Ryū: Fast Float-to-String Conversion

Ulf Adams
Google Germany
ulfjack@google.com

Abstract

We present Ryū, a new routine to convert binary floating point numbers to their decimal representations using only fixed-size integer operations, and prove its correctness. Ryū is simpler and approximately three times faster than the previously fastest implementation.

CCS Concepts • **Computing methodologies → Representation of mathematical objects**;

Keywords float, string, performance

ACM Reference Format:
Ulf Adams. 2018. 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, 13 pages. https://doi.org/10.1145/3192366.3192369

1 Introduction

Some applications output many floating point numbers. For these applications, finding a simple, fast conversion routine is valuable. For example, JavaScript engines use floating point for their primary number type, so any time they output a number they use such a routine. The most common use case in practice is conversion from binary floating point numbers to decimal strings.

We use the same three correctness criteria as Steele and White [15]: (1) **Information preservation** — a correct parser must return the original floating point value from the output. (2) **Minimum-length output** — the output string must be as short as possible. (3) **Correct rounding** — the output string must be as close to the input as possible.

We do not apply their fourth criteria, which is left-to-right generation. Output is usually buffered, which makes the order in which a routine generates digits inconsequential.

Routines following these criteria can still produce different output due to two edge cases. (1) Depending on the rounding mode of the correct parser, the conversion routine may or may not output the exact halfway point between two consecutive floating point numbers. (2) If the input lies exactly halfway between two shortest information-preserving strings, then these criteria do not provide a rule to break such ties. Our routines support all rounding modes for the parser, as well as for breaking ties.

Section 2 describes a simple, complete, and correct conversion routine that uses arbitrary precision arithmetic.

Based on this, Section 3 introduces Ryū, which significantly reduces the number of required bits. It is simple to implement, leading to a robust and fast implementation.

Section 4 evaluates the performance of Ryū, comparing our C implementation to Grisu3, and our Java implementation to OpenJDK and Jaffer's algorithm [10]. Our results indicate that Ryū is about three times faster than the best existing conversion routine that follows these criteria.

Section 5 reviews the existing literature. All of the early approaches require arbitrary precision arithmetic in the general case. More recent developments work with fixed precision arithmetic, which is faster, but can increase the complexity of the code. By contrast, Ryū is simple and fast.

Section 6 briefly reviews our results.

2 Basic Conversion Routine

After briefly reviewing the IEEE floating point format in Section 2.1, Section 2.2 outlines a simple and correct routine to convert such numbers into decimal strings: (1) Decode the floating point number. (2) Compute the interval of information-preserving outputs. (3) Convert the interval to a decimal power base. (4) Determine the shortest, correctly-rounded string within this interval. (5) Print.

Of these, step 4 is the most complex — Section 2.3 introduces the `compute_shortest` algorithm, which determines an interval of minimum-length outputs within a given interval of information-preserving outputs by repeatedly removing trailing digits as long as the resulting interval is not empty. Section 2.4 then describes how to find the correctly-rounded output within this interval, and how to determine whether there is a tie.

Steps 3 and 4 both require arithmetic with a number of bits exponential in the size of the exponent of the underlying floating point type, i.e., they do not perform well for larger floating point types. Even so, the basic conversion routine is useful for validating high-performance implementations due to its simplicity. Section 4 describes how we used it to discover correctness issues in existing implementations.

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06.
https://doi.org/10.1145/3192366.3192369

2.1 IEEE Floating Point Types

The IEEE 754 standard [9] specifies 16-, 32-, 64-, 128-, and 256-bit floating point numbers, as well as general rules for even larger sizes. The bit representation of such a floating point number consists of a sign bit s, an unsigned exponent e, and an unsigned mantissa m, in that order.

In the common case, exponent values are neither all zeros nor all ones $(0\ldots0)_2 < e < (1\ldots1)_2$; we call these normalized. Interpreting a normalized value as a number involves prepending an implicit leading 1 to the mantissa, and subtracting an IEEE-defined bias from the exponent. An exponent of all zeros $e = (0\ldots0)_2$ indicates subnormal numbers which prepend an implicit leading 0 instead. An exponent value of all ones $e = (1\ldots1)_2$ indicates either ±Infinity or NaN depending on the value of the mantissa.

In summary, the IEEE 754 standard specifies the value f of a floating point number as:

$$f = \begin{cases} (-1)^s \cdot 1.m \cdot 2^{e-\text{bias}} & \text{if } (0\ldots0)_2 < e < (1\ldots1)_2 \\ (-1)^s \cdot 0.m \cdot 2^{1-\text{bias}} & \text{if } e = (0\ldots0)_2 \\ (-1)^s \cdot \text{Infinity} & \text{if } e = (1\ldots1)_2, m = 0 \\ \text{NaN} & \text{if } e = (1\ldots1)_2, m \neq 0 \end{cases}$$

2.2 Conversion Process

The following steps convert a floating point number f from IEEE binary encoding into a decimal string representation:

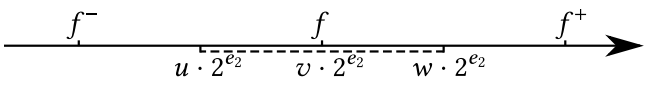
Step 1. Decode the floating point number, and unify normalized and subnormal cases.

Extract the sign bit s, exponent e, and mantissa m. Immediately return the desired output if f is ±Infinity, NaN, or ±0.0. Otherwise, f must be normalized or subnormal. We convert f into the intermediate form $f = (-1)^s \cdot m_f \cdot 2^{e_f}$ such that m_f is an unsigned integer. Making m_f an integer involves moving the decimal dot all the way to the right, which requires subtracting the length of the mantissa $\text{len}(m)$ from the exponent to compute e_f:

$$m_f = \begin{cases} 2^{\text{len}(m)} + m & \text{if } e \neq 0 \\ m & \text{if } e = 0 \end{cases}$$

$$e_f = \begin{cases} e - \text{bias} - \text{len}(m) & \text{if } e \neq 0 \\ 1 - \text{bias} - \text{len}(m) & \text{if } e = 0 \end{cases}$$

Step 2. Determine the interval of information-preserving outputs.



We compute the halfway points to the next smaller and larger floating point values f^- and f^+ of the same floating point type, and represent these as $u \cdot 2^{e_2}$ and $w \cdot 2^{e_2}$, respectively. We also convert f into the same form $v \cdot 2^{e_2}$ intentionally using the same exponent e_2, which is required for step 4. Using $e_2 = e_f - 2$ guarantees that all of u, v, and w are integers.

First consider the larger halfway point. Given $f = m_f \cdot 2^{e_f}$, the next larger number of the same precision is $f^+ = (m_f + 1) \cdot 2^{e_f}$, resulting in a halfway point of

$$(f + f^+)/2 = (2m_f + 1) \cdot 2^{e_f - 1} = (4m_f + 2) \cdot 2^{e_2}.$$

The only exception occurs if f is itself the largest representable floating point number, i.e., $m = (1\ldots1)_2$ and $e = (1..10)_2$. In that case, the next larger number is infinity, and we define the larger halfway point using the same equation as all other cases for simplicity and consistency.

Now consider the smaller halfway point. If m is not all zeros, then the next smaller number is $f^- = (m_f - 1) \cdot 2^{e_f}$, resulting in $(4m_f - 2) \cdot 2^{e_2}$. The same formula applies if f is the smallest normalized floating point value $m = 0, e = 1$. Its next smaller floating point value is $m' = (1\ldots1)_2$ and $e' = 0$, which results in the same expression.

If the mantissa is zero and the exponent is greater than one, then the next smaller number is $m' = (1\ldots1)_2$, $e' = e - 1$, resulting in $(4m_f - 1) \cdot 2^{e_2}$.

In summary, we represent the 3-tuple (smaller halfway point, f, larger halfway point) as $(u, v, w) \cdot 2^{e_2}$ with:

$$e_2 = e_f - 2$$

$$u = 4m_f - \begin{cases} 1 & \text{if } m = (0\ldots0)_2 \text{ and } e > (0..01)_2 \\ 2 & \text{otherwise} \end{cases}$$

$$v = 4m_f$$

$$w = 4m_f + 2$$

Step 3. Convert $(u, v, w) \cdot 2^{e_2}$ to a decimal power base.

We determine values for (a, b, c) and e_{10} such that $(a, b, c) \cdot 10^{e_{10}}$ equals $(u, v, w) \cdot 2^{e_2}$; there are many possible choices for e_{10}, and we choose specific values to prepare for the new conversion routine presented in Section 3. If e_2 is greater than or equal to zero, we choose e_{10} to be zero. Otherwise, we choose e_{10} to be identical to e_2, which requires multiplying all numbers by 5^{-e_2}:

$$e_2 \geq 0 \Rightarrow e_{10} = 0, (a, b, c) = (u, v, w) \cdot 2^{e_2}$$

$$e_2 < 0 \Rightarrow e_{10} = e_2, (a, b, c) = (u, v, w) \cdot 5^{-e_2}$$

Step 4. Find a shortest, correctly-rounded decimal representation in the interval of legal representations.

We determine two integers d_o and e_o such that

$$a \cdot 10^{e_{10}} < d_o \cdot 10^{e_o + e_{10}} < c \cdot 10^{e_{10}}$$

and e_o is maximal, allowing equality to the smaller or larger halfway point depending on the rounding mode. There may be multiple possible values for d_o in which case we need to pick the one closest to f. We provide an algorithm for this step in Sections 2.3 and 2.4.

Step 5. Print the decimal representation.

Finally, print $(-1)^s \cdot d_o \cdot 10^{e_o + e_{10}}$ in the desired form. For example, to print in scientific notation, we can use an existing

integer to string conversion routine to convert d_o. Let the length of the resulting string be *len*. Prepend the sign, insert the character "." after the first digit, append a character "e", and append the exponent $(e_o + e_{10} + len - 1)$ also converted to a decimal string.

2.3 Finding the Shortest Decimal Representation by Repeated Division

Let a and c be the boundaries of an interval within which we want to find the shortest decimal representation. As we will see, the algorithm requires $0 < a < c - 1$ as a precondition, which holds when a and c are computed as described in step 3.

Using the given rounding mode and the sign of f, we introduce two boolean flags *accept_smaller* and *accept_larger*, which indicate whether the smaller or larger boundary may be returned exactly, respectively. Then we determine d_o and e_o, such that

$$\begin{cases} a \le d_o \cdot 10^{e_o} & \text{if } accept\_smaller = \text{true} \\ a < d_o \cdot 10^{e_o} & \text{otherwise} \end{cases}$$

AND

$$\begin{cases} d_o \cdot 10^{e_o} \le c & \text{if } accept\_larger = \text{true} \\ d_o \cdot 10^{e_o} < c & \text{otherwise,} \end{cases}$$

(information preservation) and e_o is maximal (minimum-length output), i.e., there is no valid solution (d_t, e_t) with $e_t > e_o$.

The following algorithm computes d_o and e_o:

```
def compute_shortest(
                a, b, c, accept_smaller, accept_larger)
    i = 0, a_0 = a, c_0 = { c      if accept_larger
                          { c - 1  otherwise
    all_a_zero_0 = true
    while ⌊a_i/10⌋ < ⌊c_i/10⌋:
        all_a_zero_{i+1} = all_a_zero_i AND a_i%10 == 0
        a_{i+1} = ⌊a_i/10⌋, c_{i+1} = ⌊c_i/10⌋
        i = i + 1
    if accept_smaller AND all_a_zero_i:
        while a_i%10 == 0 :
            a_{i+1} = a_i/10, c_{i+1} = ⌊c_i/10⌋
            i = i + 1
    return  (d_o, e_o) = (c_i, i)
```

Lemma 2.1. *The* `compute_shortest` *algorithm terminates and returns a result that maintains information preservation and minimum-length output.*

Proof. **Termination**: the first loop maintains a loop invariant of $a_i < c_i$, which together with the precondition $0 < a_0$ implies $0 \le a_i < c_i$. The first loop terminates after a finite number of steps, because all the c_i are positive integers, and the sequence $c_0, c_1, \ldots$ is strictly decreasing. In the worst case, the loop continues until $a_i = 0$, and $c_i < 10$. Therefore there must be an index j such that $\lfloor a_j/10 \rfloor = \lfloor c_j/10 \rfloor$, at which point the first loop exits.

If *accept_smaller* is false or $all\_a\_zero_j$ is false, then the algorithm terminates immediately after exiting the first loop. Otherwise *accept_smaller* and $all\_a\_zero_j$ are both true. By induction over the first loop, $all\_a\_zero_j$ being true implies that all digits removed from a so far were zeros. Since no nonzero digits have been removed, and $0 < a$, there must be at least one nonzero digit left in a_j. That, in turn, implies that there must be an index $k \ge j$ for which $a_k\%10$ is not zero, at which point the second loop terminates. This index must be reached in a finite number of steps, because all the a_i are integer, and the sequence $a_j, a_{j+1}, \ldots$ is strictly decreasing.

Information preservation: by induction, we obtain for all $i \le k$ that $a_i = \lfloor a_0/10^i \rfloor$ and $c_i = \lfloor c_0/10^i \rfloor$. This implies that $c_j \cdot 10^j = \lfloor c_0/10^j \rfloor \cdot 10^j \le c_0$.

If *accept_smaller* is false, or $all\_a\_zero_j$ is false, or $a_j\%10$ is not zero, then the algorithm exits after the first loop with $(d_o, e_o) = (c_j, j)$ and it holds that

$$a < (\lfloor a_0/10^j \rfloor + 1) \cdot 10^j = (a_j + 1) \cdot 10^j \le c_j \cdot 10^j$$

AND

$$\begin{cases} c_j \cdot 10^j \le c_0 = c & \text{if } accept\_larger = \text{true} \\ c_j \cdot 10^j \le c_0 < c & \text{otherwise} \end{cases}$$

as required.

Otherwise, after exiting the first loop, both $all\_a\_zero_j$ and *accept_smaller* are true, and $a_j\%10$ is zero. After the first iteration, the second loop maintains the invariant $a_i = c_i$ and $a = a_i \cdot 10^i = c_i \cdot 10^i$ for all $j < i \le k$, again by induction. Therefore, the algorithm exits with $(d_o, e_o) = (c_k, k)$ and it holds that

$$a = c_k \cdot 10^k$$

AND

$$\begin{cases} c_k \cdot 10^k \le c & \text{if } accept\_larger = \text{true} \\ c_k \cdot 10^k < c & \text{otherwise} \end{cases}$$

as required.

Minimum-length output: we show that e_o is maximal by contradiction. Assume that there is another solution (d_t, e_t) with e_t greater than e_o.

Further assume that *accept_smaller* is false, so the algorithm exits with (c_j, j), and $e_t \ge j + 1$. It follows that

$$a_0 < d_t \cdot 10^{e_t} \le c_0 \Leftrightarrow \lfloor a_0/10^{e_t} \rfloor < d_t \le \lfloor c_0/10^{e_t} \rfloor.$$

We replace $\lfloor a_0/10^j \rfloor$ with a_j and $\lfloor c_0/10^j \rfloor$ with c_j to obtain

$$\lfloor a_j/10^{e_t - j} \rfloor < d_t \le \lfloor c_j/10^{e_t - j} \rfloor$$

with $e_t - j \ge 1$. This is a contradiction, because the first loop only terminates if $\lfloor a_j/10 \rfloor \ge \lfloor c_j/10 \rfloor$.

We still need to cover the case where *accept_smaller* is true. We have already shown that there is no pair (d_t, e_t) such that $a < d_t \cdot 10^{e_t} \le c_0$ and $e_t > j$. We therefore only

need to consider pairs such that a is equal to $d_t \cdot 10^{e_t}$ and e_t greater than k. If $all\_a\_zero_j$ is false, then there is no e_t greater than j such that a is equal to $d_t \cdot 10^{e_t}$, because $all\_a\_zero_j$ being false implies that at least one of the trailing j digits of a is nonzero.

Otherwise both $accept\_smaller$ and $all\_a\_zero_j$ are true, and the second loop is executed until $a_k\%10$ is nonzero. But if the $k+1^{\text{st}}$ trailing digit of a is nonzero, then no $e_t \geq k+1$ can exist such that $a = d_t \cdot 10^{e_t}$. □

2.4 Finding the Correctly-Rounded Decimal Representation

We now modify the `compute_shortest` algorithm to output the string that is not just a shortest string, but actually the closest to the original value. We modify `compute_shortest` in three places as follows, assuming the existence of an additional $break\_tie\_down$ parameter to control whether ties should be broken by rounding down.

(1) Add the following code to the initialization:

$digit_0 = 0$, $all\_b\_zero_0 = \text{true}$, $b_0 = b$

(2) Add the following code to the two loops before the updates to i respectively:

$digit_{i+1} = b_i\%10$
$all\_b\_zero_{i+1} = all\_b\_zero_i \text{ AND } (digit_i == 0)$
$b_{i+1} = \lfloor b_i/10 \rfloor$

(3) Replace the `return` statement with:

$is\_tie = (digit_i == 5) \text{ AND } all\_b\_zero_i$
$want\_round\_down = (digit_i < 5)$
OR $(is\_tie \text{ AND } break\_tie\_down)$
$round\_down = (want\_round\_down$
AND $(a_i \neq b_i \text{ OR } all\_a\_zero_i))$
OR $(b_i + 1 > c_i)$

$$\texttt{return } (d_0, e_0) = \left(\begin{cases} b_i & \text{if } round\_down \\ b_i + 1 & \text{otherwise} \end{cases}, i \right)$$

Lemma 2.2. *The modified* `compute_shortest` *algorithm terminates and returns a result that maintains information preservation, minimum-length output, and correct rounding.*

Proof. **Termination**: immediately follows from Lemma 2.1.

Information preservation, minimum-length output: given $a < b < c$ as a precondition, which holds by construction of step 3, it follows by induction that $a_i \leq b_i \leq c_i$. From the proof of Lemma 2.1, it also follows that the legal interval is the interval from a_i to c_i, with the lower bound being inclusive if and only if $all\_a\_zero_i$ is true, and the upper bound always being inclusive.

A return value of b_i is therefore legal if and only if $a_i \neq b_i$ or $all\_a\_zero_i$ is true, and a return value of $b_i + 1$ is illegal if and only if $b_i + 1 > c_i$; this exactly matches the added conditions for round_down in (3). Note that it is not possible for b_i and $b_i + 1$ to both be illegal at the same time as we have previously shown that c_i is a legal output and $b_i \leq c_i$.

Correct rounding: by construction, $b \cdot 10^{e_{10}}$ is equal to $v \cdot 2^{e_2}$ is equal to $m_f \cdot 2^{e_f}$ is equal to $|f|$. Furthermore, by induction $b_i = \lfloor b/10^i \rfloor$. It follows that $b_i \cdot 10^{i+e_{10}} \leq |f| < (b_i + 1) \cdot 10^{i+e_{10}}$.

Of these two, b_i is closer to $|f|$ if and only if $digit_i < 5$, and $b_i + 1$ is closer to $|f|$ if and only if $digit_i \geq 5$ and $all\_b\_zero_i$ is false. $|f|$ is exactly halfway between the two options if and only if $digit_i = 5$ and $all\_b\_zero_i$ is true. In that case, we can apply any desired rounding mode to break the tie. Given $break\_tie\_down$, this exactly matches the conditions added for $want\_round\_down$. □

3 Ryū

We now introduce Ryū. Starting from the basic conversion routine described above, we show how to reduce the number of bits of the intermediate computations as well as the number of iterations to improve the performance of the `compute_shortest` algorithm for larger floating point types.

The first two steps of the basic conversion routine above only require simple integer operations, which are fast. We follow these two steps, decode the number f, and compute the interval of legal decimal representations $(u, w) \cdot 2^{e_2}$.

We then combine the power base conversion (step 3) with the first q iterations of the algorithm's primary loop into a single step 3', with q chosen as large as possible such that the algorithm's primary loop invariant still holds. As we will see, we can choose q such that only a small number of iterations of the algorithm remain, which constitutes step 4'.

In order to skip parts of the primary loop, we need to directly compute the values of all its variables after q iterations. We subdivide this problem into three subproblems:

Lemma 3.1. *Let (a, b, c), $accept\_smaller$, and $accept\_larger$ be the inputs to the* `compute_shortest` *algorithm. We can compute intermediate values for all the variables in the primary loop if we can solve these three subproblems:*

***(1)** Choose q such that the corresponding values of a_q and c_q are small, but still differ by at least two in order to maintain the primary loop invariant, i.e., $\lfloor a/10^q \rfloor < \lfloor c/10^q \rfloor - 1$.*

Section 3.1 determines an appropriate upper bound for q, which in turn limits the magnitude of the intermediate values $\lfloor (a, b, c)/10^q \rfloor$ — they usually fit into n bits if the underlying floating point type has n bits.

***(2)** Compute $\lfloor (a, b, c)/10^q \rfloor$ directly from (u, v, w) using fixed-precision arithmetic.*

Depending on the sign of e_2, we multiply by either $\lfloor 5^q/2^k \rfloor$ or $(\lfloor 2^k/5^q \rfloor + 1)$ and shift the resulting products right by an appropriate amount. This is trivially correct for k equal to zero in the first case and large values of k in the second case. Using a software-assisted proof, Section 3.2 derives non-trivial lower / upper bounds for k for the standard IEEE floating point types. This is the key contribution of our paper: it reduces the number of bits required for the multipliers to

approximately $2n$ for all considered cases, and is the main reason for Ryū's simplicity and performance.

(3) *Finally, we need to infer whether all of the q removed digits of a, b, and c were zeros without computing their actual values, i.e., whether $a\%10^q = 0$ is true, whether $b\%10^q = 0$ is true, and whether $c\%10^q = 0$ is true.*

Section 3.3 solves this problem using partial prime factorizations. Based on these results, Section 3.4 summarizes the Ryū's new steps 3' and 4'.

Proof. We need to compute the values of all local variables after q iterations, namely a_q, b_q, c_q, and $all\_a\_zero_q$ (Section 2.3), as well as $digit_q$ and $all\_b\_zero_q$ in order to implement correct rounding (Section 2.4), and we conservatively require that the loop invariant holds, i.e., $a_q < c_q$.

We obtain $a_q = \lfloor a/10^q \rfloor$ from the proof of Lemma 2.1. Furthermore, $all\_a\_zero_q$ is true if and only if all q removed digits from a were zeros.

The value of c_q depends on the *accept_larger* flag. If *accept_larger* is true, then $c_q = \lfloor c/10^q \rfloor$. If *accept_larger* is false, then we need to compute $\lfloor (c-1)/10^q \rfloor$ instead:

$$\lfloor (c-1)/10^q \rfloor = \begin{cases} \lfloor c/10^q \rfloor - 1, & \text{if } c\%10^q = 0 \\ \lfloor c/10^q \rfloor, & \text{otherwise} \end{cases}$$

That is, we can derive c_q from $\lfloor c/10^q \rfloor$ if we also know whether all of the q removed digits from c were zeros.

Given $\lfloor a/10^q \rfloor < \lfloor c/10^q \rfloor - 1$, we can conservatively infer that the loop invariant $a_q < c_q$ still holds even if all removed digits from c were zeros and *accept_larger* is false.

Furthermore, $all\_b\_zero_q$ is true if and only if $q-1$ removed digits from b were zeros. $digit_q$ is never read inside the loop, so we do not require a value for it as long as either q is zero — in which case it must be zero — or we can ensure that the loop is executed at least once. In order to do so, we use $q' = \max(0, q-1)$ instead of q. □

3.1 Maximum Number of Iterations q That Can Be Safely Combined

The more iterations of the `compute_shortest` algorithm we can combine, the fewer bits are needed to represent the resulting intermediate values, and the fewer iterations remain. However, we have to choose q such that the primary loop invariant still holds to maintain correctness.

Lemma 3.2. *Let (u, v, w) and e_2 be given as above, as well as $(a, b, c) = (u, w) \cdot 2^{e_2}$ or $(a, b, c) = (u, v, w) \cdot 5^{-e_2}$ depending on the sign of e_2.*

- *If $e_2 \geq 0$, choose $q = \lfloor e_2 \log_{10} 2 \rfloor$ to ensure $\lfloor a/10^q \rfloor < \lfloor c/10^q \rfloor - 1$. The resulting value of $\lfloor c/10^q \rfloor$ is at most ten times larger than the original number w, i.e., $\lfloor c/10^q \rfloor \leq 10 \cdot w$, so it fits into an unsigned integer with the same number of bits as the original floating point number for all IEEE floating point types. Note that the resulting value may be a hundred times larger than w if we use $q' = \max(0, q-1)$ instead of q (see Lemma 3.1).*
- *For $e_2 < 0$, choose $q = \lfloor -e_2 \log_{10} 5 \rfloor$. It holds that $\lfloor c/10^q \rfloor \leq 10 \cdot w$.*

Proof. We only consider the case $e_2 \geq 0$, and omit the almost identical proof for $e_2 < 0$ for brevity. Starting with q, we rearrange:

$$\begin{aligned} & q = \lfloor e_2 \log_{10} 2 \rfloor \\ \Rightarrow\ & q \leq e_2 \log_{10} 2 \\ \Rightarrow\ & 10^q \leq 2^{e_2} \\ \Rightarrow\ & 2 \cdot 10^q \leq 2 \cdot 2^{e_2} \\ \Rightarrow\ & 2 \cdot 10^q < 2 \cdot 2^{e_2} + 2^{e_2} + u \cdot 2^{e_2} - u \cdot 2^{e_2} \\ \Rightarrow\ & 2 < (u+3) \cdot 2^{e_2}/10^q - u \cdot 2^{e_2}/10^q \end{aligned}$$

By construction in step 2 of the basic conversion routine, we know that $u + 3 \leq w$. We replace and simplify to get

$$2 < w \cdot 2^{e_2}/10^q - u \cdot 2^{e_2}/10^q \quad \Rightarrow \quad 2 < c/10^q - a/10^q$$

On the right-hand side, we introduce the floor function into both terms. We need to add 1 to the first term to maintain the inequality; the second term is negative, so making it slightly smaller does not affect the inequality. We simplify to get

$$\begin{aligned} & 2 < \lfloor c/10^q \rfloor + 1 - \lfloor a/10^q \rfloor \\ \Rightarrow\ & \lfloor a/10^q \rfloor < \lfloor c/10^q \rfloor - 1 \end{aligned}$$

as desired.

We now show that $\lfloor c/10^q \rfloor$ is less than or equal to $10 \cdot w$. Starting with $\lfloor c/10^q \rfloor = \left\lfloor w \cdot 2^{e_2}/10^{\lfloor e_2 \log_{10} 2 \rfloor} \right\rfloor$, we remove the inner floor function by lowering its argument to $e_2 \log_{10} 2 - 1$, which can only increase the right-hand side due to the division, and simplify to get

$$\begin{aligned} & \lfloor c/10^q \rfloor \leq \left\lfloor w \cdot 2^{e_2}/10^{e_2 \log_{10} 2 - 1} \right\rfloor \\ \Rightarrow\ & \lfloor c/10^q \rfloor \leq \left\lfloor w \cdot 2^{e_2}/(2^{e_2} \cdot 10^{-1}) \right\rfloor \\ \Rightarrow\ & \lfloor c/10^q \rfloor \leq 10 \cdot w \end{aligned}$$

as desired. □

3.2 Computing $\lfloor (a, b, c)/10^q \rfloor$

Starting with (u, v, w), computing $\lfloor (a, b, c)/10^q \rfloor$ is equivalent to computing $\lfloor (u, v, w) \cdot 2^{e_2}/10^q \rfloor$ or $\lfloor (u, v, w) \cdot 5^{-e_2}/10^q \rfloor$ depending on the sign of e_2. From Section 3.1, we also know that $q \leq e_2$, so this simplifies to $\lfloor (u, v, w) \cdot 2^{e_2-q}/5^q \rfloor$ or $\lfloor (u, v, w) \cdot 5^{-e_2-q}/2^q \rfloor$, respectively.

For performance, we turn the division by 5^q into a multiplication by $1/5^q$. This is similar to the previous work by Granlund and Montgomery [8] on compiler optimization to turn an $n \times n$-bit division by a constant into an $n \times n$-bit multiplication.

While (u, v, w) are n-bit numbers, 5^q and 5^{-e_2-q} are significantly larger. However, we only need the top-most n bits of the result, on which the lower bits of 5^q and 5^{-e_2-q} are unlikely to have any impact.

Therefore, we multiply by either $\lfloor 5^{-e_2-q}/2^k \rfloor$ or $(\lfloor 2^k/5^q \rfloor + 1)$, and show that there are non-trivial lower / upper bounds for k at least for the standard IEEE floating point types that reduce the number of bits required for the multiplier to approximately $2n$. Section 3.2.1 derives a non-trivial lower bound for k for the case $e_2 \geq 0$, and Section 3.2.2 derives a non-trivial upper bound for k for the case $e_2 < 0$. Note that the derived bounds depend on the value of e_2.

In both cases, the bounds depend on the minimum or maximum of a product modulo some power. Section 3.2.3 provides a modified Euclid's algorithm to efficiently compute such minima and maxima.

Section 3.2.4 shows that the number of bits needed of $\lfloor 5^{-e_2-q}/2^k \rfloor$ and $(\lfloor 2^k/5^q \rfloor + 1)$ is bound by relatively small constants at least for all IEEE floating point types up to 256 bits. This allows us to forgo storing individual values of k for each value of e_2, and instead compute appropriate values for k from the corresponding floating-point-type-specific constants.

3.2.1 Case $e_2 \geq 0$

Lemma 3.3. *Given a specific floating point type, let $(a, b, c) = (u, v, w) \cdot 2^{e_2}$ as before with $e_2 \geq 0$, and $q = \lfloor e_2 \log_{10} 2 \rfloor$. Then for all*

$$k > \log_2 \frac{max(w) \cdot 2^{e_2-q} 5^q}{5^q - max((w \cdot 2^{e_2-q}) \% 5^q)}$$

with max taken across all possible values of w for the given floating point type, it holds that

$$\lfloor (a, b, c)/10^q \rfloor = \lfloor (u, v, w) \cdot 2^{e_2-q-k} \cdot (\lfloor 2^k/5^q \rfloor + 1) \rfloor.$$

Proof. We only show this for c for brevity. Let δ be the distance between the untruncated right-hand side and the desired result, i.e.,

$$\begin{aligned} &\delta = w \cdot 2^{e_2-q-k} \cdot (\lfloor 2^k/5^q \rfloor + 1) - \lfloor c/10^q \rfloor \\ \Leftrightarrow\ &\delta = w \cdot 2^{e_2-q-k} \cdot (\lfloor 2^k/5^q \rfloor + 1) - \lfloor w \cdot 2^{e_2-q}/5^q \rfloor. \end{aligned}$$

If δ is between 0 and 1, then the desired equality holds. We first show that δ is greater than zero. It holds that $\lfloor 2^k/5^q \rfloor + 1$ is greater than $2^k/5^q$. We multiply both sides by $w \cdot 2^{e_2-q-k}$ and simplify to obtain

$$\begin{aligned} &w \cdot 2^{e_2-q-k} \cdot (\lfloor 2^k/5^q \rfloor + 1) > w \cdot 2^{e_2-q-k} 2^k/5^q \\ \Rightarrow\ &w \cdot 2^{e_2-q-k} \cdot (\lfloor 2^k/5^q \rfloor + 1) > \lfloor w \cdot 2^{e_2-q}/5^q \rfloor \\ \Rightarrow\ &w \cdot 2^{e_2-q-k} \cdot (\lfloor 2^k/5^q \rfloor + 1) - \lfloor w \cdot 2^{e_2-q}/5^q \rfloor > 0 \\ &\Rightarrow \delta > 0 \end{aligned}$$

as desired.

We therefore only need to show that δ is less than one.

$$\begin{aligned} &k > \log_2 \frac{\max(w) \cdot 2^{e_2-q} \cdot 5^q}{5^q - \max((w \cdot 2^{e_2-q}) \% 5^q)} \\ \Leftrightarrow\ &2^k > \frac{\max(w) \cdot 2^{e_2-q} \cdot 5^q}{5^q - \max((w \cdot 2^{e_2-q}) \% 5^q)} \\ \Leftrightarrow\ &\max(w) \cdot 2^{e_2-q} 5^q \\ &\qquad + 2^k \cdot \max((w \cdot 2^{e_2-q}) \% 5^q) < 5^q 2^k \end{aligned}$$

We erase both maximum functions, which can only decrease the left-hand side, divide by 2^k, and rearrange to obtain

$$\begin{aligned} &w \cdot 2^{e_2-q} \cdot 5^q/2^k + ((w \cdot 2^{e_2-q}) \% 5^q) < 5^q \\ \Rightarrow\ &w \cdot 2^{e_2-q} \cdot 5^q/2^k + w \cdot 2^{e_2-q} - w \cdot 2^{e_2-q} \\ &\qquad + ((w \cdot 2^{e_2-q}) \% 5^q) < 5^q \\ &\Rightarrow w \cdot 2^{e_2-q} \cdot (1 + 5^q/2^k)/5^q \\ &-(w \cdot 2^{e_2-q} - ((w \cdot 2^{e_2-q}) \% 5^q))/5^q < 1. \end{aligned}$$

We use the identity $(x - x\%y)/y = \lfloor x/y \rfloor$ to replace the modulo function and obtain:

$$w \cdot 2^{e_2-q} \cdot (1 + 5^q/2^k)/5^q - \lfloor w \cdot 2^{e_2-q}/5^q \rfloor < 1.$$

Furthermore, considering the expression $(1 + 5^q/2^k)/5^q$, we observe that $5^q/2^k \cdot \lfloor 2^k/5^q \rfloor$ is less than or equal to one and get

$$\begin{aligned} (1 + 5^q/2^k)/5^q &\geq (5^q/2^k \cdot \lfloor 2^k/5^q \rfloor + 5^q/2^k)/5^q \\ &= 2^{-k} \cdot (\lfloor 2^k/5^q \rfloor + 1). \end{aligned}$$

We substitute and simplify to obtain

$$\begin{aligned} &w \cdot 2^{e_2-q-k} \cdot (\lfloor 2^k/5^q \rfloor + 1) - \lfloor w \cdot 2^{e_2-q}/5^q \rfloor < 1 \\ &\Rightarrow \delta < 1 \end{aligned}$$

as desired. □

3.2.2 Case $e_2 < 0$

Lemma 3.4. *Given a specific floating point type, let $(a, b, c) = (u, v, w) \cdot 5^{-e_2}$ as before with $e_2 < 0$, and $q = \lfloor -e_2 \log_5 2 \rfloor$. Then for all*

$$0 \leq k \leq \log_2 \frac{min\left((w \cdot 5^{-e_2-q}) \% 2^q\right)}{max(w)}$$

with min and max taken across all possible values of w for the given floating point type, it holds that

$$\lfloor (a, b, c)/10^q \rfloor = \lfloor (u, v, w) \cdot \lfloor 5^{-e_2-q}/2^k \rfloor / 2^{q-k} \rfloor.$$

Proof. We only show this for c for brevity. Let δ be the distance between the untruncated right-hand side and the desired result, i.e.,

$$\begin{aligned} &\delta = w \cdot \lfloor 5^{-e_2-q}/2^k \rfloor / 2^{q-k} - \lfloor c/10^q \rfloor \\ \Leftrightarrow\ &\delta = w \cdot \lfloor 5^{-e_2-q}/2^k \rfloor / 2^{q-k} - \lfloor w \cdot 5^{-e_2-q}/2^q \rfloor \end{aligned}$$

If δ is greater than or equal to zero and less than one, then the desired equality holds. We first show that δ is less

than one by contradiction. Assume that δ is greater than one, which implies

$$\begin{aligned} & w \cdot \lfloor 5^{-e_2-q}/2^k \rfloor / 2^{q-k} - \lfloor w \cdot 5^{-e_2-q}/2^q \rfloor \geq 1 \\ \Rightarrow\ & w \cdot \lfloor 5^{-e_2-q}/2^k \rfloor / 2^{q-k} - 1 \geq \lfloor w \cdot 5^{-e_2-q}/2^q \rfloor \\ \Rightarrow\ & \lfloor w \cdot \lfloor 5^{-e_2-q}/2^k \rfloor / 2^{q-k} \rfloor > \lfloor w \cdot 5^{-e_2-q}/2^q \rfloor \end{aligned}$$

However, $\lfloor 5^{-e_2-q}/2^k \rfloor$ is smaller than or equal to $5^{-e_2-q}/2^k$. We multiply both sides by $w \cdot 2^k/2^q$ and take the floor function on both sides to obtain

$$\begin{aligned} & w \cdot 2^k/2^q \cdot \lfloor 5^{-e_2-q}/2^k \rfloor \leq w \cdot 2^k/2^q \cdot 5^{-e_2-q}/2^k \\ \Rightarrow\ & w/2^{q-k} \cdot \lfloor 5^{-e_2-q}/2^k \rfloor \leq w/2^q \cdot 5^{-e_2-q} \\ \Rightarrow\ & \lfloor w \cdot \lfloor 5^{-e_2-q}/2^k \rfloor / 2^{q-k} \rfloor \leq \lfloor w \cdot 5^{-e_2-q}/2^q \rfloor \end{aligned}$$

which is a contradiction.

We therefore only need to show that δ is greater than or equal to zero.

$$\begin{aligned} & k \leq \log_2 \frac{\min((w \cdot 5^{-e_2-q}) \% 2^q)}{\max(w)} \\ \Rightarrow\ & \max(w) \cdot 2^k \leq \min((w \cdot 5^{-e_2-q}) \% 2^q) \end{aligned}$$

We decrease the left-hand side by erasing the maximum function and replacing 2^k by $5^{-e_2-q} \% 2^k$. On the right-hand side, we also erase the minimum function, which can only increase the value.

$$\begin{aligned} & w \cdot 2^k \leq (w \cdot 5^{-e_2-q}) \% 2^q \\ \Rightarrow\ & w \cdot (5^{-e_2-q} \% 2^k) \leq (w \cdot 5^{-e_2-q}) \% 2^q \\ \Rightarrow\ & 0 \leq ((w \cdot 5^{-e_2-q}) \% 2^q) - w \cdot (5^{-e_2-q} \% 2^k) \\ & \qquad + w \cdot 5^{-e_2-q} - w \cdot 5^{-e_2-q} \\ \Rightarrow\ & 0 \leq w \cdot (5^{-e_2-q} - (5^{-e_2-q} \% 2^k)) \\ & \qquad - (w \cdot 5^{-e_2-q} - ((w \cdot 5^{-e_2-q}) \% 2^q)) \end{aligned}$$

We use the identity $(x - x\%y) = \lfloor x/y \rfloor \cdot y$ to replace the modulo function, and divide both sides by 2^q to obtain

$$\begin{aligned} & 0 \leq w \cdot \lfloor 5^{-e_2-q}/2^k \rfloor \cdot 2^k - \lfloor w \cdot 5^{-e_2-q}/2^q \rfloor \cdot 2^q \\ \Rightarrow\ & 0 \leq w \cdot \lfloor 5^{-e_2-q}/2^k \rfloor / 2^{q-k} - \lfloor w \cdot 5^{-e_2-q}/2^q \rfloor \end{aligned}$$

as desired. □

3.2.3 Efficiently Computing the Minimum and Maximum of a Modular Product

For smaller floating point types, it is possible to compute the minima and maxima needed for Lemmas 3.3 and 3.4 through exhaustive searching. However, this is no longer feasible for the larger types. In this section, we provide a sublinear algorithm to compute the minimum and maximum of a modular product $ax\%b$ over a given range $0 < x \leq M < b$ with $a < b$, and a and b coprime. We also provide a correctness proof, although we omit some details for brevity.

The algorithm as given here only computes a conservative approximation of the true minimum and maximum, since we may not get an exact match for M, but a slightly larger bound. This turns out not to be a problem in practice — the resulting numbers are still sufficient to support our conversion algorithm.

Given a, b, and M, the following algorithm computes (min, max), which is a conservative approximation of the true minimum and maximum:

```
def minmax_euclid(a, b, M)
    a_0 = a, b_0 = b,  s_0 = 1, t_0 = 0,  u_0 = 0, v_0 = 1
    i = 0
    while True:
        while b_i ≥ a_i: // Loop A
            b_{i+1} = b_i − a_i, u_{i+1} = u_i − s_i, v_{i+1} = v_i − t_i
            a_{i+1} = a_i, s_{i+1} = s_i, t_{i+1} = t_i
            i = i + 1
            if −u_i ≥ M return (a_i, b_i)
        if b_i == 0 return (1, b − 1)
        while a_i ≥ b_i: // Loop B
            a_{i+1} = a_i − b_i, s_{i+1} = s_i − u_i, t_{i+1} = t_i − v_i
            b_{i+1} = b_i, u_{i+1} = u_i, v_{i+1} = v_i
            i = i + 1
            if s_i ≥ M return (a_i, b_i)
        if a_i == 0 return (1, b − 1)
```

The `minmax_euclid` algorithm is an extension of the well-known extended Euclidean algorithm, which computes the greatest common divisor as well as the coefficients x and y of the Bézout identity $ax + by = 1$.

We use a non-standard formulation to simplify our proof. In particular, instead of keeping a single set of variables and alternating between remainders and coefficients, we keep them as separate sequences. In our formulation, the primary loop invariants are $as_i + bt_i = a_i$ and $au_i + bv_i = b_i$. In addition, the sequences s_i and v_i are strictly increasing, and the sequences t_i, u_i, a_i, and b_i are strictly decreasing. These properties directly follow from Knuth's [11] proof of the extended Euclidean algorithm.

As we will show below, each pair $(a_i, -b_i\%b)$ represents the minimum and maximum of $ax\%b$ over the range $0 < x \leq \max(s_i, -u_i)$. `minmax_euclid` returns $(a_i, -b_i\%b)$ as soon as $\max(s_i, -u_i) \geq M$, which is a conservative approximation to the true minimum and maximum. `minmax_euclid` returns $(1, b-1)$ if M is greater than or equal to the multiplicative inverse of a modulo b, i.e., $M > a^{-1}\%b$.

Lemma 3.5. *Algorithm* `minmax_euclid` *determines the minimum and maximum of a modular product. I.e., for all $i > 0$ and for all x with $0 < |x| \leq max(s_i, -u_i)$, it holds that $a_i \leq ax\%b \leq -b_i\%b$.*

Proof. We perform this proof by induction over i, following the structure of the code. For $i = 1$ (base case), we observe $\max(s_1, -u_1) = 1$, $a_1 = a$, and $b_1 = b - a$. It follows that

$$0 < |x| \leq 1 \rightarrow a \leq ax\%b \leq -a\%b.$$

We consider loops A and B separately, and distinguish between the first iteration and subsequent iterations of each loop. For each of these four cases, we assume that the induction statement holds for i (induction assumption), and show that it must also hold for $i + 1$ (induction hypothesis) by contradiction.

We first assume that an integer triplet x, y, p exists such that

$$\begin{gathered} ax + by = p \text{ AND} \\ \max(s_i, -u_i) < x \le \max(s_{i+1}, -u_{i+1}) \text{ AND} \\ (p < a_{i+1} \text{ OR } p > -b_{i+1}\%b). \end{gathered}$$

The existence of such a triplet contradicts the induction hypothesis, but not the induction assumption. Given x, y, p, we then construct a new triplet x', y', p' with $x' < \max(s_i, -u_i)$ and either $p' < a_i$ or $p' > -b_i\%b$. This contradicts the induction assumption. Therefore such a triplet x, y, z cannot exist, and the induction hypothesis holds, completing the proof.

There are a total of sixteen ($2 \times 2 \times 2 \times 2$) cases we need to consider: two cases for x, $x < 0$ or $x > 0$, two cases for p, $p < a_i$ or $p > -b_i\%b$, the two loops A and B, as well as whether it is the first or a subsequent iteration.

In all cases, we use one of the loop invariants $as_j + bt_j = a_j$ or $au_j + bv_j = b_j$ for either $j = i$ or $j = i + 1$, and add or subtract $ax + by = p$ to obtain the x', y', p' triplet, and then show that this triplet contradicts the induction assumption.

Figure 1 provides abbreviated proofs for all sixteen cases split into four sections corresponding to two sections for each of the loops. In each section, the first two rows provide the induction assumption and the induction hypothesis for that section. In particular, these differ in their bounds for $|x|$.

Each section is further split into four cases based on the sign of x, and whether $p < a_{i+1}$ or $p > -b_{i+1}\%b$. The first column specifies the case. The first row in the second column then gives a formula for x'. The first row in the third column gives the matching formula for p' which is fully determined by the formula for x'. The second row in the second column then first shows that x' is within the range of the induction assumption, and the second row in the third column shows that p' is outside the legal range of the induction assumption. □

Loop A, first iteration		
Pre: $-u_i < s_i, 0 < \|x\| \le s_i \to a_i \le ax\%b \le -b_i\%b$ Post: $s_i < \|x\| \le -u_{i+1} \to a_{i+1} \le ax\%b \le -b_{i+1}\%b$		
Case	x'	p'
$x > 0, p < a_i$ $\Rightarrow$	$s_i - x$ $0 < -x' < s_i$	$a_i - p$ $p' < a_i$
$x < 0, p < a_i$ $\Rightarrow$	$-u_i + x$ $0 < -x' < s_i$	$-b_i + p$ $p' > -b_i\%b$
$x > 0, p > -b_{i+1}\%b$ $\Rightarrow$	$-u_i - x$ $0 < -x' < s_i$	$-b_i - p$ $p' > -b_i\%b$
$x < 0, p > -b_{i+1}\%b$ $\Rightarrow$	$s_i + x$ $0 < -x' < s_i$	$a_i + p$ $p' > -b_i\%b$ or $p' < a_i$

Loop A, subsequent iterations		
Pre: $s_i < -u_i, 0 < \|x\| \le -u_i \to a_i \le ax\%b \le -b_i\%b$ Post: $-u_i < \|x\| \le -u_{i+1} \to a_{i+1} \le ax\%b \le -b_{i+1}\%b$		
Case	x'	p'
$x > 0, p < a_i$ $\Rightarrow$	$s_i - x$ $0 < -x' < -u_i$	$a_i - p$ $p' < a_i$
$x < 0, p < a_i$ $\Rightarrow$	$-u_i + x$ $0 < -x' < -u_i$	$-b_i + p$ $p' > -b_i\%b$
$x > 0, p > -b_{i+1}\%b$ $\Rightarrow$	$-u_{i+1} - x$ $-u_i > -x' > 0$	$-b_{i+1} - p$ $p' > -b_i\%b$
$x < 0, p > -b_{i+1}\%b$ $\Rightarrow$	$s_i + x$ $0 < -x' < -u_i$	$a_i + p$ $p' > -b_i\%b$ or $p' < a_i$

Loop B, first iteration		
Pre: $s_i < -u_i, 0 < \|x\| \le -u_i \to a_i \le ax\%b \le -b_i\%b$ Post: $-u_i < \|x\| \le s_{i+1} \to a_{i+1} \le ax\%b \le -b_{i+1}\%b$		
Case	x'	p'
$x > 0, p < a_{i+1}$ $\Rightarrow$	$s_{i+1} - x$ $-u_i > -x' > 0$	$a_{i+1} - p$ $p' < a_{i+1}$
$x < 0, p < a_{i+1}$ $\Rightarrow$	$-u_i + x$ $0 < -x' < -u_i$	$-b_i + p$ $p' > -b_i\%b$ or $p' < a_i$
$x > 0, p > -b_i\%b$ $\Rightarrow$	$-u_i - x$ $0 < -x' < -u_i$	$-b_i - p$ $p' > -b_i\%b$
$x < 0, p > -b_i\%b$ $\Rightarrow$	$s_{i+1} + x$ $-u_i > x' > 0$	$a_{i+1} + p$ $p' > -b_i\%b$ or $p' < a_{i+1}$

Loop B, subsequent iterations		
Pre: $-u_i < s_i, 0 < \|x\| \le s_i \to a_i \le ax\%b \le -b_i\%b$ Post: $s_i < \|x\| \le s_{i+1} \to a_{i+1} \le ax\%b \le -b_{i+1}\%b$		
Case	x'	p'
$x > 0, p < a_{i+1}$ $\Rightarrow$	$s_i - x$ $0 < -x' < s_i$	$a_i - p$ $p' < a_i$
$x < 0, p < a_{i+1}$ $\Rightarrow$	$-u_i + x$ $0 < -x' < s_i$	$-b_i + p$ $p' > -b_i\%b$ or $p' < a_i$
$x > 0, p > -b_i\%b$ $\Rightarrow$	$-u_i - x$ $0 < -x' < s_i$	$-b_i - p$ $p' > -b_i\%b$
$x < 0, p > -b_i\%b$ $\Rightarrow$	$s_i + x$ $0 < -x' < s_i$	$a_i + p$ $p' > -b_i\%b$ or $p' < a_i$

Figure 1. Contradictions for all sixteen cases.

3.2.4 Storing the Necessary Multipliers

Ryū uses two lookup tables to store the necessary multipliers ($\lfloor 2^k/5^q \rfloor + 1$) and $\lfloor 5^{-e_2-q}/2^k \rfloor$ for all possible values of q. As it turns out, these multipliers always fit into $2n$-bit integers for a given IEEE n-bit floating point type.

As an example, Figures 2 and 3 show the lower and upper bounds of k for the 64-bit IEEE floating point type for the cases $e_2 \geq 0$ and $e_2 < 0$, respectively, as well as the numbers of bits needed to store ($\lfloor 2^k/5^q \rfloor + 1$) and $\lfloor 5^{-e_2-q}/2^k \rfloor$: the minimum / maximum for k goes up linearly, but the minimum number of bits to store remains flat.

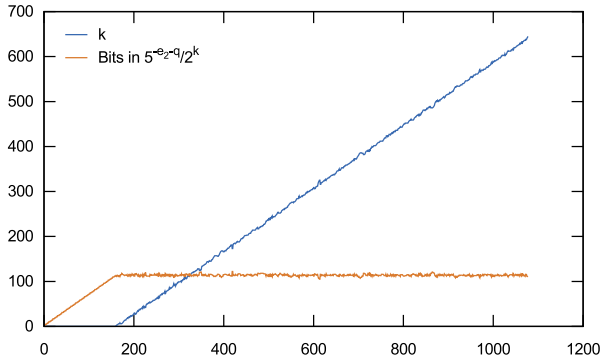


Figure 2. The lower bound for k over the range $2 \leq e_2 \leq 969$ and $\lceil k - \log_2 5^q \rceil$, which is the corresponding minimum number of bits to store $\lfloor 2^k/5^q \rfloor$, as a function of e_2 for the 64-bit IEEE floating point type.

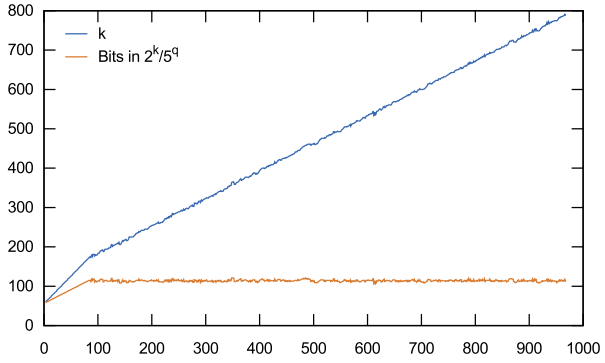


Figure 3. The upper bound for k over the range $2 \leq -e_2 \leq 1076$ and $\lceil \log_2 5^{-e_2-q} \rceil - k$, which is the corresponding minimum number of bits to store $\lfloor 5^{-e_2-q}/2^k \rfloor$, as a function of $-e_2$ for the 64-bit IEEE floating point type, and $e_2 < 0$.

The minimum number of bits is flat for all the IEEE floating point types up to 256 bit. Instead of storing a separate value for k for each value of e_2, we take the maximum number of bits to store across each range, and compute k from that. Figure 4 shows the number of bits stored for all IEEE floating point types, given as B_0 and B_1 here, as well as the combined number of entries in the two lookup tables, which together indicate the memory requirements for Ryū. Note that this table accounts for the adjustment to q in order to output correctly-rounded strings (see Section 2.4).

Type	B_0	B_1	#Entries	Total Memory
Float16	15	21	11	44 Byte
Float32	60	63	78	624 Byte
Float64	124	124	617	9,872 Byte
Float128	249	246	9,865	315,680 Byte
Float256	501	501	157,828	10,100,992 Byte

Figure 4. For each of the IEEE floating point types, this table provides the required number of bits to store ($\lfloor 2^k/5^q \rfloor + 1$) ($B_0$) and $\lfloor 5^{-e_2-q}/2^k \rfloor$ (B_1), the total size of the corresponding lookup tables, and the total memory required.

3.3 Trailing Zeros of (a, b, c)

We now solve the third subproblem of Lemma 3.1 — determining whether all the removed digits from a, b, and c are zeros. To this end, we introduce a predicate $Z(\{u, v, w\}, e_2, q)$, which is true if and only if the corresponding formula $\{a, b, c\}\%10^q == 0$ is true.

Lemma 3.6. *Given (u, v, w), e_2, and q as before:*

$$Z(\{u, v, w\}, e_2, q) = \begin{cases} \{u, v, w\}\%5^q == 0, & \text{if } e_2 \geq 0 \\ \{u, v, w\}\%2^q == 0, & \text{if } e_2 < 0 \end{cases}$$

Proof. We only discuss u here for brevity. Let $p_i(x)$ be the largest power of i that divides x. It holds that $p_{10}(x) = \min(p_2(x), p_5(x))$.

For the case $e_2 \geq 0$:

$$\begin{aligned} Z(u, e_2, q) &= (u \cdot 2^{e_2})\%10^q == 0 \\ &= p_{10}(u \cdot 2^{e_2}) \geq q \\ &= \min(p_2(u) + p_2(2^{e_2}), p_5(u)) \geq q \\ &= p_2(u) + e_2 \geq q \text{ AND } p_5(u) \geq q. \end{aligned}$$

Due to $e_2 \geq q$ (Section 3.1), and $p_2(u) \geq 0$, the first term is always true, so this simplifies to $p_5(u) \geq q$, or equivalently $u\%5^q == 0$.

For the case $e_2 < 0$, it holds that

$$\begin{aligned} Z(u, e_2, q) &= (u \cdot 5^{-e_2})\%10^q == 0 \\ &= \min(p_2(u), p_5(u) - e_2) \geq q \\ &= p_2(u) \geq q \text{ AND } p_5(u) - e_2 \geq q \\ &= p_2(u) \geq q \\ &= u\%2^q == 0. \end{aligned}$$

□

Note that the number of trailing zeros is always limited by the magnitude of (u, v, w). That means that the assumed rounding mode for the correct parser only affects the output for a tiny fraction of all possible floating point numbers.

3.4 Putting It All Together

Taken together, Sections 3.1 to 3.3 allow us to speed up the basic conversion routine. We add a preprocessing step to precompute and store $(\lfloor 2^k/5^q \rfloor + 1)$ and $\lfloor 5^{-e_2-q}/2^k \rfloor$ in lookup tables for every possible value of e_2.

Step 0. Precompute lookup tables for the given floating point type.

Given a specific floating point type, we determine the range of e_2. We also determine appropriate constants B_0 and B_1 of how many bits of $(\lfloor 2^k/5^q \rfloor + 1)$ and $\lfloor 5^{-e_2-q}/2^k \rfloor$ need to be stored.

For each possible value of e_2 that is greater than or equal to zero, we determine q as $\max\left(0, \lfloor e_2 \log_{10} 2 \rfloor - 1\right)$ (Lemma 3.2), and then use B_0 to determine a legal value for k as $(B_0 + \lfloor \log_2 5^q \rfloor)$ (Lemma 3.3). We compute $(\lfloor 2^k/5^q \rfloor + 1)$ using arbitrary precision arithmetic — its result has no more than B_0 bits — and store it in a lookup table TABLE_GTE indexed by q.

For each possible value of e_2 that is less than zero, we determine q as $\max\left(0, \lfloor -e_2 \log_5 2 \rfloor - 1\right)$ (Lemma 3.2), and then use B_1 to determine a legal value for k as $(\lceil \log_2 5^q \rceil - B_1)$ (Lemma 3.4). We compute $\lfloor 5^{-e_2-q}/2^k \rfloor$ using arbitrary precision arithmetic — this result has no more than B_1 bits — and store it in a lookup table TABLE_LT indexed by $-e_2 - q$.

Using these lookup tables, we can now replace steps 3 and 4 with the more efficient steps 3' and 4'.

Step 3'. Convert to a decimal power base and simultaneously remove most digits.

Like before, we compute q and use one of the constants B_0 and B_1 to determine a legal value for k. Then, we look up the correct factor from the corresponding lookup table. Finally, we determine whether all removed digits were zeros (Lemma 3.6).

Case $e_2 \geq 0$:

$$\begin{aligned} q &= \max\left(0, \lfloor e_2 \log_{10} 2 \rfloor - 1\right) \\ k &= B_0 + \lfloor \log_2 5^q \rfloor \\ (a_q, b_q, c_q) &= \lfloor ((u, v, w) \cdot \text{TABLE\_GTE}[q]) / 2^{-e_2+q+k} \rfloor \\ (z_a, z_b, z_c) &= (u\%5^q == 0, v\%5^{q-1} == 0, w\%5^q == 0) \end{aligned}$$

Depending on the underlying machine and floating point type, the multiplication and shift operations may not fit into a native integer type, but can always be decomposed into a small, constant number of multiplications, additions, and shift operations.

Case $e_2 < 0$:

$$\begin{aligned} q &= \max\left(0, \lfloor -e_2 \log_5 2 \rfloor - 1\right) \\ k &= \lceil \log_2 5^q \rceil - B_1 \\ (a_q, b_q, c_q) &= \lfloor ((u, v, w) \cdot \text{TABLE\_LT}[-e_2 - q]) / 2^{q-k} \rfloor \\ (z_a, z_b, z_c) &= (u\%2^q == 0, v\%2^{q-1} == 0, w\%2^q == 0) \end{aligned}$$

Step 4'. Find the shortest, correctly-rounded decimal representation in the interval.

We use the results from the previous step to run the `compute_shortest` algorithm from Section 2.3 to completion. We first determine whether the smaller and larger halfway points may be output, resulting in the two flags *accept_smaller* and *accept_larger*. For example, for the round-even rounding mode, we use

$$accept\_smaller = accept\_larger = (m\%2 == 0).$$

We then compute

$$\begin{aligned} (d_o, e_o) = (0, q) + \texttt{compute\_shortest}(& \\ & a_q, b_q, c_q, \\ & accept\_smaller \text{ AND } z_a, \\ & accept\_larger \text{ OR NOT } z_c). \end{aligned}$$

We also need to pass in z_b to be used as *all_b_zero* and perform tiebreaking, which require changing the signature of *compute_shortest*, and are omitted here for brevity.

Theorem 3.7. *The given algorithm terminates, and generates correct output.*

Proof. Follows directly from Lemmas 2.1–3.6. □

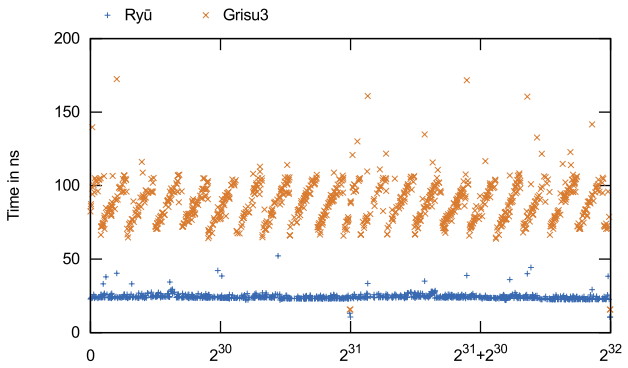
4 Performance Evaluation

We implemented four versions of Ryū, covering 32- and 64-bit floating point numbers each with a C and a Java implementation. We then experimentally compared our C implementations against what we believe to be an implementation of Grisu3 [6] that is publicly available on GitHub. We also compared our Java implementation against the implementations by Jaffer [10] and the OpenJDK. Our implementation performs correct rounding as described in Section 2.4; leaving out those modifications further improves the performance.

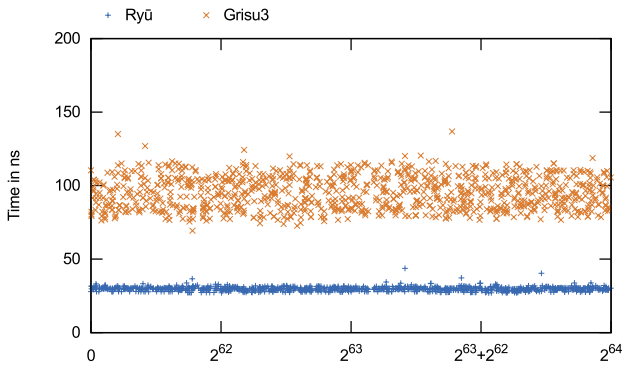
We generated a sequence of 32- and 64-bit values using the Mersenne Twister random number generator with a fixed initial seed and interpreted those as floating point numbers. We then ran each algorithm for 1,000 iterations, measured the time, divided by 1,000, and removed outliers. We performed all experiments on an Intel(R) Core(TM) i7-4770K CPU at 3.50GHz running Ubuntu Linux 17.10. We used clang 3.9 with the `-O2` flag for C compilation and tested Java performance with OpenJDK 1.8.0_131.

Figure 5 shows our experimental results. All subfigures exhibit translational symmetry because the sign bit is the highest bit in the binary floating point representation and the algorithms all treat negative and positive numbers alike.

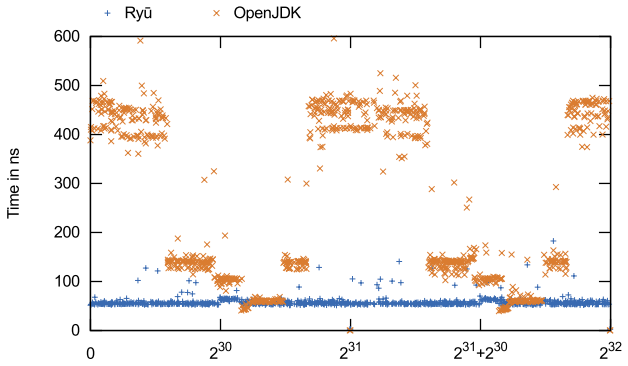
Grisu3 has several outliers, not shown in the graphs, which correspond to cases where it has to fall back to a slower arbitrary-precision implementation. The C implementation of Ryū has better performance than Grisu3 and is also less volatile. On average, Ryū is roughly three times faster than Grisu3 for 32-bit and 64-bit floating point numbers.



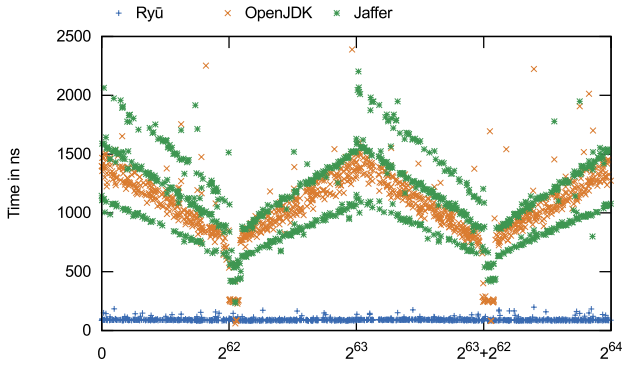
(a) 32-bit C implementation, Ryū vs. Grisu3



(b) 64-bit C implementation, Ryū vs. Grisu3



(c) 32-bit Java implementation, Ryū vs. OpenJDK



(d) 64-bit Java implementation, Ryū vs. Jaffer vs. OpenJDK

Figure 5. Figures (a) and (b) show the experimental performance results for our C implementation of Ryū compared to (a) Grisu3 on 32-bit floating point numbers, (b) Grisu3 on 64-bit floating point numbers. Figures (c) and (d) show the results for our Java implementation compared to (c) OpenJDK on 32-bit floating point numbers, and (d) OpenJDK and Jaffer's algorithm on 64-bit floating point numbers. Each point plots the time in nanoseconds of a single value over the value of the binary representation of a randomly generated floating point number. Some points are outside the plotted range.

The Java implementation of Ryū is slightly slower than the C implementation. For the C implementation, we were careful to avoid memory allocation in the inner loop. The Java implementations all allocate memory for the result and then call the `java.lang.String` constructor, which defensively performs a copy of the data to guarantee immutability. These two operations are likely responsible for the performance difference and higher volatility of the Java implementation.

The 32-bit implementation in OpenJDK uses an extensive case distinction and fixed-precision specializations for each case, which causes the stair shape in Figure 5(c). Both OpenJDK and Jaffer use an arbitrary precision library to perform computations involving numbers that grow with the floating point exponent of the input, which causes the butterfly shape in Figure 5(d).

5 Related Work

The first radix conversion algorithms were described hundreds of years ago (see Knuth [12] for a summary). Samelson and Bauer [14] may have been the first to discuss floating point conversion.

More recently, Coonen [4, 5] provided a very brief description of an algorithm for binary to decimal conversion. It does not attempt to return the shortest possible string, and it seems to require extended floating point types, i.e., floating point types with higher precision than the value to be converted. Coonen also discusses strategies to reduce the size of the lookup tables of powers of ten needed for his algorithm.

Steele and White [15] describe an algorithm to generate digits from left to right. Given a floating point number f, find the smallest integer k such that $f \leq 10^k$. Scale f by $1/10^k$ to obtain a number between 0 and 1. Repeatedly multiply the number by 10, print the integer part of the value, and keep the fractional part with each iteration printing one decimal digit. They use an iteratively updated error bound to determine when to stop printing. In the original form, this algorithm requires arbitrary precision operations and does

not handle all possible rounding modes. Some consider this to be a generalization of Taranto's algorithm [16].

Both Gay [7] and Burger and Dybvig [3] subsequently described improved versions of this algorithm, still using arbitrary precision arithmetic. Both describe faster algorithms to estimate the smallest power of 10 that is larger than the given number. Gay also found special cases in which it is sufficient to use floating point arithmetic. Burger and Dybvig introduced the concept of the smaller and larger halfway points, which they use to guarantee the shortest output while handling all possible rounding modes.

Jaffer [10] describes a simple algorithm and provides a Java implementation. It uses arbitrary precision arithmetic to compute the exact first few decimal digits of the output and uses an iterative approach to determine the shortest output by repeatedly converting the potential output back to a binary floating point number and checking whether the result is identical to the original floating point value.

We compared Jaffer's algorithm with our Java implementation of Ryū on a sample of 1,000,000 values obtained from an implementation of the Mersenne Twister using a fixed seed. Our Java implementation follows the specification of the Java `Double.toString` method and prints at least two decimal digits, which sometimes leads to longer output than Jaffer's implementation — we ignored these cases. Other than that, its output is shorter in 17 cases, all of which are incorrect. It is longer in 142,163 cases, most of which are due to a bug in the handling of negative floating point values.

Loitsch [13] introduced a family of algorithms that perform the float to string conversion using only integer operations. The first variant, Grisu, always outputs 21 digits and therefore does not maintain the minimum-length output criterion. The third variant, Grisu3, does, but rejects some inputs — according to the author approximately 0.5% of possible inputs — and has to fall back to a slower algorithm that uses arbitrary precision arithmetic.

Grisu3 is similar to Ryū in that it computes the lower and upper bounds to determine a legal range of decimal representations. However, whereas Ryū uses simple integer operations to obtain exact results directly, Grisu3 emulates a higher-precision floating point type, detects if the exact result is too close to the bounds, and falls back to a slower implementation if so.

Andrysco et al. [2] describe Errol, another family of fixed-width algorithms. Like Grisu, Errol emulates higher-precision floating point operations and detects if these emulated operations are insufficiently accurate to give an exact result. Unlike Grisu, Errol uses pairs of 64-bit floating point values rather than integers. It also uses a lookup table as a fallback rather than a slower implementation. Nevertheless, the final variant, Errol3, is slightly slower than Grisu3 in the common case where no fallback is necessary.

The OpenJDK implements a conversion scheme that branches depending on the exponent of the original floating point value to improve performance. However, it is still significantly slower than both the Grisu and Errol families of algorithms.

We have discovered that the OpenJDK implementation sometimes outputs numbers that are too long. We compared the OpenJDK implementation with our Java implementation of Ryū on the same pseudo-random sample of 1,000,000 values as above: its output is longer in 2,877 cases, all of which are unnecessary additional digits.

We did not compare our implementation against the C standard library function `printf`, as its specification does not include the correctness criteria set forth by Steele and White [15], and, accordingly, neither the glibc nor the MacOS implementation does.

6 Summary

We described a new algorithm, Ryū, to convert binary floating point numbers to decimal strings. We proved it correct and provided benchmark results. We also provide an open source implementation [1] for public scrutiny. The primary reason for Ryū's excellent performance is its simplicity: with no special cases and significantly less code than Grisu3 or Errol3, Ryū is easier to optimize for both humans and compilers. For example, Ryū generates the final digits to print as a single integer, and then uses the fastest existing integer to string conversion code, whereas Grisu3 generates digits one by one into an intermediate buffer and then copies them into the final output buffer as a separate step.

The main disadvantage of Ryū is its reliance on lookup tables. These are quite reasonable for the typical 32-bit and 64-bit floating point types but grow exponentially for larger types. It is up to future work to determine whether it is possible to reduce their size or even avoid them entirely.

Using Ryū's predecessor, our basic conversion routine, we have discovered that some of the competing implementations produce output that is either too long or too short. Any conversion routine that claims consistency with the criteria set forth by Steele and White should compare outputs with our basic conversion routine for at least a subset of numbers. Given an existing arbitrary-precision arithmetic library, its implementation is less than hundred lines of straightforward Java code, which is much easier to inspect for errors than the hundreds if not thousands of lines of high-performance conversion code.

Finally, even though binary float to decimal string conversion is the most common case, it would be interesting to see if Ryū's ideas also apply to other floating point conversion problems.

Acknowledgments

I am indebted to my wife Sara Adams for affording me time and space for this work, and for reviewing dozens of drafts. Thanks also to Christian Rosenke and Tobias Werth.

References

[1] Ulf Adams. 2018. ulfjack/ryu. (Feb. 2018). https://github.com/ulfjack/ryu

[2] Marc Andrysco, Ranjit Jhala, and Sorin Lerner. 2016. 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 '16)*. ACM, New York, NY, USA, 555–567. https://doi.org/10.1145/2837614.2837654

[3] Robert G. Burger and R. Kent Dybvig. 1996. Printing Floating-point Numbers Quickly and Accurately. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. ACM, New York, NY, USA, 108–116. https://doi.org/10.1145/231379.231397

[4] Jerome Toby Coonen. 1980. An Implementation Guide to a Proposed Standard for Floating Point Arithmetic. *Computer* 13, 1 (Jan. 1980), 68–79. https://doi.org/10.1109/MC.1980.1653344 See errata in [5].

[5] Jerome Toby Coonen. 1981. Errata: An Implementation Guide to a Proposed Standard for Floating Point Arithmetic. *Computer* 14, 3 (March 1981), 62. https://doi.org/10.1109/C-M.1981.220378 See also [4].

[6] Florian Loitsch et al. 2017. google/double-conversion. (September 2017). https://github.com/google/double-conversion commit fe9b384793c4e79bd32133dc9053f27b75a5eeae.

[7] David M. Gay. 1990. *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions*. Technical Report. AT&T Bell Laboratories. Numerical Analysis Manuscript 90-10.

[8] Torbjörn Granlund and Peter L. Montgomery. 1994. Division by Invariant Integers Using Multiplication. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/773473.178249

[9] IEEE Computer Society. Microprocessor Standards Committee and Institute of Electrical and Electronics Engineers and IEEE-SA Standards Board. 2008. *754-2008 - IEEE Standard for Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers (IEEE), New York. https://doi.org/10.1109/IEEESTD.2008.4610935

[10] Aubrey Jaffer. 2013. Easy Accurate Reading and Writing of Floating-Point Numbers. (October 2013). https://arxiv.org/abs/1310.8121v6 Updated January 2015.

[11] Donald Ervin Knuth. 1997. *The Art of Computer Programming* (3rd ed.). Vol. I: Fundamental Algorithms. Addison-Wesley, Boston, Massachusetts, United States, Chapter 1.2.1 Mathematical Induction, p. 13–17.

[12] Donald Ervin Knuth. 1997. *The Art of Computer Programming* (3rd ed.). Vol. II: Seminumerical Algorithms. Addison-Wesley, Boston, Massachusetts, United States, Chapter 4.4 Radix Conversion, p. 326.

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

[14] Klaus Samelson and Friedrich L. Bauer. 1953. Optimale Rechengenauigkeit bei Rechenanlagen mit gleitendem Komma. *Zeitschrift für angewandte Mathematik und Physik (ZAMP)* 4, 4 (Jul 1953), 312–316. https://doi.org/10.1007/BF02074638

[15] Guy L. Steele, Jr. and Jon L. White. 1990. How to Print Floating-point Numbers Accurately. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 112–126. https://doi.org/10.1145/93542.93559

[16] Donald Taranto. 1959. Binary Conversion, with Fixed Decimal Precision, of a Decimal Fraction. *Commun. ACM* 2, 7 (July 1959), p. 27. https://doi.org/10.1145/368370.368376