambda)} r_{\kappa-\lambda} < 10^{\kappa_0} < 2^q.$$

Computation of $10^{\kappa_0-(\kappa-\lambda)}\delta^{(i)}$ is also okay, because we already know $r_{\kappa_0} \ge \delta^{(i)}$ from the beginning.

On the other hand, if our conclusion is that the inequality (3) is satisfied, then we can maybe find a bigger κ that still satisfies the condition, so we do not replace κ, and just repeat the procedure with a smaller λ.

We continue iterating until we are sure that the current κ is the smallest κ that fails to satisfy (3). Then, we compute the corresponding $s_{\kappa-1}$ again using (8) and (9) and replace κ by $\kappa - 1$.

Now, with the newly computed s_κ and κ we are almost done, but when the right endpoint is not contained in the interval we still need to deal with the case when $z = s_\kappa$. Note that $z = s_\kappa$ means $r_\kappa = z^{(f)} = 0$, so we first check if these are true. Details of how to check if $z^{(f)} = 0$ will be given in Section 3.10. If we confirmed that $z = s_\kappa$ is the case and is not allowed, then we need to compute the greatest integer $\kappa' \le \kappa$ satisfying

$$10^{\kappa'} \le \delta \quad \text{or} \quad 10^{\kappa'} < \delta \tag{10}$$

depending on the boundary condition on the left endpoint. Note that if we start from $\kappa' = \kappa$ and successively compare $10^{\kappa'}$ with δ and decrease κ' by 1 when $10^{\kappa'}$ is still bigger,[13] then the procedure always terminates before κ' becomes negative because of the condition on α:

$$\alpha \ge -(q - p - 4),$$

because this condition gives us

$$2 \le 2^{q-p-3+\alpha} \le \delta.$$

Therefore, we almost never need to look at the fractional part of δ when checking if (10) is satisfied. As usual, $10^{\kappa'} < \delta^{(i)}$ implies $10^{\kappa'} < \delta$ and $10^{\kappa'} > \delta^{(i)}$ implies $10^{\kappa'} > \delta$. Since $10^{\kappa'}$ is always an integer, $10^{\kappa'} = \delta^{(i)}$ implies either $10^{\kappa'} = \delta$ when δ is an integer, or $10^{\kappa'} < \delta$ otherwise. When $x \in I$ so that the inequality we are looking at is $10^{\kappa'} \le \delta$, then thus we completely do not need to care about the fractional part of δ, and when $x \notin I$, we only need to know if $\delta^{(f)} = 0$ or not. In fact, inspecting if $\delta^{(f)} = 0$ holds or not is very simple; see Section 3.10 for details.

To initiate the procedure, we need to know the value 10^κ. However, in fact this is not necessary. From the search procedure for κ, we know $10^{\kappa_0 - \kappa}\delta^{(i)}$. Since comparing 10^κ with $\delta^{(i)}$ is equivalent to comparing 10^{κ_0} to $10^{\kappa_0-\kappa}\delta^{(i)}$, we can just reuse these precomputed values.

3.8.2 Case II: Increasing Search (When $10^{\kappa_0} s_{\kappa_0} \in I$)

Next, consdier the case when $z \mod 10^{\kappa_0}$ is smaller, so $10^{\kappa_0} s_{\kappa_0} \in I$. This means our choice $\kappa = \kappa_0$ can be possibly too small, so for a fixed positive integer λ, we try to compute $s_{\kappa+\lambda}$ and $r_{\kappa+\lambda}$ to see if $z \mod 10^{\kappa+\lambda}$ is still smaller than δ. Note that if we write

$$s_\kappa = 10^\lambda s + \eta$$

for nonegative integers s, η with $\eta < 10^\lambda$ and plug it into (7), we get

$$z^{(i)} = 10^{\kappa+\lambda} s + (10^\kappa \eta + r_\kappa),$$

so we conclude

$$s_{\kappa+\lambda} = s, \quad r_{\kappa+\lambda} = 10^\kappa \eta + r_\kappa.$$

We can compute s and η by performing integer division of s_κ by 10^λ, and since 10^λ is a known constant, Barrett reduction can be applied. Of course, computation of $r_{\kappa+\lambda}$ cannot overflow because $r_{\kappa+\lambda} \le z^{(i)}$ by definition.

Again, we compare $r_{\kappa+\lambda}$ with δ, and if $r_{\kappa+\lambda}$ is too large, that means $\kappa + \lambda$ is too large, so do the same thing with a smaller λ. If $r_{\kappa+\lambda}$ is smaller than δ, then still we might be able to choose a bigger κ, so replace κ by $\kappa + \lambda$ and do the same thing with a smaller λ, and this iteration continues until we are sure that the current κ is the maximum κ satisfying the inequality (3). Note that in this iteration we need to multiply η by 10^κ, so we need to store the value of 10^κ and replace it by $10^{\kappa+\lambda}$ when necessary.

After that, we repeat the procedure of finding κ' if necessary as we did for the first case. Here, we need to know the initial value of $10^{\kappa'} = 10^\kappa$, but we already have keep track of that value so we can just use it.

3.9 Comparing Fractional Parts

Now we explain how to compare $z^{(f)}$ and $\delta^{(f)}$. Note that we only need to compare them when we encounter the situation $r_\kappa = \delta^{(i)}$ for some κ. This means that

$$z = 10^\kappa s_\kappa + r_\kappa + z^{(f)} = 10^\kappa s_\kappa + \delta + (z^{(f)} - \delta^{(f)}).$$

Remeber the definition $\delta = z - x$; hence, it follows that

$$x = 10^\kappa s_\kappa + (z^{(f)} - \delta^{(f)}).$$

Note that since when $\kappa = 0$ we absolutely have the inequality (3), so we actually do not need to compare $z^{(f)}$ and $\delta^{(f)}$ when we are sure that $\kappa < 1$. That means we can safely assume $\kappa \ge 1$ here. Thus, $10^\kappa s_\kappa$ is an even integer. Since

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

we can draw the following conclusion:

1. $z^{(f)} \ge \delta^{(f)}$ if and only if $x^{(i)}$ is an even integer,
2. $z^{(f)} < \delta^{(f)}$ if and only if $x^{(i)}$ is an odd integer, and
3. $z^{(f)} = \delta^{(f)}$ if and only if x is an integer,

where $x^{(i)} := \lfloor x \rfloor$ is the integer part of x. Therefore, the problem of comparing $z^{(f)}$ and $\delta^{(f)}$ can be reduced into checking the parity of the integer part of x, and inspecting if x is an integer or not. Since we are able to compute the integer part of $x = w_L \cdot 10^k$ just like z and δ, it only remains to contrive a way of checking if x is an integer or not.[14]

We can indeed check that by looking at the definition

$$x = w_L \cdot 10^k = f_L \cdot 2^{e+k} \cdot 5^k.$$

Recall that f_L is one of f_m^-, f^-, or f_c, depending on the rounding mode. More precisely,

[13] Thus, searching procedure for κ' is, not like that for κ, linear. There might be a way to optimize this, but the frequency of occasions where the search is actually needed is not that large anyway.

[14] It is worth mentioning that although it is possible to compute the integer part of x, it is best to avoid computing it, because the computation of integer part of x or z is a relatively heavy operation. This is the reason why we try hard to compute everything in terms of $\lfloor z \rfloor$ and $\lfloor \delta \rfloor$ as much as possible.

1. $f_L = f_c - 2^{q-p-3}$ if the rounding mode is one of round to nearest's, and $F_w = 1$ and $E_w \neq E_{\min}$. Since $F_w = 1$ implies $f_c = 2^{q-1}$, we have $f_L = 2^{q-p-3}(2^{p+2} - 1)$ in this case.
2. $f_L = f_c - 2^{q-p-2}$ if the rounding mode is one of round to nearest's, and $F_w \neq 1$ or $E_w = E_{\min}$.
3. $f_L = f_c - 2^{q-p-2}$ if $F_w = 1$ and $E_w \neq E_{\min}$ and one of the following conditions are satisfied:
   (a) The rounding mode is round toward $+\infty$ and the input is a positive number, or
   (b) The rounding mode is round toward $-\infty$ and the input is a negative number, or
   (c) The rounding mode is round away from 0.

   Since $F_w = 1$ implies $f_c = 2^{q-1}$, we have $f_L = 2^{q-p-2}(2^{p+1} - 1)$ in this case.
4. $f_L = f_c - 2^{q-p-1}$ if $F_w \neq 1$ or $E_w = E_{\min}$, and one of the following conditions are satisfied:
   (a) The rounding mode is round toward $+\infty$ and the input is a positive number, or
   (b) The rounding mode is round toward $-\infty$ and the input is a negative number, or
   (c) The rounding mode is round away from 0.
5. $f_L = f_c$ if one of the following conditions are satisfied:
   (a) The rounding mode is round toward $-\infty$ and the input is a positive number, or
   (b) The rounding mode is round toward $+\infty$ and the input is a negative number, or
   (c) The rounding mode is round toward 0.

Since the minimum unit of F_w is 2^{-p} and $f_c := F_w 2^{q-1}$, it follows that f_c is always a multiple of 2^{q-p-1}. With this information, we deal with each case separately.

3.9.1 Case I: $f_L = f_c - 2^{q-p-3}$, $F_w = 1$ and $E_w \neq E_{\min}$

In this case, we know $f_L = 2^{q-p-3}(2^{p+2} - 1)$, thus

$$x = 2^{(q-p-3)+e+k} \cdot 5^k \cdot (2^{p+2} - 1).$$

Since $2^{p+2} - 1$ is an odd integer, x is an integer if and only if:

1. $(q - p - 3) + e + k \geq 0$ and
2. Either $k \geq 0$ or $5^{-k} \mid (2^{p+2} - 1)$.

Note that for both binary32 ($p = 23$) and binary64 ($p = 52$) formats, $2^{p+2} - 1$ is not a multiple of 5.[15] Thus, the second condition is actually equivalent to $k \geq 0$. According to our choice of k:

$$k = \lceil (\alpha - e - 1) \log_{10} 2 \rceil ,$$

the condition $k \geq 0$ is equivalent to

$$(\alpha - e - 1) \log_{10} 2 > -1,$$

which can be rewritten as

$$\alpha - e - 1 > -\log_2 10, \quad e < \alpha - 1 + \log_2 10,$$

or equivalently,

$$e \leq \alpha + 2.$$

On the other hand, for the first condition, note that

$$(q - p - 3) + e + \lceil (\alpha - e - 1) \log_{10} 2 \rceil \geq 0$$

if and only if

$$(q - p - 3) + e + (\alpha - e - 1) \log_{10} 2 > -1,$$

if and only if

$$e \log_{10} 5 > -(q - p - 2) - (\alpha - 1) \log_{10} 2.$$

Simplfiying the above gives

$$\begin{aligned} e &> -(q - p - 2) \log_5 10 - (\alpha - 1) \log_5 2 \\ &= -(q - p - 3 + \alpha) \log_5 2 - (q - p - 2), \end{aligned}$$

which is equivalent to

$$e \geq \lfloor -(q - p - 3 + \alpha) \log_5 2 \rfloor - (q - p - 3).$$

Therefore, x is an integer if and only if

$$\lfloor -(q - p - 3 + \alpha) \log_5 2 \rfloor - (q - p - 3) \leq e \leq \alpha + 2.$$

It is worth mentioning that due to our condition on α given in Section 3.6, the above range is always nonempty; that means,

$$\lfloor -(q - p - 3 + \alpha) \log_5 2 \rfloor - (q - p - 3) \leq \alpha + 2.$$

To see why, recall that the condition $\alpha \geq -(q - p - 4)$ is equivalent to

$$\lceil (q - p - 3 + \alpha) \log_{10} 2 \rceil \geq 1,$$

thus

$$(q - p - 3 + \alpha) \log_{10} 2 > 0,$$

which implies

$$-(q - p - 3 + \alpha) \log_5 2 < 0.$$

Therefore,

$$\lfloor -(q - p - 3 + \alpha) \log_5 2 \rfloor \leq -1,$$

so

$$\begin{aligned} &\lfloor -(q - p - 3 + \alpha) \log_5 2 \rfloor - (q - p - 3) \\ &\quad \leq -(q - p - 2) \leq \alpha - 2 \leq \alpha + 2. \end{aligned}$$

[15] It is easy to verify that $2^n - 1$ is a multiple of 5 if and only if n is a multiple of 4. Both 25 and 54 are not multiples of 4.

3.9.2 Case II: $f_L = f_c - 2^{q-p-2}$ and $F_w \neq 1$ or $E_w = E_{\min}$

Recall that f_c is an integer multiple of 2^{q-p-1}. Therefore, $f_L/2^{q-p-2}$ should be an odd integer. Therefore,

$$x = f_L \cdot 2^{e+k} \cdot 5^k$$

is an integer if and only if:

1. $(q-p-2)+e+k \geq 0$, and
2. Either $k \geq 0$ or $5^{-k} \mid f_L$.

Just like the previous case, the first condition can be rewritten as

$$e \geq \lfloor -(q-p-2+\alpha)\log_5 2 \rfloor - (q-p-2),$$

and the condition $k \geq 0$ can be rewritten as

$$e \leq \alpha + 2,$$

so the resulting range for $k \geq 0$ is

$$\lfloor -(q-p-2+\alpha)\log_5 2 \rfloor - (q-p-2) \leq e \leq \alpha + 2.$$

When $k < 0$, from the inequality

$$\lfloor -(q-p-3+\alpha)\log_5 2 \rfloor - (q-p-3) \leq \alpha + 2$$

we know $e > \alpha + 2$ implies $(q-p-3)+e+k \geq 0$, so in particular $(q-p-2)+e+k \geq 0$. Therefore, it suffices to chekc if 5^{-k} divides f_L. To avoid the burden of computing the division by 5^{-k} on-the-fly, we first derive an upper bound for $-k$. Note that 5^{-k} divides f_L if and only if it divides $\frac{f_L}{2^{q-p-2}} = \frac{f_c}{2^{q-p-2}} - 1$. Since the maximum possible value of f_c is $2^q - 2^{q-p-1}$, it follows that

$$\frac{f_L}{2^{q-p-2}} \leq \frac{2^q - 2^{q-p-1}}{2^{q-p-2}} - 1 = 2^{p+2} - 3.$$

Hence, if $5^{-k} > 2^{p+2} - 3$, we can never have $5^{-k} \mid f_L$. Therefore, we should have

$$5^{-k} \leq 2^{p+2} - 3,$$

or equivalently,

$$-k \leq \left\lfloor \log_5(2^{p+2} - 3) \right\rfloor .$$

In fact, it can be computationally verified that for all reasonable values of p (e.g., in the range $[2, 256]$), we have

$$\left\lfloor \log_5(2^{p+2} - 3) \right\rfloor = \lfloor (p+2)\log_5 2 \rfloor ,$$

so the condition is equivalent to

$$-k \leq \lfloor (p+2)\log_5 2 \rfloor .$$

In terms of e, we can rewrite the above as

$$\lceil (\alpha - e - 1)\log_{10} 2 \rceil \geq -\lfloor (p+2)\log_5 2 \rfloor ,$$

or equivalently,

$$(\alpha - e - 1)\log_{10} 2 > -\lfloor (p+2)\log_5 2 \rfloor - 1,$$

thus we get

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

and we can rewrite this as

$$e \leq (\alpha - 2) + \lceil (\lfloor (p+2)\log_5 2 \rfloor + 1)\log_2 10 \rceil .$$

Now, if e satisfies

$$\alpha + 3 \leq e \leq (\alpha - 2) + \lceil (\lfloor (p+2)\log_5 2 \rfloor + 1)\log_2 10 \rceil ,$$

we know

$$0 < -k \leq \lfloor (p+2)\log_5 2 \rfloor .$$

Thus we have $-k \in [1, 10]$ for binary32 format ($p = 23$), and $-k \in [1, 23]$ for binary64 format ($p = 52$).

For the range $-k \in [1, 10]$, table-based approach makes sense: compute the remainder of f_L when divided by one of pre-computed constants $5^1, \cdots, 5^{10}$ according to the runtime value of $-k$. Then we can conclude x is an integer if and only if the remainder is zero. For the range $-k \in [1, 23]$, completely table-based approach might be too much. Instead, we can utilize a binary search trick. First, check if $-k$ is greater than or equal to 12. If that is the case, divide f_L by the constant 5^{12}. If the remainder is not zero, then x is not an integer. If the remainder is zero, then replace $-k$ by $-k - 12$. Then, we have $-k \in [0, 11]$, and now table-based approach makes sense.

3.9.3 Case III: $f_L = f_c - 2^{q-p-2}$, $F_w = 1$ and $E_w \neq E_{\min}$

In this case, we know $f_L = 2^{q-p-2}(2^{p+1} - 1)$, thus

$$x = 2^{(q-p-2)+e+k} \cdot 5^k \cdot (2^{p+1} - 1).$$

This case is almost same as the Case I. However, for binary32 format ($p = 23$), $2^{p+1} - 1$ is actually a multiple of 5. It is not nonetheless a multiple of 5^2, so the x is an integer if and only if:

1. $(q-p-2)+e+k \geq 0$, and
2. $k \geq -1$ for binary32 format, $k \geq 0$ for binary64 format.

As we have seen in Case II, the first condition is equivalent to

$$e \geq \lfloor -(q-p-2+\alpha)\log_5 2 \rfloor - (q-p-2),$$

and the second condition can be equivalent to

$$e \leq \alpha + 5 \quad \text{or} \quad e \leq \alpha + 2,$$

thus we conclude that x is an integer if and only if

$$\lfloor -(q-p-2+\alpha)\log_5 2 \rfloor - (q-p-2) \leq e \leq \alpha + 5$$

for binary32 format and

$$\lfloor -(q-p-2+\alpha)\log_5 2 \rfloor - (q-p-2) \leq e \leq \alpha + 2$$

for binary64 format.

3.9.4 Case IV: $f_L = f_c - 2^{q-p-1}$ and $F_w \neq 1$ or $E_w = E_{\min}$

Since f_c is an integer multiple of 2^{q-p-1}, so is f_L. Again, just like Case II, since

$$x = f_L \cdot 2^{e+k} \cdot 5^k,$$

we can conclude that x is an integer if and only if:

1. Either $(q - p - 1) + e + k \geq 0$ or $2^{-e-k} \mid f_L$, and
2. Either $k \geq 0$ or $5^{-k} \mid f_L$.

We know that $k \geq 0$ is equivalent to $e \leq \alpha + 2$, and $e > \alpha + 2$ implies $(q - p - 3) + e + k \geq 0$, so in particular $(q - p - 1) + e + k \geq 0$. Hence, the above condition is equivalent to:

1. $\lfloor -(q - p - 1 + \alpha) \log_5 2 \rfloor - (q - p - 1) \leq e \leq \alpha + 2$, or
2. $e \geq \alpha + 3$ and $5^{-k} \mid f_L$, or
3. $e \leq \lfloor -(q - p - 1 + \alpha) \log_5 2 \rfloor - (q - p)$ and $2^{-e-k} \mid f_L$.

For the second subcase, again we derive an upper bound for $-k$:

$$\frac{f_L}{2^{q-p-1}} \leq \frac{2^q - 2^{q-p-1}}{2^{q-p-1}} - 1 = 2^{p+1} - 2,$$

thus

$$-k \leq \left\lfloor \log_5(2^{p+1} - 2) \right\rfloor = \lfloor (p + 1) \log_5 2 \rfloor$$

where the last equality holds for all reasonable values for p (e.g., in the range $[2, 256]$), and again this is equivalent to

$$e \leq (\alpha - 2) + \lceil (\lfloor (p + 1) \log_5 2 \rfloor + 1) \log_2 10 \rceil .$$

If e is in the range

$$\alpha + 3 \leq e \leq (\alpha - 2) + \lceil (\lfloor (p + 1) \log_5 2 \rfloor + 1) \log_2 10 \rceil ,$$

then we have $-k \in [1, 10]$ for binary32 format ($p = 23$) and $-k \in [1, 22]$ for binary64 format ($p = 52$), so we can proceed just like Case II.

For the third subcase, note that checking $2^{-e-k} \mid f_L$ is nothing but comparing $-e - k$ to the the number of trailing zeros of the bit representation of f_L. Some modern CPU's have instructions doing exactly that, but there are also simple ways of comparing those two without using such instructions. For example, produce q-bit mask consisting of $-e - k$ number of trailing ones padded with leading zeros, and perform bitwise AND with f_L. If the result is zero, then f_L has at least $-e-k$ number of trailing zeros, so 2^{-e-k} divides f_L. If the result is different from f_L, then 2^{-e-k} does not divide f_L. Or, we can perform bitwise shifts of f_L to the right and then to the left by $-e - k$ bits, and then compare the result with f_L. Of course, doing these bitwise operations might require some care regarding the possibility $-e - k \geq q$, but that is not a big deal. In fact, one can derive that the inequality $-e - k < q$ is equivalent to

$$e \geq -q + 1 + \lfloor -(q + \alpha - 1) \log_5 2 \rfloor .$$

3.9.5 Case V: $f_L = f_c$

This case is not really different from Case IV. From

$$x = f_L \cdot 2^{e+k} \cdot 5^k,$$

again we conclude that x is an integer if and only if:

1. Either $(q - p - 1) + e + k \geq 0$ or $2^{-e-k} \mid f_L$, and
2. Either $k \geq 0$ and $5^{-k} \mid f_L$,

and everything we have discussed for Case IV equally applies here as well.

3.10 Checking If z and δ Are Integers

In order to prevent the algorithm to return z as its output when the right endpoint of the interval I is not included in I, we need to know if z and δ is an integer or not. This also can be done similarly. Recall that

$$z = f_R \cdot 2^{e+k} \cdot 5^k$$

and f_R is one of f_m^+, f^+, or f_c, depending on the rounding mode. More precisely,

1. $f_R = f_c + 2^{q-p-2}$ if the rounding mode is one of round to nearest's.
2. $f_R = f_c + 2^{q-p-1}$ if one of the following conditions are satisfied:
   (a) The rounding mode is round toward $-\infty$ and the input is a positive number, or
   (b) The rounding mode is round toward $+\infty$ and the input is a negative number, or
   (c) The rounding mode is round toward 0.
3. $f_R = f_c$ if one of the following conditions are satisfied:
   (a) The rounding mode is round toward $+\infty$ and the input is a positive number, or
   (b) The rounding mode is round toward $-\infty$ and the input is a negative number, or
   (c) The rounding mode is round away from 0.

Dealing with these are not different from Section 3.9. For the Case I, we proceed just like Case II of 3.9, and for the Case II and III, we proceed just like Case IV or V of 3.9.

To check if δ is an integer or not, recall that

$$\delta = (f_R - f_L) \cdot 2^{e+k} \cdot 5^k$$

and $f_R - f_L$ is one of the followings:

1. $f_R - f_L = 3 \cdot 2^{q-p-3}$ if the rounding mode is one of round to nearest's, $F_w = 1$ and $E_w \neq E_{\min}$.
2. $f_R - f_L = 2^{q-p-2}$ if $F_w = 1$ and $E_w \neq E_{\min}$ and one of the following conditions are satisfied:
   (a) The rounding mode is round toward $+\infty$ and the input is a positive number, or

(b) The rounding mode is round toward $-\infty$ and the input is a negative number, or

(c) The rounding mode is round away from 0.

3. $f_R - f_L = 2^{q-p-1}$ for all other cases.

Hence, δ is an integer if and only if:

1. $e + k$ is greater than or equal to one of $-(q-p-3)$, $-(q-p-2)$, or $-(q-p-1)$, depending on the conditions above, and
2. $k \geq 0$.

As we have seen in Section 3.9, this is equivalent to

$$\lfloor -(q-p-j+\alpha)\log_5 2 \rfloor - (q-p-j) \leq e \leq \alpha + 2$$

where $j = 3, 2,$ or 1.

3.11 Correct Rounding Search

3.11.1 Some Theoretical Conclusions

So far, we have seen how to find the greatest number with the smallest number of digits in the given interval. Next, we will see among those numbers with the smallest number of digits, how to find out the one that is closest to the original input number.

First of all, note that if our search interval is of the form $(w^- \cdot 10^k, w \cdot 10^k]$, then we do not need to do any additional things, because what we have found should be the one that is closest to the original input. On the other hand, if the search interval is of the form $[w \cdot 10^k, w^+ \cdot 10^k)$, then we just find the smallest number with the same number of digits with the number we have just found. This can be easily done using binary search, for example. Therefore, we will only focus on nearest rounding modes in this section.

For simplicity of presentation, let us assume we did not need to find κ', so $\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa$ is the greatest number in I with the smallest number of digits. When κ' was actually necessary, we can just replace κ in the following discussions by κ'.

Among all numbers with the same number of digits with $\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa$,

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

are the numbers that are closest to y. Note that these two are actually the same except only when the fractional part of $\frac{y}{10^\kappa}$ is exactly $\frac{1}{2}$. In fact, these numbers are almost always in the search interval I. To see why, first, note that

$$\left\lfloor \frac{y}{10^\kappa} \right\rfloor \leq \left\lceil \frac{y}{10^\kappa} - \frac{1}{2} \right\rceil \leq \left\lfloor \frac{y}{10^\kappa} + \frac{1}{2} \right\rfloor \leq \left\lfloor \frac{y}{10^\kappa} \right\rfloor + 1.$$

If the fractional part of $\frac{y}{10^\kappa}$ is strictly less than $\frac{1}{2}$, then the first two inequalities are equalities, and if it is strictly greater than $\frac{1}{2}$, then the last two inequalities are equalities. If it is exactly $\frac{1}{2}$, then the first and the last inequalities are equalities.

Suppose that

$$\left\lfloor \frac{y}{10^\kappa} \right\rfloor = \left\lfloor \frac{z}{10^\kappa} \right\rfloor.$$

In this case, $\left\lfloor \frac{y}{10^\kappa} \right\rfloor 10^\kappa$ is in I by definition of κ.[16] Hence, when the fractional part of $\frac{y}{10^\kappa}$ is strictly less than $\frac{1}{2}$, we always have $y^{(rd)} = y^{(ru)} \in I$. If the fractional part is greater than or equal to $\frac{1}{2}$, which means

$$\frac{y}{10^\kappa} \geq \left\lfloor \frac{y}{10^\kappa} \right\rfloor + \frac{1}{2}, \tag{11}$$

then we have

$$\frac{z-y}{10^\kappa} \leq \frac{z}{10^\kappa} - \left\lfloor \frac{y}{10^\kappa} \right\rfloor - \frac{1}{2}.$$

Since $y - x \leq z - y$ in general, we get

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

which can be rearranged as

$$\left\lfloor \frac{y}{10^\kappa} \right\rfloor + 1 \leq \frac{z}{10^\kappa} - \left(\frac{y-x}{10^\kappa} - \frac{1}{2} \right).$$

Note that since $\left\lfloor \frac{y}{10^\kappa} \right\rfloor 10^\kappa = \left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa \geq x$, thus we get

$$\frac{y}{10^\kappa} \geq \frac{x}{10^\kappa} + \frac{1}{2}$$

from (11), which implies

$$\left(\left\lfloor \frac{y}{10^\kappa} \right\rfloor + 1 \right) 10^\kappa \leq z.$$

In fact, since $\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa$ is the greatest integer in $I \cup \{z\}$ with the smallest number of digits, this cannot happen. Thus, in order to have

$$\left\lfloor \frac{y}{10^\kappa} \right\rfloor = \left\lfloor \frac{z}{10^\kappa} \right\rfloor,$$

actually the fractional part of $\frac{y}{10^\kappa}$ must be strictly smaller than $\frac{1}{2}$.

Next, suppose that

$$\left\lfloor \frac{y}{10^\kappa} \right\rfloor < \left\lfloor \frac{z}{10^\kappa} \right\rfloor.$$

When the fractional part of $\frac{y}{10^\kappa}$ is strictly bigger than $\frac{1}{2}$, then

$$y^{(rd)} = y^{(ru)} = \left(\left\lfloor \frac{y}{10^\kappa} \right\rfloor + 1 \right) 10^\kappa \in \left(x, \left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa \right],$$

thus $y^{(rd)} = y^{(ru)}$ is in I. Of course, this argument requires some care when we need to replace κ by κ', because in

[16] In fact, when we replace κ by κ', this might be no longer true. However, the necessity of κ' only arises when $\frac{z}{10^\kappa}$ is an integer. Since y is strictly less than z, this enforces $\left\lfloor \frac{y}{10^{\kappa'}} \right\rfloor < \left\lfloor \frac{z}{10^{\kappa'}} \right\rfloor$.

that case $\left\lfloor \frac{z}{10^{\kappa'}} \right\rfloor 10^{\kappa'}$ may not be in I. Fortunately, we do not have any issue even for that case. The only potentially problematic case is when

$$z = \left\lfloor \frac{z}{10^{\kappa'}} \right\rfloor 10^{\kappa'} = \left(\left\lfloor \frac{y}{10^{\kappa'}} \right\rfloor + 1 \right) 10^{\kappa'},$$

but then by definition of κ', we have

$$\left\lfloor \frac{y}{10^{\kappa'}} \right\rfloor 10^{\kappa'} \in I.$$

Since the fractional part of $\frac{y}{10^{\kappa'}}$ is assumed to be strictly bigger than $\frac{1}{2}$, we have

$$\frac{y}{10^{\kappa'}} > \left\lfloor \frac{y}{10^{\kappa'}} \right\rfloor + \frac{1}{2} \geq \frac{x}{10^{\kappa'}} + \frac{1}{2}, \tag{12}$$

so

$$\frac{z-y}{10^{\kappa'}} \geq \frac{y-x}{10^{\kappa'}} > \frac{1}{2}.$$

However, this implies

$$\frac{z}{10^{\kappa'}} > \frac{y}{10^{\kappa'}} + \frac{1}{2} > \left\lfloor \frac{y}{10^{\kappa'}} \right\rfloor + 1 = \frac{z}{10^{\kappa'}}$$

by (12), thus contradiction. Therefore, we always have $y^{(rd)} = y^{(ru)} \in I$ when the fractional part of $\frac{y}{10^{\kappa}}$ is strictly bigger than $\frac{1}{2}$.

On the other hand, suppose that the fractional part is less than or equal to $\frac{1}{2}$, which means

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

then it follows that

$$\frac{z-y}{10^{\kappa}} \geq \frac{z}{10^{\kappa}} - \left\lfloor \frac{y}{10^{\kappa}} \right\rfloor - \frac{1}{2}. \tag{13}$$

When $F_w \neq 1$ or $E_w = E_{\min}$, we always have

$$z - y = y - x,$$

thus the above inequality becomes

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

which can be rearranged as

$$\frac{x}{10^{\kappa}} + \left(\frac{z-y}{10^{\kappa}} - \frac{1}{2} \right) \leq \left\lfloor \frac{y}{10^{\kappa}} \right\rfloor$$

in that case. By (13), we get

$$\frac{x}{10^{\kappa}} + \left(\frac{z}{10^{\kappa}} - \left\lfloor \frac{y}{10^{\kappa}} \right\rfloor - 1 \right) \leq \left\lfloor \frac{y}{10^{\kappa}} \right\rfloor.$$

By the assumption, we have

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

thus

$$x \leq \left\lfloor \frac{y}{10^{\kappa}} \right\rfloor 10^{\kappa}.$$

Hence, under the assumption $F_w \neq 1$ or $E_w = E_{\min}$, $y^{(ru)}$ is always in I, and $y^{(rd)}$ is in I except when following conditions are all true:

1. $\frac{z}{10^{\kappa}}$ is an integer (or equivalently, $r_\kappa = z^{(f)} = 0$),
2. $\delta = 10^{\kappa}$, and
3. $x \notin I$.

These conditions cannot be all true for nearest-to-even or nearest-to-odd rounding modes, because $z = \frac{z}{10^{\kappa}} 10^{\kappa}$ is in I if the first condition is true, which implies $x \in I$ as well for those rounding modes.[17]

When $F_w = 1$ and $E_w = E_{\min}$, it is possible to have $y^{(ru)} \notin I$ as well as $y^{(rd)} \notin I$, and if that happens,

$$\left(\left\lfloor \frac{y}{10^{\kappa}} \right\rfloor + 1 \right) 10^{\kappa}$$

is the number in I with the smallest number of digits that is closest to y. Note that $y^{(rd)} \notin I$ or $y^{(ru)} \notin I$ can only happen when they were $\left\lfloor \frac{y}{10^{\kappa}} \right\rfloor 10^{\kappa}$ rather than $\left(\left\lfloor \frac{y}{10^{\kappa}} \right\rfloor + 1 \right) 10^{\kappa}$. Since

$$\left(\left\lfloor \frac{y}{10^{\kappa}} \right\rfloor + 1 \right) 10^{\kappa} \in \left(x, \left\lfloor \frac{z}{10^{\kappa}} \right\rfloor 10^{\kappa} \right],$$

$\left(\left\lfloor \frac{y}{10^{\kappa}} \right\rfloor + 1 \right) 10^{\kappa}$ might not be in I only when we have the issue of κ'. However, by the same reason we have seen before, this actually cannot happen, because assuming

$$z = \left\lfloor \frac{z}{10^{\kappa'}} \right\rfloor 10^{\kappa'} = \left(\left\lfloor \frac{y}{10^{\kappa'}} \right\rfloor + 1 \right) 10^{\kappa'},$$

implies

$$\left\lfloor \frac{y}{10^{\kappa'}} \right\rfloor 10^{\kappa'} \in I$$

by definition of κ', which is a contradiction.

In fact, whenever $y^{(rd)} \notin I$ or $y^{(ru)} \notin I$ happens, we must have

$$\left(\left\lfloor \frac{y}{10^{\kappa}} \right\rfloor + 1 \right) 10^{\kappa} = \left\lfloor \frac{z}{10^{\kappa}} \right\rfloor 10^{\kappa}.$$

To see why, let us consider the case of $y^{(rd)}$ only, as the case of $y^{(ru)}$ is similar. Note that this can happen only when the fractional part of $\frac{y}{10^{\kappa}}$ is less than or equal to $\frac{1}{2}$. When $F_w \neq 1$ or $E_w = E_{\min}$, it is clear that the above equality is true if $y^{(rd)} \notin I$. When $F_w = 1$ and $E_w \neq E_{\min}$, from

$$\left\lfloor \frac{y}{10^{\kappa}} \right\rfloor \leq \frac{x}{10^{\kappa}} \quad \text{and} \quad \left\lfloor \frac{y}{10^{\kappa}} \right\rfloor + \frac{1}{2} \geq \frac{y}{10^{\kappa}}$$

we have

$$\frac{1}{2} \geq \frac{y-x}{10^{\kappa}} = \frac{1}{2} \cdot \frac{z-y}{10^{\kappa}},$$

thus

$$\frac{z}{10^{\kappa}} \leq \frac{y}{10^{\kappa}} + 1, \tag{14}$$

[17] Again, we need to be more careful when replacing κ by κ' because $\frac{z}{10^{\kappa'}} 10^{\kappa'}$ need not be in I. However, at least $\left(\frac{z}{10^{\kappa'}} - 1 \right) 10^{\kappa'} = z - \delta = x$ should be in I when the first two conditions are true, by definition of κ'. Thus, the third condition cannot hold in this case.

which implies

$$\left\lfloor \frac{z}{10^{\kappa}} \right\rfloor \leq \left\lfloor \frac{y}{10^{\kappa}} \right\rfloor + 1 \leq \left\lfloor \frac{z}{10^{\kappa}} \right\rfloor.$$

Now, we will explain an algorithm to find the closest integer in I with the same number of digits with $\left\lfloor \frac{z}{10^{\kappa}} \right\rfloor 10^{\kappa}$. We deal with the cases $\kappa > 0$ and $\kappa = 0$ separately.

3.11.2 The Search Algorithm for $\kappa > 0$

The very first thing to do is to compute $\lfloor z - y \rfloor$. For notational simplicity, let us denote

$$\epsilon := z - y, \quad \epsilon^{(i)} := \lfloor z - y \rfloor, \quad \text{and} \quad \epsilon^{(f)} := \epsilon - \epsilon^{(i)}.$$

Computation of $\epsilon^{(i)}$ is basically the same as that of $z^{(i)}$ or $\delta^{(i)}$ as we have seen in Section 3.7, but actually it is simpler, because ϵ is always equal to $2^{q-p-2+e} \cdot 10^{k}$. Thus,

$$\epsilon^{(i)} = \left\lfloor 2^{q-p-2-Q+\beta} \tilde{\varphi}_k \right\rfloor,$$

which is nothing but the first $q - p - 2 + \beta$ bits of $\tilde{\varphi}_k$. Since

$$z = \left\lfloor \frac{z}{10^{\kappa}} \right\rfloor 10^{\kappa} + r_{\kappa} + z^{(f)},$$ [18]

it follows that

$$\frac{y}{10^{\kappa}} \pm \frac{1}{2} = \left\lfloor \frac{z}{10^{\kappa}} \right\rfloor - \frac{1}{10^{\kappa}} \left(\epsilon^{(i)} - r_{\kappa} \mp \frac{10^{\kappa}}{2} \right) + \frac{z^{(f)} - \epsilon^{(f)}}{10^{\kappa}},$$

therefore,

$$\left\lceil \frac{y}{10^{\kappa}} - \frac{1}{2} \right\rceil = \left\lfloor \frac{z}{10^{\kappa}} \right\rfloor - \left\lfloor \frac{1}{10^{\kappa}} \left(\epsilon^{(i)} - r_{\kappa} + \frac{10^{\kappa}}{2} \right) - \frac{z^{(f)} - \epsilon^{(f)}}{10^{\kappa}} \right\rfloor,$$

and similarly

$$\left\lfloor \frac{y}{10^{\kappa}} + \frac{1}{2} \right\rfloor = \left\lfloor \frac{z}{10^{\kappa}} \right\rfloor - \left\lceil \frac{1}{10^{\kappa}} \left(\epsilon^{(i)} - r_{\kappa} - \frac{10^{\kappa}}{2} \right) - \frac{z^{(f)} - \epsilon^{(f)}}{10^{\kappa}} \right\rceil.$$

We will compute the above quantities.

First, consider $\left\lceil \frac{y}{10^{\kappa}} - \frac{1}{2} \right\rceil$. Let

$$n := \left\lfloor \frac{(\epsilon^{(i)} - r_{\kappa} + (10^{\kappa}/2)) - (z^{(f)} - \epsilon^{(f)})}{10^{\kappa}} \right\rfloor.$$

Since we have assumed $\kappa > 0$, $\frac{10^{\kappa}}{2}$ is an integer. Define

$$N := \epsilon^{(i)} - r_{\kappa} + \frac{10^{\kappa}}{2},$$

then n is the unique nonnegative integer satisfying

$$10^{\kappa} n \leq N - b < 10^{\kappa}(n+1),$$

where

$$b := \begin{cases} 1 & \text{if } z^{(f)} > \epsilon^{(f)} \\ 0 & \text{if } z^{(f)} \leq \epsilon^{(f)} \end{cases}.$$

We first find the unique n' satisfying

$$10^{\kappa} n' \leq N < 10^{\kappa}(n'+1).$$

Then we can conclude $n = n'$ except possibly when $10^{\kappa} n' = N$. This happens rarely, thus we can almost ignore the fractional parts. Of course, when $10^{\kappa} n' = N$ really is the case, we do need to compare $z^{(f)}$ and $\epsilon^{(f)}$. Similarly to Section 3.9, this can be done by looking at the integer parts of ϵ and y. Note that

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

Therefore, $z^{(f)} > \epsilon^{(f)}$ if and only if

$$y > z^{(i)} - \epsilon^{(i)}.$$

To inspect this inequality, we first compute the integer part of y using the method introduced in Section 3.7, and if $\lfloor y \rfloor = z^{(i)} - \epsilon^{(i)}$, we check if y is an integer or not using the method introduced in Section 3.9. Hence, we can check if the inequality $z^{(f)} > \epsilon^{(f)}$ holds or not. If $z^{(f)} > \epsilon^{(f)}$ is actually the case, we conclude $n = n' - 1$, and if not, we conclude $n = n'$.

Computation of n' is basically the division $N/10^{\kappa}$, which is notriously slow. However, we already know the list of possible values of n: $0, 1, 2, 3, 4, 5, 6, 7$, or 8. This is because $y^{(rd)}$ is almost always in the search interval I, and there should be no number in I which has less number of digits than $\left\lfloor \frac{z}{10^{\kappa}} \right\rfloor 10^{\kappa}$. Hence, $\frac{y^{(rd)}}{10^{\kappa}}$ and $\left\lfloor \frac{z}{10^{\kappa}} \right\rfloor$ should be identical except possibly at the last digit. This is even true when $y^{(rd)}$ is not in I, because in that case $\left\lfloor \frac{z}{10^{\kappa}} \right\rfloor$ and $\frac{y^{(rd)}}{10^{\kappa}}$ should differ by 1. Therefore, we can find out n' without computing division; instead, we can use binary search.[19]

If we did increasing search for κ (that is, $10^{\kappa_0} s_{\kappa_0} \in I$; see Section 3.8.2), we have computed 10^{κ} and r_{κ}, thus we can proceed as described in the previous paragraph. If we did decreasing search for κ (that is, $10^{\kappa_0} s_{\kappa_0} \notin I$; see Section 3.8.1), we do not know 10^{κ} and r_{κ}. Instead, we know 10^{κ_0} and $10^{\kappa_0 - \kappa} r_{\kappa}$. However, this is not a big problem because we can just replace N by $10^{\kappa_0 - \kappa} N$ and 10^{κ} by 10^{κ_0} and proceed.

[18] When we did the κ' search procedure, we use 0 instead of r_{κ}.

[19] In fact, it is extremely rare to have $n_1 = 6, 7, 8, 9$. According to a test with uniformly randomly generated data, $n_1 = 0$ with probability about 50%, $n_1 = 1$ with probability about 30%, $n_1 = 2$ or 3 or 4 with probability about 15%, and $n_1 = 5$ with probability less than 1%. Thus, when we do binary search, it is better to divide the search interval into two with unequal sizes, favoring small n.

Next, consider $\left\lfloor \frac{y}{10^\kappa} + \frac{1}{2} \right\rfloor$. Let

$$n := \left\lceil \frac{(\epsilon^{(i)} - r_\kappa - (10^\kappa/2)) - (z^{(f)} - \epsilon^{(f)})}{10^\kappa} \right\rceil.$$

Note that

$$n = \left\lfloor \frac{(\epsilon^{(i)} - r_\kappa + (10^\kappa/2)) - (z^{(f)} - \epsilon^{(f)})}{10^\kappa} \right\rfloor$$

if the number inside $\lfloor\cdot\rfloor$ is not an integer, and

$$n = \left\lfloor \frac{(\epsilon^{(i)} - r_\kappa + (10^\kappa/2)) - (z^{(f)} - \epsilon^{(f)})}{10^\kappa} \right\rfloor - 1$$

otherwise. Define

$$N := \epsilon^{(i)} - r_\kappa + \frac{10^\kappa}{2},$$

then n is the unique nonnegative integer satisfying

$$10^\kappa n \leq N - b < 10^\kappa(n+1),$$

where

$$b := \begin{cases} 1 & \text{if } z^{(f)} \geq \epsilon^{(f)} \\ 0 & \text{if } z^{(f)} < \epsilon^{(f)} \end{cases}.$$

Then the rest is similar to the case of $\left\lceil \frac{y}{10^\kappa} - \frac{1}{2} \right\rceil$.

Now, we know the exact value of $y^{(rd)}$ and $y^{(ru)}$. Next, we need to check if they are inside I or not. Actually, we need to check this only when they differ by exactly 1 from $\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa$. The check can be done by comparing δ and

$$z - y^{(r)} = r_\kappa + 10^\kappa + z^{(f)},$$

where $y^{(r)}$ is either $y^{(rd)}$ or $y^{(ru)}$. If the above quantity is greater than (or greater than or equal to depending on the boundary condition) δ, then $y^{(r)}$ is not in I so we need to return $\left\lfloor \frac{z}{10^\kappa} \right\rfloor 10^\kappa$ instead.

3.11.3 The Search Algorithm for $\kappa = 0$

In this case, we cannot isolate the consideration of fractional parts of z and ϵ with integer parts easily. Instead, we utilize the followings:

$$y^{(rd)} = \left\lceil y - \frac{1}{2} \right\rceil = \left\lceil \frac{2y-1}{2} \right\rceil = \left\lfloor \frac{\lceil 2y \rceil}{2} \right\rfloor,$$
$$y^{(ru)} = \left\lfloor y + \frac{1}{2} \right\rfloor = \left\lfloor \frac{2y+1}{2} \right\rfloor = \left\lfloor \frac{\lfloor 2y \rfloor + 1}{2} \right\rfloor.$$

To see why these are true, write $2y = 2k + \rho$ or $2y = 2k + 1 + \rho$ for some $0 \leq \rho < 1$ and a nonnegative integer k. For the first case,

$$\left\lceil \frac{2y-1}{2} \right\rceil = \left\lceil k - \frac{1-\rho}{2} \right\rceil = k = \left\lfloor \frac{\lceil 2y \rceil}{2} \right\rfloor$$

and

$$\left\lfloor \frac{2y+1}{2} \right\rfloor = \left\lfloor k + \frac{1+\rho}{2} \right\rfloor = k = \left\lfloor \frac{\lfloor 2y \rfloor + 1}{2} \right\rfloor$$

and for the second case,

$$\left\lceil \frac{2y-1}{2} \right\rceil = \left\lceil k + \frac{\rho}{2} \right\rceil = \begin{cases} k & \text{if } \rho = 0 \\ k+1 & \text{otherwise} \end{cases} = \left\lfloor \frac{\lceil 2y \rceil}{2} \right\rfloor$$

and

$$\left\lfloor \frac{2y+1}{2} \right\rfloor = \left\lfloor k + 1 + \frac{\rho}{2} \right\rfloor = k + 1 = \left\lfloor \frac{\lfloor 2y \rfloor + 1}{2} \right\rfloor.$$

Hence, by computing $\lfloor 2y \rfloor$ and $\lceil 2y \rceil$, we can compute $y^{(rd)}$ and $y^{(ru)}$. Since $\lceil 2y \rceil = \lfloor 2y \rfloor + 1$ if and only if $2y$ is not an integer and $\lceil 2y \rceil = \lfloor 2y \rfloor$ if and only if $2y$ is an integer, we can compute $\lceil 2y \rceil$ by first computing $\lfloor 2y \rfloor$ and then check if $2y$ is an integer using a method similar to that introduced in Section 3.9. More concretely, one can show that $2y = f_c \cdot 2^{e+k+1} \cdot 5^k$ is an integer if and only if:

1. $\lfloor -(q-p+\alpha)\log_5 2 \rfloor - (q-p) \leq e \leq \alpha + 2$, or
2. $e \geq \alpha + 3$ and $5^{-k} \mid f_c$, or
3. $e \leq \lfloor -(q-p+\alpha)\log_5 2 \rfloor - (q-p+1)$ and $2^{-e-k-1} \mid f_c$.

Therefore, our first goal is to compute $\lfloor 2y \rfloor$. This can be done just like $\lfloor z \rfloor$ or $\lfloor \delta \rfloor$ as introduced in Section 3.7, after replacing β by $\beta + 1$. However, one need to be careful that unless $\gamma \leq -1$,[20] the result can overflow. Hence, if $\gamma = 0$, we just take the least significant bit of the result and then compute $\lfloor y \rfloor$ instead. In this way, we can compute $y^{(rd)}$ and $y^{(ru)}$.

Unlike the case $\kappa > 0$, in this case in fact we can be sure that $y^{(rd)}$ and $y^{(ru)}$ compated above are actually inside I. Recall that by (14), we know

$$z - y \leq 1$$

whenever $y^{(rd)}$ or $y^{(ru)}$ is not in I. Note that this implies

$$\delta \leq 2(z - y) \leq 2.$$

However, for nearest rounding modes, we have

$$\begin{aligned} \delta &= (f_m^+ - f_m^-)\varphi_k 2^{-Q} 2^\beta \\ &\geq 3 \cdot 2^{q-p-3} \cdot 2^{Q-1} \cdot 2^{-Q} \cdot 2^\beta \\ &= 3 \cdot 2^{q-p-4} \cdot 2^\beta \geq 3 \cdot 2^{q-p-4+\alpha} \geq 3 \end{aligned}$$

since $\alpha \geq -(q-p-4)$, thus $y^{(rd)}$ and $y^{(ru)}$ should be in I.

4. Correctness of Integer Part Computation

The paper [2] introduced a way to compute a good upper bound on the required amount of bits to reliably compute $\lfloor x \cdot 10^k \rfloor$. The paper introduced the following two lemmas; the proofs of these lemmas, which are presented in the original paper, are included in this paper for completeness.

[20] Our assumption on γ in Section 3.6 is $\gamma \leq 0$. However, our preferred choice of γ is -2, anyway.

Lemma 4.1 (Adams, 2018)**.**
Let k be a nonnegative integer, b an integer, and g a positive integer. Then for any integer u satisfying

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

we have

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

Proof. Define

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

then it suffices to show that $0 \leq \delta < 1$. First, $\delta \geq 0$ is obvious because

$$\delta > g \cdot 2^{b-u} \frac{2^u}{5^k} - \frac{g \cdot 2^b}{5^k} = 0.$$

To show $\delta < 1$, note that we have

$$2^{u-b} > \frac{5^k g}{5^k - (2^b g \mod 5^k)}$$

from the assumption on u. Hence,

$$\begin{aligned} g \cdot 2^{b-u} &< 1 - \frac{1}{5^k} \left((g \cdot 2^b) \mod 5^k \right) \\ &= 1 - \frac{1}{5^k} \cdot 5^k \left(\frac{g \cdot 2^b}{5^k} - \left\lfloor \frac{g \cdot 2^b}{5^k} \right\rfloor \right), \end{aligned}$$

so

$$g \cdot 2^{b-u} + \frac{g \cdot 2^b}{5^k} - \left\lfloor \frac{g \cdot 2^b}{5^k} \right\rfloor < 1,$$

and from

$$\frac{g \cdot 2^b}{5^k} \geq g \cdot 2^{b-u} \left\lfloor \frac{2^u}{5^k} \right\rfloor,$$

we conclude

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

□

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

$$l \leq \log_2 \max \left\{ 1, \frac{5^k g \mod 2^b}{g} \right\},$$

we have

$$\left\lfloor \frac{g \cdot 5^k}{2^b} \right\rfloor = \left\lfloor g \cdot 2^{l-b} \left\lfloor \frac{5^k}{2^l} \right\rfloor \right\rfloor.$$

Proof. The equality trivially holds for $l \leq 0$, thus we may assume

$$1 \leq \frac{5^k g \mod 2^b}{g}.$$

Define

$$\delta := g \cdot 2^{l-b} \left\lfloor \frac{5^k}{2^l} \right\rfloor - \left\lfloor \frac{g \cdot 5^k}{2^b} \right\rfloor,$$

then it suffices to show that $0 \leq \delta < 1$. First, $\delta < 1$ is obvious because

$$\delta \leq g \cdot 2^{l-b} \frac{5^k}{2^l} - \left\lfloor \frac{g \cdot 5^k}{2^b} \right\rfloor = \frac{g \cdot 5^k}{2^b} - \left\lfloor \frac{g \cdot 5^k}{2^b} \right\rfloor < 1.$$

To show $\delta \geq 0$, note that we have

$$2^l \leq \frac{5^k g \mod 2^b}{g}$$

from the assumption on l. Hence,

$$\begin{aligned} g \cdot 2^{l-b} &\leq \frac{1}{2^b} \left((g \cdot 5^k) \mod 2^b \right) \\ &= \frac{1}{2^b} \cdot 2^b \left(\frac{g \cdot 5^k}{2^b} - \left\lfloor \frac{g \cdot 5^k}{2^b} \right\rfloor \right), \end{aligned}$$

so

$$\frac{g \cdot 5^k}{2^b} - g \cdot 2^{l-b} - \left\lfloor \frac{g \cdot 5^k}{2^b} \right\rfloor \geq 0,$$

and from

$$g \cdot 2^{l-b} \left\lfloor \frac{5^k}{2^l} \right\rfloor > g \cdot 2^{l-b} \left(\frac{5^k}{2^l} - 1 \right) = \frac{g \cdot 5^k}{2^b} - g \cdot 2^{l-b},$$

we conclude

$$\delta = g \cdot 2^{l-b} \left\lfloor \frac{5^k}{2^l} \right\rfloor - \left\lfloor \frac{g \cdot 5^k}{2^b} \right\rfloor > 0.$$

□

Based on these lemmas, we will justify computations in Section 3.7 and Section 3.11.3. Note that

$$\left\lfloor f \cdot 2^e \cdot 10^k \right\rfloor = \left\lfloor g \cdot 2^{q-p-2+e+k} \cdot 5^k \right\rfloor$$

or

$$\left\lfloor f \cdot 2^e \cdot 10^k \right\rfloor = \left\lfloor (2^{p+2} - 1) \cdot 2^{q-p-3+e+k} \cdot 5^k \right\rfloor$$

for some nonnegative integer $g \in [0, 2^{p+2}]$, where f is one of f_c, f^-, f^+, f_m^-, and f_m^+. Sicne the case $f = 0$ is vacuous, we assume $f \neq 0$, so $g \in [1, 2^{p+2}]$. Also,

$$\left\lfloor 2 f_c \cdot 2^e \cdot 10^k \right\rfloor = \left\lfloor g \cdot 2^{q-p+e+k} \cdot 5^k \right\rfloor$$

for some positive integer $g \in [1, 2^{p+1} - 1]$.

4.1 Case I: $k < 0$

First, consider the case $k < 0$. Recall from Section 3.9 that $k < 0$ is equivalent to $e \geq \alpha + 3$, and this implies $(q - p - 3) + e + k \geq 0$. Therefore, $\lfloor f \cdot 2^e \cdot 10^k \rfloor$ is indeed of the form

$$\left\lfloor \frac{g \cdot 2^b}{5^{-k}} \right\rfloor$$

for some $b \geq 0$. By Lemma 4.1, we know

$$\lfloor f \cdot 2^e \cdot 10^k \rfloor = \left\lfloor f \cdot 2^{e+k-u} \cdot \left(\left\lfloor \frac{2^u}{5^{-k}} \right\rfloor + 1 \right) \right\rfloor$$

if u satisfies

$$u \geq (q - p - 2) + e + k + 1 \\ + \max_{g=1,\cdots,2^{p+2}} \left\lfloor \log_2 \frac{5^{-k} g}{5^{-k} - (2^{(q-p-2)+e+k} g \bmod 5^{-k})} \right\rfloor$$

and

$$u \geq (q - p - 3) + e + k + 1 \\ + \left\lfloor \log_2 \frac{5^{-k}(2^{p+2} - 1)}{5^{-k} - (2^{(q-p-3)+e+k}(2^{p+2} - 1) \bmod 5^{-k})} \right\rfloor .$$

Also, we have

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

if u satisfies

$$u \geq (q - p - 2) + e + k + 2 \\ + \max_{g=1,\cdots,2^{p+1}} \left\lfloor \log_2 \frac{5^{-k} g}{5^{-k} - (2^{(q-p)+e+k} g \bmod 5^{-k})} \right\rfloor .$$

We want to set

$$u = k - e_k = k - \lfloor k \log_2 10 \rfloor + Q - 1 \\ = Q - \lfloor k \log_2 5 \rfloor - 1,$$

so that

$$\tilde{\varphi}_k := \left\lfloor \frac{2^u}{5^{-k}} \right\rfloor + 1 = \lfloor 2^{u-k} \cdot 10^k \rfloor + 1 \\ = \lfloor \varphi_k \cdot 2^{u-k+e_k} \rfloor + 1 = \lfloor \varphi_k \rfloor + 1.$$

Therefore, in order to guarantee correctness of computations in Section 3.7 and Section 3.11.3, it is sufficient that Q satisfies the following three inequalities for all $e \geq \alpha + 3$:

1.

$$Q \geq q + e + \lfloor k \log_2 10 \rfloor + 2 + \\ \max_{g=1,\cdots,2^{p+2}} \left\lfloor \log_2 \frac{5^{-k}}{5^{-k} - (2^{(q-p-2)+e+k} g \bmod 5^{-k})} \right\rfloor ,$$

2.

$$Q \geq q + e + \lfloor k \log_2 10 \rfloor + 1 + \\ \left\lfloor \log_2 \frac{5^{-k}}{5^{-k} - (2^{(q-p-3)+e+k}(2^{p+2} - 1) \bmod 5^{-k})} \right\rfloor ,$$

3.

$$Q \geq q + e + \lfloor k \log_2 10 \rfloor + 2 + \\ \max_{g=1,\cdots,2^{p+1}} \left\lfloor \log_2 \frac{5^{-k}}{5^{-k} - (2^{(q-p)+e+k} g \bmod 5^{-k})} \right\rfloor .$$

Using the min-max Euclid algorithm introduced in [2] (see Section 4.3), one can computationally verify that $Q = 2q$ satisfies all of these inequalities for both binary32 and binary64 formats, with our specific choice of α; see Figure 4.1. Our reference implementataion [4] includes a program computing these lower bounds shown on the figure.

4.2 Case II: $k \geq 0$

Next, consider the case $k \geq 0$. Recall from Section 3.9 that $k \geq 0$ is equivalent to $e \leq \alpha + 2$. By Lemma 4.2, we know

$$\lfloor f \cdot 2^e \cdot 10^k \rfloor = \left\lfloor f \cdot 2^{l+e+k} \cdot \left\lfloor \frac{5^k}{2^l} \right\rfloor \right\rfloor$$

if l satisfies

$$l \leq \max \left\{ 0, \min_{g=1,\cdots,2^{p+2}} \left\lfloor \log_2 \frac{5^k g \bmod 2^{-e-k-(q-p-2)}}{g} \right\rfloor \right\}$$

and

$$l \leq \max \left\{ 0, \left\lfloor \log_2 \frac{5^k (2^{p+2} - 1) \bmod 2^{-e-k-(q-p-3)}}{2^{p+2} - 1} \right\rfloor \right\} .$$

Also, we have

$$\lfloor 2 f_c \cdot 2^e \cdot 10^k \rfloor = \left\lfloor f_c \cdot 2^{l+e+k+1} \cdot \left\lfloor \frac{5^k}{2^l} \right\rfloor \right\rfloor$$

if l satisfies

$$l \leq \max \left\{ 0, \min_{g=1,\cdots,2^{p+1}} \left\lfloor \log_2 \frac{5^k g \bmod 2^{-e-k-(q-p)}}{g} \right\rfloor \right\} .$$

We want to set

$$l = e_k - k = \lfloor k \log_2 10 \rfloor - Q + 1 - k \\ = \lfloor k \log_2 5 \rfloor - Q + 1,$$

so that

$$\tilde{\varphi}_k := \left\lfloor \frac{5^k}{2^l} \right\rfloor = \lfloor 2^{-k-l} \cdot 10^k \rfloor \\ = \lfloor \varphi_k \cdot 2^{e_k - k - l} \rfloor = \lfloor \varphi_k \rfloor .$$

Therefore, in order to guarantee correctness of computations in Section 3.7 and Section 3.11.3, it is sufficient that Q satisfies the following three inequalities for all $e \leq \alpha + 2$:

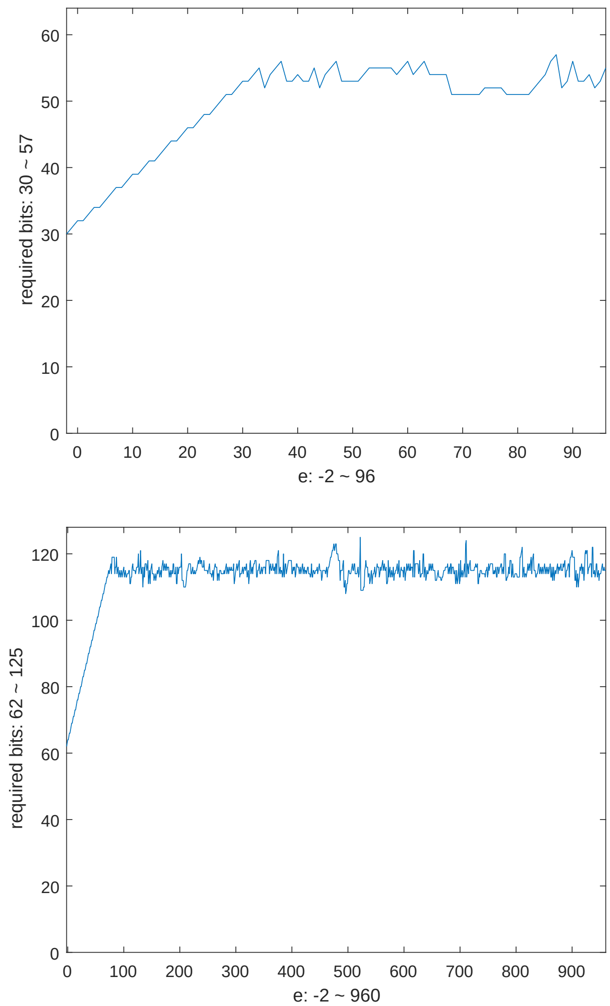

Figure 1. Lower bounds on Q for each e with $k < 0$ (top: binary32, bottom: binary64); the maximum value is 57 for binary32, 125 for binary64.

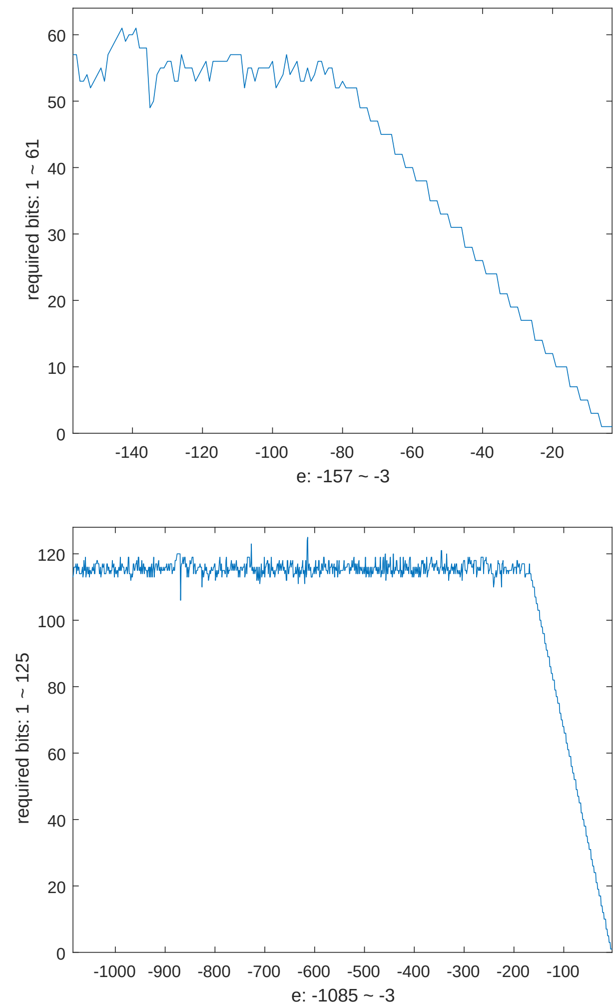

Figure 2. Lower bounds on Q for each e with $k \geq 0$ (top: binary32, bottom: binary64); the maximum value is 61 for binary32, 125 for binary64.

1.
$$Q \geq \lfloor k \log_2 5 \rfloor + 1 - \max \Big\{ 0, -p - 2 + \min_{g=1,\,\cdots\,,2^{p+2}} \left\lfloor \log_2 \left(5^k g \mod 2^{-e-k-(q-p-2)} \right) \right\rfloor \Big\},$$

2.
$$Q \geq \lfloor k \log_2 5 \rfloor + 1 - \max \Big\{ 0, -p - 2 + \left\lfloor \log_2 \left(5^k (2^{p+2} - 1) \mod 2^{-e-k-(q-p-3)} \right) \right\rfloor \Big\},$$

3.
$$Q \geq \lfloor k \log_2 5 \rfloor + 1 - \max \Big\{ 0, -p - 1 + \min_{g=1,\,\cdots\,,2^{p+1}} \left\lfloor \log_2 \left(5^k g \mod 2^{-e-k-(q-p)} \right) \right\rfloor \Big\}.$$

Again, using the min-max Euclid algorithm introduced in [2] (see Section 4.3), one can computationally verify that $Q = 2q$ satisfies all of these inequalities for both binary32 and binary64 formats, with our specific choice of α; see Figure 4.2. Our reference implementataion [4] includes a program computing these lower bounds shown on the figure.

4.3 Min-Max Euclid Algorithm

In order to compute lower bounds given in Section 4.1 and Section 4.2, we need to compute the minimum and the maximum of numbers of the form

$$ag \mod b$$

where a, b are powers of 2 or 5, and g runs over a range $[1, N] \cap \mathbb{Z}$. Since N is very large ($\sim 2^{25}$ or $\sim 2^{54}$), it is computationally too heavy to compute the minimum and the maximum directly. To resolve this issue, [2] introduced a nice algorithm, which the author called *min-max Euclid algorithm*, to compute conservative bounds on these values. Here, we will not use this algorithm straightly; rather, we use a variant, which is faster than the original version.

We wish we can post a pseudocode and explanation of the algorithm later.

5. Benchmark Results

We did a benchmark testing performances of Grisu-Exact, Grisu-Exact without performing correct rounding search (that is, omitting the procedure explained in Section 3.11), and Ryū, for the task of producing a decimal string representation of a given floating-point number. The source code for the benchmark is contained in our reference implementation [4]. As advertised before, Grisu-Exact outperforms Ryū in small-digits regime. For example, according to the benchmark results shown in Figure 5, when the number of digits is 2, the average performances are:

- Grisu-Exact: $24.92\,\mathrm{ns}$ (binary32), $39.42\,\mathrm{ns}$ (binary64)
- Grisu-Exact without correct rounding search: $23.92\,\mathrm{ns}$ (binary32), $40.16\,\mathrm{ns}$ (binary64)
- Ryū: $38.22\,\mathrm{ns}$ (binary32), $72.50\,\mathrm{ns}$ (binary64)

and when the number of digits is 6, the average performaces are:

- Grisu-Exact: $27.93\,\mathrm{ns}$ (binary32), $41.28\,\mathrm{ns}$ (binary64)
- Grisu-Exact without correct rounding search: $26.24\,\mathrm{ns}$ (binary32), $40.91\,\mathrm{ns}$ (binary64)
- Ryū: $29.46\,\mathrm{ns}$ (binary32), $62.74\,\mathrm{ns}$ (binary64)

Thus, for binary64, Grisu-Exact is about 84% faster than Ryū when the number of digits is 2, and is about 52% faster than Ryū when the number of digits is 6.

However, Grisu-Exact's performance is not really better than Ryū when the number of digits is very large, as shown in Figure 5. According to our benchmarks, it performs worse for binary32-encoded numbers with 8 or 9 digits, and it performs slightly better in terms of average for binary64-encoded long numbers but the difference is marginal. Since most exisiting floating-point numbers are of almost maximum length, it is expected that Grisu-Exact performs slightly worse than Ryū for binary32 and slightly better than Ryū for binary64 if subjected to uniformly randomly generated floating-point numbers. This is indeed the case, as shown in Figure 5. According to the benchmark results shown in Figure 5, the average performance of Grisu-Exact is about 7% slower than Ryū for binary32 format and about 5.4% faster than Ryū for binary64 format.

Nevertheless, practical input data of float-to-string conversion will not be uniformly distributed. By (mis)applying Zipf's law, it seems reasonable to assume that the distribution of the number of digits of input data might be roughly uniform. If that is indeed the case, then Grisu-Exact will certainly outperform Ryū. According to our benchmarks, the total averages of average performances over all possible numbers of digits are:

1. Grisu-Exact: $27.42\,\mathrm{ns}$ (binary32), $42.64\,\mathrm{ns}$ (binary64)
2. Grisu-Exact without correct rounding search: $25.62\,\mathrm{ns}$ (binary32), $42.21\,\mathrm{ns}$ (binary64)
3. Ryū: $32.16\,\mathrm{ns}$ (binary32), $56.41\,\mathrm{ns}$ (binary64)

Hence, if we assume the uniform distribution of number of digits, Grisu-Exact is about 17% faster than Ryū for binary32 data and is about 32% faster than Ryū for binary64 data, according to the benchmark results shown in Figure 5.

5.1 Some Notes on Our Benchmarks

5.1.1 Random Floating-Point Numbers with Given Number of Digits

It is not easy to uniformly randomly generate a floating-point number with the given number of digits. Our method is to uniformly randomly generate an integer with the given number of digits, combine it with a uniformly randomly generated exponent (among all valid decimal exponents) and a uniformly randomly generated sign, convert the result into a string, and then convert it back to a floating-point number. Of course, this will not give us the uniform distribution, because some string representations might represent the same floating-point number. Nonetheless, one can claim this will give a good approximation to the uniform distribution when the number of digits is small. When the number of digits is close to the maximum, it is highly probable the resulting number actually have a shorter representation. However, we could not think of an easy way to correctly produce a random floating-point number with a given number of digits. With this method, we produced and tested $100,000$ random floating-point numbers for each given number of digits and for each of binary32 and binary64. We used the same data set for all the algorithms.

5.1.2 Uniformly Random Floating-Point Numbers

We uniformly randomly generated $1,000,000$ q-bit integers, reinterpreted them as floating-point numbers, and then tested them. We used the same data set for all the algorithms. Since it is too hard to recognize anything if all of $1,000,000$ points are drawn on a single figure, we sampled $10,000$ of them for producing Figure 5. Averages, standard deviations, and

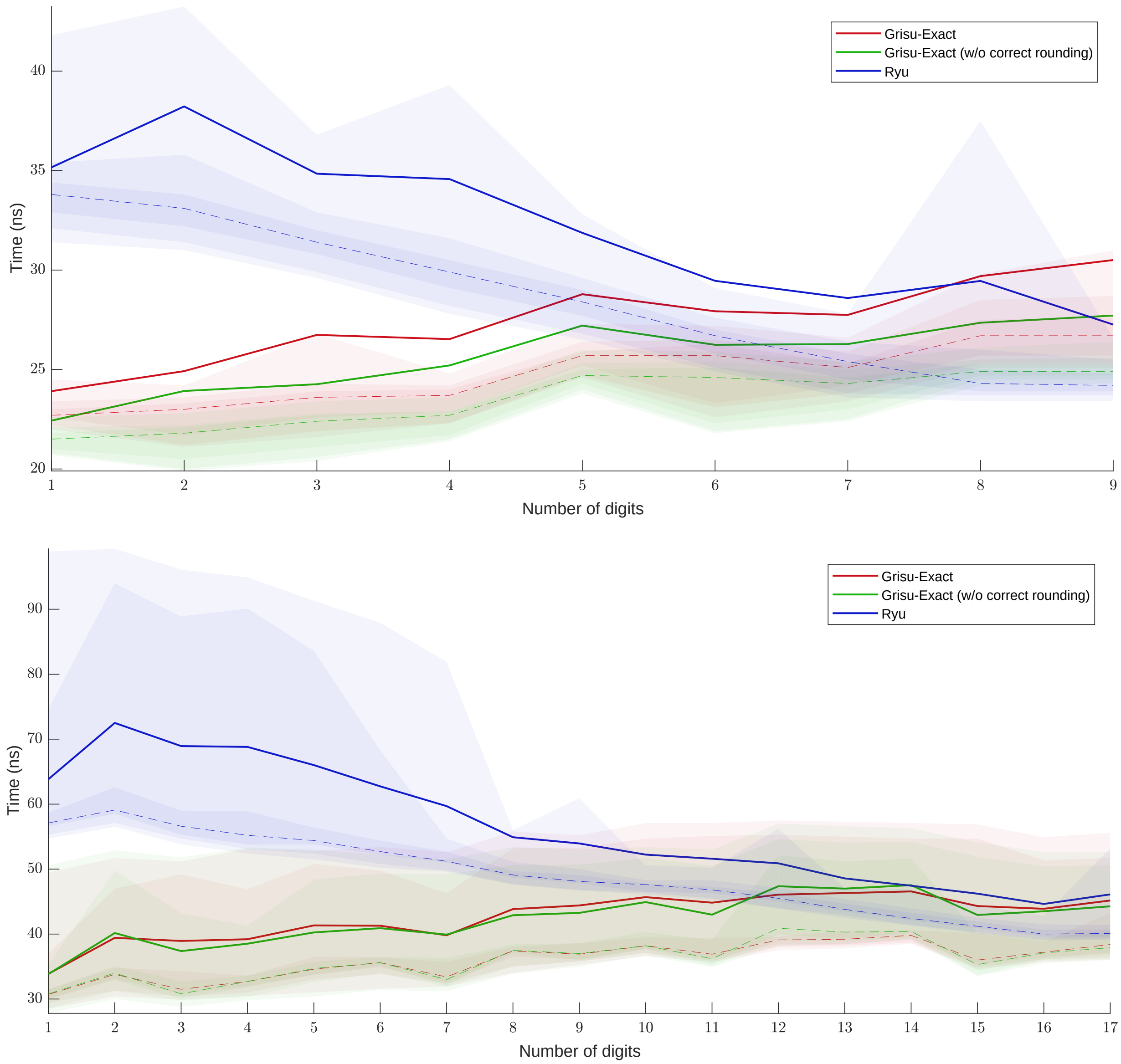

Figure 3. Performances of Grisu-Exact, Grisu-Exact without performing correct rounding search, and Ryū for random floating-point numbers with given number of digits; solid lines are averages, dashed lines are medians, and shaded regions show 30%, 50%, and 70% percentiles. (top: binary32, bottom: binary64)

medians shown in Figure 5 are calculated from the original data.

5.1.3 Procedure for Actual String Generation

Strictly speaking, Grisu-Exact as an algorithm does not include actual generation of human-readable string. Rather, it only produces a pair of integers representing the decimal significand and the decimal exponent of a given floating-point number. For the benchmark, we copied (and modified a little bit) the function producing a human-readable string from this pair from Ryū's reference implementation [6].

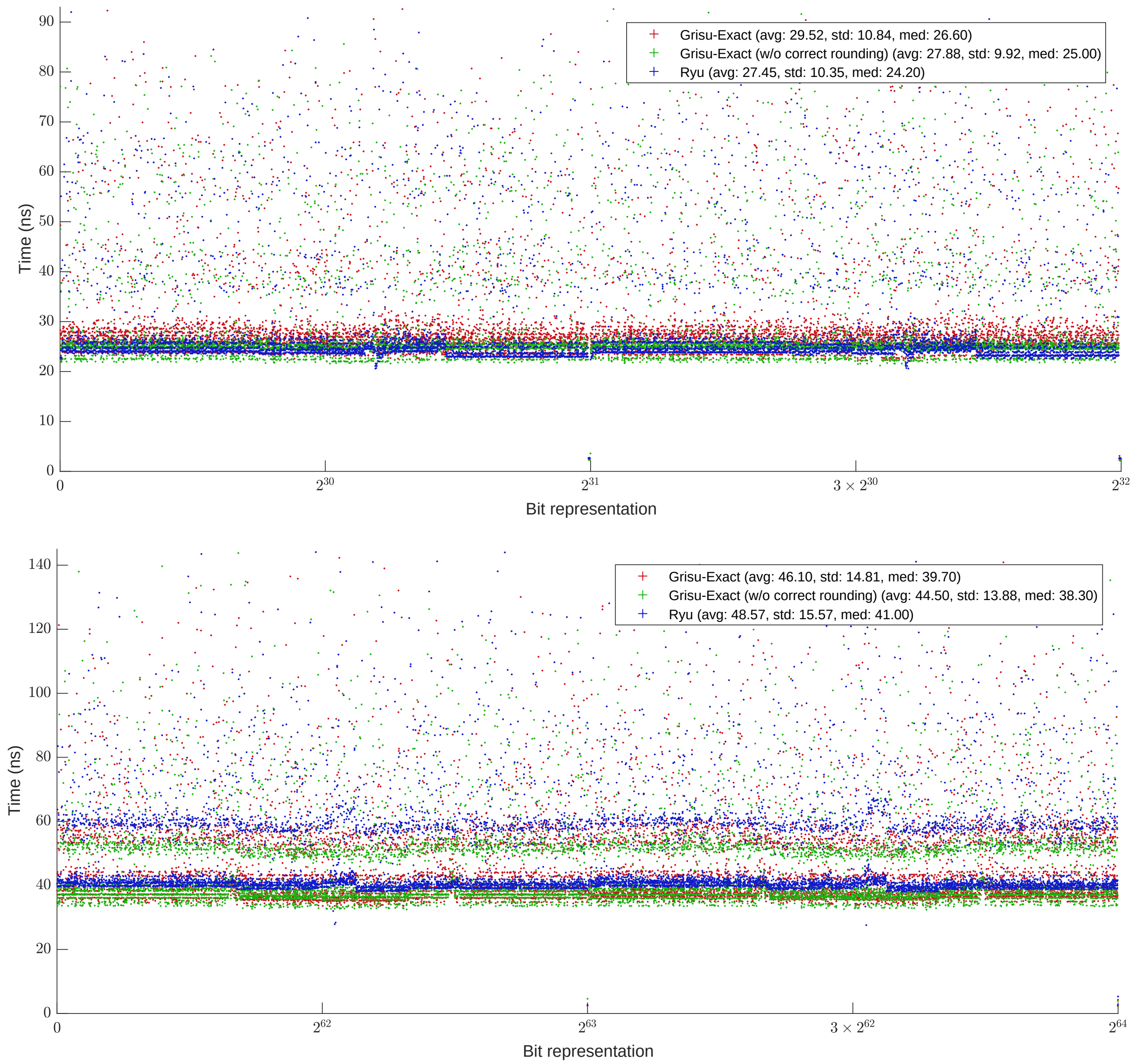

Figure 4. Performances of Grisu-Exact, Grisu-Exact without performing correct rounding search, and Ryū for uniform random floating-point numbers (top: binary32, bottom: binary64)

References

[1] 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

[2] 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

[3] 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

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

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

[6] https://github.com/ulfjack/ryu. (Jan. 2020)