computers

MDPI

Article

xjb: Fast Float to String Algorithm

Junbo Xiang and Tiejun Wang *

School of Computer Science, Chengdu University of Information Technology, Chengdu 610225, China; 3220604004@stu.cuit.edu.cn

* Correspondence: tjw@cuit.edu.cn; Tel.: +86-138-8090-0111

Abstract

Efficiently and accurately converting floating-point numbers to decimal strings remains a fundamental challenge in numerical computation, data serialization, and human–computer interaction. While modern algorithms such as Ryū, Dragonbox, and Schubfach rigorously satisfy the Steele–White criteria for correctness and minimal output length, their performance is frequently constrained by branch mispredictions, high-precision multiplication overhead, and suboptimal utilization of instruction-level parallelism. This paper introduces xjb, a novel floating-point–string conversion algorithm derived from Schubfach that systematically overcomes these bottlenecks. By restructuring the core computation to reduce instruction dependencies, adopting branchless decision logic, and exploiting SIMD instruction sets for decimal-to-ASCII formatting, xjb delivers state-of-the-art throughput across diverse hardware platforms. The algorithm requires only a single 64-by-128-bit multiplication for IEEE 754 binary64 conversions and a single 64-by-64-bit multiplication for binary32, drastically decreasing arithmetic complexity. Extensive benchmarking on AMD R7-7840H and Apple M1/M5 processors demonstrates that xjb consistently outperforms leading contemporary implementations. Notably, on the Apple M5, xjb achieves speedups of approximately 20% and 136% for binary64 and binary32 conversions, respectively, when compared to the highly optimized zmij library. The algorithm is fully compliant with the Steele–White principle; exhaustive validation over the entire binary32 space and extensive random testing across the binary64 range confirm both its theoretical soundness and practical robustness.

Keywords: floating point; printing; algorithm; performance; SIMD; branchless

Academic Editor: Paolo Bellavista

Received: 29 March 2026
Revised: 20 April 2026
Accepted: 24 April 2026
Published: 27 April 2026

1. Introduction

Floating-point–decimal string conversion is a fundamental operation in computer systems, with widespread applications across numerous domains. From scientific computing and financial systems to web services and database management, the ability to efficiently and accurately convert binary floating-point representations into human-readable decimal strings is essential. Despite its apparent simplicity, this conversion problem presents significant challenges in balancing the competing demands of correctness, performance, and output compactness.

1.1. Background and Motivation

In 1990, Steele and White [1,2] established the foundational principles for optimal floating-point printing algorithms, now widely known as the Steele–White (SW) principle. The SW principle comprises four key requirements:

- **Information Preservation**: The printed result must be parsable back to the original floating-point number without loss of precision.
- **Minimum Length**: The output string should be as short as possible while maintaining information preservation.
- **Correct Rounding**: When multiple representations satisfy the first two criteria, the algorithm must correctly round to the nearest value, with ties broken by selecting the even value.
- **Left-to-Right Generation**: The output digits should be generated sequentially from the most significant to the least significant digit.

The SW principle ensures that floating-point numbers have a unique, well-defined decimal representation that is both human-readable and machine-parsable. Algorithms satisfying the SW principle guarantee that the conversion process is reversible and produces the shortest possible output, which is crucial for data exchange, serialization, and user interface display. Over the past three decades, significant research efforts have been devoted to developing efficient algorithms. Early approaches, such as Dragon4, provided correct results but suffered from performance limitations due to arbitrary-precision arithmetic, and they were later improved by dtoa.c [3]. Grisu3 [4] pioneered the use of precomputed powers of ten to avoid expensive operations, although it occasionally fell back to slower methods. Errol [5] reduced the fallback rate through more precise error analysis. Ryū [6,7] established a new performance baseline through careful instruction scheduling and lookup table optimizations. Schubfach [8] introduced a compact and elegant solution based on the pigeonhole principle, while Grisu-Exact [9] eliminated fallbacks entirely. Dragonbox [10] reduced the number of multiplications, at the cost of more branches; yy_double [11] and yy_json [12] explored alternative computational strategies to minimize multiplication costs, but they still retained a few branches; uscale [13–15], proposed by Russ Cox, enhances floating-point printing performance in the Go programming language. Finally, zmij [16] builds upon yy_double with extensive code-level optimizations.

Despite these advances, existing algorithms still face several performance challenges that limit their effectiveness in high-throughput scenarios:

1. **Branch Prediction Penalties**: Many algorithms rely heavily on conditional branches to handle different cases, leading to frequent branch mispredictions on modern pipelined processors.
2. **High-Precision Multiplication Overhead**: The conversion process requires high-precision arithmetic operations, particularly multiplications involving large precomputed constants, which can be expensive on standard hardware.
3. **Instruction Dependency Chains**: Sequential dependencies between operations limit instruction-level parallelism and prevent efficient utilization of modern superscalar processors.
4. **Limited SIMD Utilization**: Most existing algorithms do not exploit vector instruction sets (SIMD) that are now ubiquitous in contemporary processors.

Despite the progress made by prior works, a critical research gap remains: no existing algorithm holistically addresses the four fundamental performance bottlenecks—branch mispredictions, high-precision multiplication overhead, instruction dependency chains, and underutilization of SIMD capabilities. For example,

- Schubfach [8] offers an elegant approach but suffers from suboptimal performance due to unoptimized computation flow.
- Dragonbox [10] reduces multiplications at the cost of increased branches, trading off one bottleneck for another.

- yy_double [11] and yy_json [12] are highly optimized but do not use SIMD instructions.
- zmij [16] provides competitive performance but still leaves room for improvement in instruction dependency reduction and branch optimization.

This gap necessitates a new approach that systematically integrates multiple optimization strategies within a cohesive framework, rather than trading off one bottleneck for another.

1.2. Contributions

This paper presents **xjb**, a novel floating-point–string conversion algorithm that achieves superior performance through systematic optimization of the underlying computational structure. The xjb algorithm is derived from the Schubfach algorithm and incorporates insights from yy_double and Dragonbox, but it introduces several key innovations that directly address the limitations of existing frameworks:

1. **Reduced Instruction Dependencies**: Unlike other algorithms that suffer from sequential dependencies, xjb carefully restructures the computation by decomposing d (introduce on Section 3.2) into $d//10$ and $d\%10$ instead of computing d directly. This minimizes data dependencies between operations, enabling better instruction-level parallelism and improved pipeline utilization on modern superscalar processors.
2. **Minimized Multiplication Operations**: Building on insights from yy_double, but without the trade-off of increased branches, xjb reduces the number of expensive high-precision multiplications required during conversion, significantly decreasing the computational overhead while maintaining branch efficiency. For IEEE 754 binary64, only one 64-bit by 128-bit multiplication is required, and for IEEE 754 binary32, only one 64-bit by 64-bit multiplication is needed.
3. **Mitigated Branch Prediction Penalties**: Through branchless programming techniques and careful case analysis, xjb addresses the branch prediction problem that plagues algorithms like Dragonbox. All branches in xjb are designed as unlikely branches, and the core conversion of normal floating-point numbers is completely branch-free, minimizing conditional branches that could lead to prediction failures.
4. **SIMD Instruction Utilization**: Unlike most existing algorithms that neglect SIMD potential, xjb is designed from the ground up to leverage SIMD instructions (NEON for ARM64, AVX512/SSE4.1/SSE2 for x86-64) for efficient decimal-to-ASCII conversion, fully exploiting the vector processing capabilities of contemporary processors.
5. **Concise Core Implementation**: Despite its sophisticated optimizations, xjb maintains a compact and readable core implementation, facilitating adoption and maintenance.

These innovations work synergistically to address all key performance bottlenecks simultaneously, filling the research gap identified in the previous section. The xjb algorithm supports both IEEE 754 single-precision (binary32) and double-precision (binary64) floating-point formats, which are the most widely used floating-point representations in modern computing. For simplicity, this paper uses float to refer to IEEE 754 binary32 and double to refer to IEEE 754 binary64.

1.3. Evaluation Overview

We conducted extensive benchmarking of xjb across diverse hardware platforms, including AMD R7-7840H and Apple M1/M5 processors. Our evaluation demonstrates that xjb outperforms state-of-the-art algorithms in most scenarios while maintaining full compliance with the SW principle. The algorithm exhibits excellent portability and scalability, making it suitable for deployment across a wide range of systems, from embedded devices to high-performance servers.

The remainder of this paper is organized as follows: Section 2 reviews the IEEE 754 floating-point representation and establishes the mathematical foundation for the conversion problem. Section 3 elucidates the core principles and derivation of the xjb algorithm and describes the implementation details and optimizations. Section 4 presents experimental results comparing xjb against existing algorithms. To conclude, Section 5 offers a comprehensive summary of the paper.

1.4. Explanation of Special Symbols in This Article

We provide special explanations for the special symbols used in the formulae of this article, as shown in Table 1.

Table 1. Explanation of special symbols in this article.

Symbol	Brief Explanation	Example
%	Integer modulus operation	2 = 8%3
//	Integer division operation	1 = 5//3
<< or >>	Left or right shift of binary values	8 = 1<<3
? :	Similar to the ternary operator in C syntax	a = 1?a:b

2. IEEE 754 Floating-Point Number Representation

This section establishes the mathematical foundation for the representation of floating-point numbers and defines the notation used throughout this paper. We focus on the IEEE 754 standard, which is the most widely adopted floating-point arithmetic standard in modern computing systems.

2.1. Scope and Assumptions

For clarity of presentation, we make the following simplifying assumptions:

- We consider only positive floating-point numbers, as negative numbers differ only by a leading minus sign.
- We excluded special values (zero, NaN, and infinity) from our analysis, since these are handled separately in practice.

These assumptions are standard in the literature and do not affect the generality of our algorithm, as excluded cases can be handled with straightforward special-case logic.

2.2. Binary Representation

The IEEE 754 standard [17,18] defines two primary floating-point formats relevant to this work:

Double Precision (binary64): A 64-bit format consisting of the following:

- One sign bit (s): indicates positive ($s = 0$) or negative ($s = 1$).
- Eleven exponent bits (e): biased exponent in the range $[0, 2047]$.
- Fifty-two fraction bits (f): significant fraction in the range $[0, 2^{52} - 1]$.

Single Precision (binary32): A 32-bit format consisting of the following:

- one sign bit (s): indicates positive ($s = 0$) or negative ($s = 1$).
- Eight exponent bits (e): biased exponent in the range $[0, 255]$.
- Twenty-three fraction bits (f): significant fraction in the range $[0, 2^{23} - 1]$.

2.3. Classification of Floating-Point Numbers

We classify floating-point numbers into three categories based on their exponent and fraction fields:

1. **Subnormal Numbers** ($e = 0$ and $f \neq 0$): These represent very small values close to zero, where the implicit leading bit of the significand is 0 instead of 1.
2. **Normal Numbers** ($e \neq 0$ and $f \neq 0$): The standard case, where the implicit leading bit of the significand is 1.
3. **Irregular Numbers** ($e \neq 0$ and $f = 0$): Numbers with zero fraction field, representing powers of two.

We use the term **regular** to refer to both subnormal and normal numbers (i.e., all cases where $f \neq 0$). Unless otherwise specified, this article discusses only regular values—that is, non-irregular values.

2.4. Value Representation

The real value v of a positive floating-point number can be expressed in the unified form $v = c \cdot 2^q$, where c is the integer significand and q is the exponent. The general formula covering all cases is as follows:

$$\begin{aligned} \text{double}: v &= \left(f + (e \neq 0 \,?\, 2^{52} : 0)\right) \cdot 2^{\max(e,1)-1023-52} = c \cdot 2^q \\ \text{float}: v &= \left(f + (e \neq 0 \,?\, 2^{23} : 0)\right) \cdot 2^{\max(e,1)-127-23} = c \cdot 2^q \end{aligned} \tag{1}$$

For each category, the values decompose as follows:
Subnormal Numbers ($e = 0, f \neq 0$):

$$\text{subnormal}: \begin{cases} \text{double}: & v = f \cdot 2^{-1074} \\ \text{float}: & v = f \cdot 2^{-149} \end{cases} \tag{2}$$

Normal Numbers ($e \neq 0, f \neq 0$):

$$\text{normal}: \begin{cases} \text{double}: & v = (f + 2^{52}) \cdot 2^{e-1075} \\ \text{float}: & v = (f + 2^{23}) \cdot 2^{e-150} \end{cases} \tag{3}$$

Irregular Numbers ($e \neq 0, f = 0$):

$$\text{irregular}: \begin{cases} \text{double}: & v = 2^{52} \cdot 2^{e-1075} \\ \text{float}: & v = 2^{23} \cdot 2^{e-150} \end{cases} \tag{4}$$

2.5. Rounding Interval

A critical concept for accurate floating-point printing is the **rounding interval** R_v, which defines the range of real numbers that round to the given floating-point value v when parsed. The rounding interval is bounded by

$$\begin{aligned} v_l &= \begin{cases} \left(c - \frac{1}{2}\right) \cdot 2^q = v - 2^{q-1} & \text{if } f \neq 0 \text{ or } e \leq 1 \\ \left(c - \frac{1}{4}\right) \cdot 2^q = v - 2^{q-2} & \text{if } f = 0 \end{cases} \\ v_r &= \left(c + \frac{1}{2}\right) \cdot 2^q = v + 2^{q-1} \\ R_v &= \begin{cases} [v_l, v_r] & \text{if } f \bmod 2 = 0 \text{ (even significand)} \\ (v_l, v_r) & \text{if } f \bmod 2 = 1 \text{ (odd significand)} \end{cases} \end{aligned} \tag{5}$$

The **rounding radius** for regular floating-point numbers is $2^{q-1} = v_r - v$. The distinction between closed and open intervals at the boundaries depends on the parity of the significand, ensuring correct rounding according to the round-to-even rule specified

in the IEEE 754 standard. Any decimal number within R_v will parse back to the original floating-point value v, which is essential for ensuring the information preservation property of the SW principle.

Table 2 summarizes the valid ranges for c and q across different categories.

Table 2. Valid ranges for significand c and exponent q.

Category	Float (Binary32)	Double (Binary64)
Subnormal	$1 \le c \le 2^{23}-1, q = -149$	$1 \le c \le 2^{52}-1, q = -1074$
Normal	$2^{23}+1 \le c \le 2^{24}-1$ $-148 \le q \le 104$	$2^{52}+1 \le c \le 2^{53}-1$ $-1073 \le q \le 971$
Irregular	$c = 2^{23}, -149 \le q \le 104$	$c = 2^{52}, -1074 \le q \le 971$

3. Algorithm Principles

This section presents the algorithmic principles and mathematical foundation of the xjb floating-point–string conversion algorithm. We first introduce the overall architecture and design goals, followed by the mathematical formulation of the conversion problem.

3.1. Design Overview

This paper focuses on converting float (single-precision) and double (double-precision) floating-point numbers to decimal strings. The conversion process consists of two stages:

1. **Float-to-Decimal Conversion**: Converting binary floating-point values to decimal significand–exponent pairs (d, k).
2. **Decimal-to-String Conversion**: Formatting (d, k) into human-readable strings.

Table 3 presents the corresponding optimization methods for the identified limitations and the chapter information where they are located.

Table 3. Mapping between identified limitations and xjb's solutions.

Identified Limitation	Corresponding Solution in xjb
Frequent branch mispredictions	Branchless programming for core decision logic (Sections 3.6, 3.7 and 3.10)
High-precision multiplication overhead	Minimized multiplication count via lookup-table restructuring (Sections 3.4–3.6)
Long instruction dependency chains	Restructured computation flow to expose instruction-level parallelism (Section 3.10)
Limited SIMD utilization	SIMD-optimized ASCII generation for decimal-to-string stage (Section 3.10, Table 5)

3.2. Mathematical Foundation

Before presenting the algorithm details, we establish the mathematical framework for the float-to-decimal conversion problem.

Recall from Section 2 that any floating-point value v can be expressed in the form $v = c \cdot 2^q$, where c is the integer significand and q is the exponent. Our goal is to find the optimal decimal representation $opt = d \cdot 10^k$ that satisfies the SW principle.

As established in Section 2, regular floating-point numbers (which include all normal and subnormal numbers with non-zero fraction fields) account for the vast majority of possible floating-point values. For the purposes of algorithm derivation, we focus primarily on regular numbers, as special cases can be handled with minimal additional logic.

The valid ranges for the significand c and exponent q for regular floating-point numbers are as follows:

$$\begin{aligned} \text{float}: &\begin{cases} 1 \le c \le 2^{24}-1, c \ne 2^{23}, & q = -149 \\ 2^{23}+1 \le c \le 2^{24}-1, & -148 \le q \le 104 \end{cases} \\ \text{double}: &\begin{cases} 1 \le c \le 2^{53}-1, c \ne 2^{52}, & q = -1074 \\ 2^{52}+1 \le c \le 2^{53}-1, & -1073 \le q \le 971 \end{cases} \end{aligned} \tag{6}$$

For irregular floating-point numbers (powers of two), the ranges are

$$\begin{aligned} \text{float}: &\quad c = 2^{23}, \quad -149 \le q \le 104 \\ \text{double}: &\quad c = 2^{52}, \quad -1074 \le q \le 971 \end{aligned} \tag{7}$$

For subnormal numbers,

$$\begin{aligned} \text{float}: &\quad c \le 2^{23}-1, \quad q = -149 \\ \text{double}: &\quad c \le 2^{52}-1, \quad q = -1074 \end{aligned} \tag{8}$$

The conversion problem can now be formally stated as follows: given a floating-point value $v = c \cdot 2^q$, find the optimal decimal representation $opt = d \cdot 10^k$ such that

$$\begin{aligned} &v = c \cdot 2^q \rightarrow opt = d \cdot 10^k \\ &\text{subject to: } opt \in R_v, \quad d \in \mathbb{Z}^+, \quad k \in \mathbb{Z} \end{aligned} \tag{9}$$

where R_v is the rounding interval of v, as defined in Section 2.

For example, consider the IEEE 754 binary64 floating-point number representing 1.3. Its actual value is 1.3000000000000000444089209850062616169452667236328125, with the hexadecimal representation `3ff4cccccccccccd`. The optimal decimal representation opt satisfying the SW principle is simply `1.3`. Similarly, for the binary32 representation of 1.3, with an actual value of 1.299999952316284179687 5 and a hexadecimal representation of `3fa66666`, the optimal representation is also `1.3`.

3.3. Overview of the Schubfach Algorithm and Derivation of Our Method

This section reviews the Schubfach algorithm and presents our derivation of an optimized variant. We begin by establishing the mathematical foundation for determining the optimal decimal representation.

3.3.1. Candidate Values for the Significand d

The Schubfach algorithm identifies four candidate values for the decimal significand d:

$$d \in \{10\lfloor v \cdot 10^{-k-1} \rfloor, \lfloor 10v \cdot 10^{-k-1} \rfloor, \lfloor 10v \cdot 10^{-k-1} \rfloor + 1, 10\lfloor v \cdot 10^{-k-1} \rfloor + 10\} \tag{10}$$

The exponent k is computed as follows:

$$k = \begin{cases} \lfloor q \cdot \log_{10}(2) \rfloor & \text{if } v \text{ is regular} \\ \lfloor q \cdot \log_{10}(2) - \log_{10}(4/3) \rfloor & \text{otherwise} \end{cases} \tag{11}$$

For efficient computation on modern processors, Equation (11) can be implemented using integer arithmetic:

$$\begin{aligned} \text{double}: &\quad k = (q \cdot 315653 - (v \text{ is regular} \,?\, 0 : 131072)) \gg 20 \\ \text{float}: &\quad k = (q \cdot 1233 - (v \text{ is regular} \,?\, 0 : 512)) \gg 12 \end{aligned} \tag{12}$$

3.3.2. Decomposition into Integer and Fractional Parts

We will decompose $v \cdot 10^{-k-1}$ into its integer component $\lfloor v \cdot 10^{-k-1} \rfloor$ (floor function) and fractional component $v \cdot 10^{-k-1} - \lfloor v \cdot 10^{-k-1} \rfloor$. Let $m = \lfloor v \cdot 10^{-k-1} \rfloor$ denote the integer part and $n = v \cdot 10^{-k-1} - m$ the fractional part, where $0 \leq n < 1$. Substituting into Equation (10),

$$d \in \{10m, \lfloor 10(m+n) \rfloor, \lfloor 10(m+n) \rfloor + 1, 10m + 10\} \tag{13}$$

Since $\lfloor 10(m+n) \rfloor = 10m + \lfloor 10n \rfloor$, the candidates simplify to:

$$d \in \{10m, 10m + \lfloor 10n \rfloor, 10m + \lfloor 10n \rfloor + 1, 10m + 10\} \tag{14}$$

Based on the Schubfach algorithm and the SW principle, if the value $10m$ or $10m + 10$ falls within the range R_v, it is selected as the optimal solution d. In cases where neither $10m$ nor $10m + 10$ lies within R_v, the optimal value d is determined as either $10m + \lfloor 10n \rfloor$ or $10m + \lfloor 10n \rfloor + 1$ in accordance with the rules of correct rounding, as shown in Equation (15). We decompose d as $d = ten + one$, where $ten = 10m$ and $one \in \{0, \lfloor 10n \rfloor, \lfloor 10n \rfloor + 1, 10\}$. The problem thus reduces to determining the appropriate value of one.

$$one = \begin{cases} 0; \text{ if } 10m \in R_v \\ 10; \text{ else if } 10m + 10 \in R_v \\ \lfloor 10n \rfloor \text{ or } \lfloor 10n \rfloor + 1; \text{ else apply correct rounding} \end{cases} \tag{15}$$

3.3.3. Selection Criteria for *one*

The selection of *one* depends on the relationship between the rounding interval and the candidate values. Recall that the rounding interval for a regular floating-point number $v = c \cdot 2^q$ is $R_v = (f\%2 = 0) \,?\, [v - 2^{q-1}, v + 2^{q-1}] : (v - 2^{q-1}, v + 2^{q-1})$, where the rounding radius 2^{q-1} represents the half-unit in the last place (ulp) of v.

- **Case** $one = 0$ **(i.e.,** $d = 10m$**):** This case applies when $10m \cdot 10^k$ falls inside the rounding interval R_v. The condition is derived as follows:
  The lower bound of the rounding interval must be less than $10m \cdot 10^k$:

$$\begin{aligned} c \cdot 2^q - 10m \cdot 10^k &< 2^{q-1} \\ c \cdot 2^q - \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor \cdot 10^{k+1} &< 2^{q-1} \\ c \cdot 2^q \cdot 10^{-k-1} - \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor &< 2^{-1} \cdot 2^q \cdot 10^{-k-1} \\ n &< 2^{q-1} \cdot 10^{-k-1} \end{aligned} \tag{16}$$

  When equality holds ($n = 2^{q-1} \cdot 10^{-k-1}$, or equal to $10m \cdot 10^k = v - 2^{q-1}$), we apply the round-to-even rule, requiring c to be even:

$$one = 0 \quad \text{if} \quad 2^{q-1} \cdot 10^{-k-1} > n \;\text{ or }\; \left(2^{q-1} \cdot 10^{-k-1} = n \text{ and } c \bmod 2 = 0\right) \tag{17}$$

- **Case** $one = 10$ **(i.e.,** $d = 10m + 10$**):** This case applies when $(10m + 10) \cdot 10^k$ falls inside the rounding interval R_v. The condition is derived similarly:

The upper bound of the rounding interval must be greater than $(10m + 10) \cdot 10^k$:

$$\begin{aligned} (10m + 10) \cdot 10^k - c \cdot 2^q &< 2^{q-1} \\ \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor \cdot 10^{k+1} + 10^{k+1} - c \cdot 2^q &< 2^{q-1} \\ 1 - \left(c \cdot 2^q \cdot 10^{-k-1} - \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor \right) &< 2^{-1} \cdot 2^q \cdot 10^{-k-1} \\ 1 - n &< 2^{q-1} \cdot 10^{-k-1} \end{aligned} \tag{18}$$

When equality holds ($1 - n = 2^{q-1} \cdot 10^{-k-1}$, or equal to $(10m + 10) \cdot 10^k = v + 2^{q-1}$), we again apply round-to-even:

$$one = 10 \quad \text{if} \quad 2^{q-1} \cdot 10^{-k-1} > 1 - n \ \text{ or } \ \left(2^{q-1} \cdot 10^{-k-1} = 1 - n \text{ and } c \bmod 2 = 0 \right) \tag{19}$$

- **Case** $one \in \{\lfloor 10n \rfloor, \lfloor 10n \rfloor + 1\}$**:** When neither boundary condition applies, the optimal value lies between $10m + \lfloor 10n \rfloor$ and $10m + \lfloor 10n \rfloor + 1$. We determine one by rounding $10n$ to the nearest integer:
  - If the fractional part $\{10n\} < 0.5$: $one = \lfloor 10n \rfloor$;
  - If the fractional part $\{10n\} > 0.5$: $one = \lfloor 10n \rfloor + 1$;
  - If the fractional part $\{10n\} = 0.5$: apply round-to-even.

  For irregular floating-point numbers (powers of two), additional verification is required to ensure that the selected value lies within the rounding interval R_v, as the interval boundaries differ for these special cases.

3.3.4. Algorithm Overview

Algorithm 1 summarizes our optimized variant of the Schubfach algorithm (xjb32 and xjb64 for float and double, respectively.). Given inputs c and q, the algorithm returns d and k such that $d \cdot 10^k$ satisfies the SW principle. The computation of k follows Equatio (12); the remainder of this chapter focuses on efficient computation of d for regular floating-point numbers.

We will provide a detailed introduction to the efficient implementation of Algorithm 1 in the following sections:

1. Lookup table precomputation;
2. Efficient computation of m;
3. Fast boundary condition testing for $one \in \{0, 10\}$;
4. Efficient computation of $\lfloor 10n \rfloor$ and rounding;
5. Handling of irregular floating-point numbers;
6. Implementation of pseudocode.

In the last section of this chapter, we will briefly introduce the second stage: decimal-to-string conversion.

In essence, xjb diverges from the baseline Schubfach by transforming the conditional boundary tests (which cause branch mispredictions) into integer arithmetic and lookup operations. Furthermore, it reorders the computation of m and n to reduce data dependencies, directly addressing the instruction-level parallelism limitation identified in Section 1.1.

3.4. Lookup Table Precomputation

The algorithm in this paper employs a lookup table to store precomputed values of 10^{-k-1} for different ranges of q: $[-149, 104]$ for float and $[-1074, 971]$ for double. These lookup tables use extended precision: 64-bit for float and 128-bit for double. The reference implementation is available in (`gen.py`) https://github.com/xjb714/xjb/blob/main/py_test/gen.py (accessed on 20 April 2026).

3.4.1. Fundamental Calculation

Let B denote the bit length of each entry in the lookup table, with $B = 64$ for float and $B = 128$ for double. For any integer e_{10} (representing a power of 10), we aim to represent $10^{e_{10}}$ in the form $f \cdot 2^{\lfloor e_2 \rfloor}$, where $1 \leqslant f < 2$ and e_2 is a real number. This gives

$$f \cdot 2^{\lfloor e_2 \rfloor} = 2^{e_2} = 10^{e_{10}} \tag{20}$$

Taking the logarithm base 2 of both sides, we get

$$\lfloor e_2 \rfloor = \lfloor e_{10} \cdot \log_2(10) \rfloor \tag{21}$$

Algorithm 1: The xjb Algorithm for Float-to-Decimal Conversion

Require: Floating-point components c (significand) and q (exponent)
Ensure: Decimal representation $d \cdot 10^k$ satisfying the SW principle

$c \cdot 2^q \leftarrow v$
if v is regular **then**
  $k \leftarrow \lfloor q \cdot \log_{10}(2) \rfloor$
else
  $k \leftarrow \lfloor q \cdot \log_{10}(2) - \log_{10}(4/3) \rfloor$
end if
$m \leftarrow \lfloor v \cdot 10^{-k-1} \rfloor$
$n \leftarrow v \cdot 10^{-k-1} - m$
$ten \leftarrow 10m$
$\delta \leftarrow 10n - \lfloor 10n \rfloor$ {fractional part of $10n$}
if $\delta = 0.5$ **then**
  if $\lfloor 10n \rfloor \bmod 2 = 0$ **then**
    $one \leftarrow \lfloor 10n \rfloor$ {round to even}
  else
    $one \leftarrow \lfloor 10n \rfloor + 1$
  end if
else if $\delta < 0.5$ **then**
  $one \leftarrow \lfloor 10n \rfloor$ {round to nearest}
else
  $one \leftarrow \lfloor 10n \rfloor + 1$ {round to nearest}
end if
if v is irregular **then**
  if $\delta > 2^{q-2} \cdot 10^{-k}$ **then**
    $one \leftarrow \lfloor 10n \rfloor + 1$
  end if
  if $2^{q-2} \cdot 10^{-k-1} \geq n$ **then**
    $one \leftarrow 0$
  end if
else
  if $2^{q-1} \cdot 10^{-k-1} > n$ or $\left(2^{q-1} \cdot 10^{-k-1} = n \text{ and } c \bmod 2 = 0\right)$ **then**
    $one \leftarrow 0$ {minimum length}
  end if
  if $2^{q-1} \cdot 10^{-k-1} > 1 - n$ or $\left(2^{q-1} \cdot 10^{-k-1} = 1 - n \text{ and } c \bmod 2 = 0\right)$ **then**
    $one \leftarrow 10$ {minimum length}
  end if
end if
$d \leftarrow ten + one$ {information preservation}

return d, k

Solving for f gives

$$f = \frac{10^{e_{10}}}{2^{\lfloor e_{10} \cdot \log_2(10) \rfloor}} \tag{22}$$

The lookup table entries are computed using upward rounding:

$$\begin{aligned} lookup[e_{10}] &= \lceil f \cdot 2^{B-1} \rceil \\ &= \lceil \frac{10^{e_{10}}}{2^{\lfloor e_{10} \cdot \log_2(10) \rfloor}} \cdot 2^{B-1} \rceil \\ &= \lceil 10^{e_{10}} \cdot 2^{B-1-\lfloor e_{10} \cdot \log_2(10) \rfloor} \rceil \end{aligned} \tag{23}$$

Notably, $f \cdot 2^{B-1}$ becomes an integer for certain ranges of e_{10}: $0 \leqslant e_{10} \leqslant 27$ for float and $0 \leqslant e_{10} \leqslant 55$ for double.

3.4.2. Detailed Calculation Process

The detailed calculation process is as follows:

- **Float**

  The range of $-k-1$ is calculated to be $[-32, 44]$ through the q value range in Equation (6), so the lookup table contains representation values from 10 to the power of -32 to 10 to the power of 44. The calculation process is as follows:

$$\begin{aligned} & -32 \leqslant e_{10} \leqslant 44 \\ & e_2 = |\lfloor e_{10} \cdot \log_2(10) \rfloor - 63| \\ & pow10t = \begin{cases} 2^{e_2} // 10^{|e10|}; \text{if } e_{10} < 0 \\ 10^{|e10|} // 2^{e_2}; \text{if } e_{10} \geqslant 20 \\ 10^{|e10|} \cdot 2^{e_2}; \text{if } 1 \leqslant e_{10} \leqslant 19 \end{cases} \\ & f_{1,e_{10}} = pow10 = pow10t + (0 \leqslant e_{10} \leqslant 27 \,?\, 0 : 1) \end{aligned} \tag{24}$$

  When $0 \leqslant e_{10} \leqslant 27$, the lookup table variable indicates that the values $f_{1,e_{10}} \cdot 2^{\lfloor e_{10} \cdot \log_2(10) \rfloor - 63}$ and $10^{e_{10}}$ are equal. In other cases, the relative error is less than 2^{-63}, expressed as follows:

$$\begin{aligned} r_{1,e_{10}} &= \frac{f_{1,e_{10}} \cdot 2^{\lfloor e_{10} \cdot \log_2(10) \rfloor - 63}}{10^{e_{10}}} \\ &\in \begin{cases} 1; \text{if } 0 \leqslant e_{10} \leqslant 27 \\ (1, 1 + 2^{-63}); \text{if } e_{10} < 0 \text{ or } e_{10} > 27 \end{cases} \end{aligned} \tag{25}$$

- **Double**

  The range of $-k-1$ is calculated to be $[-293, 323]$ through the q value range in Equation (6), so the lookup table contains representation values from 10 to the power of -293 to 10 to the power of 323. The calculation process is as follows:

$$\begin{aligned} & -293 \leqslant e_{10} \leqslant 323 \\ & e_2 = |\lfloor e_{10} \cdot \log_2(10) \rfloor - 127| \\ & pow10t = \begin{cases} 2^{e_2} // 10^{|e10|}; \text{if } e_{10} < 0 \\ 10^{|e10|} // 2^{e_2}; \text{if } e_{10} \geqslant 39 \\ 10^{|e10|} \cdot 2^{e_2}; \text{if } 1 \leqslant e_{10} \leqslant 38 \end{cases} \\ & f_{1,e_{10}} = pow10 = pow10t + (0 \leqslant e_{10} \leqslant 55 \,?\, 0 : 1) \end{aligned} \tag{26}$$

When $0 \leqslant e_{10} \leqslant 55$, the lookup table variable indicates that the values $f_{1,e_{10}} \cdot 2^{\lfloor e_{10} \cdot \log_2(10) \rfloor - 127}$ and $10^{e_{10}}$ are equal. In other cases, the relative error is less than 2^{-127}, expressed as follows:

$$r_{1,e_{10}} = \frac{f_{1,e_{10}} \cdot 2^{\lfloor e_{10} \cdot \log_2(10) \rfloor - 127}}{10^{e_{10}}} \in \begin{cases} 1; \text{if } 0 \leqslant e_{10} \leqslant 55 \\ (1, 1 + 2^{-127}); \text{if } e_{10} < 0 \text{ or } e_{10} > 55 \end{cases} \tag{27}$$

Let r_1 denote the error for float lookup table entries, r_2 for double entries, and r for both. In Algorithm 1, we retrieve 10^{-k-1} from the lookup table. The lookup table provides exact values when q falls within specific ranges:

$$\begin{aligned} \text{float} &: 0 \leqslant -k - 1 \leqslant 27 \Rightarrow -93 \leqslant q \leqslant -1 \\ \text{double} &: 0 \leqslant -k - 1 \leqslant 55 \Rightarrow -186 \leqslant q \leqslant -1 \end{aligned} \tag{28}$$

For q outside these ranges, the lookup table entries have bounded relative errors:

$$\begin{aligned} \text{float} &: 0 < r_1 - 1 < 2^{-63} \\ \text{double} &: 0 < r_2 - 1 < 2^{-127} \end{aligned} \tag{29}$$

3.4.3. Storage Requirements

The float lookup table requires 616 bytes of storage, calculated as $(44 - (-32) + 1) \times 8$ bytes (77 entries × 8 bytes each). The double lookup table requires 9872 bytes, calculated as $(323 - (-293) + 1) \times 16$ bytes (617 entries × 16 bytes each).

3.4.4. Implementation Notes

The lookup table precomputation uses efficient integer arithmetic to avoid precision loss during calculations. The conditional logic in Equations (24) and (26) optimizes the computation based on the sign and magnitude of e_{10}, ensuring efficient generation of accurate lookup table entries.

3.5. Efficient Computation of m

This section presents an efficient method for calculating m in Algorithm 1, which is defined as $m = \lfloor v \cdot 10^{-k-1} \rfloor$.

3.5.1. Key Proof

In Algorithm 1, we need to compute $m = \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor$. We aim to prove

$$m = \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor c \cdot 2^q \cdot r \cdot 10^{-k-1} \rfloor \tag{30}$$

where r is the lookup table error defined in Equations (25) and (27). When condition Equation (28) is met, $r = 1$, and the equation holds trivially. For $r \neq 1$,

$$\begin{aligned} \text{float} &: 1 < r < 1 + 2^{-63} \\ \text{double} &: 1 < r < 1 + 2^{-127} \end{aligned} \tag{31}$$

Calculate the range of $2^q \cdot 10^{-k-1}$, and we get

$$2^q \cdot 10^{-k-1} = 10^{-1} \cdot \left(2^q \cdot 10^{-\lfloor q \cdot \lg(2) \rfloor}\right) = 10^{-1} \cdot \left(10^{q \cdot \lg(2) - \lfloor q \cdot \lg(2) \rfloor}\right) \tag{32}$$

When q is not 0, Equation (32) exists:

$$\begin{aligned} q \cdot \lg(2) &\neq \lfloor q \cdot \lg(2) \rfloor \\ 0 < q \cdot \lg(2) &- \lfloor q \cdot \lg(2) \rfloor < 1 \end{aligned} \tag{33}$$

When q is 0, $q \cdot \lg(2) - \lfloor q \cdot \lg(2) \rfloor = 0$, so the final conclusion is

$$10^{-1} \leqslant 2^q \cdot 10^{-k-1} < 1 \tag{34}$$

because there is

$$c \cdot 2^q \cdot 10^{-k-1} = c \cdot \frac{2^{q-k-1}}{5^{k+1}} \in [0.1c, c) \tag{35}$$

Therefore,

$$c \cdot 2^q \cdot 10^{-k-1} = \begin{cases} \frac{c \cdot 2^{q-k-1}}{5^{k+1}}; q \geqslant 1 \\ \frac{c}{2^{1+k-q} \cdot 5^{k+1}} = \frac{c}{10}; q = 0 \\ \frac{c \cdot 5^{-k-1}}{2^{1+k-q}}; q < 0 \end{cases} \tag{36}$$

Suppose

$$c \cdot 2^q \cdot 10^{-k-1} = c \cdot \frac{x}{y} < c \tag{37}$$

Then, there are

$$(x, y) = \begin{cases} \left(2^{q-k-1}, 5^{k+1}\right); q \geqslant 1 \\ (1, 10); q = 0 \\ \left(5^{-k-1}, 2^{1+k-q}\right); q < 0 \end{cases} \tag{38}$$

3.5.2. Bit Width Calculation

Define maximum values for c:

$$\begin{aligned} \text{float} : c \leqslant c_{max} &= C_1 = 2^{24} - 1 \\ \text{double} : c \leqslant c_{max} &= C_2 = 2^{53} - 1 \end{aligned} \tag{39}$$

Let C denote either C_1 or C_2, depending on the precision.

For $y > C$, compute P^* and Q^* for each q using Appendix A.5 $f(C, x, y)$, and find the minimum BIT such that

$$\frac{x}{y}(1 + 2^{-BIT}) < \frac{P^*}{Q^*} \tag{40}$$

For $y \leqslant C$,

$$c \cdot \frac{x}{y}\left(1 + \frac{1}{Cy}\right) = \frac{cx + \frac{c}{C} \cdot \frac{x}{y}}{y} < \frac{cx + 1}{y} \tag{41}$$

Thus,

$$\lfloor c \cdot \frac{x}{y} \rfloor = \lfloor c \cdot \frac{x}{y}\left(1 + \frac{1}{Cy}\right) \rfloor \tag{42}$$

Similarly, find the minimum BIT such that

$$\frac{x}{y}(1 + 2^{-BIT}) < \frac{x}{y}\left(1 + \frac{1}{Cy}\right) \tag{43}$$

3.5.3. Results

The maximum of the minimum BIT values for all q (calculated in (`test1.py`) https://github.com/xjb714/xjb/blob/main/py_test/test1.py (accessed on 20 April 2026) in about 1–2 s) is

$$\begin{aligned} \text{float} : BIT_{max} &= 52 \\ \text{double} : BIT_{max} &= 113 \end{aligned} \tag{44}$$

Thus,

$$\begin{aligned} \text{float} : \lfloor c \cdot \frac{x}{y} \rfloor &= \lfloor c \cdot \frac{x}{y} \cdot (1 + 2^{-52}) \rfloor = \lfloor c \cdot \frac{x}{y} \cdot r_1 \rfloor \\ \text{double} : \lfloor c \cdot \frac{x}{y} \rfloor &= \lfloor c \cdot \frac{x}{y} \cdot (1 + 2^{-113}) \rfloor = \lfloor c \cdot \frac{x}{y} \cdot r_2 \rfloor \end{aligned} \tag{45}$$

This result confirms that m can be calculated efficiently using the lookup table values, even with their inherent errors. Once m is determined, $ten = 10m$ can be computed quickly.

3.6. Fast Boundary Condition Testing for $one = 0$ and $one = 10$

In Algorithm 1, the conditions for determining $one = 0$ and $one = 10$ appear on lines 31 and 34, respectively. This section introduces an optimized method to quickly test these boundary conditions using equivalent mathematical formulations.

3.6.1. Equivalent Conditions for Boundary Testing

We start by deriving equivalent mathematical conditions for testing $one = 0$ and $one = 10$.

- **Case 1: Testing** $one = 0$
  When $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$, this is equivalent to

$$\begin{aligned} c \cdot 2^q \cdot 10^{-k-1} - \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor &= 2^{-1} \cdot 2^q \cdot 10^{-k-1} \\ (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} &= \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor \end{aligned} \tag{46}$$

- **Case 2: Testing** $one = 10$
  When $2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n$, this is equivalent to

$$\begin{aligned} \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor - c \cdot 2^q \cdot 10^{-k-1} + 1 &= 2^{-1} \cdot 2^q \cdot 10^{-k-1} \\ (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} &= \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor + 1 \end{aligned} \tag{47}$$

3.6.2. Integer Testing Analysis

To further analyze these conditions, we start with the range of $2^{q-1} \cdot 10^{-k-1}$ from Equation (34):

$$2^{q-1} \cdot 10^{-k-1} \in [0.05, 0.5) \tag{48}$$

- **Analysis for** $one = 0$
  For the $one = 0$ case, we can derive

$$\begin{aligned} \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor - 1 &< c \cdot 2^q \cdot 10^{-k-1} - 0.5 \\ &< (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} \\ &\leqslant c \cdot 2^q \cdot 10^{-k-1} - 0.05 < \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor + 1 \end{aligned} \tag{49}$$

  This implies that when $(2c-1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is an integer, it must equal $\lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor$.
- **Analysis for** $one = 10$

Similarly, for the $one = 10$ case,

$$\begin{aligned} \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor &< c \cdot 2^q \cdot 10^{-k-1} + 0.05 \\ &\leqslant (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} \\ &< c \cdot 2^q \cdot 10^{-k-1} + 0.5 < \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor + 2 \end{aligned} \tag{50}$$

This implies that when $(2c+1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is an integer, it must equal $\lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor + 1$.

3.6.3. Key Insight: Integer Divisibility Test

The key insight is that testing for $one = 0$ or $one = 10$ is equivalent to checking whether $(2c \pm 1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is an integer. This can be rewritten as follows:

$$(2c \pm 1) \cdot 2^{q-1} \cdot 10^{-k-1} = (2c \pm 1) \cdot 2^{q-k-2} \cdot 5^{-k-1} \tag{51}$$

We analyze different ranges of q to simplify this condition:

- **Case** $q \geqslant 2$
  From $q \geqslant 2$, we get $k \geqslant 0$. The expression simplifies to checking whether $(2c \pm 1) \cdot 2^{q-k-2}$ is divisible by 5^{k+1}. Since 2 and 5 are coprime, this reduces to checking whether $(2c \pm 1)$ is divisible by 5^{k+1}:

$$(2c \pm 1) \mod 5^{k+1} = 0 \tag{52}$$

  Let t be a positive integer such that $2c \pm 1 = t \cdot 5^{k+1}$. Since $2c \pm 1$ is odd, t must also be odd. Considering the ranges of c for float and double,

$$\begin{aligned} \text{float} &: 2c - 1 \in [2^{24} + 1, 2^{25} - 3];\ 2c + 1 \in [2^{24} + 3, 2^{25} - 1]; \\ \text{double} &: 2c - 1 \in [2^{53} + 1, 2^{54} - 3];\ 2c + 1 \in [2^{53} + 3, 2^{54} - 1]; \end{aligned} \tag{53}$$

  This gives us the range for t:

$$\begin{aligned} \text{float} &: \frac{2^{24}+1}{5^{k+1}} \leqslant t \leqslant \frac{2^{25}-1}{5^{k+1}}; \\ \text{double} &: \frac{2^{53}+1}{5^{k+1}} \leqslant t \leqslant \frac{2^{54}-1}{5^{k+1}}; \end{aligned} \tag{54}$$

  The maximum values of k where t can be at least one odd integer are

$$\begin{aligned} \text{float} &: k_{\max} = 9 \Rightarrow q_{\max} = 33, t = 3 \\ \text{double} &: k_{\max} = 22 \Rightarrow q_{\max} = 76, t = 1 \end{aligned} \tag{55}$$

- **Case** $1 \geqslant q \geqslant 0$
  The denominator $2^{2+k-q} \cdot 5^{k+1}$ is even, while the numerator $(2c \pm 1)$ is odd, so no solution exists.
- **Case** $q < 0$
  The denominator 2^{2+k-q} is even, while the numerator $(2c \pm 1) \cdot 5^{-k-1}$ is odd, so no solution exists.

3.6.4. Summary of Boundary Conditions

In summary, the situations when $(2c \pm 1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is an integer are as follows:

$$\begin{aligned} \text{float} &: 2 \leqslant q \leqslant 33 \,\&\&\, (2c \pm 1) \mod 5^{k+1} = 0; \\ \text{double} &: 2 \leqslant q \leqslant 76 \,\&\&\, (2c \pm 1) \mod 5^{k+1} = 0; \end{aligned} \tag{56}$$

The range of $-k-1$ is as follows:

$$\begin{aligned} \text{float} &: -10 \leqslant -k-1 \leqslant -1 \\ \text{double} &: -23 \leqslant -k-1 \leqslant -1 \end{aligned} \tag{57}$$

3.6.5. Efficient Implementation

We can further simplify the testing conditions using bitwise operations. For $one = 0$, when $2^{-1} \cdot 2^{q} \cdot 10^{-k-1} = n$,

$$\begin{aligned} \text{float} &: \lfloor 2^{35} \cdot 2^{q} \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor \\ \text{double} &: \lfloor 2^{63} \cdot 2^{q} \cdot 10^{-k-1} \rfloor = \lfloor 2^{64} \cdot n \rfloor \end{aligned} \tag{58}$$

For $one = 10$, when $2^{-1} \cdot 2^{q} \cdot 10^{-k-1} = 1 - n$, the following conclusions can be drawn:

$$\begin{aligned} \text{float} &: \begin{cases} 2^{35} \cdot 2^{q} \cdot 10^{-k-1} = 2^{36} - 2^{36} \cdot n \Rightarrow \\ \lfloor 2^{35} \cdot 2^{q} \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor = 2^{36} - 1 - \lfloor 2^{36} \cdot n \rfloor \end{cases} \\ \text{double} &: \begin{cases} 2^{63} \cdot 2^{q} \cdot 10^{-k-1} = 2^{64} - 2^{64} \cdot n \Rightarrow \\ \lfloor 2^{63} \cdot 2^{q} \cdot 10^{-k-1} \rfloor = \lfloor 2^{64} - 2^{64} \cdot n \rfloor = 2^{64} - 1 - \lfloor 2^{64} \cdot n \rfloor \end{cases} \end{aligned} \tag{59}$$

The discussion on whether $\lfloor 2^{36} - 2^{36} \cdot n \rfloor = 2^{36} - 1 - \lfloor 2^{36} \cdot n \rfloor$ in Equation (59) holds true—that is, whether $2^{36} \cdot n$ in Equation (59) is an integer—or equivalent to discussing whether the following values are integers when Equation (56) holds true (the same applies to double):

$$\begin{aligned} \text{float} &: 2^{36} \cdot (m+n) = c \cdot 2^{q+36} \cdot 10^{-k-1} = c \cdot 2^{q-k+35} \cdot 5^{-k-1} = c \cdot \frac{2^{q-k+35}}{5^{k+1}} \\ \text{double} &: 2^{64} \cdot (m+n) = c \cdot 2^{q+64} \cdot 10^{-k-1} = c \cdot 2^{q-k+63} \cdot 5^{-k-1} = c \cdot \frac{2^{q-k+63}}{5^{k+1}} \end{aligned} \tag{60}$$

Suppose c can divide 5^{k+1} evenly (where t is a temporary integer variable):

$$c = t \cdot 5^{k+1}; t \geqslant 1 \tag{61}$$

Therefore, when Equation (61) was established, there were

$$2c \pm 1 = 2 \cdot t \cdot 5^{k+1} \pm 1 \tag{62}$$

Expression Equation (62) cannot divide 5^{k+1} evenly, which contradicts Equation (56), so c cannot divide 5^{k+1} evenly. Therefore, for float, $c \cdot 2^{q+36} \cdot 10^{-k-1}$ and $2^{36} \cdot n$ are not integers. For double, $c \cdot 2^{64+q} \cdot 10^{-k-1}$ and $2^{64} \cdot n$ are not integers; that is,

$$\begin{aligned} \text{float} &: \lfloor 2^{36} - 2^{36} \cdot n \rfloor = 2^{36} + \lfloor -2^{36} \cdot n \rfloor = 2^{36} - 1 - \lfloor 2^{36} \cdot n \rfloor \\ \text{double} &: \lfloor 2^{64} - 2^{64} \cdot n \rfloor = 2^{64} + \lfloor -2^{64} \cdot n \rfloor = 2^{64} - 1 - \lfloor 2^{64} \cdot n \rfloor \end{aligned} \tag{63}$$

Therefore, the conclusion Equation (59) is correct. Discuss the necessary and sufficient conditions for whether $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor$ is $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$. The same applies to double, expressed as follows:

$$\begin{aligned} \text{float}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} = n \Leftrightarrow \lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor \\ \text{double}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} = n \Leftrightarrow \lfloor 2^{63} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{64} \cdot n \rfloor \end{aligned} \tag{64}$$

Similarly, the necessary and sufficient condition for whether $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor$ is $2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n$. The same applies to double, expressed as follows:

$$\begin{aligned} \text{float}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n \Leftrightarrow \lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor \\ \text{double}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n \Leftrightarrow \lfloor 2^{63} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{64} - 2^{64} \cdot n \rfloor \end{aligned} \tag{65}$$

The sufficient conditions of Equations (64) and (65) are obviously established. Introduce the proof that Equation (64) holds. For float, only the necessary conditions need to be discussed; that is, whether $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$ must hold true when $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor$ holds, or equivalent to $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor \neq \lfloor 2^{36} \cdot n \rfloor$ must hold true when $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq n$. The following is proved by proof by contradiction.

Assume that $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor$ holds when $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq n$. Then there is

$$\begin{aligned} & \lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor \\ \Rightarrow & 0 < \left| 2^{35} \cdot 2^q \cdot 10^{-k-1} - 2^{36} \cdot n \right| < 1 \\ \Rightarrow & 0 < \left| (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} - m \right| < 2^{-36} \end{aligned} \tag{66}$$

As is known from Equation (49), there is

$$m - 1 < (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} < m + 1 \tag{67}$$

Suppose that the decimal part of $(2c-1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is represented as n^-; thus, we have

$$\left| (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} - m \right| = \begin{cases} n^-; \text{if } (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} > m \\ 1 - n^-; \text{if } (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} < m \end{cases} \tag{68}$$

Substitute Equation (68) into Equation (66), and we get

$$\begin{aligned} & 0 < \left| (2c-1) \cdot 2^{q-1} \cdot 10^{-k-1} - m \right| < 2^{-36} \\ \Rightarrow & 0 < n^- < 2^{-36} \text{ or } 0 < 1 - n^- < 2^{-36} \end{aligned} \tag{69}$$

Similarly, it can be known that the double range is the range of n^-. Therefore, there is

$$\begin{aligned} \text{float}: & n^- \in \left(0, 2^{-36}\right) \cup \left(1 - 2^{-36}, 1\right) \\ \text{double}: & n^- \in \left(0, 2^{-64}\right) \cup \left(1 - 2^{-64}, 1\right) \end{aligned} \tag{70}$$

When $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq n$, it is known from Equation (46) that $(2c-1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is not an integer. Therefore, there is

$$0 < n^- < 1 \tag{71}$$

It is only necessary to prove that Equation (70) does not hold. Discuss the range of the decimal part n^- when $(2c-1)\cdot 2^{q-1}\cdot 10^{-k-1}$ is not an integer. According to Equation (51), there are

$$(2c-1)\cdot 2^{q-1}\cdot 10^{-k-1} = (2c-1)\cdot\frac{x}{y} = \begin{cases} \frac{(2c-1)\cdot 2^{q-k-2}}{5^{k+1}}; q \geqslant 2 \\ \frac{(2c-1)}{2^{2+k-q}\cdot 5^{k+1}}; 1 \geqslant q \geqslant 0 \\ \frac{(2c-1)\cdot 5^{-k-1}}{2^{2+k-q}}; q < 0 \end{cases} \tag{72}$$

The maximum value of $2c-1$ is

$$\begin{aligned} \text{float}: (2c-1)_{\max} &= 2^{25}-3 \\ \text{double}: (2c-1)_{\max} &= 2^{54}-3 \end{aligned} \tag{73}$$

Discuss based on the denominator range in Equation (72).

- $y \leqslant (2c-1)_{\max}$
  When $y \leqslant (2c-1)_{\max}$, $y_{\max}$ is the expression Equation (73), the following holds true:

$$\begin{aligned} \frac{1}{y_{\max}} \leqslant n^- &\leqslant 1-\frac{1}{y_{\max}} \\ \frac{1}{y_{\max}} \leqslant 1-n^- &\leqslant 1-\frac{1}{y_{\max}} \end{aligned} \tag{74}$$

  Therefore, when $y \leqslant (2c-1)_{\max}$, Equation (70) does not hold true.
- $y > (2c-1)_{\max}$
  Call function Appendix A.5 to calculate the approximation results $\frac{P_*}{Q_*}$ and $\frac{P^*}{Q^*}$ of all possible upper and lower limit rational numbers:

$$\left(\frac{P_*}{Q_*}, \frac{P^*}{Q^*}\right) = f((2c-1)_{\max}, x, y) \tag{75}$$

  Therefore, for n^-, the following conclusion can be drawn from Appendix A.4.

$$n^- \in \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \tag{76}$$

  By exhausting all possibilities, we thus have the following (the test code file is (`test3.py`) https://github.com/xjb714/xjb/blob/main/py_test/test3.py) (accessed on 20 April 2026):

$$\begin{aligned} \text{float}: 2^{-33} < n^- &< 1-2^{-29} \\ \text{double}: 2^{-62} < n^- &< 1-2^{-63} \end{aligned} \tag{77}$$

$$\begin{aligned} \text{float}: \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] &\cap \left(0, 2^{-36}\right) = \varnothing \\ \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] &\cap \left(1-2^{-36}, 1\right) = \varnothing \\ \text{double}: \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] &\cap \left(0, 2^{-64}\right) = \varnothing \\ \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] &\cap \left(1-2^{-64}, 1\right) = \varnothing \end{aligned} \tag{78}$$

Therefore, when $y > (2c-1)_{\max}$, Equation (70) does not hold true.

In summary, when $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq n$, Equation (70) does not hold true; that is, $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor \neq \lfloor 2^{36} \cdot n \rfloor$ must hold true. Therefore, when $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} \cdot n \rfloor$ holds, $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$ must hold true. Therefore, Equation (64) holds.

Similarly, it can be proved that when $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor$ holds, $2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n$ must hold true. The same applies to double. Similarly, by proof of contradiction, for float, it is assumed that when $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq 1 - n$ holds, $\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor$ holds. That is,

$$\begin{aligned} &\lfloor 2^{35} \cdot 2^q \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor \\ &\Rightarrow 0 < \left| 2^{35} \cdot 2^q \cdot 10^{-k-1} - 2^{36} + 2^{36} \cdot n \right| < 1 \\ &\Rightarrow 0 < \left| 2^{q-1} \cdot 10^{-k-1} - 1 + n \right| < 2^{-36} \\ &\Rightarrow -2^{-36} < (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} - m - 1 < 2^{-36} \end{aligned} \tag{79}$$

As is known from Equation (50), there is

$$m < (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} < m + 2 \tag{80}$$

Suppose that the decimal part of $(2c+1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is represented as n^+; thus, we have

$$(2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} - m - 1 = \begin{cases} n^+; \text{if } (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} > m+1 \\ 1 - n^+; \text{if } (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} < m+1 \end{cases} \tag{81}$$

Substitute Equation (81) into Equation (79), and we get

$$\begin{aligned} &0 < \left| (2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} - m - 1 \right| < 2^{-36} \\ &\Rightarrow 0 < 1 - n^+ < 2^{-36} \text{ or } 0 < n^+ < 2^{-36} \end{aligned} \tag{82}$$

Similarly, it can be known that the double range is the range of n^+. Therefore, there is

$$\begin{aligned} \text{float} &: n^+ \in \left(0, 2^{-36}\right) \cup \left(1 - 2^{-36}, 1\right) \\ \text{double} &: n^+ \in \left(0, 2^{-64}\right) \cup \left(1 - 2^{-64}, 1\right) \end{aligned} \tag{83}$$

When $2^{-1} \cdot 2^q \cdot 10^{-k-1} \neq 1 - n$, it is known from Equation (47) that $(2c+1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is not an integer. Therefore, there is

$$0 < n^+ < 1 \tag{84}$$

It is only necessary to prove that Equation (83) does not hold. Discuss the range of the decimal part n^+ when $(2c+1) \cdot 2^{q-1} \cdot 10^{-k-1}$ is not an integer. According to Equation (51), there are

$$(2c+1) \cdot 2^{q-1} \cdot 10^{-k-1} = (2c+1) \cdot \frac{x}{y} = \begin{cases} \frac{(2c+1) \cdot 2^{q-k-2}}{5^{k+1}}; q \geqslant 2 \\ \frac{(2c+1)}{2^{2+k-q} \cdot 5^{k+1}}; 1 \geqslant q \geqslant 0 \\ \frac{(2c+1) \cdot 5^{-k-1}}{2^{2+k-q}}; q < 0 \end{cases} \tag{85}$$

The maximum value of $2c+1$ is

$$\begin{aligned} \text{float} &: (2c+1)_{\max} = 2^{25} - 1 \\ \text{double} &: (2c+1)_{\max} = 2^{54} - 1 \end{aligned} \tag{86}$$

Discuss based on the denominator range in Equation (85).

- $y \leqslant (2c+1)_{\max}$
  When $y \leqslant (2c+1)_{\max}$, $y_{\max}$ is the expression Equation (86), the following holds true:

$$\begin{aligned} \frac{1}{y_{\max}} \leqslant n^{+} \leqslant 1 - \frac{1}{y_{\max}} \\ \frac{1}{y_{\max}} \leqslant 1 - n^{+} \leqslant 1 - \frac{1}{y_{\max}} \end{aligned} \tag{87}$$

  Therefore, when $y \leqslant (2c+1)_{\max}$, Equation (83) does not hold true.
- $y > (2c+1)_{\max}$
  Call function Appendix A.5 to calculate the approximation results $\frac{P_*}{Q_*}$ and $\frac{P^*}{Q^*}$ of all possible upper and lower limit rational numbers:

$$\left(\frac{P_*}{Q_*}, \frac{P^*}{Q^*}\right) = f((2c+1)_{\max}, x, y) \tag{88}$$

  Therefore, for n^{+}, the following conclusion can be drawn from formula A.4.

$$n^{+} \in \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \tag{89}$$

  By exhausting all possibilities, we thus have the following (the test code file is (`test7.py`) https://github.com/xjb714/xjb/blob/main/py_test/test7.py (accessed on 20 April 2026)):

$$\begin{aligned} \text{float}: & 2^{-33} < n^{+} < 1 - 2^{-29} \\ \text{double}: & 2^{-62} < n^{+} < 1 - 2^{-63} \end{aligned} \tag{90}$$

$$\begin{aligned} \text{float}: & \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(0, 2^{-36}\right) = \varnothing \\ & \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(1 - 2^{-36}, 1\right) = \varnothing \\ \text{double}: & \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(0, 2^{-64}\right) = \varnothing \\ & \left[\frac{(Q_* x)\%y}{y}, \frac{(Q^* x)\%y}{y}\right] \cap \left(1 - 2^{-64}, 1\right) = \varnothing \end{aligned} \tag{91}$$

  Therefore, when $y > (2c+1)_{\max}$, Equation (83) does not hold true.

In summary, when $2^{-1} \cdot 2^{q} \cdot 10^{-k-1} \neq 1 - n$, Equation (83) does not hold true; that is, $\lfloor 2^{35} \cdot 2^{q} \cdot 10^{-k-1} \rfloor \neq \lfloor 2^{36} - 2^{36} \cdot n \rfloor$ must hold true. Therefore, when $\lfloor 2^{35} \cdot 2^{q} \cdot 10^{-k-1} \rfloor = \lfloor 2^{36} - 2^{36} \cdot n \rfloor$ holds, $2^{-1} \cdot 2^{q} \cdot 10^{-k-1} = 1 - n$ must hold true. The same is true for double. Therefore, Equation (65) holds.

The following conclusions hold:

$$\begin{aligned} \text{float}: \lfloor 2^{36} - 2^{36} \cdot n \rfloor &= \begin{cases} 2^{36} - 1 - \lfloor 2^{36} \cdot n \rfloor; \text{if } c \cdot 2^{36+q} \cdot 10^{-k-1} \notin Z \\ 2^{36} - \lfloor 2^{36} \cdot n \rfloor; \text{if } c \cdot 2^{36+q} \cdot 10^{-k-1} \in Z \end{cases} \\ \text{double}: \lfloor 2^{64} - 2^{64} \cdot n \rfloor &= \begin{cases} 2^{64} - 1 - \lfloor 2^{64} \cdot n \rfloor; \text{if } c \cdot 2^{64+q} \cdot 10^{-k-1} \notin Z \\ 2^{64} - \lfloor 2^{64} \cdot n \rfloor; \text{if } c \cdot 2^{64+q} \cdot 10^{-k-1} \in Z \end{cases} \end{aligned} \tag{92}$$

Discuss whether the following Equation (93) holds when conditions Equations (56) and (57) are met:

$$\begin{aligned}
\text{float}: &\lfloor c \cdot \frac{2^{q+35-k}}{5^{k+1}} \rfloor = \lfloor c \cdot \frac{2^{q+35-k}}{5^{k+1}} \cdot r \rfloor \\
&= \lfloor c \cdot \frac{2^{q+35-k}}{5^{k+1}} \cdot \frac{\left(2^{63-\lfloor(-k-1)\cdot\log_2(10)\rfloor} // 10^{k+1}\right)+1}{10^{-k-1}} \cdot 2^{\lfloor(-k-1)\cdot\log_2(10)\rfloor-63} \rfloor \\
\text{double}: &\lfloor c \cdot \frac{2^{q+63-k}}{5^{k+1}} \rfloor = \lfloor c \cdot \frac{2^{q+63-k}}{5^{k+1}} \cdot r \rfloor \\
&= \lfloor c \cdot \frac{2^{q+63-k}}{5^{k+1}} \cdot \frac{\left(2^{127-\lfloor(-k-1)\cdot\log_2(10)\rfloor} // 10^{k+1}\right)+1}{10^{-k-1}} \cdot 2^{\lfloor(-k-1)\cdot\log_2(10)\rfloor-127} \rfloor
\end{aligned} \tag{93}$$

There are

$$\begin{aligned}
\text{float}: &\lfloor c \cdot \frac{2^{q+35-k}}{5^{k+1}} \rfloor = \lfloor 2^{36} \cdot (m+n) \rfloor = 2^{36} \cdot m + \lfloor 2^{36} \cdot n \rfloor \\
\text{double}: &\lfloor c \cdot \frac{2^{q+63-k}}{5^{k+1}} \rfloor = \lfloor 2^{64} \cdot (m+n) \rfloor = 2^{64} \cdot m + \lfloor 2^{64} \cdot n \rfloor
\end{aligned} \tag{94}$$

It has been proven earlier that m can be accurately calculated. Then, when Equation (93) holds true, the values $\lfloor 2^{36} \cdot n \rfloor$ and $\lfloor 2^{64} \cdot n \rfloor$ on the right side of equations Equations (58) and (59) can be accurately calculated.

From Equation (51), we have

$$c = \frac{t \cdot 5^{k+1} - 1}{2} \tag{95}$$

Substituting Equation (95) into Equation (93), we have

$$\begin{aligned}
\text{float}: &c \cdot \frac{2^{q+35-k}}{5^{k+1}} = t \cdot 2^{q+34-k} - \frac{2^{q+34-k}}{5^{k+1}} \\
\text{double}: &c \cdot \frac{2^{q+63-k}}{5^{k+1}} = t \cdot 2^{q+62-k} - \frac{2^{q+62-k}}{5^{k+1}}
\end{aligned} \tag{96}$$

When the conditions of Equations (56) and (57) are met, $t \cdot 2^{q+34-k}$ and $t \cdot 2^{q+62-k}$ are integers. Under the condition of meeting condition Equation (56), the decimal part of expression Equation (96) is represented as follows:

$$\begin{aligned}
\text{float}: &\frac{2^{q+34-k} \% 5^{k+1}}{5^{k+1}}; 2 \leqslant q \leqslant 33 \\
\text{double}: &\frac{2^{q+62-k} \% 5^{k+1}}{5^{k+1}}; 2 \leqslant q \leqslant 76
\end{aligned} \tag{97}$$

It is only necessary to prove that the increase in the value $c \cdot \frac{2^{q+35-k}}{5^{k+1}} \cdot r$ on the right side of the expression compared to the value $c \cdot \frac{2^{q+35-k}}{5^{k+1}}$ on the left side plus the decimal part of the value on the left side is less than 1 for Equation (93) to hold true. That is,

$$\begin{aligned}
\text{float}: &\frac{2^{q+34-k} \% 5^{k+1}}{5^{k+1}} + \left(c \cdot \frac{2^{q+35-k}}{5^{k+1}} \cdot r - c \cdot \frac{2^{q+35-k}}{5^{k+1}} \right) < 1 \\
\text{double}: &\frac{2^{q+62-k} \% 5^{k+1}}{5^{k+1}} + \left(c \cdot \frac{2^{q+63-k}}{5^{k+1}} \cdot r - c \cdot \frac{2^{q+63-k}}{5^{k+1}} \right) < 1
\end{aligned} \tag{98}$$

By exhaustively calculating the maximum possible c value under each q and substituting it into Equation (98), it holds. The calculation result can be found at in (`test2.py`) https://github.com/xjb714/xjb/blob/main/py_test/test2.py (accessed on 20 April 2026). The calculation results show that, for the float range and the double range, Equation (98) always holds true. Therefore, Equation (93) holds true, and thus, the values of $\lfloor 2^{36} \cdot n \rfloor$ and $\lfloor 2^{64} \cdot n \rfloor$ on the right side of Equations (58) and (59) can be accurately calculated. The values of $\lfloor 2^{35} \cdot 2^{q} \cdot 10^{-k-1} \rfloor$ and $\lfloor 2^{63} \cdot 2^{q} \cdot 10^{-k-1} \rfloor$ on the left side of Equations (58) and (59) can be calculated through lookup tables.

$$\begin{aligned} \text{float}: \lfloor 2^{35} \cdot 2^{q} \cdot 10^{-k-1} \rfloor &= pow10 \gg (28 - q - \lfloor (-k-1) \cdot \log_2(10) \rfloor) \\ \text{double}: \lfloor 2^{63} \cdot 2^{q} \cdot 10^{-k-1} \rfloor &= pow10 \gg (64 - q - \lfloor (-k-1) \cdot \log_2(10) \rfloor) \end{aligned} \tag{99}$$

The code file for verifying the validity of Equation (99) is (`test4.py`) https://github.com/xjb714/xjb/blob/main/py_test/test4.py (accessed on 20 April 2026). Therefore, when the conditions of Equations (56) and (57) are met, the values of both sides of Equations (58) and (59) can be accurately calculated.

Discuss the relationship between the following two values within all ranges of floating-point numbers:

$$\begin{aligned} \text{float}: &\lfloor c \cdot 2^{q+36} \cdot 10^{-k-1} \rfloor; \lfloor c \cdot 2^{q+36} \cdot r \cdot 10^{-k-1} \rfloor; \\ \text{double}: &\lfloor c \cdot 2^{q+64} \cdot 10^{-k-1} \rfloor; \lfloor c \cdot 2^{q+64} \cdot r \cdot 10^{-k-1} \rfloor; \end{aligned} \tag{100}$$

When $r = 1$, it is obvious that the two values in Equation (100) are equal. When $r \neq 1$, or equivalent to $r > 1$,

$$\begin{aligned} \text{float}: c \cdot 2^{q+36} \cdot r \cdot 10^{-k-1} &= c \cdot 2^{q+36} \cdot 10^{-k-1} + c \cdot 2^{q+36} \cdot (r-1) \cdot 10^{-k-1} \\ &< c \cdot 2^{q+36} \cdot 10^{-k-1} + 2^{24} \cdot 2^{36} \cdot 2^{q} \cdot 10^{-k-1} \cdot (r-1) \\ &< c \cdot 2^{q+36} \cdot 10^{-k-1} + 2^{-3} \\ \lfloor c \cdot 2^{q+36} \cdot r \cdot 10^{-k-1} \rfloor &\leqslant \lfloor c \cdot 2^{q+36} \cdot 10^{-k-1} \rfloor + 1 \\ \text{double}: c \cdot 2^{q+64} \cdot r \cdot 10^{-k-1} &= c \cdot 2^{q+64} \cdot 10^{-k-1} + c \cdot 2^{q+64} \cdot (r-1) \cdot 10^{-k-1} \\ &< c \cdot 2^{q+64} \cdot 10^{-k-1} + 2^{53} \cdot 2^{64} \cdot 2^{q} \cdot 10^{-k-1} \cdot (r-1) \\ &< c \cdot 2^{q+64} \cdot 10^{-k-1} + 2^{-10} \\ \lfloor c \cdot 2^{q+64} \cdot r \cdot 10^{-k-1} \rfloor &\leqslant \lfloor c \cdot 2^{q+64} \cdot 10^{-k-1} \rfloor + 1 \end{aligned} \tag{101}$$

Therefore, there is

$$\begin{aligned} \text{float}: 0 &\leqslant \lfloor c \cdot 2^{q+36} \cdot r \cdot 10^{-k-1} \rfloor - \lfloor c \cdot 2^{q+36} \cdot 10^{-k-1} \rfloor \leqslant 1 \\ \text{double}: 0 &\leqslant \lfloor c \cdot 2^{q+64} \cdot r \cdot 10^{-k-1} \rfloor - \lfloor c \cdot 2^{q+64} \cdot 10^{-k-1} \rfloor \leqslant 1 \end{aligned} \tag{102}$$

because there is

$$\lfloor c \cdot 2^{q} \cdot 10^{-k-1} \rfloor = \lfloor c \cdot 2^{q} \cdot r \cdot 10^{-k-1} \rfloor = m \tag{103}$$

$$\begin{aligned} \text{float}: \lfloor c \cdot 2^{q+36} \cdot 10^{-k-1} \rfloor &= 2^{36} \cdot m + \lfloor 2^{36} \cdot n \rfloor \\ \text{double}: \lfloor c \cdot 2^{q+64} \cdot 10^{-k-1} \rfloor &= 2^{64} \cdot m + \lfloor 2^{64} \cdot n \rfloor \end{aligned} \tag{104}$$

Suppose

$$n_r = c \cdot 2^{q} \cdot r \cdot 10^{-k-1} - m \tag{105}$$

Therefore, the following conclusion can be drawn: when Equation (56) is met, from Equation (93), we have

$$\begin{aligned} \text{float}: 2 \leqslant q \leqslant 33 \;\&\&\; (2c \pm 1) \% 5^{k+1} = 0 \Rightarrow \lfloor 2^{36} \cdot n \rfloor = \lfloor 2^{36} \cdot n_r \rfloor \\ \text{double}: 2 \leqslant q \leqslant 76 \;\&\&\; (2c \pm 1) \% 5^{k+1} = 0 \Rightarrow \lfloor 2^{64} \cdot n \rfloor = \lfloor 2^{64} \cdot n_r \rfloor \end{aligned} \tag{106}$$

Within the range of floating-point numbers, there exists

$$\begin{aligned} \text{float}: &\lfloor 2^{36} \cdot n \rfloor \leqslant \lfloor 2^{36} \cdot n_r \rfloor \leqslant \lfloor 2^{36} \cdot n \rfloor + 1 \\ \text{double}: &\lfloor 2^{64} \cdot n \rfloor \leqslant \lfloor 2^{64} \cdot n_r \rfloor \leqslant \lfloor 2^{64} \cdot n \rfloor + 1 \end{aligned} \tag{107}$$

To simplify the expression, *even* is used to indicate whether c is an even number:

$$even = (c+1)\%2 \in \{0, 1\} \tag{108}$$

When $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$ or $2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n$, $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$ is the boundary condition for $one = 0$, and $2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n$ is the boundary condition for $one = 10$. Whether *one* is 0 or 10 is determined based on whether c is an even number. Therefore, the following exists:

$$\begin{aligned} \text{float}: &\begin{cases} one = 0 : \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even > \lfloor 2^{36} \cdot n_r \rfloor \\ one = 10 : \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even > 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \end{cases} \\ \text{double}: &\begin{cases} one = 0 : \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even > \lfloor 2^{64} \cdot n_r \rfloor \\ one = 10 : \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even > 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor \end{cases} \end{aligned} \tag{109}$$

Therefore, when $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$ or $2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n$, we can use the condition Equation (110) to determine whether $one = 0$ or $one = 10$.

$$\begin{aligned} \text{float}: &\begin{cases} \text{if } \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even > \lfloor 2^{36} \cdot n_r \rfloor : one = 0 \\ \text{if } \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even > 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor : one = 10 \end{cases} \\ \text{double}: &\begin{cases} \text{if } \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even > \lfloor 2^{64} \cdot n_r \rfloor : one = 0 \\ \text{if } \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even > 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor : one = 10 \end{cases} \end{aligned} \tag{110}$$

When $2^{-1} \cdot 2^q \cdot 10^{-k-1} > n$ or $2^{-1} \cdot 2^q \cdot 10^{-k-1} > 1 - n$, we can also use the above condition Equation (110) to determine whether $one = 0$ or $one = 10$. When $2^{-1} \cdot 2^q \cdot 10^{-k-1} < n$ or $2^{-1} \cdot 2^q \cdot 10^{-k-1} < 1 - n$, we can also use the above condition Equation (110) to determine whether $one \neq 0$ or $one \neq 10$. There are a total of four situations. The proof is as follows:

(1) When $2^{-1} \cdot 2^q \cdot 10^{-k-1} < n$, there must exist $one \neq 0$, and there is

$$\begin{aligned} \text{float}: &2^{-1} \cdot 2^q \cdot 10^{-k-1} - n = n^- - 1 \in \left(2^{-33} - 1, -2^{-29}\right) \\ \text{double}: &2^{-1} \cdot 2^q \cdot 10^{-k-1} - n = n^- - 1 \in \left(2^{-62} - 1, -2^{-63}\right) \end{aligned} \tag{111}$$

Therefore, the following exists:

$$\begin{aligned} \text{float}: &2^{q+35} \cdot 10^{-k-1} - 2^{36} \cdot n \in \left(2^3 - 2^{36}, -2^7\right) \\ \text{double}: &2^{q+63} \cdot 10^{-k-1} - 2^{64} \cdot n \in \left(4 - 2^{64}, -2\right) \end{aligned} \tag{112}$$

Suppose that there are two real numbers a and b, and the following relationship must exist:

$$\begin{aligned} 0 \leqslant & b - \lfloor b \rfloor < 1 \\ a - \lfloor a \rfloor - 1 < & b - \lfloor b \rfloor < 1 + a - \lfloor a \rfloor \\ a - b - 1 < & \lfloor a \rfloor - \lfloor b \rfloor < a - b + 1 \end{aligned} \tag{113}$$

When $a = 2^{q+35} \cdot 10^{-k-1}$ and $b = 2^{36} \cdot n$ or $a = 2^{q+63} \cdot 10^{-k-1}$ and $b = 2^{64} \cdot n$, the following exists:

$$\begin{aligned} \text{float}: &\lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor - \lfloor 2^{36} \cdot n \rfloor < 2^{q+35} \cdot 10^{-k-1} - 2^{36} \cdot n + 1 \\ \text{double}: &\lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor - \lfloor 2^{64} \cdot n \rfloor < 2^{q+63} \cdot 10^{-k-1} - 2^{64} \cdot n + 1 \end{aligned} \tag{114}$$

From Equation (112), we have

$$\begin{aligned} \text{float}: &\lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor - \lfloor 2^{36} \cdot n \rfloor < 1 - 2^7 < 0 \\ \text{double}: &\lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor - \lfloor 2^{64} \cdot n \rfloor < 1 - 2 < 0 \end{aligned} \tag{115}$$

Therefore, there is

$$\begin{aligned} \text{float}: \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &\leqslant \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + 1 \\ &< \lfloor 2^{36} \cdot n \rfloor \leqslant \lfloor 2^{36} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &< \lfloor 2^{36} \cdot n_r \rfloor \\ \text{double}: \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &\leqslant \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + 1 \\ &< \lfloor 2^{64} \cdot n \rfloor \leqslant \lfloor 2^{64} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &< \lfloor 2^{64} \cdot n_r \rfloor \end{aligned} \tag{116}$$

Therefore, when $2^{-1} \cdot 2^q \cdot 10^{-k-1} < n$, the condition Equation (110) can be used to determine that $one \neq 0$.

(2) When $2^{-1} \cdot 2^q \cdot 10^{-k-1} > n$, there must exist $one = 0$, and there is

$$\begin{aligned} \text{float}: &2^{-1} \cdot 2^q \cdot 10^{-k-1} - n = n^- \in \left(2^{-33}, 1 - 2^{-29}\right) \\ \text{double}: &2^{-1} \cdot 2^q \cdot 10^{-k-1} - n = n^- \in \left(2^{-62}, 1 - 2^{-63}\right) \end{aligned} \tag{117}$$

Therefore, the following exists:

$$\begin{aligned} \text{float}: &2^{q+35} \cdot 10^{-k-1} - 2^{36} \cdot n \in \left(2^3, 2^{36} - 2^7\right) \\ \text{double}: &2^{q+63} \cdot 10^{-k-1} - 2^{64} \cdot n \in \left(4, 2^{64} - 2\right) \end{aligned} \tag{118}$$

When $a = 2^{q+35} \cdot 10^{-k-1}$ and $b = 2^{36} \cdot n$ or $a = 2^{q+63} \cdot 10^{-k-1}$ and $b = 2^{64} \cdot n$, from Equation (113), the following exists:

$$\begin{aligned} \text{float}: &\lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor - \lfloor 2^{36} \cdot n \rfloor > 2^{q+35} \cdot 10^{-k-1} - 2^{36} \cdot n - 1 \\ \text{double}: &\lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor - \lfloor 2^{64} \cdot n \rfloor > 2^{q+63} \cdot 10^{-k-1} - 2^{64} \cdot n - 1 \end{aligned} \tag{119}$$

From Equation (118), we have

$$\begin{aligned} \text{float}: &\lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor - \lfloor 2^{36} \cdot n \rfloor > 2^3 - 1 \geqslant 0 \\ \text{double}: &\lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor - \lfloor 2^{64} \cdot n \rfloor > 4 - 1 \geqslant 0 \end{aligned} \tag{120}$$

Therefore, there is

$$\begin{aligned} \text{float}: \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &\geqslant \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor \\ &> \lfloor 2^{36} \cdot n \rfloor + 1 \geqslant \lfloor 2^{36} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &> \lfloor 2^{36} \cdot n_r \rfloor \\ \text{double}: \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &\geqslant \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor \\ &> \lfloor 2^{64} \cdot n \rfloor + 1 \geqslant \lfloor 2^{64} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &> \lfloor 2^{64} \cdot n_r \rfloor \end{aligned} \tag{121}$$

Therefore, when $2^{-1} \cdot 2^q \cdot 10^{-k-1} > n$, the condition Equation (110) can be used to determine that $one = 0$.

(3) When $2^{-1} \cdot 2^q \cdot 10^{-k-1} < 1 - n$, there must exist $one \neq 10$, and there is

$$\begin{aligned} \text{float}: 2^{-1} \cdot 2^q \cdot 10^{-k-1} + n = n^+ &\in \left(2^{-33}, 1 - 2^{-29}\right) \\ \text{double}: 2^{-1} \cdot 2^q \cdot 10^{-k-1} + n = n^+ &\in \left(2^{-62}, 1 - 2^{-63}\right) \end{aligned} \tag{122}$$

Therefore, the following exists:

$$\begin{aligned} \text{float}: 2^{q+35} \cdot 10^{-k-1} + 2^{36} \cdot n &\in \left(2^3, 2^{36} - 2^7\right) \\ \text{double}: 2^{q+63} \cdot 10^{-k-1} + 2^{64} \cdot n &\in \left(4, 2^{64} - 2\right) \end{aligned} \tag{123}$$

Suppose that there are two real numbers a and b, and the following relationship must exist:

$$\begin{aligned} a - 1 < \lfloor a \rfloor &\leqslant a \\ b - 1 < \lfloor b \rfloor &\leqslant b \\ a + b - 2 < \lfloor a \rfloor + \lfloor b \rfloor &\leqslant a + b \end{aligned} \tag{124}$$

When $a = 2^{q+35} \cdot 10^{-k-1}$ and $b = 2^{36} \cdot n$ or $a = 2^{q+63} \cdot 10^{-k-1}$ and $b = 2^{64} \cdot n$, the following exists:

$$\begin{aligned} \text{float}: \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \lfloor 2^{36} \cdot n \rfloor &\leqslant 2^{q+35} \cdot 10^{-k-1} + 2^{36} \cdot n \\ \text{double}: \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + \lfloor 2^{64} \cdot n \rfloor &\leqslant 2^{q+63} \cdot 10^{-k-1} + 2^{64} \cdot n \end{aligned} \tag{125}$$

From Equation (123), we have

$$\begin{aligned} \text{float}: \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \lfloor 2^{36} \cdot n \rfloor &< 2^{36} - 2^7 \\ \text{double}: \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + \lfloor 2^{64} \cdot n \rfloor &< 2^{64} - 2 \end{aligned} \tag{126}$$

Therefore, there is:

$$\begin{aligned} \text{float}: \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &\leqslant \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + 1 \\ &< 2^{36} - 2 - \lfloor 2^{36} \cdot n \rfloor \\ &< 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &< 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \\ \text{double}: \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &\leqslant \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + 1 \\ &< 2^{64} - 2 - \lfloor 2^{64} \cdot n \rfloor \\ &< 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &< 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor \end{aligned} \tag{127}$$

Therefore, when $2^{-1} \cdot 2^q \cdot 10^{-k-1} < 1 - n$, the condition Equation (110) can be used to determine that $one \neq 10$.

(4) When $2^{-1} \cdot 2^q \cdot 10^{-k-1} > 1 - n$, there must exist $one = 10$, and there is

$$\begin{aligned} \text{float}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} + n = n^+ + 1 \in \left(1 + 2^{-33}, 2 - 2^{-29}\right) \\ \text{double}: & 2^{-1} \cdot 2^q \cdot 10^{-k-1} + n = n^+ + 1 \in \left(1 + 2^{-62}, 2 - 2^{-63}\right) \end{aligned} \tag{128}$$

Therefore, the following exists:

$$\begin{aligned} \text{float}: & 2^{q+35} \cdot 10^{-k-1} + 2^{36} \cdot n \in \left(2^3 + 2^{36}, 2^{37} - 2^7\right) \\ \text{double}: & 2^{q+63} \cdot 10^{-k-1} + 2^{64} \cdot n \in \left(4 + 2^{64}, 2^{65} - 2\right) \end{aligned} \tag{129}$$

When $a = 2^{q+35} \cdot 10^{-k-1}$ and $b = 2^{36} \cdot n$ or $a = 2^{q+63} \cdot 10^{-k-1}$ and $b = 2^{64} \cdot n$, from Equation (124), the following exists:

$$\begin{aligned} \text{float}: & \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \lfloor 2^{36} \cdot n \rfloor > 2^{q+35} \cdot 10^{-k-1} + 2^{36} \cdot n - 2 \\ \text{double}: & \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + \lfloor 2^{64} \cdot n \rfloor > 2^{q+63} \cdot 10^{-k-1} + 2^{64} \cdot n - 2 \end{aligned} \tag{130}$$

From Equation (129), we have

$$\begin{aligned} \text{float}: & \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \lfloor 2^{36} \cdot n \rfloor > 2^{36} + 2^3 - 2 > 2^{36} \\ \text{double}: & \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + \lfloor 2^{64} \cdot n \rfloor > 2^{64} + 2 - 2 > 2^{64} \end{aligned} \tag{131}$$

Therefore, there is

$$\begin{aligned} \text{float}: \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &\geqslant \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor \\ &> 2^{36} - \lfloor 2^{36} \cdot n \rfloor \\ &> 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even &> 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \\ \text{double}: \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &\geqslant \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor \\ &> 2^{64} - \lfloor 2^{64} \cdot n \rfloor \\ &> 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor \\ \Rightarrow \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even &> 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor \end{aligned} \tag{132}$$

Therefore, when $2^{-1} \cdot 2^q \cdot 10^{-k-1} > 1 - n$, the condition Equation (110) can be used to determine that $one = 10$.

From the above proof, it can be seen that when Equation (56) is met, the condition Equation (110) can be used to determine whether $one = 0$ or $one = 10$ when $2^{-1} \cdot 2^q \cdot 10^{-k-1} = n$ or $2^{-1} \cdot 2^q \cdot 10^{-k-1} = 1 - n$. When $2^{-1} \cdot 2^q \cdot 10^{-k-1} > n$ or $2^{-1} \cdot 2^q \cdot 10^{-k-1} > 1 - n$, the condition Equation (110) can be used to determine whether $one = 0$ or $one = 10$. When $2^{-1} \cdot 2^q \cdot 10^{-k-1} < n$ or $2^{-1} \cdot 2^q \cdot 10^{-k-1} < 1 - n$, the condition Equation (110) can be used to determine whether $one \neq 0$ or $one \neq 10$.

The proof process of this section is completed. In the code implementation, the two judgment conditions can be quickly calculated using addition and subtraction shift operations, and they can be compiled by the compiler into cmov instructions, thereby reducing the impact of branch prediction failure on performance.

Ultimately, we reached the following conclusion: *one* can quickly determine whether *one* equals 0 or 10 by using the following method:

$$\begin{aligned} \text{float}: \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even > \lfloor 2^{36} \cdot n_r \rfloor &\Rightarrow one = 0 \\ \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + even > 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor &\Rightarrow one = 10 \\ \text{double}: \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even > \lfloor 2^{64} \cdot n_r \rfloor &\Rightarrow one = 0 \\ \lfloor 2^{q+63} \cdot 10^{-k-1} \rfloor + even > 2^{64} - 1 - \lfloor 2^{64} \cdot n_r \rfloor &\Rightarrow one = 10 \end{aligned} \tag{133}$$

3.7. Efficient Computation of $\lfloor 10n \rfloor$ *and Rounding*

Determine whether *one* is $\lfloor 10n \rfloor$ or $\lfloor 10n \rfloor + 1$ based on the decimal part of $10n$. There are two cases: the decimal part of $10n$ is 0.5, or it is not 0.5.

3.7.1. $10n - \lfloor 10n \rfloor = 0.5$

When the decimal part of $10n$ is 0.5, there must be

$$\begin{aligned} 10n - \lfloor 10n \rfloor &= 0.5 \\ \Rightarrow 10 \cdot c \cdot 2^q \cdot 10^{-k-1} - \lfloor 10 \cdot c \cdot 2^q \cdot 10^{-k-1} \rfloor &= 0.5 \\ \Rightarrow c \cdot 2^q \cdot 10^{-k} - \lfloor c \cdot 2^q \cdot 10^{-k} \rfloor &= 0.5 \\ \Rightarrow c \cdot 2^q \cdot 10^{-k} = \lfloor c \cdot 2^q \cdot 10^{-k} \rfloor &+ 0.5 \\ \Rightarrow 2c \cdot 2^q \cdot 10^{-k} = 2\lfloor c \cdot 2^q \cdot 10^{-k} \rfloor &+ 1 \end{aligned} \tag{134}$$

so $2c \cdot 2^q \cdot 10^{-k}$ is an odd number. Then, the following expression is odd:

$$c \cdot 2^{q+1} \cdot 10^{-k} = c \cdot 2^{q-k+1} \cdot 5^{-k} \tag{135}$$

According to the range of q, there are

$$c \cdot 2^{q+1} \cdot 10^{-k} = \begin{cases} \frac{c \cdot 2^{q-k+1}}{5^k}; q \geqslant 0 \\ c \cdot 2 \cdot 5^{-k}; q = -1 \\ \frac{c \cdot 5^{-k}}{2^{k-q-1}}; q \leqslant -2 \end{cases} \tag{136}$$

According to the range of q, the following situations are discussed:

- $q \geqslant 0$
  When $q \geqslant 0$, it can be concluded that $q - k + 1 \geqslant 1$, the numerator $c \cdot 2^{q-k+1}$ is even, and the denominator 5^k is odd, which does not meet the condition.
- $q = -1$
  When $q = -1$, it can be concluded that $c \cdot 2 \cdot 5^{-k}$ is even, which does not meet the condition.
- $q \leqslant -2$
  5^{-k} is an odd number, and c is an odd multiple of 2^{k-q-1}, so

$$\begin{aligned} \text{float}: c &\geqslant 2^{k-q-1} \Rightarrow k - q - 1 \leqslant 22 \Rightarrow q \geqslant -34 \\ \text{double}: c &\geqslant 2^{k-q-1} \Rightarrow k - q - 1 \leqslant 51 \Rightarrow q \geqslant -75 \end{aligned} \tag{137}$$

Therefore, when q meets the above conditions, c must be an odd multiple of 2^{k-q-1}. Therefore, when the following conditions are met, the expression of Equation (135) is an odd number:

$$\begin{aligned} \text{float}: -34 \leqslant q \leqslant -2 \,\&\&\, c\%2^{k-q} &= 2^{k-q-1} \\ \text{double}: -75 \leqslant q \leqslant -2 \,\&\&\, c\%2^{k-q} &= 2^{k-q-1} \end{aligned} \tag{138}$$

When q is within the range of Equation (138), $r = 1$ is derived from Equation (28). Therefore, there is

$$n_r = n \tag{139}$$

The following equation holds:

$$20m + 20n = c \cdot 2^q \cdot 10^{-k+1} = c \cdot 2^{q-k+1} \cdot 5^{-k} = \frac{c}{2^{k-q-1}} \cdot 5^{-k} \tag{140}$$

Since $-k \geqslant 1$, 5^{-k} is multiple of 5 and is an odd number. Since $\frac{c}{2^{k-q-1}}$ and 5^{-k} are both odd numbers, $20m$ is an even number, and $20n$ is multiple of 5 and is an odd number. Therefore, there is

$$\begin{aligned} 20n &\in \{5, 15\} \\ \Rightarrow n &\in \{0.25, 0.75\} \\ \Rightarrow n_r &\in \{0.25, 0.75\} \end{aligned} \tag{141}$$

The result of *one* is an even number between $\lfloor 10n \rfloor$ and $\lfloor 10n \rfloor + 1$. Therefore,

$$one = \begin{cases} \lfloor 10n \rfloor = 2, \text{if } n = 0.25 \\ \lfloor 10n \rfloor + 1 = 8, \text{if } n = 0.75 \end{cases} \Rightarrow one = \lfloor 20n + 1 \rfloor // 2 - (n = 0.25 \,?1 : 0) \tag{142}$$

3.7.2. $10n - \lfloor 10n \rfloor \neq 0.5$

When the decimal part of $10n$ is not 0.5, round to the nearest integer value based on the decimal part of $10n$. Therefore, there is

$$one = \begin{cases} \lfloor 10n \rfloor, \text{if } 10n - \lfloor 10n \rfloor < 0.5 \\ \lfloor 10n \rfloor + 1, \text{if } 10n - \lfloor 10n \rfloor > 0.5 \end{cases} \Rightarrow one = \lfloor 10n + 0.5 \rfloor = \lfloor 20n + 1 \rfloor // 2 \tag{143}$$

Since $\lfloor 20n + 1 \rfloor = \lfloor 20n \rfloor + 1$, it is only necessary to accurately calculate the value of $\lfloor 20n \rfloor$, and there is

$$\begin{aligned} d &= ten + one \\ &= 10m + \lfloor 20n + 1 \rfloor // 2 \\ &= (\lfloor 20m + 20n \rfloor + 1) // 2 \end{aligned} \tag{144}$$

Suppose that there are

$$20m + 20n = c \cdot 2^{q+1} \cdot 10^{-k} = c \cdot 2^{q-k+1} \cdot 5^{-k} = c \cdot \frac{x}{y} \tag{145}$$

Suppose that the decimal part of $20n$ is n_{20}.

When $y \leqslant c_{\max} = C$, the range of the decimal part must include the following:

$$\begin{aligned} \text{float} : \frac{1}{2^{24} - 1} = \frac{1}{C} \leqslant n_{20} \leqslant 1 - \frac{1}{C} = \frac{2^{24} - 2}{2^{24} - 1} \\ \text{double} : \frac{1}{2^{53} - 1} = \frac{1}{C} \leqslant n_{20} \leqslant 1 - \frac{1}{C} = \frac{2^{53} - 2}{2^{53} - 1} \end{aligned} \tag{146}$$

When $y > c_{\max} = C$, the range of the decimal part must include (the test file is (`test5.py`) https://github.com/xjb714/xjb/blob/main/py_test/test5.py (accessed on 20 April 2026)):

$$\begin{aligned} \text{float} : 2^{-32} < n_{20} < 1 - 2^{-30} \\ \text{double} : 2^{-64} < n_{20} < 1 - 2^{-62} \end{aligned} \tag{147}$$

Therefore, the range of n_{20} satisfies Equation (147). In the code implementation, for float, only the high 36 bits of n_r are retained, and for double, only the high 70 bits of n_r

are retained. Suppose that the discarded part of a float is represented as n_{36}, and similarly, the discarded part of a double is represented as n_{70}. Therefore, there is

$$\begin{aligned} \text{float}: & n_{36} \in \left[0, 2^{-36}\right) \\ \text{double}: & n_{70} \in \left[0, 2^{-70}\right) \end{aligned} \tag{148}$$

Calculate the boundary conditions of the following expression:

$$\begin{aligned} \text{float}: & F = 20 \cdot \left(c \cdot 2^q \cdot r \cdot 10^{-k-1} - n_{36}\right) \\ \text{double}: & F = 20 \cdot \left(c \cdot 2^q \cdot r \cdot 10^{-k-1} - n_{70}\right) \end{aligned} \tag{149}$$

Therefore, there is

$$\begin{aligned} \text{float}: F_{\min} &> 20 \cdot \left(c \cdot 2^q \cdot 10^{-k-1} - 2^{-36}\right) \\ &= 20m + 20n - 20 \cdot 2^{-36} \\ F_{\max} &< 20 \cdot \left(c \cdot 2^q \cdot \left(1 + 2^{-63}\right) \cdot 10^{-k-1} - 0\right) \\ &< 20m + 20n + 20 \cdot 2^{-63} \cdot c \\ &< 20m + \lfloor 20n \rfloor + 1 \\ \text{double}: F_{\min} &> 20 \cdot \left(c \cdot 2^q \cdot 10^{-k-1} - 2^{-70}\right) \\ &= 20m + 20n - 20 \cdot 2^{-70} \\ &> 20m + \lfloor 20n \rfloor \\ F_{\max} &< 20 \cdot \left(c \cdot 2^q \cdot \left(1 + 2^{-127}\right) \cdot 10^{-k-1} - 0\right) \\ &< 20m + 20n + 20 \cdot 2^{-127} \cdot c \\ &< 20m + \lfloor 20n \rfloor + 1 \end{aligned} \tag{150}$$

Therefore, there is

$$\begin{aligned} \text{float}: & \lfloor F \rfloor = 20m + \lfloor 20n \rfloor \\ \text{double}: & \lfloor F \rfloor = 20m + \lfloor 20n \rfloor \end{aligned} \tag{151}$$

In fact, in the above proof process, for float, $\lfloor F_{min} \rfloor \neq 20m + \lfloor 20n \rfloor$ may exist, but the code implementation has passed the exhaustive test. For the float type, our code implementation has passed all tests for all possible input values, so this not-so-perfect proof process can be ignored. Therefore, the calculation of d can be simplified as follows:

$$\begin{aligned} d &= ten + one \\ &= (\lfloor F \rfloor + 1)//2 \\ &= (\lfloor 20 \cdot (c \cdot 2^q \cdot r \cdot 10^{-k-1} - n_x) \rfloor + 1)//2 \end{aligned} \tag{152}$$

For the float range, $n_x = n_{36}$; for the double range, $n_x = n_{70}$.

3.7.3. Efficient Implementation of $n = 0.25$ for Double

For double, quickly determine that $n = 0.25$ in Equation (142).

When $n = 0.25, \lfloor 2^{64} \cdot n_r \rfloor = \lfloor 2^{64} \cdot n \rfloor = 2^{62}$. Therefore, the following condition can be used to quickly determine whether $n = 0.25$:

$$\text{double}: n = 0.25 \text{ if } \lfloor 2^{64} \cdot n_r \rfloor = 2^{62} \tag{153}$$

When $n \neq 0.25$, calculate the range of the decimal part of the following expression:

$$4m + 4n = c \cdot 2^{q+2} \cdot 10^{-k-1} \tag{154}$$

Therefore, when Equation (154) is not an integer, we have the following (the test file is (`test6.py`) https://github.com/xjb714/xjb/blob/main/py_test/test6.py (accessed on 20 April 2026)):

$$2^{-62} < 4n - \lfloor 4n \rfloor < 1 - 2^{-62} \tag{155}$$

Calculate the two boundary cases of $4n$ that are closest to 1:

$$\begin{aligned} \lfloor 4n \rfloor = 0 \Rightarrow 4n - 0 < 1 - 2^{-62} \Rightarrow \lfloor 2^{64} \cdot n \rfloor \leqslant 2^{62} - 2 \\ \lfloor 4n \rfloor = 1 \Rightarrow 4n - 1 > 2^{-62} \Rightarrow \lfloor 2^{64} \cdot n \rfloor \geqslant 2^{62} + 1 \end{aligned} \tag{156}$$

Then, there are

$$\begin{aligned} \lfloor 2^{64} \cdot n \rfloor \neq 2^{62} \,\&\&\, \lfloor 2^{64} \cdot n \rfloor + 1 \neq 2^{62} \\ \Rightarrow \lfloor 2^{64} \cdot n_r \rfloor \neq 2^{62} \end{aligned} \tag{157}$$

Therefore, the following condition can be used to quickly determine whether $n \neq 0.25$:

$$\text{double}: n \neq 0.25 \,\text{if}\, \lfloor 2^{64} \cdot n_r \rfloor \neq 2^{62} \tag{158}$$

In summary, for double, the following condition can be used to quickly determine whether $n = 0.25$:

$$\begin{aligned} &\text{double}: n = 0.25 \,\text{if}\, \lfloor 2^{64} \cdot n_r \rfloor = 2^{62} \\ &\text{double}: n \neq 0.25 \,\text{if}\, \lfloor 2^{64} \cdot n_r \rfloor \neq 2^{62} \end{aligned} \tag{159}$$

3.7.4. Efficient Calculation of *one* for Double

In the double range, introduce another faster way to calculate *one*:

$$\text{double}: one = \lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 + \left((n = 0.25)?0 : \left(2^{-1} + \frac{6}{2^{64}} \right) \right) \rfloor \tag{160}$$

The proof of Equation (160) is as follows:

when $n = 0.25$, $\lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 \rfloor = \lfloor 10n \rfloor = 2$;
when $n \neq 0.25$, Equation (160) can be equivalent to the following:

$$\text{double}: one = \lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 + 2^{-1} + \frac{6}{2^{64}} \rfloor \tag{161}$$

According to the $10n - \lfloor 10n \rfloor$ range, *one* is represented as follows:

$$\text{double}: one = \begin{cases} \lfloor 10n \rfloor, \text{if } 10n - \lfloor 10n \rfloor < 0.5 \\ 8, \text{if } 10n - \lfloor 10n \rfloor = 0.5 \\ \lfloor 10n \rfloor + 1, \text{if } 10n - \lfloor 10n \rfloor > 0.5 \end{cases} = \lfloor 20n + 1 \rfloor // 2 \tag{162}$$

Therefore,when $n \neq 0.25$, we need to prove that the following equation holds:

$$\lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 + 2^{-1} + \frac{6}{2^{64}} \rfloor = \begin{cases} \lfloor 10n \rfloor, \text{if } 10n - \lfloor 10n \rfloor < 0.5 \\ 8, \text{if } 10n - \lfloor 10n \rfloor = 0.5 \\ \lfloor 10n \rfloor + 1, \text{if } 10n - \lfloor 10n \rfloor > 0.5 \end{cases} = \lfloor 20n + 1 \rfloor // 2 \tag{163}$$

From the range of n, there is

$$\frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \in \left(n_r - 2^{-64}, n_r \right] \tag{164}$$

because the following conditions exist:

$$\begin{aligned} c \cdot 2^q \cdot 10^{-k-1} &= m + n \\ c \cdot 2^q \cdot r \cdot 10^{-k-1} &= m + n_r \end{aligned} \tag{165}$$

Therefore, the following relationship can be concluded:

$$\begin{aligned} n_r - n &= (r-1) \cdot c \cdot 2^q \cdot 10^{-k-1} \\ n_r &= (r-1) \cdot (m+n) + n \\ \Rightarrow n \leqslant n_r &< 2^{-127} \cdot c + n \\ n \leqslant n_r &< 2^{-127} \cdot 2^{53} + n \\ n \leqslant n_r &< 2^{-74} + n \end{aligned} \tag{166}$$

From Equations (164) and (166), it can be concluded that

$$\begin{aligned} \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} &\in \left(n - 2^{-64}, n + 2^{-74} \right) \\ \Rightarrow \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 &\in \left(10n - 10 \cdot 2^{-64}, 10n + 10 \cdot 2^{-74} \right) \\ \Rightarrow \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 20 &\in \left(20n - 20 \cdot 2^{-64}, 20n + 20 \cdot 2^{-74} \right) \\ \Rightarrow \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 20 &\in \left(\lfloor 20n \rfloor + n_{20} - 20 \cdot 2^{-64}, \lfloor 20n \rfloor + n_{20} + 20 \cdot 2^{-74} \right) \end{aligned} \tag{167}$$

Discuss the range of values of x when the following conditions are met:

$$\lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 20 + 1 + x \rfloor // 2 = \lfloor 20n + 1 \rfloor // 2 = one \tag{168}$$

Therefore, the following conclusions can be drawn:

$$\begin{aligned} \lfloor 20n \rfloor + n_{20} - 20 \cdot 2^{-64} + 1 + x \geqslant \lfloor 20n + 1 \rfloor &\Rightarrow x \geqslant 20 \cdot 2^{-64} - n_{20} \\ \lfloor 20n \rfloor + n_{20} + 20 \cdot 2^{-74} + 1 + x < \lfloor 20n + 2 \rfloor &\Rightarrow x < 1 - 20 \cdot 2^{-74} - n_{20} \end{aligned} \tag{169}$$

Suppose $x = 12 \cdot 2^{-64}$. From Equation (169), all floating-point numbers that do not meet the following conditions can be obtained:

$$x = 12 \cdot 2^{-64} \geqslant 20 \cdot 2^{-64} - n_{20} \tag{170}$$

All floating-point numbers that do not meet the conditions of Equation (170) are as follows (hexadecimal and the printed results that meet the SW principle):

$$\begin{aligned} &\text{0x0d17c0747bd76fa1 , 1.3588129002659584e-245} \\ &\text{0x0d27c0747bd76fa1 , 2.7176258005319167e-245} \\ &\text{0x4d73de005bd620df , 1.3076622631878654e+65} \\ &\text{0x4d83de005bd620df , 2.6153245263757307e+65} \\ &\text{0x4d93de005bd620df , 5.230649052751461e+65} \end{aligned} \tag{171}$$

From Equation (169), all floating-point numbers that do not meet the following conditions can be obtained:

$$x = 12 \cdot 2^{-64} < 1 - 20 \cdot 2^{-74} - n_{20} \tag{172}$$

All floating-point numbers that do not meet the conditions of Equation (172) are as follows (hexadecimal and the printed results that meet the SW principle):

$$\begin{aligned} &\text{0x612491daad0ba280 , 9.03725590277404e+159} \\ &\text{0x6159b651584e8b20 , 9.03725590277404e+160} \\ &\text{0x619011f2d73116f4 , 9.03725590277404e+161} \\ &\text{0x61c4166f8cfd5cb1 , 9.03725590277404e+162} \\ &\text{0x61d4166f8cfd5cb1 , 1.807451180554808e+163} \end{aligned} \tag{173}$$

There are

$$2(\frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 + 2^{-1} + \frac{6}{2^{64}}) = \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 20 + 1 + x \tag{174}$$

When the floating-point number is not within the range specified in Equations (171) and (173), the condition of Equation (169) is satisfied. We have tested all floating-point numbers within the above-mentioned range (Equations (171) and (173)), and the algorithm implementation code output the correct result; that is, it satisfies the SW principle. The test process file is (`test8.py`) https://github.com/xjb714/xjb/blob/main/py_test/test8.py (accessed on 20 April 2026).

In summary, Equations (163) and (160) hold. Therefore, Equation (160) can be used to quickly calculate *one*.

3.8. Irregular Number

Due to the limited and small number of irregular floating-point numbers, there are a total of 2046 double floating-point numbers and 254 float floating-point numbers. The correctness of the algorithm code in this paper can be proved by the exhaustive method. Compared with the Schubfach algorithm, all of the irregular values produced exactly the same output results; therefore, it is not introduced in this article. In the code implementation, we use separate branches to calculate the irregular values. For the specific implementation process, please refer to the source code.

3.9. Implementation of Pseudocode

This subsection details the pseudocode implementation for converting regular floating-point numbers to decimal representation. The handling of irregular floating-point numbers is omitted here due to space constraints; interested readers may refer to the source code for the complete implementation.

3.9.1. Single-Precision Floating-Point Numbers

$$\begin{aligned}
\text{uint32 } vi &= \text{bit_copy_from_f32}(v) \\
(\text{uint64 } c, \text{int32 } q) &= \text{extract}(vi) \\
\text{const int BIT} &= 36 \\
\text{const uint64 offset} &= (1\text{ULL} \ll (\text{BIT} - 2)) - 7 \\
\text{int32 } k &= (q \cdot 1233) \gg 12 \\
\text{int32 } h &= q + ((1701 \cdot (-k - 1)) \gg 9) \\
\text{uint64 pow10} &= \text{get_pow10}(-k - 1) \\
\text{uint64 } cb &= c \ll (h + \text{BIT} + 1) \\
\text{uint64 hi64} &= \text{umul_hi}_{64\times 64}(cb, \text{pow10}) \\
\text{bool even} &= (vi + 1)\&1 \\
\text{uint64 half} &= (\text{pow10} \gg (65 - (h + \text{BIT} + 1))) + \text{even} \\
\text{uint32 shorter} &= (\text{hi64} + \text{half}) \gg \text{BIT} \\
\text{uint64 bias} &= (\text{hi64} \gg (\text{BIT} - 4))\&15 \\
\text{uint32 longer} &= (5 \cdot \text{hi64} + \text{offset} + \text{bias}) \gg (\text{BIT} - 1) \\
\text{bool updown} &= \text{shorter} > ((\text{hi64} - \text{half}) \gg \text{BIT}) \\
\text{uint32 } d &= \text{updown ? shorter} \cdot 10 : \text{longer}
\end{aligned} \tag{175}$$

Pseudocode Equation (175) outlines the procedure for computing d and k for single-precision floating-point numbers.

Since c contains at most 23 significant bits and pow10 has 64 significant bits, their product contains at most 87 significant bits. Given that the range of h is $[-4, -1]$, the value $cb = c \ll (h + \text{BIT} + 1)$ does not overflow, preserving all bits of c. When $e_{10} = -k - 1$, pow10 is computed using Equation (24). From Equation (25), we derive

$$\text{pow10} \cdot 2^{\lfloor (-k-1)\cdot \log_2 10 \rfloor - 63} = 10^{-k-1} \cdot r_{1,-k-1} \tag{176}$$

From Equation (45),

$$\begin{aligned}
m &= \lfloor v \cdot 10^{-k-1} \rfloor \\
&= \lfloor v \cdot 10^{-k-1} \cdot r_{1,-k-1} \rfloor \\
&= \lfloor v \cdot \text{pow10} \cdot 2^{\lfloor (-k-1)\cdot \log_2 10 \rfloor - 63} \rfloor \\
&= \lfloor c \cdot \text{pow10} \cdot 2^{q + \lfloor (-k-1)\cdot \log_2 10 \rfloor - 63} \rfloor \\
&= (c \cdot \text{pow10}) \gg (63 - q - \lfloor (-k-1) \cdot \log_2 10 \rfloor) \\
&= (c \cdot \text{pow10}) \gg (63 - h) \\
&= ((c \ll (37 + h)) \cdot \text{pow10}) \gg (36 + 64) \\
&= \text{hi64} \gg 36
\end{aligned} \tag{177}$$

Let up indicate whether $one = 10$. From Equations (132) and (99),

$$\begin{aligned}
\text{bool up} &= \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \text{even} > 2^{36} - 1 - \lfloor 2^{36} \cdot n_r \rfloor \\
&= \lfloor 2^{q+35} \cdot 10^{-k-1} \rfloor + \text{even} + \lfloor 2^{36} \cdot n_r \rfloor \geqslant 2^{36} \\
&= (\text{pow10} \gg (28 - h)) + \text{even} + \lfloor 2^{36} \cdot n_r \rfloor \geqslant 2^{36} \\
&= (\text{half} + \lfloor 2^{36} \cdot n_r \rfloor) \gg 36
\end{aligned} \tag{178}$$

When $one \neq 10$, we have $d//10 = m$; when $one = 10$, we have $d//10 = m + 1$. Therefore,

$$d//10 = m + \text{up} \tag{179}$$

The upper 28 bits of hi64 represent m, while the lower 36 bits represent $\lfloor 2^{36} \cdot n_r \rfloor$:

$$\text{hi64} = (m \ll 36) + \lfloor 2^{36} \cdot n_r \rfloor \tag{180}$$

Consequently,

$$\text{shorter} = d//10 = (\text{hi64} + \text{half}) \gg 36 \tag{181}$$

Let updown indicate whether $one \in \{0, 10\}$, or equivalently whether the last digit of d is zero:

$$\text{updown} = (d \bmod 10 = 0) \tag{182}$$

For brevity, let $\text{dot\_one} = \lfloor 2^{36} \cdot n_r \rfloor$. From Equations (128) and (134),

$$\begin{aligned} one = 0 &: \text{half} > \text{dot\_one} \\ one = 10 &: \text{half} > 2^{36} - 1 - \text{dot\_one} \end{aligned} \tag{183}$$

The following equations hold:

$$\begin{aligned} one = 0 &: (\text{hi64} - \text{half}) \gg 36 = m - 1 \\ one = 10 &: (\text{hi64} - \text{half}) \gg 36 = m \end{aligned} \tag{184}$$

Therefore,

$$\begin{aligned} one = 0 &: \text{shorter} = m > (\text{hi64} - \text{half}) \gg 36 = m - 1 \\ one = 10 &: \text{shorter} = m + 1 > (\text{hi64} - \text{half}) \gg 36 = m \end{aligned} \tag{185}$$

Thus, the condition for $one \in \{0, 10\}$ is

$$\text{shorter} > (\text{hi64} - \text{half}) \gg 36 \implies d \bmod 10 = 0 \tag{186}$$

When updown is true, $d = 10 \cdot \text{shorter}$. Otherwise, we compute d using Equation (152). When $10n - \lfloor 10n \rfloor \neq 0.5$,

$$\begin{aligned} d &= (\lfloor 20 \cdot (v \cdot 10^{-k-1} \cdot r - n_{36}) \rfloor + 1)//2 \\ &= (\lfloor 20 \cdot (\text{hi64} \cdot 2^{-36}) \rfloor + 1)//2 \\ &= (\lfloor 5 \cdot \text{hi64} \cdot 2^{-34} \rfloor + 1)//2 \\ &= (\lfloor (5 \cdot \text{hi64} + 2^{34}) \cdot 2^{-34} \rfloor)//2 \\ &= ((5 \cdot \text{hi64} + 2^{34}) \gg 34)//2 \\ &= (5 \cdot \text{hi64} + 2^{34}) \gg 35 \end{aligned} \tag{187}$$

Adding 2^{34} serves as a rounding operation. When $10n - \lfloor 10n \rfloor = 0.5$, Equation (142) requires checking whether $n = 0.25$. The derivation of longer involves careful handling of edge cases; the correctness of this computation has been verified through exhaustive testing. Similarly, for the float type, our code implementation has passed all tests for possible input values. From the above pseudocode, it can be seen that, for the float type, we only need to perform one 64-bit multiplication of 64-bit integers.

3.9.2. Double-Precision Floating-Point Numbers

$$\begin{aligned}
\text{uint64 } vi &= \text{bit_copy_from_f64}(v)\\
(\text{uint64 } c, \text{int } q) &= \text{extract}(vi)\\
\text{const int BIT} &= 6\\
\text{int } k &= (q \cdot 78913) \gg 18\\
\text{int } h &= q + ((217707 \cdot (-k-1)) \gg 16)\\
\text{uint128 pow10} &= \text{get_pow10}(-k-1)\\
\text{uint64 } cb &= c \ll (h + \text{BIT} + 1)\\
\text{uint128 hi128} &= \text{umul_hi}_{64\times128}(cb, \text{pow10})\\
\text{bool even} &= (vi+1)\&1\\
\text{uint64 half} &= (\text{pow10} \gg (64-h)) + \text{even}\\
\text{uint64 dot_one} &= (\text{uint64})(\text{hi128} \gg \text{BIT})\\
\text{uint64 ten} &= 10 \cdot (\text{hi128} \gg (\text{BIT}+64))\\
\text{uint64 offset_num} &= (\text{dot_one} = 2^{62})\,?\,0 : 2^{63}+6\\
\text{uint64 } one &= ((\text{uint128})10 \cdot \text{dot_one} + \text{offset_num}) \gg 64\\
one &= (\text{half} > \text{dot_one})\,?\,0 : one\\
one &= (\text{half} > 2^{64} - 1 - \text{dot_one})\,?\,10 : one\\
\text{uint64 } d &= \text{ten} + one
\end{aligned} \tag{188}$$

Pseudocode Equation (188) presents the corresponding procedure for double-precision floating-point numbers; pow10 is computed using Equation (26), satisfying

$$\text{pow10} \cdot 2^{\lfloor (-k-1)\cdot \log_2 10 \rfloor - 127} = 10^{-k-1} \cdot r_{1,-k-1} \tag{189}$$

From Equation (45),

$$\begin{aligned}
m &= \lfloor v \cdot 10^{-k-1} \rfloor\\
&= \lfloor v \cdot 10^{-k-1} \cdot r_{1,-k-1} \rfloor\\
&= \lfloor v \cdot \text{pow10} \cdot 2^{\lfloor (-k-1)\cdot \log_2 10 \rfloor - 127} \rfloor\\
&= \lfloor c \cdot \text{pow10} \cdot 2^{q + \lfloor (-k-1)\cdot \log_2 10 \rfloor - 127} \rfloor\\
&= (c \cdot \text{pow10}) \gg (127 - q - \lfloor (-k-1)\cdot \log_2 10 \rfloor)\\
&= (c \cdot \text{pow10}) \gg (127 - h)\\
&= ((c \ll (7+h)) \cdot \text{pow10}) \gg (64+70)\\
&= \text{hi128} \gg 70
\end{aligned} \tag{190}$$

Given $h \in [-4, -1]$ and c having at most 53 significant bits, cb does not overflow, ensuring accurate computation of ten $= 10 \cdot m$. By definition, dot_one $= \lfloor 2^{64} \cdot n_r \rfloor$. From Equation (99),

$$\begin{aligned}
\text{half} &= (\text{pow10} \gg (64-h)) + \text{even}\\
&= \lfloor 2^{63} \cdot 2^{q} \cdot 10^{-k-1} \rfloor + \text{even}
\end{aligned} \tag{191}$$

The value of *one* is first computed using Equation (160), and then adjusted if the conditions for $one = 0$ or $one = 10$ are met:

$$\begin{aligned}
one &= \left\lfloor \frac{\lfloor 2^{64} \cdot n_r \rfloor}{2^{64}} \cdot 10 + \left((n = 0.25)\,?\,0 : (2^{-1} + \frac{6}{2^{64}}) \right) \right\rfloor\\
&= (\text{dot_one} \cdot 10 + (\text{dot_one} = 2^{62}\,?\,0 : 2^{63}+6)) \gg 64
\end{aligned} \tag{192}$$

An equivalent formulation is

$$one = (\text{dot\_one} = 2^{62})\,?\,2 : (\text{dot\_one} \cdot 10 + 2^{63} + 6) \gg 64 \tag{193}$$

The conditions $one = 0$ and $one = 10$ are determined by Equations (121) and (132), respectively. From the above pseudocode, it can be seen that, for the double type, we only need one 64-bit multiplication by a 128-bit integer.

In the C/C++ implementation, the only branch occurs when computing c and q for subnormal floating-point numbers (marked as unlikely). The conversion of normal floating-point numbers is branch-free, eliminating branch misprediction penalties. As shown in Pseudocode Equations (175) and (188), the core algorithm is remarkably concise. Irregular numbers are handled in a separate branch—a worthwhile trade-off given their rarity.

3.10. Decimal-to-String Conversion

The floating-point number printing algorithm consists of two main phases. While the first phase, which computes the decimal digits d and exponent k, has been discussed in previous sections, this section focuses on the second phase: converting the computed decimal representation into a string.

Floating-point numbers are typically printed in two formats: fixed-point notation and scientific notation. Table 4 illustrates examples of both formats.

Table 4. Examples of floating-point number printing results.

Float Number	Fixed-Point	Scientific
2.34	"2.34"	"2.34"
12	"12.0"	"1.2"
120	"120.0"	"1.2×10^2"
0.012	"0.012"	"1.2×10^{-2}"

Having computed d and k, we can now convert the floating-point number to a string based on these values. According to the Schubfach algorithm, d may contain trailing zeros, meaning d mod 10 could be zero. In order to reduce instruction dependencies, we decompose d into two parts: $d//10$ and d mod 10. Let up indicate whether one equals 10, while $updown$ represents the cases where one is either 0 or 10. The following relationships hold:

$$\begin{aligned} d//10 &= m + up \\ updown &= (one = 0 \vee one = 10) = (d \bmod 10 = 0) \\ d &= 10 \cdot (m + up) + (updown\,?\,0 : one) \end{aligned} \tag{194}$$

By calculating the approximate range of m, we obtain

$$\begin{aligned} m &= \lfloor c \cdot 2^q \cdot 10^{-k-1} \rfloor \\ c \cdot 2^q \cdot 10^{-k-1} &\in [0.1 \cdot c, c) \\ \Rightarrow \lfloor 0.1 \cdot c_{\min} \rfloor \leqslant \lfloor 0.1 \cdot c \rfloor \leqslant m \leqslant \lfloor c \rfloor &\leqslant \lfloor c_{\max} \rfloor \end{aligned} \tag{195}$$

Based on the range of c, we derive

$$\begin{aligned} \text{float} &: \begin{cases} \text{normal} : [\frac{2^{23}+1}{10}, 2^{24} - 1] = [838860.9, 16777215] \\ \text{subnormal} : [\frac{1}{10}, 2^{23} - 1] = [0.1, 8388607] \end{cases} \\ \text{double} &: \begin{cases} \text{normal} : [\frac{2^{52}+1}{10}, 2^{53} - 1] = [4.5 \times 10^{14}, 9 \times 10^{15}] \\ \text{subnormal} : [\frac{1}{10}, 2^{52} - 1] = [0.1, 4.5 \times 10^{15}] \end{cases} \end{aligned} \tag{196}$$

Let $len10(x)$ denote the number of decimal digits in x. For $x > 0$,

$$len10(x) = \lfloor \log_{10}(x) \rfloor + 1 \tag{197}$$

We define $len10(0) = 0$. For example, $len10(12345) = 5$. Consequently,

$$\begin{aligned} \text{float} &: \begin{cases} \text{normal} : len10(m + up) \in [6, 8] \\ \text{subnormal} : len10(m + up) \in [0, 7] \end{cases} \\ \text{double} &: \begin{cases} \text{normal} : len10(m + up) \in [15, 16] \\ \text{subnormal} : len10(m + up) \in [0, 16] \end{cases} \end{aligned} \tag{198}$$

Since $d//10 = m + up$, we have $len10(d) = len10(m + up) + 1$. Let $tz10(x)$ denote the number of trailing zeros in the decimal representation of x; for example, $tz10(12300) = 2$. The number of significant digits $sig10(x)$ equals $len10(x) - tz10(x)$. When $updown = 0$, $one \neq 0$; thus, $d \bmod 10 = one \neq 0$ and $tz10(d) = 0$. Therefore,

$$\begin{aligned} sig10(d) &= updown \ ? \ len10(d) - tz10(d) : len10(d) \\ \text{float} &: sig10(d) \in [1, 9] \\ \text{double} &: sig10(d) \in [1, 17] \end{aligned} \tag{199}$$

For normal floating-point numbers, $len10(d)$ can be computed as follows:

$$\begin{aligned} \text{float} &: len10(d) = len(m + up) + 1 = 9 - [m + up < 10^7] - [m + up < 10^6] \\ \text{double} &: len10(d) = len(m + up) + 1 = 16 + [(m + up) \geqslant 10^{15}] \end{aligned} \tag{200}$$

where $[P]$ denotes the Iverson bracket, which evaluates to 1 if predicate P is true and 0 otherwise.

Computing the ASCII representation of d reduces to computing the ASCII codes of $m + up$ and one. From the above, we have

$$\begin{aligned} \text{float} &: m + up < 10^8 \\ \text{double} &: m + up < 10^{16} \end{aligned} \tag{201}$$

These bounds allow efficient computation of the ASCII code for $m + up$ using CPU SIMD instruction sets.

Scalar Implementation

We first present a scalar method that does not rely on SIMD instructions.

Let $x \in [0, 10^8)$ and $dec\_to\_ascii8(x)$ be a function that computes the ASCII representation of x. Algorithm 2 describes this process. This version is an optimized version of the itoa algorithm written by Paul Khuong [19].

The function simultaneously computes $tz10(x)$ as the return value tz. Example: X = 12345600, Y ="12345600", tz = 2.

For $x \in [0, 10^{16})$, let $dec\_to\_ascii16(x)$ compute the ASCII representation of x. Algorithm 3 describes this process. Example: X = 123456001234500, Y ="1234560012345600", tz = 2.

Handling Undefined Behavior

Since $\_\_builtin\_ctzll(0)$ is undefined behavior in C language, special handling is required. For single-precision floating-point numbers, when $m + up = 0$ (the six smallest subnormal numbers), $updown = 0$, avoiding the undefined case. For double-precision

numbers, $m + up = 0$ only occurs for 5×10^{-324} (the smallest subnormal number). In our implementation, we handle this special case separately at the function entry.

Algorithm 2: Convert an 8-digit decimal number to ASCII: dec_to_ascii8(x)

Input: X (type: uint64, range: [0, 10^8))
Output: Y (type: uint64), tz (type: uint32)

```
uint64 abcd_efgh = X + (0x100000000 - 10000) * ((X * 0x68db8bbULL) >> 40)
uint64 ab_cd_ef_gh = abcd_efgh + (0x10000 - 100) * (((abcd_efgh * 0x147b) >> 19)
& 0x7f0000007f)
uint64 a_b_c_d_e_f_g_h = ab_cd_ef_gh + (0x100 - 10) * (((ab_cd_ef_gh * 0x67) >> 10)
& 0xf000f000f000f)
uint32 tz = __builtin_ctzll(a_b_c_d_e_f_g_h) >> 3
uint64 BCD = CPU_IS_LITTLE_ENDIAN ? byteswap(a_b_c_d_e_f_g_h) :
a_b_c_d_e_f_g_h
uint64 Y = BCD + 0x3030303030303030
return Y, tz
```

Algorithm 3: Convert a 16-digit decimal number to ASCII: dec_to_ascii16(x)

Input: X (type: uint64, range: [0, 10^{16}))
Output: Y (type: uint128), tz (type: uint32)

```
uint64 abcdefgh = X // 10^8
uint64 ijklmnop = X mod 10^8
(uint64 abcdefgh_ascii, uint tz1) = dec_to_ascii8(abcdefgh)
(uint64 ijklmnop_ascii, uint tz2) = dec_to_ascii8(ijklmnop)
uint128 Y = CPU_IS_LITTLE_ENDIAN ? (ijklmnop_ascii << 64) + abcdefgh_ascii :
(abcdefgh_ascii << 64) + ijklmnop_ascii
uint32 tz = (ijklmnop == 0) ? 8 + tz1 : tz2
return Y, tz
```

SIMD Implementation

The SIMD implementations of *dec_to_ascii*8 and *dec_to_ascii*16 follow the same principles as the scalar version. For the ARM64 architecture, we use the NEON instruction set; for the x86-64 architecture, we support three variants: AVX512, SSE4.1, and SSE2. Table 5 summarizes these implementations. Due to the limited space, it might not be appropriate to fully demonstrate the implementation process. Readers can refer to the source code design on their own.

Table 5. SIMD implementations of *dec_to_ascii*8 and *dec_to_ascii*16.

SIMD Implementation	Description
NEON [20]	Original author: Dougall Johnson. Runs on ARM processors with NEON instruction set.
SSE2 [21]	Based on scalar version; requires only SSE2 instruction set.
SSE4.1	Nearly identical to SSE2 implementation; requires SSE4.1 instruction set.
AVX512 [22]	Original author: Daniel Lemire. Requires AVX512IFMA and AVX512VBMI instruction sets.

Using the methods above, we compute the ASCII representation of $m + up$ and the number of significant digits. Printing d is equivalent to printing $m + up$ and one. Based on $sig10(d)$, we determine which buffer contents to retain. In our implementation, we adopt different formats for floating-point numbers in different ranges to enhance readability, as shown in Table 6.

Table 6. Printing formats for different ranges.

Type	Fixed-Point	Scientific
Float	$[10^{-3}, 10^{7})$	Other ranges
Double	$[10^{-4}, 10^{16})$	Other ranges

In scientific notation, the result includes an exponent. For example, in "1.2×10^{-2}", "10^{-2}" represents the exponent. Converting $d \cdot 10^k$ to standard decimal scientific notation yields

$$d \cdot 10^k = \left(d \cdot 10^{-\lfloor \log_{10}(d) \rfloor}\right) \cdot 10^{k+\lfloor \log_{10}(d) \rfloor} \tag{202}$$

Since $d \cdot 10^{-\lfloor \log_{10}(d) \rfloor} \in [1, 10)$, the exponent is determined by $k + \lfloor \log_{10}(d) \rfloor$. Let $E_{10} = k + \lfloor \log_{10}(d) \rfloor$. Then,

$$E_{10} = k + \lfloor \log_{10}(d) \rfloor = k + \lfloor \log_{10}(m + up) \rfloor + 1 \tag{203}$$

For single-precision floating-point numbers, fixed-point notation is used when $E_{10} \in [-3, 6]$. For double-precision numbers, fixed-point notation is used when $E_{10} \in [-4, 15]$.

The floating-point number printing algorithm proposed in this paper is illustrated as Algorithm 4. The key distinction between our algorithm and others is the use of SIMD instruction sets for converting $m + up$ to ASCII values. For single-precision numbers, we use dec_to_ascii8; for double-precision, we use $dec\_to\_ascii16$. The detailed implementation is available in the source code.

Algorithm 4: Floating-point number printing algorithm

Input: v (type: float/double), buffer (type: char*)
Output: buffer (type: char*)

1: compute $m + up$, $updown$, one, k from v
2: compute $sig10(d) = updown\,?\,len10(m + up) - tz10(m + up) : len10(m + up) + 1$
3: compute $E_{10} = k + \lfloor \log_{10}(m + up) \rfloor + 1$
4: convert $m + up$, one to ASCII codes and store in buffer
5: based on $sig10(d)$ and E_{10}, determine which buffer contents to retain
6: if result is in scientific notation, print E_{10}
7: **return** buffer

In our C implementation, all branches are designed as unlikely branches to minimize the impact of branch prediction failures. By transforming the computation of d into computing $m + up$ and one, we reduce instruction dependencies and enhance instruction-level parallelism. Unlike other algorithms that compute the exact value of d before printing, we avoid computing d directly and instead quickly compute $d//10 = m + up$.

3.11. Summary

This section explains how to quickly calculate d and k, as well as the printing optimization. Since $d = 10m + one$, the process of calculating d is transformed into calculating m and one.

- Section 3.5 introduces the method for quickly calculating m.
- Sections 3.6 and 3.7 introduce the methods for quick calculation of one.

- Section 3.9 provides a detailed description of the pseudocode implementation.
- Section 3.10 discusses print optimization.

4. Experimental Evaluation

This section presents a comprehensive experimental evaluation of the xjb algorithm, comparing its performance against state-of-the-art floating-point–string conversion algorithms across multiple hardware platforms and compilers.

4.1. Correctness Verification

Prior to performance evaluation, we conducted rigorous correctness verification to ensure that xjb fully complies with the Steele–White (SW) principle and produces accurate results. The verification process covered two key aspects: floating-point–decimal conversion, and floating-point–string conversion.

- **Single-Precision (binary32)**: Given the manageable size of the binary32 search space (2^{32} possible values), we performed exhaustive testing across the entire range. Each output was compared against the reference Schubfach algorithm to ensure identical results, guaranteeing complete correctness for the binary32 format.
- **Double-Precision (binary64)**: Exhaustive testing of all 2^{64} binary64 values is computationally infeasible. Instead, we employed a comprehensive testing strategy that included the following:
  - Large-scale random testing with statistically significant sample sizes.
  - Targeted testing of edge cases including subnormal numbers, extreme exponents, and near-power-of-two values.

All test results confirmed that xjb produces outputs identical to the Schubfach algorithm while fully satisfying the SW principle. The complete verification test suite is publicly available at (`check.cpp`) https://github.com/xjb714/xjb/blob/main/bench/check.cpp (accessed on 20 April 2026).

4.2. Experimental Setup

4.2.1. Hardware Platforms

To evaluate cross-platform performance and portability, we conducted benchmarks on three representative hardware platforms spanning both x86-64 and ARM64 architectures:

- **AMD R7-7840H**: A modern high-performance x86-64 processor with support for AVX2 and AVX-512 instruction sets, running Ubuntu 26.04. This platform represents state-of-the-art x86-64 computing (max frequency: 5.1 GHz).
- **Apple M1**: A first-generation Apple Silicon ARM64 processor with NEON SIMD support, running macOS 26.4. This platform serves as a baseline for ARM64 performance (max frequency: 3.2 GHz).
- **Apple M5**: A recent-generation Apple Silicon ARM64 processor with NEON SIMD support, running macOS 26.4. This platform represents the latest ARM64 technology (max frequency: 4.46 GHz).

4.2.2. Compilers and Compilation Flags

Each platform uses its native compiler toolchain to ensure optimal code generation:

- **AMD R7-7840H**: Intel C++ Compiler (icpx) version 2025.0.4.
- **Apple M1/M5**: Apple Clang version 21.0.0.

All benchmarks were compiled with `-O3 -march=native` to enable maximum compiler optimizations and generate architecture-specific instructions, ensuring fair comparison across platforms.

4.2.3. Benchmark Methodology

Our benchmark methodology was designed to ensure fairness, reproducibility, and statistical significance:

1. **Input Generation**: Generate 2^{24} (16,777,216) random floating-point numbers, excluding special values (NaN, and infinity) to focus on the core conversion logic.
2. **Warm-Up Phase**: Execute the benchmark multiple times before measurement to eliminate cold-start effects and ensure consistent cache behavior.
3. **Measurement**: Measure the total wall-clock time required to convert all numbers through multiple iterations.
4. **Analysis**: Calculate the average conversion time per floating-point number, discarding outliers to ensure robust results.

This methodology minimizes system noise and provides reliable, reproducible performance measurements while avoiding skewed results from special-case handling.

4.3. Algorithms Compared

We compared the xjb algorithm with the other algorithms listed in Table 7. However, for some special cases, we will provide advance explanations.

Table 7. All algorithms in the benchmark test.

Algorithm	Float	Double	Description: Author and Source Code
Schubfach [8]	Schubfach32	Schubfach64	Raffaello Giulietti, https://github.com/abolz/Drachennest/tree/master/src (accessed on 4 December 2025).
Schubfach_xjb [23]	Schubfach32_xjb	Schubfach64_xjb	The computation flow in the Schubfach source code has been modified by me, without altering the original output results, https://github.com/xjb714/xjb/tree/main/bench/schubfach_xjb (accessed on 4 December 2025).
Ryū [6,7]	Ryū32	Ryū64	Ulf Adams, https://github.com/ulfjack/ryu (accessed on 4 December 2025).
Dragonbox [10]	Dragonbox32	Dragonbox64	Junekey Jeon, https://github.com/jk-jeon/Dragonbox (accessed on 4 December 2025).
fmt [24]	fmt32	fmt64	Victor Zverovich, https://github.com/fmtlib/fmt version:12.1.0 (accessed on 4 December 2025)
yy_double [11]	-	yy_double	Guo YaoYuan, https://github.com/ibireme/c_numconv_benchmark/blob/master/vendor/yy_double/yy_double.c (accessed on 4 December 2025).
yy_json [12]	yy_json32	yy_json64	Guo YaoYuan, https://github.com/ibireme/yyjson version:0.12.0 (accessed on 4 December 2025)
teju_jagua [25]	teju32	teju64	Cassio Neri, https://github.com/cassioneri/teju_jagua (accessed on 4 December 2025).
xjb	xjb32	xjb64	This paper, https://github.com/xjb714/xjb (accessed on 4 December 2025).
zmij [16]	zmij32	zmij64	Victor Zverovich, https://github.com/vitaut/zmij (accessed on 8 April 2026).
jnum [26]	jnum32	jnum64	Jing Leng, https://github.com/lengjingzju/json/jnum.c (accessed on 4 December 2025).
uscalec [13]	-	uscalec	Russ Cox, https://github.com/rsc/fpfmt commit 6255750 (accessed on 19 January 2026).

For floating-point numbers to decimal/string:

- **teju_jagua**: Only implements float/double-to-decimal conversion.
- **jnum**: Only implements double-to-string conversion. When comparing float to string, we convert the double value to a float value. Strictly speaking, the jnum algorithm does not satisfy the SW principle. However, its performance is also quite excellent, so we still included it in the benchmark.

For data type:

- **yy_double, uscalec**: Only the double data type is supported.

4.4. Performance Results

We evaluated xjb through two primary conversion interfaces: floating-point–decimal and floating-point–string.

- **Float/Double-to-Decimal Conversion**: Table 8 summarizes the benchmark results for float-to-decimal and double-to-decimal conversions across the AMD R7-7840H and Apple M1/M5 platforms. All benchmarks use random values, excluding NaN and infinity, to focus on core conversion performance.

Table 8. Float/double-to-decimal benchmark results (time in nanoseconds).

Algorithm	AMD R7-7840H		Apple M1		Apple M5	
	Icpx 2025.0.4		Apple Clang 21.0.0		Apple Clang 21.0.0	
	Float	Double	Float	Double	Float	Double
Schubfach	12.20	11.51	11.64	13.12	7.59	7.71
Schubfach_xjb	4.44	6.33	5.16	6.58	3.15	3.75
Ryū	14.02	13.08	15.75	14.16	10.23	9.50
Dragonbox	10.19	10.05	11.78	12.03	7.56	7.39
yy_json	4.67	5.72	3.97	4.46	2.40	2.72
yy_double	–	5.24	–	4.08	–	2.71
teju_jagua	14.99	14.37	20.25	18.66	13.49	12.71
zmij	4.76	4.78	4.11	3.83	2.82	2.14
uscalec	–	11.27	–	15.26	–	9.61
xjb	**2.24**	**3.76**	**2.15**	**2.58**	**1.44**	**1.55**

Note that different algorithms may produce semantically equivalent but syntactically varying decimal outputs. For example, some algorithms (including Dragonbox, uscale, and Ryū) omit trailing zeros in their output representations. Although the results were not consistent, the real values represented by all of the algorithms' outputs all met the SW principle.

- **Float/Double-to-String Conversion**: Figures 1 and 2 present comprehensive benchmark results for double-to-string and float-to-string conversions on the three processor platforms.

  Specifically,

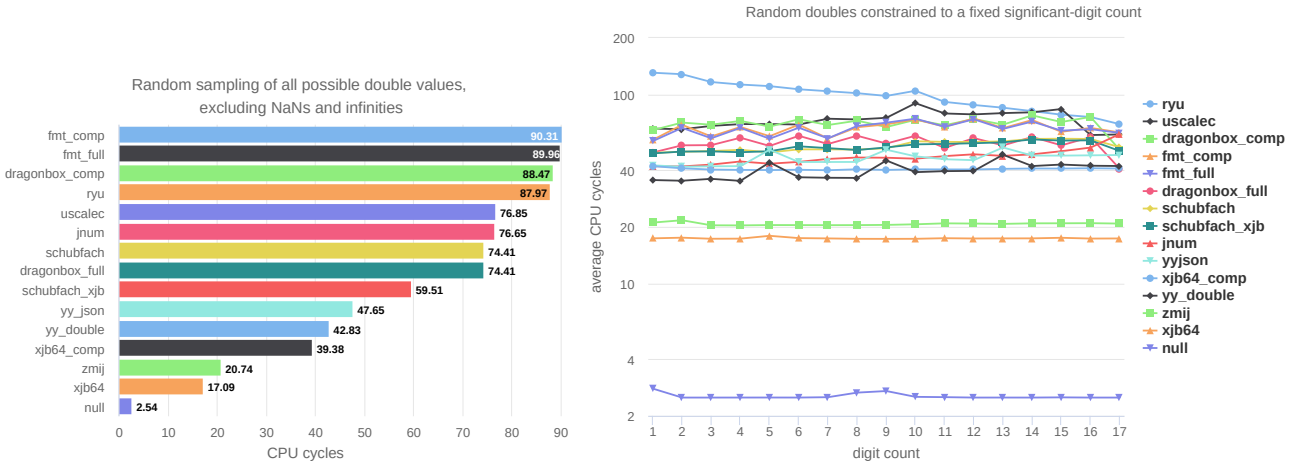
  - Figure 1a,c,e show results for completely random double-precision floating-point numbers.
  - Figure 2a–c show results for completely random single-precision floating-point numbers.
  - Figure 1b,d,f present results for fixed-length significant digits (ranging from 1 to 17 digits).

  In Figures 1 and 2, the suffixes denote specific implementations:

  - `_comp` (e.g., `fmt_comp`, `dragonbox_comp`, `xjb32_comp`, `xjb64_comp`): Versions using compressed constant tables for reduced memory footprint.
  - `_full` (e.g., `fmt_full`, `dragonbox_full`): Versions using uncompressed constant tables for potentially faster access.

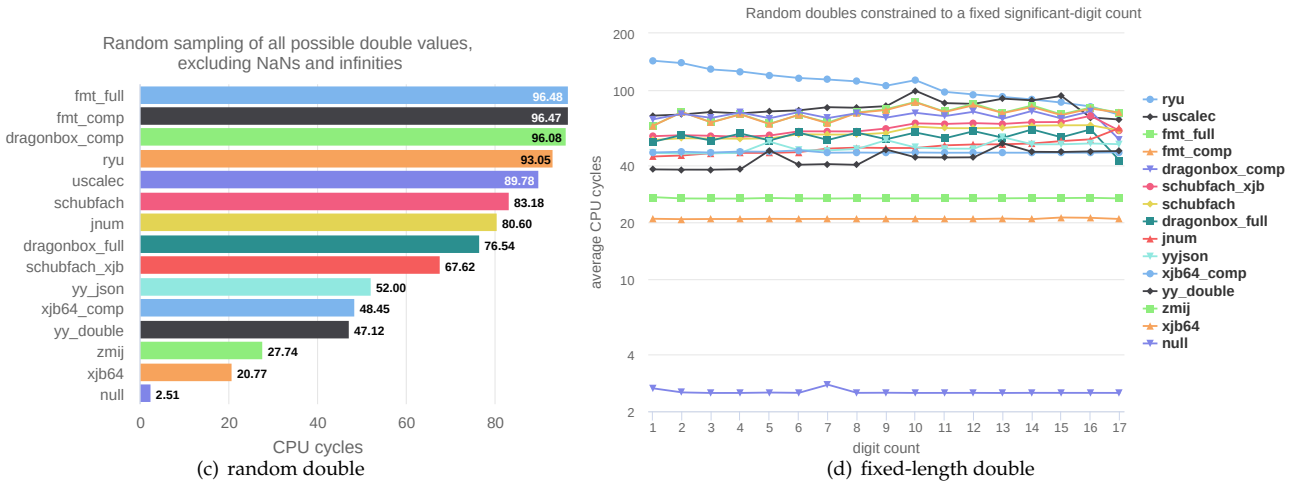
– `null`: An empty function used to isolate and measure the overhead of function calls.

Running on Apple M5 compiled with Apple clang 21.0.0



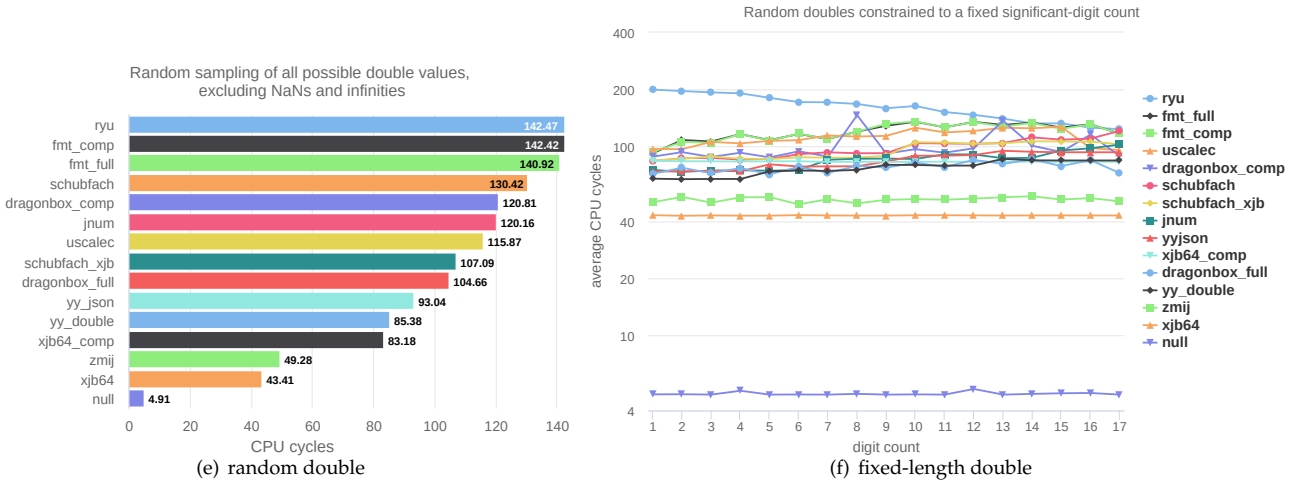
(a) random double (b) fixed-length double

Running on Apple M1 compiled with Apple clang 21.0.0



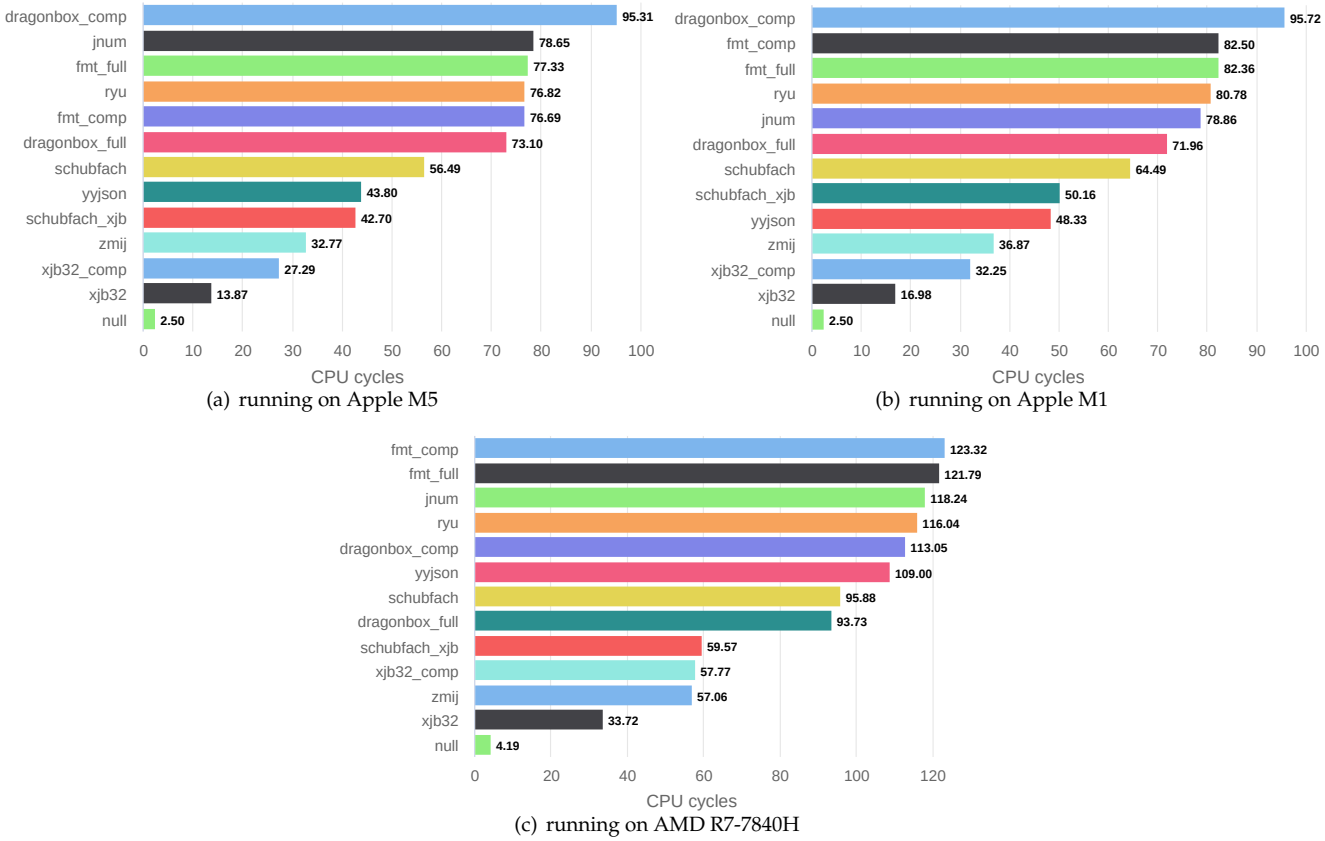
(c) random double (d) fixed-length double

Running on AMD R7-7840H compiled with icpx 2025.0.4



(e) random double (f) fixed-length double

Figure 1. Benchmark results for random and fixed-length double-precision numbers (excluding NaN and Inf).



(a) running on Apple M5 (b) running on Apple M1

(c) running on AMD R7-7840H

Figure 2. Benchmark results for random float-precision numbers (excluding NaN and Inf).

4.5. Analysis and Discussion

4.5.1. Performance Comparison

The benchmark results demonstrate that xjb achieves state-of-the-art performance across all three hardware platforms and both conversion interfaces. For decimal conversion, we compared it against the baseline Schubfach algorithm; for string conversion, we focused on comparisons with zmij, a representative modern high-performance algorithm.

On AMD R7-7840H (x86-64):

- Float-to-decimal: 2.24 ns, 5.44× faster than Schubfach (12.2 ns);
- Double-to-decimal: 3.76 ns, 3.06× faster than Schubfach (11.51 ns);
- Float-to-string: 33.72 cycle, 70% faster than zmij (57.06 cycle);
- Double-to-string: 43.41 cycle, 13% faster than zmij (49.28 cycle).

On Apple M1 (ARM64):

- Float-to-decimal: 2.15 ns, 5.41× faster than Schubfach (11.64 ns);
- Double-to-decimal: 2.58 ns, 5.08× faster than Schubfach (13.12 ns);
- Float-to-string: 16.98 cycle, 117% faster than zmij (36.87 cycle);
- Double-to-string: 20.77 cycle, 25% faster than zmij (27.74 cycle).

On Apple M5 (ARM64):

- Float-to-decimal: 1.44 ns, 5.27× faster than Schubfach (7.59 ns);
- Double-to-decimal: 1.55 ns, 4.97× faster than Schubfach (7.71 ns);
- Float-to-string: 13.87 cycle, 136% faster than zmij (32.77 cycle);
- Double-to-string: 17.09 cycle, 20% faster than zmij (20.74 cycle).

These results establish xjb as the new performance leader in floating-point–string conversion.

4.5.2. Performance Consistency

A defining characteristic of xjb is its consistent performance across diverse input distributions. By designing all conditional branches as unlikely branches, xjb achieves near-optimal branch prediction rates regardless of input patterns. This property is particularly valuable in real-world applications, where input distributions can be unpredictable.

This design addresses a key limitation of algorithms like Dragonbox [10], which trade branch efficiency for reduced multiplication operations. In contrast, xjb simultaneously achieves both minimal multiplication operations and efficient branch handling, combining the strengths of Schubfach [8] and Dragonbox [10] while avoiding their respective trade-offs.

4.5.3. Cross-Platform Performance

The benchmark results demonstrate xjb's strong portability across processor architectures:

- **x86-64 (AMD R7-7840H)**: The algorithm benefits from the compiler's ability to generate highly optimized code for arithmetic operations and SIMD instructions.
- **ARM64 (Apple M1/M5)**: xjb maintains consistent performance advantages across both generations of Apple Silicon, demonstrating the algorithm's robustness and effectiveness across different instruction set architectures.

This cross-platform consistency is achieved through careful algorithm design that works harmoniously with compiler optimizations on diverse platforms.

4.5.4. Comparison with Some Related Algorithms

Placing xjb in the context of prior work reveals several key insights:

- **vs. Schubfach**: With 3–5× speedup over the baseline, xjb validates that Schubfach's elegant mathematical framework can be substantially optimized through computational restructuring, branch optimization, and SIMD utilization.
- **vs. yy_json/yy_double**: While these algorithms represent excellent engineering for JSON serialization, xjb outperforms them by 2–3×, demonstrating that SIMD instruction set utilization unlocks significant additional optimization potential.
- **vs. zmij**: Achieving 1.2–2.3× speedup over zmij highlights the benefits of xjb's approach to instruction dependency reduction, which enables better instruction-level parallelism on modern superscalar processors.
- **vs. Ryū and Dragonbox**: xjb outperforms these established algorithms by 2.5–6×, demonstrating that systematic optimization of multiple bottlenecks simultaneously yields substantial performance gains over approaches that focus on single aspects of the problem.

4.5.5. Fixed-Length Performance Analysis

The fixed-length benchmark results (Figure 1b,d,f) reveal important performance characteristics:

- **Consistent Performance**: All xjb variants (xjb32, xjb32_comp, xjb64, xjb64_comp) maintain consistent performance across all digit lengths (1–17 digits). In contrast, some competing algorithms show significant performance variations depending on the number of output digits.
- **Predictable Latency**: The branch-free core design ensures that the conversion time remains relatively constant regardless of output length. This predictability is a valuable property for real-time systems and high-throughput applications, where consistent latency is as important as raw performance.

- **Compression Trade-Off**: The compressed table variant (_comp) maintains strong competitiveness in performance compared to other algorithms, while also reducing memory usage. This demonstrates the efficiency of xjb in terms of memory utilization.

4.6. Summary

This comprehensive experimental evaluation establishes xjb as the new state-of-the-art method in floating-point–string conversion across multiple hardware platforms. The key findings can be summarized as follows:

1. **Superior Performance**: xjb consistently outperforms all competing algorithms across both x86-64 (AMD R7-7840H) and ARM64 (Apple M1/M5) architectures.
2. **Significant Speedups**: Achieves 3–5× improvement over the baseline Schubfach algorithm.
3. **Performance Consistency**: Maintains stable performance across diverse input distributions and output lengths due to effective branch prediction optimization and branch-free core design.

These results validate the effectiveness of our holistic optimization approach: minimizing multiplication operations, reducing instruction dependencies, optimizing branch patterns, and leveraging SIMD instructions—working synergistically to deliver exceptional performance on modern processor architectures.

5. Conclusions

This paper presented xjb, a novel high-performance algorithm for converting IEEE 754 floating-point numbers to decimal string representations. Building upon the Schubfach algorithm, xjb introduces several key optimizations that significantly improve conversion speed while maintaining full compliance with the Steele–White principle for accurate and minimal-length output.

5.1. Improvements to the Schubfach Algorithm

The xjb algorithm represents a significant advancement over the baseline Schubfach algorithm through several targeted improvements:

1. **Restructured Computation Flow**: By decomposing the significand calculation into integer and fractional parts, xjb minimizes instruction dependencies, enabling better instruction-level parallelism and improved pipeline utilization on modern superscalar processors.
2. **Minimized Multiplication Operations**: xjb reduces the number of expensive high-precision multiplications required during conversion. For IEEE 754 binary64, only one 64-bit by 128-bit multiplication is needed, and for binary32, only one 64-bit by 64-bit multiplication is required, significantly decreasing computational overhead.
3. **Branch Optimization**: The algorithm employs branchless programming techniques for core conversion logic and structures remaining branches as unlikely paths, enabling efficient branch prediction on modern processors and resulting in consistent performance across diverse input distributions.
4. **SIMD Instruction Utilization**: Unlike Schubfach and many other existing algorithms, xjb leverages SIMD instructions (NEON for ARM64, AVX512/SSE4.1/SSE2 for x86-64) for efficient decimal-to-ASCII conversion, fully exploiting the vector processing capabilities of contemporary processors.

5.2. Key Findings

Our extensive experimental evaluation across AMD R7-7840H (x86-64) and Apple M1/M5 (ARM64) platforms reveals several key findings that advance the state of the art in floating-point–string conversion:

- **Significant Performance Improvement**: xjb achieves a remarkable 3–5× speedup over the baseline Schubfach algorithm, representing a substantial leap in performance compared to prior work. On Apple M5, xjb achieves an impressive 1.44 ns for float-to-decimal conversion and 1.55 ns for double-to-decimal conversion, setting a new performance benchmark in the field.
- **Superior to State-of-the-Art Algorithms**: xjb consistently outperforms other high-performance algorithms, including yy_json, yy_double, and zmij, by margins of 1.2–3×. This indicates the performance improvement achieved by using the SIMD instruction set and reducing instruction dependencies. On the Apple M5 processor, compared with zmij, our algorithm achieves approximately 20% and 136% speedups for double-to-string and float-to-string conversion, respectively.
- **Consistent Performance across Platforms**: Unlike prior work, which often shows significant performance variations between architectures, xjb maintains its performance advantage across both x86-64 and ARM64. This portability is achieved through careful algorithm design that works well with compiler optimizations on different platforms.
- **Stable Performance across Input Distributions**: The algorithm maintains consistent performance regardless of input patterns and output digit lengths. This stability can be attributed to our branch-free core design and effective branch prediction optimization for all conditional branches, making xjb ideal for applications requiring predictable performance.
- **Synergistic Optimization Effects**: The combination of instruction dependency reduction, multiplication minimization, branch optimization, and SIMD utilization works synergistically to deliver performance gains that exceed what any single optimization could achieve in isolation.
- **Concise Core Implementation**: The core conversion logic of xjb is implemented in a concise manner, with minimal code lines and clear logic flow. This design simplifies maintenance and allows for easy integration into larger software systems.

These key findings collectively demonstrate that xjb successfully addresses the research gap identified in the Introduction, providing a comprehensive solution to the limitations of existing floating-point–string conversion algorithms.

5.3. Practical Implications

The xjb algorithm has immediate practical applications in numerous domains:

- **Data Serialization**: JSON and other text-based data formats require efficient floating-point–string conversion for serialization operations.
- **Scientific Computing**: Applications that output numerical results in a human-readable format benefit from faster conversion without sacrificing accuracy.
- **Database Systems**: Export operations and query result formatting can leverage xjb for improved throughput.
- **Web Services**: RESTful APIs and web applications that return numerical data can achieve lower latency with efficient conversion.

5.4. Limitations and Future Work

While xjb demonstrates strong performance, several areas warrant further investigation:

1. **Extended Precision Support**: Future work could extend xjb to support extended precision formats (e.g., 16-bit, 80-bit, 128-bit, and 256-bit floating-point numbers) for applications requiring higher precision.
2. **SIMD Vectorization**: Although xjb is designed to be SIMD-friendly, explicit vectorization using AVX-512 or NEON could yield additional performance gains for batch conversion workloads.
3. **Compiler Compatibility**: Further optimization for different compilers (particularly MSVC) would improve portability across development environments.
4. **Memory-Constrained Environments**: Investigating memory-efficient variants of xjb could benefit embedded systems and other resource-constrained platforms.

5.5. Availability

The complete implementation of the xjb algorithm, along with benchmark tools and test suites, is publicly available at https://github.com/xjb714/xjb/releases/tag/v1.5.0 (accessed on 20 April 2026). We encourage the community to integrate, test, and contribute to the ongoing development of this work.

Author Contributions: Methodology, J.X.; Software, J.X.; Validation, J.X.; Writing—original draft, J.X.; Funding acquisition, T.W.; Writing—review & editing, T.W.; Visualization, T.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research is supported by the Sichuan Science and Technology Program (2024ZDZX0001).

Data Availability Statement: All the source code files in our GitHub repository at https://github.com/xjb714/xjb (accessed on 20 April 2026) are freely accessible.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Mathematical Foundations of Fractional Part Boundary

In this Appendix, we collect several elementary facts concerning rational approximation and Farey sequences. These results are used in the main text to bound fractional parts and to compute best rational approximations efficiently.

Appendix A.1. Notation and Assumptions

Let n, P, Q, $n_{\max}$ be positive integers satisfying the following conditions:

- P and Q are coprime and $P < Q$;
- $1 \leqslant n \leqslant n_{\max}$;
- $Q > n_{\max}$.

For a given maximal denominator $n_{\max}$, we denote by

$$\frac{P_*}{Q_*} \quad \text{and} \quad \frac{P^*}{Q^*}$$

the *best rational approximations* of P/Q from below and from above, respectively, i.e.,

$$\frac{P_*}{Q_*} = \max_{\substack{1 \leqslant n \leqslant n_{\max} \\ Q_* \leqslant n_{\max}}} \frac{\lfloor nP/Q \rfloor}{n}, \qquad \frac{P^*}{Q^*} = \min_{\substack{1 \leqslant n \leqslant n_{\max} \\ Q^* \leqslant n_{\max}}} \frac{\lceil nP/Q \rceil}{n}.$$

Appendix A.2. Basic Identities

If nP is not a multiple of Q, then clearly

$$\lfloor n \cdot \tfrac{P}{Q} \rfloor + 1 = \lceil n \cdot \tfrac{P}{Q} \rceil.$$

When nP is a multiple of Q, the left-hand side is one larger than the right-hand side; the non-divisibility assumption avoids this degenerate case.

Appendix A.3. A Useful Equivalence

Assume that, for a real number ξ, the equality

$$\lfloor n \cdot \tfrac{P}{Q} \rfloor = \lfloor n\xi \rfloor$$

holds for all $1 \leqslant n \leqslant n_{\max}$. Then, ξ must lie between the best lower and upper rational approximations of P/Q with the denominator bounded by $n_{\max}$. More precisely,

$$\frac{P_*}{Q_*} = \max_{1 \leqslant n \leqslant n_{\max}} \frac{\lfloor nP/Q \rfloor}{n} \leqslant \xi < \min_{1 \leqslant n \leqslant n_{\max}} \frac{\lfloor nP/Q \rfloor + 1}{n} = \min_{1 \leqslant n \leqslant n_{\max}} \frac{\lceil nP/Q \rceil}{n} = \frac{P^*}{Q^*}.$$

Consequently, the admissible range for ξ is the half-open interval

$$\frac{P_*}{Q_*} \leqslant \xi < \frac{P^*}{Q^*}.$$

Appendix A.4. Range of the Fractional Parts

The fractional part of $n \cdot P/Q$ is defined as $\{nP/Q\} = nP/Q - \lfloor nP/Q \rfloor$. Within the set $\{1, 2, \ldots, n_{\max}\}$, this quantity attains its minimum at $n = Q_*$ and its maximum at $n = Q^*$. Hence, the fractional parts lie exactly in the interval

$$\Big[\frac{(Q_* P) \bmod Q}{Q}, \frac{(Q^* P) \bmod Q}{Q} \Big]. \tag{A1}$$

where $(x \bmod Q)$ denotes the remainder of x upon division by Q, i.e., $0 \leqslant (x \bmod Q) < Q$.

Proof of the Fractional Part Range:

We prove that for all $1 \leqslant n \leqslant n_{\max}$,

$$\{Q_* P/Q\} \leqslant \{nP/Q\} \leqslant \{Q^* P/Q\},$$

with equality at $n = Q_*$ and $n = Q^*$, respectively.

(1) Notation

Let $a = P_*$, $b = Q_*$, $c = P^*$, $d = Q^*$. By definition, a/b and c/d are adjacent terms in the Farey sequence $F_{n_{\max}}$ and satisfy $a/b < P/Q < c/d$ and $bc - ad = 1$. For any n with $1 \leqslant n \leqslant n_{\max}$, set $k = \lfloor nP/Q \rfloor$. The fraction k/n cannot lie strictly between a/b and c/d, because $n \leqslant n_{\max}$ and the two are neighbors in $F_{n_{\max}}$. Consequently,

$$\frac{k}{n} \leqslant \frac{a}{b} \quad \text{and} \quad \frac{k+1}{n} \geqslant \frac{c}{d}.$$

(2) Integer Linear Representation

We now establish a convenient parameterization of the integers n and $k = \lfloor nP/Q \rfloor$ using the coefficients of the adjacent Farey fractions a/b and c/d. Because a/b and c/d are neighbors in the Farey sequence $F_{n_{\max}}$, they satisfy the well-known unimodular relation

$$bc - ad = 1.$$

This means that the 2×2 matrix

$$M = \begin{pmatrix} b & d \\ a & c \end{pmatrix}$$

has the determinant $\det M = bc - ad = 1$. Any integer matrix with a determinant of 1 is invertible over the integers; its inverse is given explicitly by

$$M^{-1} = \begin{pmatrix} c & -d \\ -a & b \end{pmatrix}.$$

For a given n and the corresponding $k = \lfloor nP/Q \rfloor$, consider the column vector $\begin{pmatrix} n \\ k \end{pmatrix}$. Since M is unimodular, there exists a unique pair of integers (x, y) such that

$$M \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} n \\ k \end{pmatrix} \quad \Longleftrightarrow \quad \begin{pmatrix} b & d \\ a & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} n \\ k \end{pmatrix}.$$

Multiplying both sides on the left by M^{-1} yields the explicit formulae for x and y:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} c & -d \\ -a & b \end{pmatrix} \begin{pmatrix} n \\ k \end{pmatrix} = \begin{pmatrix} cn - dk \\ -an + bk \end{pmatrix}.$$

Thus, we obtain the two relations

$$x = cn - dk, \qquad y = bk - an.$$

Now, recall that $k/n \leqslant a/b$. Because $n > 0$ and $b > 0$, we may cross-multiply without changing the inequality to obtain $bk \leqslant an$. This immediately implies that

$$y = bk - an \leqslant 0.$$

Let us define $y' = -y \geqslant 0$. Substituting $y = -y'$ back into the original system gives

$$n = bx + d(-y') = bx - dy', \qquad k = ax + c(-y') = ax - cy'.$$

We therefore arrive at the non-negative parameterization

$$n = bx - dy', \qquad k = ax - cy',$$

with integers $x \geqslant 1$ and $y' \geqslant 0$. If $x \leqslant 0$, then $n = bx - dy' \leqslant 0$, contradicting $n \geqslant 1$; hence x must be strictly positive.

This representation is the key to expressing the fractional part $\{nP/Q\}$ as a simple linear combination of two fundamental positive quantities, which will be exploited in the next step.

(3) Minimum of the Fractional Parts

Let $\varepsilon = P/Q - a/b > 0$. Then, $\{bP/Q\} = b\varepsilon$. Using the representation above,

$$\{nP/Q\} = \frac{nP}{Q} - k = \frac{P}{Q}(bx - dy') - (ax - cy') = x\Big(\frac{bP}{Q} - a\Big) - y'\Big(\frac{dP}{Q} - c\Big).$$

Since $dP/Q - c = -(c - dP/Q) = -d\delta$, where $\delta = c/d - P/Q > 0$, we have

$$\{nP/Q\} = x \cdot b\varepsilon + y' \cdot d\delta.$$

Both $b\varepsilon$ and $d\delta$ are positive, and $x \geqslant 1$, $y' \geqslant 0$. Therefore,

$$\{nP/Q\} \geqslant 1 \cdot b\varepsilon + 0 \cdot d\delta = b\varepsilon = \{bP/Q\},$$

with equality if and only if $x = 1$ and $y' = 0$, i.e., $n = b$ and $k = a$. Thus, the minimum is attained at $n = Q_*$.

(4) Maximum of the Fractional Parts

A completely symmetric argument using the upper approximation gives

$$1 - \{nP/Q\} = \frac{k+1}{n} \cdot n - \frac{nP}{Q} = n\Big(\frac{k+1}{n} - \frac{P}{Q}\Big).$$

We have $(k+1)/n \geqslant c/d$, and using the unimodular relations one can show that

$$1 - \{nP/Q\} = x' \cdot d\delta + y'' \cdot b\varepsilon$$

for some non-negative integers x', y'' with $x' \geqslant 1$. Hence,

$$1 - \{nP/Q\} \geqslant d\delta = 1 - \{dP/Q\},$$

which is equivalent to $\{nP/Q\} \leqslant \{dP/Q\}$. The maximum is therefore achieved at $n = d = Q^*$. For a detailed proof of the upper bound, one may also consult the theory of continued fractions: the largest fractional part among $n\theta \pmod 1$ for $n \leqslant N$ occurs at the denominator of the best upper approximation.

(5) Conclusion

The fractional parts all lie in the interval $[\{Q_*P/Q\}, \{Q^*P/Q\}]$, and the endpoints are exactly $(Q_*P \bmod Q)/Q$ and $(Q^*P \bmod Q)/Q$, as claimed.

Appendix A.5. Computation via Farey Sequences

The best rational approximations with bounded denominators can be obtained efficiently from the Farey sequence of order C. We define the function

$$\big(\tfrac{P_*}{Q_*}, \tfrac{P^*}{Q^*}\big) = (DN, UP) = f(C, P, Q) \tag{A2}$$

which returns the two adjacent terms in the C-th Farey sequence F_C that bracket P/Q. The implementation of f relies on the mediant property of Farey sequences and is provided in the accompanying code (see `test1.py`) https://github.com/xjb714/xjb/blob/main/py_test/test1.py (accessed on 20 April 2026)).

Using this function, the bounds in Appendix A.4 can be computed in negligible time, thereby providing a fast method to determine the range of fractional parts discussed in the main text.

References

1. Steel, G.L., Jr.; White, J.L. 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, 1990; pp. 112–126. https://doi.org/10.1145/93542.93559.
2. Steel, G.L., Jr.; White, J.L. How to Print Floating-Point Numbers Accurately (Retrospective). ACM SIGPLAN Notices 39(4), April 2004 (Best of PLDI, 1979–1999). Available online: https://dl.acm.org/doi/10.1145/989393.989431 (accessed on 1 April 2004).

3. Burger, R.G.; Dybvig, R.K. Printing Floating-point Numbers Quickly and Accurately. In *Proceedings of the ACM SIGPLAN1996 Conference on Programming Language Design and Implementation (PLDI '96)*; ACM: New York, NY, USA, 1996; pp. 108–116. https://doi.org/10.1145/231379.231397.
4. Loitsch, F. 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, 2010; pp. 233–243. https://doi.org/10.1145/1806596.1806623.
5. Andrysco, M.; Jhala, R.; Lerner, S. Printing Floating-Point Numbers: A Faster, Always Correct Method. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*; ACM: New York, NY, USA, 2016; pp. 555–567. https://doi.org/10.1145/2837614.2837654.
6. Adams, U. Ryū: Fast Float-to-String Conversion. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*; ACM: New York, NY, USA, 2018; pp. 270–282. https://doi.org/10.1145/3192366.3192369.
7. Adams, U. Ryū Revisited: Printf Floating Point Conversion. *Proc. ACM Program. Lang.* **2019**, *3*, 169. https://doi.org/10.1145/3360595.
8. Giulietti, R. The Schubfach Way to Render Doubles. 2020. Available online: https://drive.google.com/file/d/1KLtG_LaIbK9ETXI290zqCxvBW94dj058/view (accessed on 1 September 2020).
9. Jeon, J. Grisu-Exact: A Fast and Exact Floating-Point Printing Algorithm. 2020. Available online: https://github.com/jk-jeon/Grisu-Exact/blob/master/other_files/Grisu-Exact.pdf. (accessed on 1 September 2020).
10. Jeon, J. Dragonbox: A New Floating-Point Binary-to-Decimal Conversion Algorithm. 2024. Available online: https://github.com/jk-jeon/Dragonbox (accessed on 1 July 2024).
11. Guo, Y.Y. Available online: https://github.com/ibireme/c_numconv_benchmark/blob/master/vendor/yy_double/yy_double.c (accessed on 1 January 2025).
12. Guo, Y.Y. Available online: https://github.com/ibireme/yyjson (accessed on 1 August 2025).
13. Cox, R. Available online: https://github.com/rsc/fpfmt (accessed on 1 January 2026).
14. Cox, R. Floating-Point Printing and Parsing Can Be Simple And Fast. Available online: https://research.swtch.com/fp (accessed on 1 January 2026).
15. Cox, R. Fast Unrounded Scaling: Proof by Ivy. Available online: https://research.swtch.com/fp-proof (accessed on 1 January 2026).
16. Zverovich, V. Available online: https://github.com/vitaut/zmij (accessed on 1 March 2026).
17. *ANSI/IEEE Std 754-1985*; IEEE Standard for Binary Floating-Point Arithmetic. IEEE: New York, NY, USA, 1985; pp. 1–20. https://doi.org/10.1109/IEEESTD.1985.82928.
18. *IEEE Std 754-2019*; (Revision of IEEE 754-2008) IEEE Standard for Floating-Point Arithmetic. IEEE: New York, NY, USA, 2019; pp. 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229.
19. Khuong, P. How to Print Integers Really Fast (with Open Source AppNexus Code!). Available online: https://pvk.ca/Blog/2017/12/22/appnexus-common-framework-its-out-also-how-to-print-integers-faster/ (accessed on 1 December 2017).
20. Johnson, D. Converting Integers to Fixed-Width Strings Faster with Neon SIMD on the Apple M1. Available online: https://dougallj.wordpress.com/2022/04/01/converting-integers-to-fixed-width-strings-faster-with-neon-simd-on-the-apple-m1/ (accessed on 1 April 2022).
21. Muła, W. SSE: Conversion Integers to Decimal Representation. Available online: http://0x80.pl/notesen/2011-10-21-sse-itoa.html (accessed on 1 October 2011).
22. Lemire, D. Converting Integers to Decimal Strings Faster with AVX-512. Available online: https://lemire.me/blog/2022/03/28/converting-integers-to-decimal-strings-faster-with-avx-512/ (accessed on 1 March 2022).
23. Xiang, J. Available online: https://github.com/xjb714/xjb/tree/main/bench/schubfach_xjb (accessed on 1 April 2026).
24. Zverovich, V. Available online: https://github.com/fmtlib/fmt (accessed on 1 October 2025).
25. Neri, C. Available online: https://github.com/cassioneri/teju_jagua (accessed on 1 November 2025).
26. Leng, J. Available online: https://github.com/lengjingzju/json/jnum.c (accessed on 1 November 2025).