Ryū Revisited: Printf Floating Point Conversion

ULF ADAMS, Google, Germany

Ryū Printf is a new algorithm to convert floating-point numbers to decimal strings according to the `printf` `%f`, `%e`, and `%g` formats: `%f` generates 'full' output (integer part of the input, dot, configurable number of digits), `%e` generates scientific output (one leading digit, dot, configurable number of digits, exponent), and `%g` generates the shorter of the two. Ryū Printf is based on the Ryū algorithm, which converts binary floating-point numbers to the shortest equivalent decimal floating-point representation. We provide quantitative evidence that Ryū Printf is between 3.8 and 55 times faster than existing `printf` implementations.

Furthermore, we show that both Ryū and Ryū Printf generalize to arbitrary number bases. This finding implies the existence of a fast algorithm to convert from base-10 to base-2, as long as the maximum precision of the input is known a priori.

CCS Concepts: • **Computing methodologies** → **Representation of mathematical objects**.

Additional Key Words and Phrases: float, string, printf, performance

ACM Reference Format:
Ulf Adams. 2019. Ryū Revisited: Printf Floating Point Conversion. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 169 (October 2019), 23 pages. https://doi.org/10.1145/3360595

1 INTRODUCTION

Converting numbers from one base to another is a common operation in today's software, supported by virtually every language's standard library. This conversion can be performance-critical for languages that extensively use floating-point numbers, such as JavaScript, or when the software outputs files with a lot of floating-point numbers such as JSON or SVG files. With the upcoming `C++17` standard adding more floating-point conversion and formatting routines such as `<charconv>`, a set of simple, highly efficient conversion algorithms becomes increasingly important.

We build extensively on the previously discovered Ryū algorithm [Adams 2018], an exceptionally simple and fast algorithm for the specific use case of generating the shortest possible decimal representation of a binary floating-point number.

In general, the `printf` output of a 64-bit floating-point number can have up to approximately 800 non-zero digits, and existing `printf` implementations support almost arbitrary precision parameters. However, shortest output never requires more than 17 significant digits, and Ryū is fast in part due to this fact. Adjusting Ryū to generate enough digits to handle the worst-case `printf` output would require prohibitively expensive high-precision multiplications. Note also that generating output with Ryū and then padding or cutting to the desired precision for `printf` would round twice, which is generally incorrect.

Furthermore, Ryū can only convert from base-2 to base-10. We are not aware of any current applications for floating-point numbers in number bases other than 10, 2, or small powers of 2. Even the Setun machine [Brusentsov and Ramil Alvarez 2011] developed at Moscow State University in the 1950s, which used ternary integer arithmetic, seems to have used base-10 floating-point

Author's address: Ulf Adams, Google, Germany, ulfjack@google.com.

2475-1421/2019/10-ART169
https://doi.org/10.1145/3360595

numbers. However, the previous work left open whether Ryū could also be used to convert from base-10 to base-2, which is an important use case, e.g., to parse a decimal string into a binary floating-point number, or to convert from IEEE-754 [2008] decimal to binary format.

This paper describes the Ryū Printf algorithm, which generates `printf`-identical output for the `%f`, `%e`, and `%g` formats with arbitrary runtime-provided precision parameters, i.e., `printf("%.<p>f",<f>)`, `printf("%.<p>e",<f>)`, and `printf("%.<p>g",<f>)` for any precision p and floating-point value f. Ryū Printf introduces a novel segmentation mechanism that computes the output segment-by-segment to avoid high-precision multiplications, generalizing Ryū to a much wider range of use cases. Each segment is computed independently of each other segment, which could facilitate even faster implementations on machines that support vector instructions.

Additionally, we show that Ryū and Ryū Printf both generalize to arbitrary source and target bases, which further strengthens the theory behind Ryū, and also increases the range of use cases. In particular, this result implies the existence of a fast algorithm to parse a short decimal string into a binary floating-point number, the inverse operation of Ryū. Note that this does not include the case of parsing arbitrary-length decimal input strings, as the segmentation mechanism is not obviously reversible.

Section 2 briefly recapitulates the relevant parts of the `printf` specification for the `%f` and `%e` conversion formats, and how the precision parameter affects the output. We do not discuss the `%g` format in detail as it is merely a combination format: depending on the provided value, the implementation decides whether to use `%f` or `%e` format.

Section 3 summarizes the Ryū algorithm.

Section 4 generalizes Ryū to arbitrary source and target bases by showing how to adjust each operation in the algorithm. This section includes correctness proofs, with some references to proofs in previous work for brevity. Our contribution is the existence proof of a fast algorithm to convert decimal floating-point to binary floating-point numbers for cases where the maximum precision of the source and the target formats are known a priori.

Section 5 then develops Ryū Printf, introducing a novel segmentation approach, which is key to the performance of Ryū Printf.

Section 6 provides quantitative performance results from our C implementation, which is a drop-in replacement for rendering the `%e` and `%f` formats including runtime-provided precision parameters of the C `printf` family of functions. We show that our implementation is between 3.8 and 55 times faster than existing implementations on Linux (glibc and musl), Mac (Apple libc), and Windows (Microsoft Visual C and msys). Our measurements allow us to draw some conclusions about the underlying implementations.

This section also briefly discusses the size of the necessary lookup tables used by Ryū Printf, and how they compare to the tables used by Ryū.

Section 7 reviews the existing literature, and Section 8 summarizes our results.

2 PRINTF %F, %E, AND %G CONVERSION

The `printf` function has been a staple of the C programming language for decades and has also found its way into many other programming languages such as Go, Haskell, Java, and Python [Anonymous 2018]. It accepts a format string and a variable number of parameters. The format string contains formatting instructions for the parameters; the formatted parameters replace the formatting instructions in the output. Of particular interest here are the `%f`, `%e`, and `%g` formats, which are used to render floating-point numbers as decimal strings.

While the various specifications of `printf` [Free Software Foundation, Inc. 2018; IBM 2018; Microsoft 2016] differ slightly in wording, we have not observed any major differences in output (see Section 6 for details). For brevity, out of all the flags that can be passed to `printf`, we only discuss the precision parameter.

In short, `printf("%.<precision>f",<f>)` emits a minus sign if the floating-point number is negative, followed by the full integer part of the floating-point number, regardless of the magnitude, followed by a dot, followed by the fractional part rounded to `<precision>` digits. If no precision is given, a default value of six is used. The specification does not prescribe a rounding mode but all tested implementations round to even.

By contrast, `printf("%.<precision>e",<f>)` generates output in scientific format: it emits a minus sign if the floating-point number is negative, followed by a single non-zero digit (unless the number is exactly zero), followed by a dot, followed by a fractional part rounded to `<precision>` digits, followed by the character 'e', followed by a signed ('+'/'-') integer exponent in base-10 using at least two digits. Note that previous versions of MSVC generated three-digit exponents by default.

For the `%g` specifier, `printf` picks either `%f` or `%e` formatting, depending on the exponent, and also omits trailing zeros from the result. See Figure 1 for examples.

input	%f	%e	%g
1.23456e-7	0.000000	1.234560e-07	1.23456e-07
1.23456e-4	0.000123	1.234560e-04	0.000123456
0.123456	0.123456	1.234560e-01	0.123456
1.23456e+2	123.456000	1.234560e+02	123.456
1.23456e+5	123456.000000	1.234560e+05	123456
1.23456e+8	123456000.000000	1.234560e+08	1.23456e+08
1.23456e+106	12345...336.000000	1.234560e+106	1.23456e+106

Fig. 1. Examples of `%f`, `%e`, and `%g` `printf` output. In the last row, the `%f` format results in 104 characters of output, almost all of which we omitted for brevity.

The `%e` format usually produces precision+6 characters (negative numbers, 3-digit exponents, and 4-digit exponents require additional characters). The `%f` format produces additional digits to represent the full integer part of the floating-point value, which can require hundreds of characters in the 64-bit case. Note that there is no specified limit for the precision parameter, even though binary floating-point numbers can generally only represent a small number of decimal digits exactly.

3 RYŪ

The original Ryū algorithm [Adams 2018] converts a binary floating-point number to a decimal string in five steps:

(1) Given a floating-point number f, Ryū decodes it, handles Infinity, NaN, and ±0.0, and unifies normalized and subnormal cases into a single representation $f = (-1)^s \cdot m_f \cdot 2^{e_f}$ such that m_f is an unsigned integer and e_f a signed integer. Assuming rounding to the closest number breaking ties to the closest even number as recommended by IEEE-754 [2008], the sign s is only relevant for step (5); we therefore consider this a pair (m_f, e_f).

(2) Ryū computes the halfway points to the next smaller and larger floating-point values of the same type, representing both as pairs of integers as above. If the next smaller and larger floating-point numbers are f^- and f^+, respectively, then the halfway points are $(f + f^-)/2$ and $(f + f^+)/2$. This step is necessary to determine the shortest representation, which is not relevant for Ryū Printf.

(3) Ryū converts all three pairs to a decimal power base. That is, it converts each pair (m_2, e_2) representing a binary floating-point number to a pair (m_{10}, e_{10}) representing a decimal floating-point number that is less than or equal to the binary floating-point number. As a constraint, this

step ensures that the resulting mantissa has sufficiently many digits. This conversion step is also a critical part of Ryū Printf.

The key contribution of the original publication is an efficient low-precision conversion algorithm. Specifically, it showed that:

$$\left\lfloor \frac{m_2 \cdot 2^{e_2}}{10^{e'}} \right\rfloor = \begin{cases} \left\lfloor \frac{m_2}{2^k} \cdot \left\lfloor \frac{2^{k+e_2}}{10^{e'}} + 1 \right\rfloor \right\rfloor, & \text{if } e_2 \geq 0 \\ \left\lfloor \frac{m_2}{2^k} \cdot \left\lfloor \frac{2^{k+e_2}}{10^{e'}} \right\rfloor \right\rfloor, & \text{otherwise} \end{cases}$$

given appropriate values for k and e'. For k, the original publication gave an approximation involving a modified version of Euclid's algorithm called `minmax_euclid` which takes two coprime integers a and b as well as an integer parameter M and determines the min and max of $(ma)\%b$, where % is the modulo operator, for $0 < m < M$.

(4) Ryū removes unnecessary trailing digits using the `compute_shortest` algorithm with the three decimal floating-point numbers, corresponding to the original number f as well as the smaller and larger halfway points, as inputs. To do this, Ryū analyzes the prime factorization by 2 and 5 of the integer mantissae computed in steps (1) and (2) to determine whether to round the result up or down. Ryū Printf also uses prime factorizations of its inputs to determine correct rounding, but does not use the `compute_shortest` algorithm.

(5) Ryū converts the resulting decimal floating-point number to a suitable string representation by determining the individual decimal digits of the mantissa and exponent and laying them out in a string.

4 GENERALIZING RYŪ TO ARBITRARY BASES

This section generalizes Ryū to arbitrary source and target bases. To that end, this section discusses which steps have to be modified and how, and then provides correctness proofs for the modified operations.

Since Ryū Printf reuses several of these operations, the proofs simultaneously contribute to Ryū Printf as well. Applying Ryū Printf for binary to decimal conversion does not require support for arbitrary source and target bases, but does require the generalized approach for correct rounding (Section 4.3), as its requirements are slightly different from Ryū.

Ryū implicitly assumes that the integer number base of the underlying machine and the floating-point number base are identical, i.e., that they are both base-2, whereas the output is base-10. However, generalizing Ryū to arbitrary bases requires distinguishing between all three bases: the source base b_s, the target base b_t, and the underlying machine base b_m. If the machine base is different from the source base, i.e., $b_m \neq b_s$, then we have to perform a base conversion as part of step (1) such that the resulting integers are stored in machine registers in the underlying machine base.

Step (2) of the original Ryū performs integer operations only; it applies them to the machine registers in the machine base.

Step (3) also restricts itself to integer operations to evaluate simple integer functions. The fundamental operation here is replacing a large multiplication with a smaller one, and the validity of this operation requires certain bounds on modular products, which the `minmax_euclid` algorithm [Adams 2018] computes. However, the `minmax_euclid` algorithm requires that its two primary inputs are coprime.

Section 4.1 first extends the `minmax_euclid` algorithm to accept inputs that are not coprime. Section 4.2 then shows that replacing a large multiplication with a smaller one generalizes to arbitrary bases.

The `compute_shortest` algorithm in step (4) works unmodified for arbitrary bases. However, the previous approach for correct rounding is not immediately transferable, especially for odd target bases. Section 4.3 generalizes the underlying theory of prime factorizations to handle all cases.

Finally, step (5) converts the intermediate integers to the target base, which involves repeated integer division by the target base. This works regardless of whether the integers are internally stored in machine base or source base.

4.1 `minmax_euclid`

The `minmax_euclid` algorithm [Adams 2018] accepts three positive integers a, b, and M, with a and b coprime, and computes the minimum and maximum of $(ma)\%b$ over the range $0 < m < M$. We now show that this algorithm can also be used when a and b are not coprime:

Lemma 4.1. *Let a, b, and M be positive integers. Furthermore, let c be the greatest common divisor of a and b. Then the minimum and maximum over the range $0 < m < M$ of $(ma)\%b$ is $c \cdot \texttt{minmax\_euclid}(a/c, b/c, M)$.*

Proof. We have that

$$\begin{aligned}(ma)\%b &= ma - b\left\lfloor \frac{ma}{b} \right\rfloor \\ &= \frac{c}{c} \cdot \left(ma - b\left\lfloor \frac{ma}{b} \cdot \frac{c}{c} \right\rfloor\right) \\ &= c \cdot \left(m \cdot \frac{a}{c} - \frac{b}{c}\left\lfloor \frac{m \cdot \frac{a}{c}}{\frac{b}{c}} \right\rfloor\right) \\ &= c \cdot ((ma/c)\%(b/c)) .\end{aligned}$$

Additionally, a/c and b/c are coprime. We can therefore apply $\texttt{minmax\_euclid}(a/c, b/c, M)$, which returns the minimum and maximum of $(m \cdot (a/c))\%(b/c)$ over the range $0 < m < M$. The lemma immediately follows. □

4.2 Direct Mantissa Conversion

We restate the primary result from Adams [2018] here slightly differently to unify the cases where the exponent is less than and where it is greater than or equal to zero; note that this involves a different interpretation for the bound k, which becomes a lower bound in both cases. We also generalize it to apply to arbitrary source and target bases. The correctness proof follows the same steps and is also restated here for completeness.

Lemma 4.2. *Let b_s and b_t be positive integers greater than one, let e_s and e' be integers, and let M be a positive integer.*

Case 1: $e_s \geq 0$ and $e' \geq 0$. For all $0 < m_s < M$ and

$$k > \log_{b_s} \frac{M \cdot b_t^{e'}}{b_t^{e'} - \max\left((m_s \cdot b_s^{e_s})\%b_t^{e'}\right)}$$

follows that

$$\left\lfloor \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \right\rfloor = \left\lfloor \frac{m_s}{b_s^k} \cdot \left\lfloor \frac{b_s^{k+e_s}}{b_t^{e'}} + 1 \right\rfloor \right\rfloor .$$

Case 2: $e_s < 0$ and $e' < 0$. For all $0 \leq m_s < M$ and

$$k > \log_{b_s} \frac{M}{\min\left((m_s \cdot b_t^{-e'})\% b_s^{-e_s}\right)} - e_s$$

follows that

$$\left\lfloor \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \right\rfloor = \left\lfloor \frac{m_s}{b_s^k} \cdot \left\lfloor \frac{b_s^{k+e_s}}{b_t^{e'}} \right\rfloor \right\rfloor .$$

Proof. Case 1: We first show that the left-hand side is less than or equal to the untruncated right-hand side, regardless of the value of k. We then show that the difference between the untruncated right-hand side and the left-hand side is strictly less than one given the inequality for k. These two conditions are sufficient to prove the original claim.

From the definition of the floor function, we have that $x < \lfloor x \rfloor + 1$, and therefore:

$$\begin{aligned} \frac{b_s^{k+e_s}}{b_t^{e'}} &< \left\lfloor \frac{b_s^{k+e_s}}{b_t^{e'}} \right\rfloor + 1 \\ \Leftrightarrow \frac{m_s \cdot b_s^{k+e_s}}{b_s^k \cdot b_t^{e'}} &< \frac{m_s}{b_s^k} \cdot \left(\left\lfloor \frac{b_s^{k+e_s}}{b_t^{e'}} \right\rfloor + 1 \right) \\ \Rightarrow \left\lfloor \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \right\rfloor &< \frac{m_s}{b_s^k} \cdot \left\lfloor \frac{b_s^{k+e_s}}{b_t^{e'}} + 1 \right\rfloor . \end{aligned}$$

We now show that the difference between the untruncated right-hand side and the left-hand side is strictly less than one, beginning with the inequality for k:

$$\begin{aligned} k &> \log_{b_s} \frac{M \cdot b_t^{e'}}{b_t^{e'} - \max\left((m_s \cdot b_s^{e_s})\% b_t^{e'}\right)} \\ \Rightarrow b_s^k &> \frac{M \cdot b_t^{e'}}{b_t^{e'} - \max\left((m_s \cdot b_s^{e_s})\% b_t^{e'}\right)} \\ \Rightarrow b_s^k \cdot \left(b_t^{e'} - \max\left((m_s \cdot b_s^{e_s})\% b_t^{e'}\right)\right) &> M \cdot b_t^{e'} \\ \Rightarrow b_s^k \cdot b_t^{e'} &> M \cdot b_t^{e'} + b_s^k \cdot \max\left((m_s \cdot b_s^{e_s})\% b_t^{e'}\right) \end{aligned}$$

On the right-hand side, we replace M with m_s and erase the maximum function, both of which can only decrease the right-hand side due to the precondition $0 \leq m_s < M$:

$$\begin{aligned} \Rightarrow b_s^k \cdot b_t^{e'} &> m_s \cdot b_t^{e'} + b_s^k \cdot \left((m_s \cdot b_s^{e_s})\% b_t^{e'}\right) \\ \Rightarrow 1 &> \frac{m_s}{b_s^k} + \frac{(m_s \cdot b_s^{e_s})\% b_t^{e'}}{b_t^{e'}} \end{aligned}$$

We add $\frac{m_s \cdot b_s^{e_s}}{b_t^{e'}}$ on both sides:

$$\begin{aligned} \Rightarrow \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} + 1 &> \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} + \frac{m_s}{b_s^k} + \frac{(m_s \cdot b_s^{e_s})\% b_t^{e'}}{b_t^{e'}} \\ \Rightarrow \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} - \frac{(m_s \cdot b_s^{e_s})\% b_t^{e'}}{b_t^{e'}} + 1 &> \frac{m_s}{b_s^k} \cdot \left(\frac{b_s^{k+e_s}}{b_t^{e'}} + 1 \right) \end{aligned}$$

On the left-hand side, we use the identity $\frac{x - x\%y}{y} = \left\lfloor \frac{x}{y} \right\rfloor$, and we further decrease the right-hand side by introducing the floor function, completing the proof:

$$\Rightarrow \left\lfloor \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \right\rfloor + 1 > \frac{m_s}{b_s^k} \cdot \left\lfloor \frac{b_s^{k+e_s}}{b_t^{e'}} + 1 \right\rfloor .$$

Case 2: We first show that the difference between the untruncated right-hand side and the left-hand side is less than one, regardless of the value of k. We then show that the left-hand side is less than the untruncated right-hand side given the inequality for k. These two conditions are sufficient to prove the original claim.

From the definition of the floor function, we have that $\lfloor x \rfloor + 1 > x$, and therefore:

$$\begin{aligned} \left\lfloor \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \right\rfloor + 1 &> \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \\ \Rightarrow \left\lfloor \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \right\rfloor + 1 &> \frac{m_s \cdot b_s^{k+e_s}}{b_s^k \cdot b_t^{e'}} \\ \Rightarrow \left\lfloor \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \right\rfloor + 1 &> \frac{m_s}{b_s^k} \cdot \left\lfloor \frac{b_s^{k+e_s}}{b_t^{e'}} \right\rfloor \end{aligned}$$

We now show that the left-hand side is less than the untruncated right-hand side, beginning with the inequality for k:

$$\begin{aligned} k &> \log_{b_s} \frac{M}{\min\left((m_s \cdot b_t^{-e'}) \% b_s^{-e_s}\right)} - e_s \\ \Rightarrow b_s^{k+e_s} &> \frac{M}{\min\left((m_s \cdot b_t^{-e'}) \% b_s^{-e_s}\right)} \\ \Rightarrow b_s^{k+e_s} \cdot \min\left((m_s \cdot b_t^{-e'}) \% b_s^{-e_s}\right) &> M \end{aligned}$$

We now increase the left-hand side by erasing the minimum function and decrease the right-hand side by replacing M with m_s:

$$\begin{aligned} \Rightarrow b_s^{k+e_s} \cdot \left((m_s \cdot b_t^{-e'}) \% b_s^{-e_s}\right) &> m_s \\ \Rightarrow \frac{(m_s \cdot b_t^{-e'}) \% b_s^{-e_s}}{b_s^{-e_s}} &> \frac{m_s}{b_s^k} \\ \Rightarrow -\frac{(m_s \cdot b_t^{-e'}) \% b_s^{-e_s}}{b_s^{-e_s}} &< -\frac{m_s}{b_s^k} \\ \Rightarrow \frac{m_s \cdot b_t^{-e'}}{b_s^{-e_s}} - \frac{(m_s \cdot b_t^{-e'}) \% b_s^{-e_s}}{b_s^{-e_s}} &< \frac{m_s \cdot b_t^{-e'}}{b_s^{-e_s}} - \frac{m_s}{b_s^k} \\ \Rightarrow \left\lfloor \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \right\rfloor &< \frac{m_s}{b_s^k} \cdot \left(\frac{b_s^{k+e_s}}{b_t^{e'}} - 1 \right) \\ \Rightarrow \left\lfloor \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \right\rfloor &< \frac{m_s}{b_s^k} \cdot \left\lfloor \frac{b_s^{k+e_s}}{b_t^{e'}} \right\rfloor . \end{aligned}$$

□

4.3 Correct Rounding

Given a fraction a/b, we generally compute some fraction $\lfloor a/b \rfloor$ and subsequently determine whether we need to round up or not. This section describes a generic approach for rounding correctly that follows the Ryū algorithm [Adams 2018] and uses prime factorizations of quantities known a priori. It works for any rounding mode.

We primarily determine whether $a/b - \lfloor a/b \rfloor > 1/2$: if so, we always have to round up. In addition, we determine whether $a/b - \lfloor a/b \rfloor = 1/2$ to detect ties which must be broken according to the rounding mode. We now show how to compute these efficiently:

Lemma 4.3. *Let a and b be positive integers. Let x and i be positive integers, and $pow_i(x)$ be the largest integer k such that i^k divides x. We have that:*

$$a/b - \lfloor a/b \rfloor \geq 1/2 \iff \lfloor 2a/b \rfloor \% 2 = 1 \tag{1}$$

$$a/b - \lfloor a/b \rfloor = 1/2 \iff \lfloor 2a/b \rfloor \% 2 = 1 \wedge pow_b(2a) \geq 1. \tag{2}$$

Let $\mathbb{P}$ be the set of all primes, and $\mathbb{N}$ the set of all positive integers. It also holds that:

$$\forall p \in \mathbb{P}, x, y \in \mathbb{N} : pow_p(x \cdot y) = pow_p(x) + pow_p(y) \tag{3}$$

$$\forall p \in \mathbb{P}, x, k \in \mathbb{N} : pow_{p^k}(x) = \left\lfloor \frac{pow_p(x)}{k} \right\rfloor \tag{4}$$

$$\forall p \in \mathbb{P}, x, k, c \in \mathbb{N}, p \text{ does not divide } c :$$

$$pow_{p^k \cdot c}(x) = min(pow_{p^k}(x), pow_c(x)) \tag{5}$$

$$\forall a, b \in \mathbb{N} : \frac{a}{b} \text{ is an integer } \iff pow_b(a) \geq 1 \tag{6}$$

Proof. **(1)** We start with the left-hand side and rearrange:

$$\begin{aligned} & a/b - \lfloor a/b \rfloor \geq 1/2 \\ \iff & 2a/b - 2 \cdot \lfloor a/b \rfloor \geq 1 \\ \iff & \lfloor 2a/b - 2 \cdot \lfloor a/b \rfloor \rfloor \geq 1 \\ \iff & \lfloor 2a/b \rfloor - 2 \cdot \lfloor a/b \rfloor \geq 1 \\ \iff & \lfloor 2a/b \rfloor - 2 \cdot \lfloor 2a/2b \rfloor \geq 1 \\ \iff & \lfloor 2a/b \rfloor - 2 \cdot \left\lfloor \frac{\lfloor 2a/b \rfloor}{2} \right\rfloor \geq 1 \\ \iff & \lfloor 2a/b \rfloor \% 2 \geq 1 \\ \iff & \lfloor 2a/b \rfloor \% 2 = 1 \end{aligned}$$

(2) We first show that the left-hand side implies the right-hand side. We have already shown part of this claim, so we only need to show that $2a/b$ must be an integer. We start with the left-hand side and rearrange:

$$\begin{aligned} & a/b - \lfloor a/b \rfloor = 1/2 \\ \iff & 2a/b - 2 \cdot \lfloor a/b \rfloor = 1 \end{aligned}$$

Since 1 and $2 \cdot \lfloor a/b \rfloor$ are both integers, this can only be true if $2a/b$ is also integer, which is equivalent to $pow_b(2a) \geq 1$.

We now show that the right-hand side implies the left-hand side. We have already shown part of this claim, namely that $a/b - \lfloor a/b \rfloor \geq 1/2$, so we only need to show that the additional constraint of $pow_b(2a) \geq 1$ is sufficient to guarantee that the left-hand side is exactly $1/2$. Let $2a/b = k$, which is an integer, and therefore $a/b = k/2$:

$$a/b - \lfloor a/b \rfloor = k/2 - \lfloor k/2 \rfloor$$
$$\Rightarrow a/b - \lfloor a/b \rfloor = \begin{cases} 0 & \text{if } k \text{ is even} \\ 1/2 & \text{if } k \text{ is odd} \end{cases}$$

Since $a/b - \lfloor a/b \rfloor \geq 1/2$, it can only be exactly $1/2$.

(3) Let $m = x/p^{pow_p(x)}$ and $n = y/p^{pow_p(y)}$. Clearly, m is an integer and can not be a multiple of p, as otherwise $p^{pow_p(x)+1}$ would divide x in contradiction with the definition of pow; similarly for n. Since p is prime, $m \cdot n$ can not be a multiple of p either. We have that

$$mn = x/p^{pow_p(x)} \cdot y/p^{pow_p(y)} = xy/\left(p^{pow_p(x)+pow_p(y)}\right)$$

Therefore $pow_p(xy) = pow_p(x) + pow_p(y)$.

(4) Let $m = pow_p(x)$. Clearly, x/p^m is an integer and can not be a multiple of p. Also, we have that

$$k \cdot \lfloor m/k \rfloor \leq m < k \cdot (\lfloor m/k \rfloor + 1).$$

Therefore

$$\frac{x}{(p^k)^{\lfloor m/k \rfloor}} = \frac{x}{p^{k \cdot \lfloor m/k \rfloor}} \text{ is an integer, and}$$
$$\frac{x}{(p^k)^{\lfloor m/k \rfloor + 1}} = \frac{x}{p^{k \cdot (\lfloor m/k \rfloor + 1)}} \text{ is not an integer.}$$

It follows that $pow_{p^k}(x) = \lfloor m/k \rfloor = \lfloor pow_p(x)/k \rfloor$.

(5) Since p and c are coprime, the maximal power of $p^k \cdot c$ that divides x must be less than or equal to the maximal power of p^k that divides x and simultaneously less than or equal to the maximal power of c that divides x. At the same time, let m be an integer. If $(p^k)^m$ divides x and c^m also divides x, then $(p^k)^m \cdot c^m = (p^k \cdot c)^m$ must also divide x. It follows that $pow_{p^k \cdot c}(x) = min(pow_{p^k}(x), pow_c(x))$.

(6) If a/b is an integer, then a is a multiple of b, and therefore $pow_b(a) \geq 1$. If $pow_b(a) \geq 1$, then a is a multiple of b, and therefore a/b must be an integer. □

Lemma 4.3 gives us the necessary tools to determine whether we need to round up. Note that $\lfloor \lfloor 2a/b \rfloor /2 \rfloor = \lfloor a/b \rfloor$, so the necessary computation may be done as part of an existing step to compute $\lfloor a/b \rfloor$, e.g., as part of Lemma 4.2.

In particular, if the target base b_t is even, then instead of computing $\lfloor 2a/b \rfloor \% 2$ we can use the equivalent operation of computing an additional digit in the target base and checking if it is greater than or equal to $b_t/2$. In some cases, we already compute one or more additional digits that would otherwise be unused. If the target base b_t is odd, or we do not already have an additional digit, then we need to make sure that the additional factor of 2 is taken into account when computing the lookup tables used for Lemma 4.2 or Lemma 5.1 below.

5 RYŪ PRINTF

This section describes our new algorithm, Ryū Printf. Section 5.1 first introduces a stronger version of Lemma 4.2, and Section 5.2 discusses the corner case where the exact value does not have a finite expansion. Section 5.4 then handles `%f` format generation, and Section 5.5 handles `%e` format generation.

5.1 Segmented Mantissa Conversion

Ryū directly obtains the leading digits of the target mantissa, and it produces sufficiently many to guarantee that it can subsequently compute the shortest representation. For `printf`, however, we need as many digits as specified by the precision parameter, which can be arbitrarily large. If we used the same approach as Ryū, then we would need larger and larger multiplications to obtain more and more digits. Unfortunately, the cost for the multiplications grows super-linearly in the size of the multipliers, and therefore in the number of generated digits.

Instead, this section describes a technique to compute segments of digits in the target base. Here, the multiplier sizes for each segment remain constant for a given floating-point type, so the overall cost is linear in the number of generated digits.

As a simplified example, consider this approach of converting 2^{70} to 10-digit decimal segments, i.e., converting it to base-10^{10}:

$$\begin{aligned} 2^{70} &= \texttt{1180591620717411303424} \\ \lfloor 2^{70}/10^{20} \rfloor \% 10^{10} &= \texttt{11} \\ \lfloor 2^{70}/10^{10} \rfloor \% 10^{10} &= \quad\texttt{8059162071} \\ \lfloor 2^{70}/10^{0} \rfloor \% 10^{10} &= \qquad\qquad\texttt{7411303424} \end{aligned}$$

That is, given a segment length l and an arbitrary exponent e', we compute

$$\left\lfloor \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \right\rfloor \% b_t^l.$$

As argued above, if we had to fully evaluate $\lfloor m_s \cdot b_s^{e_s} / b_t^{e'} \rfloor$, then this approach would not provide any benefit: taking the modulo after doing a large multiplication would only add yet another expensive step. We will show that we can move the modulo operator into the floor function, which results in a stronger version of Lemma 4.2:

Lemma 5.1. *Assume the same conditions and case distinction as Lemma 4.2. Further, let l be a positive integer, and X be either 1 (case 1) or 0 (case 2). We have that:*

$$\left\lfloor \frac{m_s \cdot b_s^{e_s}}{b_t^{e'}} \right\rfloor \% b_t^l = \left\lfloor \frac{m_s}{b_s^k} \cdot \left(\left\lfloor \frac{b_s^{k+e_s}}{b_t^{e'}} + X \right\rfloor \% \left(b_t^l \cdot b_s^k \right) \right) \right\rfloor \% b_t^l$$

Proof. This follows directly from Lemma 4.2 in combination with Lemma 5.2. □

Lemma 5.2. *Let a, b, and c be positive integers. We have that*

$$\left\lfloor \frac{a}{b} \right\rfloor \% c = \left\lfloor \frac{a \% (b \cdot c)}{b} \right\rfloor \% c.$$

Proof. We start with the right-hand side and rearrange:

$$\left\lfloor \frac{a \% (b \cdot c)}{b} \right\rfloor \% c = \left\lfloor \frac{a - bc \cdot \lfloor \frac{a}{bc} \rfloor}{b} \right\rfloor \% c = \left\lfloor \frac{a}{b} - c \cdot \left\lfloor \frac{a}{bc} \right\rfloor \right\rfloor \% c$$

The term $c \cdot \left\lfloor \frac{a}{bc} \right\rfloor$ is an integer because both factors are integers, so we can move it out of the floor function. Therefore:

$$\begin{aligned}\left\lfloor \frac{a\%(b \cdot c)}{b} \right\rfloor \%c &= \left(\left\lfloor \frac{a}{b} \right\rfloor - c \cdot \left\lfloor \frac{a}{bc} \right\rfloor\right) \%c \\ &= \left(\left\lfloor \frac{a}{b} \right\rfloor \%c - \left(c \cdot \left\lfloor \frac{a}{bc} \right\rfloor\right) \%c\right) \%c \\ &= \left(\left\lfloor \frac{a}{b} \right\rfloor \%c - 0\right) \%c \\ &= \left\lfloor \frac{a}{b} \right\rfloor \%c\end{aligned}$$

□

Lemma 5.1 allows us to reduce the size of the multiplication for each segment to a manageable size, although we need an additional lookup table entry for each segment.

Consequently, there are two competing requirements for the segment size. On the one hand, our lookup table grows with the number of segments, so fewer larger segments are better. On the other hand, the computations for larger segments require larger integer operations. As a compromise, we have chosen the segment size such that a segment fits into a single machine integer; this also simplifies the subsequent step of converting each segment into a string of digits in the target base. In our implementation, we use 32-bit integers for each segment, which results in a segment size of 9 decimal digits.

5.2 Infinite Expansion

As a corner case, if the source base has a prime factor that is not in the target base, then the exact representation of a source base floating-point number in the target base may be infinite. While Lemma 5.1 still applies, a naively-built lookup table would become infinitely large as well. Fortunately, finite tables suffice because the exact representation must be periodic in this case. In general, the length of the period of a fraction a/b in base b_t is the multiplicative order of b modulo b_t. In addition, we know a priori that there is a maximum length of the expansion before the period starts, and we can use this to limit the size of the lookup table. Once we are beyond that, we can simply copy the last `<period length>` digits as often as necessary to fill the digits up to the given precision.

5.3 Pseudocode Definitions

Before discussing the pseudocode, we need to provide some basic definitions.

We use a segment index where positive values or zero correspond to segments to the left of the decimal point, and negative values correspond to segments to the right of the decimal point. Doing so allows us to use a single two-dimensional lookup table, called `table`, for the multipliers (see Section 5.1), indexed by exponent and segment index. However, we need different constants, c_0 and c_1 corresponding to the chosen values for k in Lemma 5.1, Case 1 and 2 respectively, to compute the correct shift distances:

$$table[e][i] = \begin{cases} \left\lfloor 10^{-9i} \cdot 2^{e+c_1} + 1 \right\rfloor \% \left(10^9 \cdot 2^{c_1}\right) & i \geq 0 \\ \left\lfloor 10^{-9i} \cdot 2^{c_0-e} \right\rfloor \% \left(10^9 \cdot 2^{c_0}\right) & i < 0 \end{cases}$$

The table values are computed ahead of time using arbitrary-precision arithmetic. Note that our pseudocode does not handle table compression. In our implementation, we compress the table by reusing table rows rather than providing one row per possible exponent value: when merging

2^b rows, we need b additional bits for each entry to maintain accurate results. For each row, our implementation only contains the non-zero entries, which requires additional bounds checks.

Our primary helper function is called `mul_shift_mod`, and is defined as:

$$\texttt{mul\_shift\_mod}(t, m, e) = \left\lfloor \frac{t \cdot m}{2^e} \right\rfloor \%10^9$$

This function is performance-sensitive, and also has to handle intermediate results that do not fit into a machine integer.

`estimate_first_integer_segment` and `estimate_first_fractional_segment` estimate the first non-zero segment to the left and to the right of the decimal point, respectively. Our implementation uses fast estimators based only on the input exponent e; other trade-offs between accuracy and performance are possible, as it is generally unproblematic for them to over-estimate the correct segment.

The `append_digits(buffer, digits,` n`)` function converts the given integer to decimal and appends the least significant n digits as characters to the buffer. The pseudocode sometimes shifts digits to the right to move them into the correct place for this function; our implementation uses multiple specialized functions to cover all necessary cases.

5.4 Generating %f Format

This section describes how Ryū Printf generates the `printf %f` format, with pseudocode provided as the function `append_double_as_f` in Figure 2. We subdivide the algorithm into five steps:

Step 1. Decode the given floating-point number into $(-1)^s \cdot m_s \cdot b_s^{e_s}$ with both m_s and e_s integers stored in native machine registers, like in Ryū.

Step 2. Detect `NaN`, `±Infinity`, and `±0.0`. If applicable, generate the appropriate output and return early.

Step 3. Print the sign if necessary. Then, print the integer part of the floating-point value using the segmented mantissa generation approach outlined in Section 5.1. To this end, estimate the number of required segments and iterate through the segments from left to right; we use a simple estimator based solely on the exponent. Start printing digits upon discovering the first non-zero segment. If all segments are zero, print a single '0' character.

Step 4. Print the dot and the fractional part, also using the segmented mantissa generation approach. Determine the number of segments from the given precision, and iterate through them from left to right. In addition, determine whether the number needs to be rounded up, rounded up if it is odd, or not rounded up using the prime factorization approach outlined in Section 4.3; we use the last segment as a partial solution, generating an additional segment if the precision is divisible by the segment length.

Step 5. Round up, if necessary, by modifying the given buffer in-place from right to left. Doing so can spill over into the integer part, and may also require moving the dot and appending another digit. For example, rounding up `9.99` results in `10.00`.

5.5 Generating %e Format

This section describes how to generate the `printf %e` format, with pseudocode provided as the function `append_double_as_e` in Figure 3. The algorithm is conceptually similar to the previous one in that it iterates through the segments left-to-right. However, the location of the decimal dot is not fixed relative to the segments as before but depends on the location of the first non-zero segment.

We subdivide the algorithm into five steps:

```
def append_double_as_f(buffer, $d$, $p$)
    # 1. Decode $d$ into $(-1)^s \cdot m \cdot 2^e$, with $m$ and $e$ integer
    $(s, m, e)$ = decode($d$)

    # 2. Handle special cases NaN, Infinity, and 0.0
    if ($e$ = SPECIAL_EXPONENT) or ($e = 0$ and $m = 0$) :
        handle_special_cases(buffer, $s$, $m$, $e$)
        return

    # 3. Print the integer part
    # 3a. Append the sign
    if $s$ : buffer.append('-')
    # 3b. Loop through the integer part segments in reverse
    zero = True
    for $i$ in [estimate_first_integer_segment($e$) $-$ 1, 0] :
        digits = mul_shift_mod($m$, table[$e$][$i$], $-(c_0 - e)$)
        if !zero :
            # Append exactly 9 digits, including leading zeros
            append_digits(buffer, digits, 9)
        elif digits $\neq 0$ :
            # Append up to 9 digits starting with the first non-zero digit
            availableDigits = $\lfloor \log_{10}(\text{digits}) \rfloor + 1$
            append_digits(buffer, digits, availableDigits)
            zero = False
    # 3c. The integer part was zero; append a single '0' character
    if zero : append(buffer, '0')

    # 4. Print the fractional part
    if $p > 0$ : append(buffer, '.')
    roundUp = 0 # Tristate: 0 = do not round; 1 = round up; 2 = round up if odd
    segments = $\lfloor \frac{p}{9} \rfloor + 1$ # We add 1 to obtain an additional digit for rounding
    for $i$ in [$-1$, $-$segments] :
        digits = mul_shift_mod($m$, table[$e$][$i$], $e + c_1$)
        if $i >$ $-$segments :
            # Append exactly 9 digits, including leading zeros
            append_digits(buffer, digits, 9)
        else :
            # Append up to 8 digits, including leading zeros (length may be 0)
            length = $p - 9 \cdot$ segments
            append_digits(buffer, $\left\lfloor \frac{\text{digits}}{10^{9-\text{length}}} \right\rfloor$, length)
            # Determine whether we need to round up
            roundUp = determine_rounding(digits, length $-$ 1, $m$, $e$, $p$)

    # 5. Round up if necessary (modifies buffer in-place; can cross into integer part)
    if roundUp $\neq 0$ : round_up(roundUp, buffer)
```

Fig. 2. Pseudocode for %f conversion.

```
def append_double_as_e(buffer, $d$, $p$)
    # 1. Decode $d$ into $(-1)^s \cdot m \cdot 2^e$, with $m$ and $e$ integer
    $(s, m, e)$ = decode($d$)

    # 2. Handle special cases NaN, Infinity, and 0.0
    if ($e$ = SPECIAL_EXPONENT) or ($e = 0$ and $m = 0$) :
        handle_special_cases(buffer, $s$, $m$, $e$)
        return

    # 3. Print number
    # 3a. Append the sign
    if $s$ : append(buffer, '-')
    # 3b. Loop left-to-right through integer part segments, fractional part segments
    printedDigits = 0, remainingDigits = 0
    for (interval, c) in [
            ([estimate_first_integer_segment($e$), 0], $-c_0$),
            ([estimate_first_fractional_segment($e$), $-\infty$], $c_1$)
    ]
        for $i$ in interval :
            digits = mul_shift_mod($m$, table[$e$][$i$], $e + c$)
            if (printedDigits $\neq 0$) or (digits $\neq 0$) :
                if printedDigits $\neq 0$ :
                    availableDigits = 9
                else :
                    # Append the leading non-zero digit and a '.'
                    availableDigits = $\lfloor \log_{10}(\text{digits}) \rfloor + 1$
                    $e_{10} = 9 \cdot i +$ availableDigits $- 1$
                    append_digits(buffer, $\left\lfloor \frac{\text{digits}}{10^{\text{availableDigits}-1}} \right\rfloor$, 1)
                    if $p > 0$ : append(buffer, '.')
                    availableDigits = availableDigits $- 1$
                # Append up to 9 digits, including leading zeros
                length = min($p$ − printedDigits, availableDigits)
                append_digits(buffer, $\left\lfloor \frac{\text{digits}}{10^{\text{availableDigits}-\text{length}}} \right\rfloor$, length)
                printedDigits = printedDigits + length
                availableDigits = availableDigits − length
                if (printedDigits = $p$) and (availableDigits $\neq 0$) : break

    # 4. Round up if necessary (modifies buffer in-place; can cross into integer part)
    roundUp = determine_rounding(digits, remainingDigits, $m$, $e$, $p$)
    if roundUp $\neq 0$ : $e_{10} = e_{10}$ + round_up_scientific(roundUp, buffer)

    # 5. Print exponent
    append(buffer, '$e$')
    append(buffer, $e_{10}$)
```

Fig. 3. Pseudocode for %e conversion.

Step 1. Decode the given floating-point number into $(-1)^s \cdot m_s \cdot b_s^{e_s}$ with both m_s and e_s integers stored in native machine registers, like in Ryū.

Step 2. Detect `NaN`, `±Infinity`, and `±0.0`. If applicable, generate the appropriate output and return early.

Step 3. Print the sign if necessary. Then, iterate through the segments corresponding to both the integer and the fractional part of the input value. Upon encountering the first non-zero segment, append the first non-zero digit followed by a '.' if necessary. Determine the preliminary output exponent from the segment index i and the index j of the first non-zero digit within the segment $e_o = li + j$ where l is the segment length. Continue appending segments until `precision` digits are done; note that we need one additional digit to determine rounding.

Step 4. Determine rounding by using the additional digit and the prime factorization approach from Section 4.3. Round up, if necessary, by modifying the given buffer in-place from right to left. Note that this never requires moving the dot, but may require increasing the output exponent e_o. For example, rounding up `9.99e+10` results in `1.00e+11`.

Step 5. Print the 'e' character and output exponent e_o.

5.6 Alternative Segmentation Orders

Generating digits from left to right is not required if we assume availability of sufficient memory for the full output. We could also generate them from right to left, or even generate multiple segments in parallel, e.g., using SIMD instructions. However, we do need to know where in the output each segment needs to go, and for that, we need to know exactly how many digits are required for the integer part, after rounding, or fix the result if we guessed incorrectly:

(1) Left-to-right generation. In this case, we may have to go back to earlier segments to round correctly or hold one or more segments until we can determine that the carry is guaranteed not to propagate into the specific segment. A carry can only propagate more than one segment if that segment's value is maximal, i.e., $b_t^s - 1$, except for the last segment, which may have more digits than necessary for the given precision. An advantage of this approach is that we do not have to know the final length of the output.

(2) Right-to-left generation. In this case, we determine whether rounding up is required before or as part of generating the first segment, and propagate the carry as we go from segment to segment. However, right-to-left generation generally requires a buffer that is large enough to hold the entire number. In addition, we need to know the exact length of the number before we can start writing output to the buffer.

(3) Parallel generation. Using SIMD instructions, we can generate multiple segments in parallel. Similar to right-to-left generation, we need to know the exact length of the number before we can start writing output to the buffer. However, carry propagation becomes more involved.

The `printf` variants `sprintf` and `snprintf` always generate output into a user-specified buffer. `printf` usually generates output into an intermediate buffer before making the system call to write the output: making one system call per character would incur a very significant performance overhead. Therefore, the question of left-to-right or right-to-left generation may be academic.

For our implementation, we decided to go with the simplest possible approach: we generate the entire number into a buffer using left-to-right generation and round up by modifying the number inside the buffer afterward. This is simple and fast: carry propagation rarely requires updating more than one output digit.

6 QUANTITATIVE PERFORMANCE MEASUREMENTS

We are not only interested in the overall performance of our algorithm, but also how it behaves across the range of possible floating-point numbers, as well as across different output precisions.

An individual conversion call can take less than 100 nanoseconds, which is difficult to measure directly and is also subject to significant noise and variance due to unpredictable microarchitectural behaviors. The processors we used all support hyperthreading, which we disabled to prevent unnecessary sharing of microarchitectural resources. The performance also heavily depends on the optimizations enabled in the compiler. Unfortunately, `printf` is part of the standard library which is provided as a pre-compiled library, so it is not affected by the benchmark compiler settings. We use the '-O2' compiler setting for the benchmark with the expectation that this matches the standard library.

We have decided on the following approach: we select one thousand floating-point values pseudo-randomly, equally distributed over the 64-bit range. For each sample, we run the conversion 1000 times, measuring the time for all 1000 iterations, and take the average per iteration. We rerun the same samples for the precision values 1, 10, 100, and 1000 to give an idea of the expected performance across a large precision range without running an excessive number of experiments.

This approach reduces noise with the side effect of fully priming the microarchitectural resources, which can result in better than expected performance. We provide the same environment to all implementations and record the versions of the compiler and standard library. We only compare results from benchmark runs in identical environments, i.e., we do not directly compare results from one processor to another processor, a different compiler, or running on a different operating system.

A disadvantage of our approach is that it does not necessarily predict real-world behavior. Perfectly primed microarchitectural resources are unlikely in any scenario, and real-world applications usually also have a non-uniform distribution of floating-point values. However, we believe that the resulting graphs provide insight into the underlying algorithms. In particular, the shapes of the resulting graphs can indicate whether the underlying algorithm is linear or superlinear in the length of the output, in the precision, or the value of the exponent, with discontinuities hinting at case distinctions in the implementation. It also gives us good coverage of the floating-point range, even if a specific application would benefit from a stronger focus on a subset of possible values.

In addition to collecting performance numbers, our benchmark also compares the resulting strings to each other and outputs a warning if there is any difference in output.

6.1 Results

We implemented Ryū Printf in C, and compared it to existing C implementations of `printf` on multiple platforms; Figure 4 lists hardware models and software versions. These include glibc and musl on Ubuntu Linux, MSVC and msys on Microsoft Windows, and Apple's libc on macOS.

In all cases, we have found Ryū Printf to be significantly faster than the existing implementation. Figure 5 shows a summary of the more detailed results in Figure 6: the y-axis shows how many times longer the implementation takes compared to Ryū Printf on average for the same CPU, compiler, and operating system.

Machine	OS	Libc Implementation
AMD Ryzen 7 2700X @3.70 GHz	Ubuntu Linux 18.04	libc6 2.27-3ubuntu1
AMD Ryzen 7 2700X @3.70 GHz	Ubuntu Linux 18.04	musl 1.1.19-1
AMD Ryzen 7 2700X @3.70 GHz	Windows 10 Home 1803	MSVC 19.14.26429.4
AMD Ryzen 7 2700X @3.70 GHz	Windows 10 Home 1803	msys-runtime-devel 2.10.0-2
Intel Core i5 @2.9 GHz	macOS Mojave 10.14	Apple Libc

Fig. 4. The combinations of machine hardware, operating system version, and libc version tested.

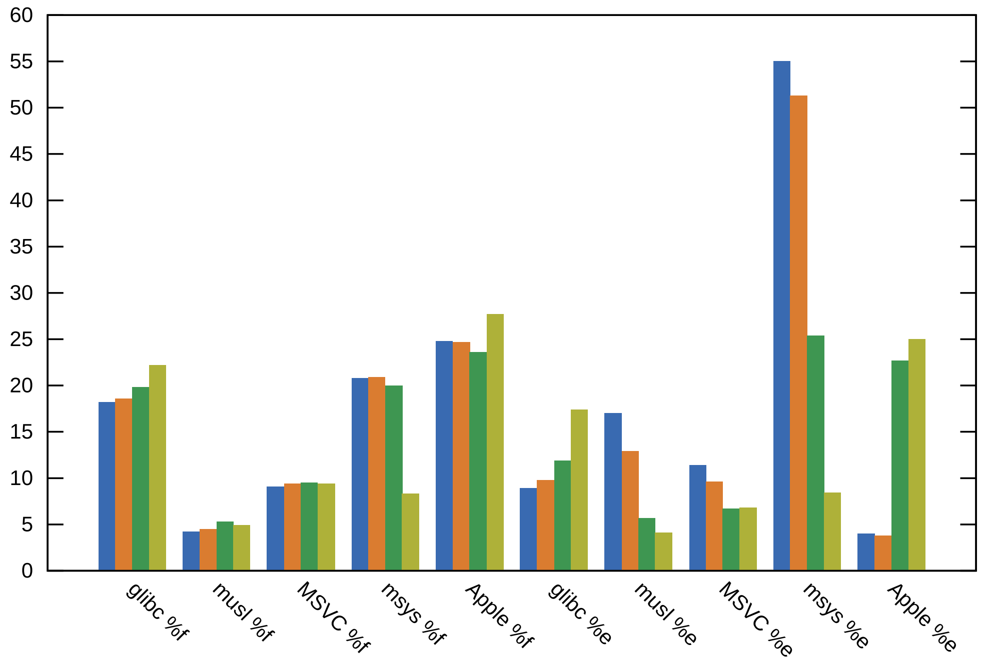


Fig. 5. Performance overview over all printf implementations. The y-axis shows how many times longer the implementation takes on average compared to our implementation of Ryū Printf on the same machine with the same inputs. Each set of four bars corresponds to precision values of 1, 10, 100, and 1000 from left to right using blue, orange, green, and yellow (the same colors as Figure 6).

Figure 6 shows detailed results with one graph per `printf` implementation. Each graph plots a pseudo-random sample of 1000 floating-point values for precision values 1, 10, 100, and 1000, as described. Each point is the average time in nanoseconds of converting a single floating-point value over the integer value of the binary representation of that floating-point number. Therefore, the x-axis has a range of $[0, 2^{64})$, which covers all 64-bit floating-point values. Some points are outside the plotted range, especially in the glibc and msys graphs. The left half of each plot $[0, 2^{63})$ are the positive and the right half $[2^{63}, 2^{64})$ the negative floating-point values. The sign bit does not affect performance, so the plots all exhibit translational symmetry.

The graphs (a-h) show `%f` performance, whereas graphs (j-q) show `%e` performance. In both cases, the rows are performance data for Linux, Windows, and macOS, respectively. The first two columns in each row are compiled with the same compiler and collected in a single run of the benchmark binary. The last column in each row, if present, uses a different compiler, and is therefore not directly comparable. The hardware used for the Linux and Windows benchmarks is identical, whereas the macOS machine is slightly less powerful.

Our implementation of Ryū Printf on Windows is slower than on Linux despite identical hardware; this is primarily due to separate code paths for the primary multiply-and-shift operation. We are using compiler extensions for high-precision integer operations on Linux which we have not yet been able to match in performance on Windows.

We generally expect an increase in output length to result in increased runtime; regardless of how the implementation works, writing more digits to memory will generally take longer. `%f` outputs the full integer part of the number, so its outputs gets longer with increasing (positive) exponent; this starts to happen around the value representing 1.0 (approximately 2^{62}) which is roughly halfway through the range of positive floating-point values (about a quarter of the entire plot width). `%e` outputs in scientific notation, which only depends on the precision.

In the best case, an implementation is linear in the length of the output: for a given precision value, `%e` shows a horizontal line across the entire plot, and `%f` shows a horizontal line turning into a sloped line, both about equal widths, in both the positive and negative halves of the plot.

For low precision values, Ryū Printf does show these patterns; in fact, there is so much overlap between the precision values for 1, 10, and 100, that the lower values are hardly distinguishable. This indicates that its implementation is linear in the output size.

However, it does not show these patterns for the largest precision value of 1000. The underlying reason is that our implementation uses an optimized code path to print leading and trailing zeros which has a much lower constant factor than the general case. The observed lines exhibit a linear relationship to the number of non-zero digits, which increases with both increasing (positive) and decreasing (negative) exponents, up to the point where the interval of non-zero digits starts to fall outside the precision. This optimization results in a trough shape that becomes more pronounced for higher precision values.

6.2 Rendering `%f`

The Ryū Printf graphs for %f rendering for low precision (< 1000) cases show a mostly horizontal line on the left-hand side for the range $[0, 2^{62}]$. Most of these floating-point values are less than 1.0, so there is no integer part, and rendering of the fractional part is primarily dependent on the precision. The high precision plot, however, exhibits a clear slope; the reason for the slope is that our implementation uses a fast path to bulk set trailing zeros as soon as it can determine that all further digits are zero, and the floating-point numbers to the left have more non-zero digits and therefore take longer to reach that point.

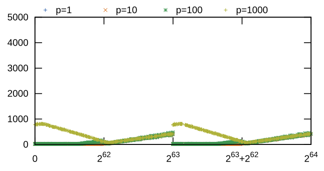
On the right-hand side, the range $[2^{62}, 2^{63}]$ shows a linear increase in time with the number of digits in the integer part of the floating-point value - most of these values are larger than 1.0 and have no fractional part. Therefore, all trailing zeros are generated on the fast path, which results in very little performance difference for different precision values.

The **glibc** graphs show a very steep increase in rendering time for floating-point values larger than 1.0; the glibc library also struggles with high precision values, for which the measurements are outside the displayed range. On the left-hand side, the graphs show several dips, which seem to indicate a case distinction based on the exponent. Glibc is about 15 times slower than Ryū Printf.

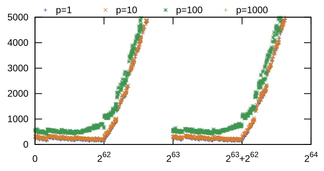
The **musl** graphs show significantly better behavior than glibc for floating-point values larger than 1.0, with the high-precision values also clearly using a fast path for the fractional part. For floating-point values smaller than 1.0, the rendering time increases as numbers get smaller, with the steepness depending on the precision. Musl is about 4 times slower than Ryū Printf, being clearly the closest in performance.

The **MSVC** graphs indicate worse performance than glibc for low precision values, but behave slightly better for high precision values. MSVC is about 9 times slower than Ryū Printf. During our testing, we rediscovered a known bug in the MSVC implementation, which does not round correctly in all cases.

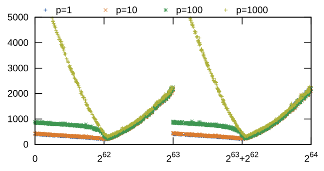
Msys has the worst total runtime, and is largely outside the plotted range, being about 20 times slower than Ryū Printf for small precision values, and about 8 times slower for the largest tested



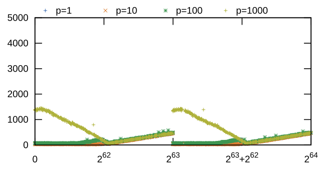
(a) Linux, Ryū %f



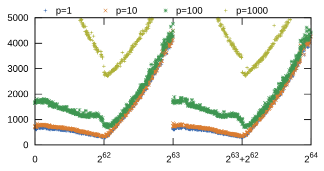
(b) Linux, glibc %f



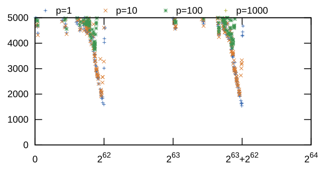
(c) Linux, musl %f



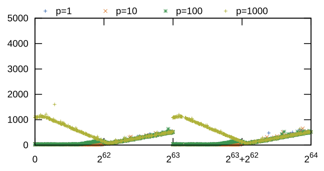
(d) Windows, Ryu %f



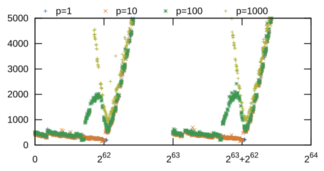
(e) Windows, MSVC %f



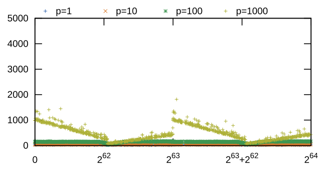
(f) Windows, msys %f



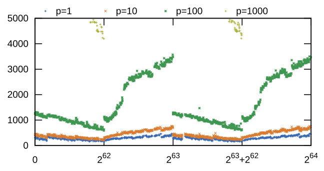
(g) macOS, Ryu %f



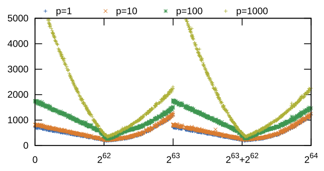
(h) macOS, Apple Libc %f



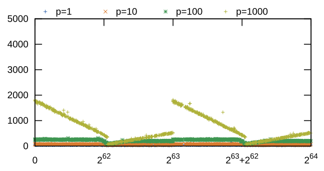
(j) Linux, Ryū %e



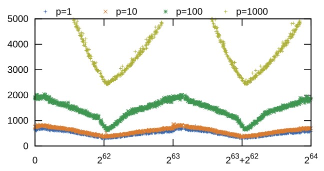
(k) Linux, glibc %e



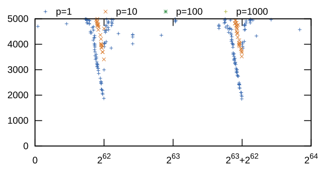
(l) Linux, musl %e



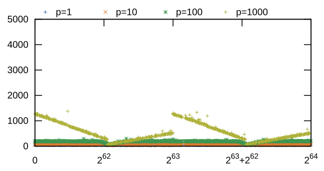
(m) Windows, Ryu %e



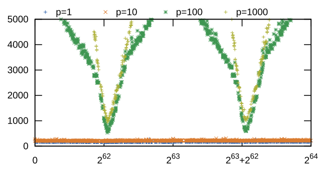
(n) Windows, MSVC %e



(o) Windows, msys %e



(p) macOS, Ryu %e



(q) macOS, Apple Libc %e

Fig. 6. Each graph shows plots for precision values 1, 10, 100, and 1000 in blue, orange, green, and yellow, respectively. 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.

precision of 1000. Note that msys only prints about 17 non-zero digits regardless of the precision value, and therefore differs from other implementations that print as many as requested.

Apple's libc shows a very steep slope with increasing floating-point value, although it also uses a fast path for the fractional part. Overall, Apple's libc is about 24 times slower than Ryū Printf.

6.3 Rendering %e

The Ryū Printf graphs for %e rendering for low precision (< 1000) cases show a mostly horizontal line across the entire range. This is expected as the performance is mainly independent of the exact floating-point value, and mainly dependent on the precision. For the high precision case, the fast path turns this into a slope on both sides of the graph, as explained above.

The **glibc** graphs show good performance across the board for low precision cases (< 100), but significant deterioration as precision goes up, with the largest precision measurements again outside the displayed range. Glibc is about 9 times slower than Ryū Printf, and does not seem to be using a fast path for large precision values.

The **musl** graphs exhibit slopes in both directions. While it is slightly worse than glibc on low precision cases, it performs significantly better as precision increases. Musl is about 17 times slower than Ryū Printf for small precision values, improving to about 4 times slower for the largest.

The **MSVC** graphs exhibit slopes in both directions, similar to musl, although the absolute performance is slightly worse. MSVC is about 11 times slower than Ryū Printf for small precision values, improving to about 6 times slower for the largest.

Msys has the worst total runtime, and is largely outside the plotted range, being about 55 times slower than Ryū Printf for small precision values, and improving to about 8 times slower for the largest precision value.

Apple's libc implementation is optimized for small precision values, for which it is almost flat. However, it becomes significantly slower as precision increases. Apple's libc is about 4 times slower than Ryū Printf for small precision values, which is the closest to Ryū Printf, but regresses to about 25 times slower for the largest.

6.4 Comparing to I/O Performance

If the typical use case for `printf` is to write floating-point numbers to a file, then the obvious question is whether its performance matters compared to disk speeds. While we cannot conclusively answer this question, we can look at individual examples based on our benchmark results in relation to typical contemporary disk write speeds: %e conversion with a precision of 10 on Linux.

The average time per conversion for Ryu printf is 42ns or about 428 MB/s single-threaded, and for glibc is 538ns or about 33 MB/s. Today's SSDs handle 200 – 3500 MB/s sequential writes. Given these numbers, writing an SVG file with large, complex path objects could certainly be bottlenecked on floating-point conversion. If doing so blocks an interactive SVG editor, this could be user-visible, even if it only takes up a small fraction of the total application runtime. However, it is likely that such usage has already been optimized given the comparatively bad performance of `printf`; for example, the open-source Inkscape SVG editor uses the Grisu3 algorithm developed by Loitsch [2010].

6.5 Comparing to Ryū

The performance of Ryū and Ryū Printf is not directly comparable due to the differences in the problem definition.

However, we observe that Ryū has a flat performance profile at around 25ns per conversion on comparable hardware to generate about 17 digits of decimal precision. The closest comparison is Ryū Printf %e rendering, for which we measure about 40ns per conversion for a precision of 1, and 60ns per conversion for a precision of 10. This seems to indicate that there is still room for improvement for Ryū Printf.

For `%f` rendering, Ryū Printf average runtime is dominated by the integer parts of large floating-point values.

6.6 Table Size

The lookup table size in our C implementation of Ryū Printf for 64-bit floating-point numbers is approximately 104 kB. This is significantly larger than the table needed for Ryū, which fits in under 10 kB. In both cases, the table also covers any smaller floating-point formats. We consider this still unproblematic for typical machines today; for example, including these tables as part of glibc would increase the binary size by only about 5%.

That said, such a table size is likely a problem for embedded devices where size is at a premium. The size could be halved to 52 kB with no performance impact, and further compressed at the expense of increased run time by using the mathematical structure of the table constants. We have been able to compress the tables for Ryū adapted to 128-bit floating-point numbers by a factor of approximately 35.

The table for Ryū Printf is computed a priori and compiled into the binary. Therefore, we exclude the precomputation cost from the benchmarks.

7 RELATED WORK

Knuth [1997] summarized the earliest radix conversion algorithms. In particular, Samelson and Bauer [1953] provided an early discussion of floating-point conversion in the context of binary computers.

More recently, Coonen [1980; 1981] provided a very brief description of an algorithm for binary to decimal floating-point conversion. It seems to require floating-point types with higher precision than the values being converted.

Steele and White [1990] defined the problem of finding the shortest round-trip safe representation in another base, and provided an algorithm, called Dragon, using arbitrary precision arithmetic. With some modifications, the core of this algorithm is also applicable to `printf`.

Gay [1990] as well as Burger and Dybvig [1996] described improved versions of Dragon, while still using arbitrary precision arithmetic. Burger and Dybvig introduced the concept of the smaller and larger halfway points, which seem critical for guaranteeing correctness.

Subsequent work has focused on the problem of finding the shortest representation and has become less and less general. Jaffer [2013] provided a Java implementation using an iterative conversion approach.

Loitsch [2010] introduced a family of algorithms that only use integer operations. The first variant, Grisu, always outputs 21 digits and therefore does not maintain the minimum-length criterion. The third variant, Grisu3, does, but rejects some inputs and has to fall back to a slower algorithm that uses arbitrary precision arithmetic.

Andrysco et al. [2016] described Errol, another family of algorithms to generate shortest output. Like Grisu, Errol emulates higher-precision floating-point operations and detects inaccuracies. Unlike Grisu, Errol uses pairs of 64-bit floating-point registers and falls back to a lookup table.

Adams [2018] described another integer algorithm to generate the shortest representation, Ryū, which is significantly faster than both Grisu and Errol, and does not require a fallback. Ryū proves that there is no need for large multiplications if only the first few digits of the result are needed.

Despite all these advances, `printf` is likely the most widely used float-to-string function. Unfortunately, algorithms that generate shortest output are not immediately useable for `printf`.

8 SUMMARY

The printf function has been a staple of the C programming language for decades and has found its way into many other programming languages as well. Based on Ryū [Adams 2018], we describe a new algorithm for converting floating-point numbers to strings according to the printf %f, %e, and %g formats. Our performance results indicate that it is about 4 times faster than the best existing implementation, which is considerable given the long history of printf. Our implementation in C is open source [Adams 2019].

Unfortunately, despite most of the previous implementations being open source, our results (Fig. 6) also demonstrate that performance improvements made in one implementation often do not make their way into other implementations, although we can only speculate about the underlying reasons.

This paper strengthens the theory behind the Ryū algorithm, significantly increasing the range of use cases for which it can be applied. We primarily cover printf formatting here, but also show applicability to the case of fixed-precision binary/decimal conversion in both directions. Furthermore, we provide a baseline performance target for library maintainers, which we hope will encourage future improvements across all implementations. Even if our new algorithm is not adopted in full, we posit that a few small changes, such as the adoption of a fast path for rendering trailing zeros, can already result in significant improvements for existing implementations.

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 Stephan T. Lavavej for discussions, encouragement, as well as source code contributions.

REFERENCES

Ulf Adams. 2018. Ryū: Fast Float-to-String Conversion. In *Proceedings of the 39th ACM SIGPLAN 2018 Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 270–282. https://doi.org/10.1145/3192366.3192369

Ulf Adams. 2019. *https://github.com/ulfjack/ryu*. https://doi.org/10.5281/zenodo.3366212

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

Anonymous. 2018. *printf format string*. https://en.wikipedia.org/wiki/Printf_format_string

Nikolay Petrovich Brusentsov and José Ramil Alvarez. 2011. Ternary Computers: The Setun and the Setun 70. In *Perspectives on Soviet and Russian Computing*, John Impagliazzo and Eduard Proydakov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 74–80.

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

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 [Coonen 1981].

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 [Coonen 1980].

Free Software Foundation, Inc. 2018. *GNU libc Printf Documentation*. https://www.gnu.org/software/libc/manual/html_node/Floating_002dPoint-Conversions.html

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

IBM. 2018. *IBM Printf Documentation*. https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_72/rtref/printf.htm

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

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

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.

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

Microsoft. 2016. *Microsoft Visual Studio 2017: Format specification syntax: printf and wprintf functions*. https://docs.microsoft.com/en-us/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions?view=vs-2017

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

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