Abstract

We present algorithms for accurately converting floating-point numbers to decimal representation. They are fast (up to 4 times faster than commonly used algorithms that use high-precision integers) and correct: any printed number will evaluate to the same number, when read again.

Our algorithms are fast, because they require only fixed-size integer arithmetic. The sole requirement for the integer type is that it has at least two more bits than the significand of the floating-point number. Hence, for IEEE 754 double-precision numbers (having a 53-bit significand) an integer type with 55 bits is sufficient. Moreover we show how to exploit additional bits to improve the generated output.

We present three algorithms with different properties: the first algorithm is the most basic one, and does not take advantage of any extra bits. It simply shows how to perform the binary-to-decimal transformation with the minimal number of bits. Our second algorithm improves on the first one by using the additional bits to produce a shorter (often the shortest) result. Finally we propose a third version that can be used when the shortest output is a requirement. The last algorithm either produces optimal decimal representations (with respect to shortness and rounding) or rejects its input. For IEEE 754 double-precision numbers and 64-bit integers roughly 99.4% of all numbers can be processed efficiently. The remaining 0.6% are rejected and need to be printed by a slower complete algorithm.

Categories and Subject Descriptors I.\textit{m} [Computing Methodologies]: Miscellaneous

General Terms Algorithms

Keywords floating-point printing, dtoa

1. Introduction

Printing floating-point numbers has always been a challenge. The naive approach is not precise enough and yields incorrect results in many cases. Throughout the 1970s and 1980s many language libraries and in particular the \texttt{printf} function of most C libraries were known to produce wrong decimal representations.

In the early 1980s Coonen published a paper \cite{Coonen1980} and a thesis \cite{Coonen1984} containing algorithms for accurate yet economical binary-decimal conversions, but his work went largely unnoticed (at least with respect to the printing algorithms).

Steele and White's paper \cite{SteeleJr.andWhite1990} had a much bigger impact. Correct printing become part of the specification of many languages and furthermore all major C libraries (and as a consequence all programs relying on the \texttt{printf} functions) adapted accurate algorithms and print correct results now.

Steele and White's algorithm, “Dragon4”, relies on high precision arithmetic (also known as “bignums”) and even though two other papers \cite{Gay1990} and \cite{Burger and Dybvig1996} proposed improvements and optimizations to the algorithm this requirement remained. It is natural to wonder if limited-precision arithmetic could suffice. Indeed, according to Steele and White's retrospective of 2003 \cite{SteeleJr. and White2004} “

[

 during the 1980s, White investigated the question of whether one could use limited-precision arithmetic [...] rather than bignums. He had earlier proved by exhaustive testing that just 7 extra bits suffice for correctly printing 36-bit PDP-10 floating-point numbers, if powers of ten used for prescaling are precomputed using bignums and rounded just once. The document continues by asking whether “[one could] derive, without exhaustive testing, the necessary amount of extra precision solely as a function of the precision and exponent range of a floating-point format”.

In this paper we will present a new algorithm Grisu, which allows us to answer this question. Grisu requires only two extra bits and a cache of precomputed powers-of-ten whose size depends on the exponent range.

However, Grisu does not supersede Dragon4 and its optimized descendants. While accurate and fast (up to 4 times faster than previous approaches) it produces suboptimal results. For instance the IEEE 754 double-precision number representing 0.3 is printed as 29999999999999998e-17. When read, both numbers will be approximated to the same floating-point number. They are hence both accurate representations of the corresponding floating-point number, but the shorter 0.3 is clearly more desirable.

With just two extra bits it is difficult to do better than in our example, but often there exists an integer type with more bits. For IEEE 754 floating-point numbers, which have a significand size of 53, one can use 64 bit integers, providing 11 extra bits. We have developed an algorithm Grisu2 that uses these extra bits to shorten the output. However, even 11 extra bits may not be sufficient in every case. There are still boundary conditions under which Grisu2 will not be able to produce the shortest representation. Since this property is often a requirement \cite{SteeleJr. and White2004}.

\footnote{A draft of this article had existed long before and had already been mentioned in “Knuth Volume 2” \cite{Knuth1981} in 1981.}
for some examples) we propose a variant, Grisu3, that detects (and aborts) when its output may not be the shortest. As a consequence Grisu3 is incomplete and will fail for some percentage of its input. Given 11 extra bits roughly 99.5% are processed correctly and are thus guaranteed to be optimal (with respect to shortness and rounding). The remaining 0.5% are rejected and need to be printed by another printing algorithm (like Dragon4).

All presented algorithms come with code snippets in C that show how they can be efficiently implemented. We use C99, as this version provides the user with a platform independent means of using 64-bit data types.

In this paper we will concentrate exclusively on IEEE 754 double-precision floating-point numbers. They are the de facto standard today and while our work applies to other floating-point representations it would unnecessarily complicate the descriptions.

We will now discuss some basics in Section 2. In Section 3 we present a custom floating-point data-type which will be used in all remaining sections. Section 4 details the requirements on the cache of powers-of-ten. In Section 5 we introduce Grisu, and in Section 6 we present a custom floating-point data-type which will be used in the remainder of this document.

We conclude this section with a definition we will use frequently throughout the text to handle signs.

Section 2.2 contains examples for all notions we introduce in this section. Readers might want to have a look at this section whenever a definition is unclear.

A floating point number, as the name suggests, has a radix point that can “float”. Concretely a floating-point number $v$ in base $b$ (usually 2) with precision $p$ is built out of an integer significand (also known as mantissa or fraction) $f_v$ of at most $p$ digits and an exponent $e_v$, such that $v = f_v \times b^{e_v}$.

Unless otherwise stated, we use the convention that the significand of a floating-point number is named $f$ with the variable’s name as subscript. Similarly the exponent is written as $e$ with the same subscript. For instance a floating-point number $w$ is assumed to be composed of $f_w$ and $e_w$.

Any significant $s$ satisfies $f = \sum_{i=0}^{p-1} d_i \times b^i$, $0 \leq d_i < b$ where the integers $d_i$ are called the digits of $f$. We call a number “normalized” if the most-significant digit $d_{p-1}$ is non-zero.

If the exponent has unlimited range any non-zero number can be normalized by “shifting” the significand to the left while adjusting the exponent accordingly. When the exponent is size-limited then some numbers cannot be normalized. We call non-normalized numbers that have the minimal exponent “denormalized”.

Note. Floating-point numbers may allow different representations for the same value (for example $12 \times 10^3$ and $1.2 \times 10^5$). The representation is however unique when all numbers are either normalized or denormal.

2.1 Rounding and Errors

Floating point numbers have only a limited size and thus a limited precision. Real numbers must hence be rounded in order to fit into this finite representation. In this section we will discuss the rounding mechanisms that are used in this document and introduce a mechanism to quantify the error they introduce.

The most natural way of rounding is to choose the nearest available floating-point number. This rounded-to-nearest approach is straightforward except for half-way cases (in the decimal system numbers ending with 5).

In this paper we will use the following strategies for half-way cases:

- **up**: picks the number closer to $+\infty$. We will use the notation $\lceil x \rceil$ when rounding $x$ by this strategy.
- **even**: picks the number that is even: $\lfloor x \rfloor$. For instance, in the decimal system, 1.5 would round to 2, whereas 0.5 would round to 0. This is the default strategy used by IEEE.

Whenever the half-way rounding strategy has no importance we will use a star to make this fact explicit: $\lfloor x \rfloor^*$.

We will use the notation $\tilde{x}$ to indicate that the floating-point number $\tilde{x}$ contains a normalized significand of size $p$ which has been computed by rounding-to-nearest using strategy $s$ (up, even, or any).

We can quantify $\tilde{x}$’s error $|\tilde{x} - x|$ as follows: $\tilde{x}$ is of form $f \times b^e$ and since $f$ has been rounded to nearest $|\tilde{x} - x| \leq 0.5 \times b^e$, or, in other words, by half a unit in the last place (of the significand).

Following established conventions we will use the shorthand $\pmulp$ to describe these units. A $\pmulp$ needs to be given with respect to a certain floating-point number. In almost all cases the associated floating-point number is clear from context. In the remaining cases we add the associated number as subscript as so: $\pmulp_{0.5}$.

During the remainder of this document we will use the tilde-notation to indicate that a number has been rounded-to-nearest. In most cases its error will be 0.5 $\pmulp$, but this is not always the case.

2.2 Neighbors and Boundaries

For floating-point number types where the value of each encoded number is unique we can define predecessors and successors.

Let $v = f_v \times b^e$ be a strictly positive floating-point number. The predecessor $v^-$ of $v$ is the next smallest number. If $v$ is minimal, then we define $0$ to be its predecessor. Similarly $v^+$ designates the successor of $v$. For the maximal $v$ we define $v^+$ to be $v^+ := v + (v - v^-)$. That is for this particular $v$ the successor $v^+$ is at the same distance than the predecessor $v^-$. We call $v^-$ and $v^+$ neighbors of $v$.

The boundary between two adjacent numbers $v_1$ and $v_2$ is simply their arithmetic mean: $m := \frac{v_1 + v_2}{2}$. By definition boundaries can not be expressed in the given floating-point number type, since its value lies between two adjacent floating-point numbers. Every floating-point number $v$ has two associated boundaries: $m^- := \frac{v_{min} + v}{2}$ and $m^+ := \frac{v + v_{max}}{2}$. Clearly, any real number $w$, such that $m^- < w < m^+$, will round to $v$. Should $w$ be equal to one of the boundaries then we assume that $w$ is rounded to even (the IEEE 754 default). That is, the rounding algorithm will chose the floating-point number with an even significand.

We conclude this section with a definition we will use frequently in the remainder of this document.

**Definition 2.1.** A printed representation $R$ of a floating-point number $v$ satisfies the internal identity requirement iff $R$ would convert to $v$ when read again.

For IEEE 754 double-precision numbers (where half-way cases are rounded to even) this implies $|R|_{\pmulp_{0.5}} = v$. In other words:

$m^- \leq V < m^+$ when $f_v$ is even, and
$m^- < V < m^+$ when $f_v$ is odd.

2.3 Examples

In this section we show some examples for the previously defined notions. For simplicity we will work in a decimal system. The significand’s size $p$ is set to 3, and any exponent is in range 0 to 10. All numbers are either normalized or denormal.
In this configuration the extreme values are \( \text{min} := 1 \times 10^0 \) and \( \text{max} := 999 \times 10^{10} \). The smallest normalized number equals \( 100 \times 10^0 \). Non-normalized representations like \( 3 \times 10^2 \) are not valid. The significand must either have three digits or the exponent must be zero.

Let \( v := 1234 \) be a real number that should be stored inside the floating-point number type. Since it contains four digits the number will not fit exactly into the representation and it must be rounded. When rounded to the nearest representation then \( \hat{v} := [v]_e := 123 \times 10^3 \) is the only possible representation. The rounding error is equal to \( 4.4 \) ulp.

Contrary to \( v \) the real number \( w := 1245 \) lies exactly between to possible representations. Indeed, \( 124 \times 10^3 \) and \( 125 \times 10^3 \) are both at distance 5. The chosen representation depends on the rounding mechanism. If rounded up then the significand 125 is chosen. If rounded to even then 124 is chosen. For \( w = 1235 \) both rounding mechanisms would have chosen 124 as significand.

The neighbors of \( w \) are \( w^- := 123 \times 10^3 \) and \( w^+ := 125 \times 10^3 \). Its respective boundaries are therefore \( m^- = 123 \times 10^3 \) and \( m^+ = 125 \times 10^3 \). In this case the neighbors were both at the same distance. This is not true for \( r := 100 \times 10^3 \), with neighbors \( r^- := 99 \times 10^3 \) and \( r^+ := 101 \times 10^3 \). Clearly \( r^- \) is closer to \( r \) than is \( r^+ \).

For the sake of completeness we now show the boundaries for the extreme values and the smallest normalized number. The number min has its lower (resp. upper) boundary at \( 0.5 \times 10^1 \) (resp. \( 1.5 \times 10^1 \)). For max, the boundaries are \( 99.8 \times 10^{10} \) and \( 999.5 \times 10^{10} \).

The boundaries for the smallest normalized number are special: even though its significand is equal to 100 the distance to its lower neighbor \( (99 \times 10^0) \) is equal to 1 ulp and not just 0.5 ulp. Therefore its boundaries are \( 99.5 \times 10^0 \) and \( 100.5 \times 10^0 \).

### 2.4 IEEE 754 Double-Precision

An IEEE 754 double-precision floating-point number, or simply “double”, is defined as a base 2 data type consisting of 64 bits. The first bit is the number sign, followed by 11 bits reserved for the exponent \( e_{\text{IEEE}} \), and 52 bits for the significand \( f_{\text{IEEE}} \). For the purpose of this paper the sign-bit is irrelevant and we will assume to work with positive numbers.

With the exception of some special cases (which will be discussed shortly) all numbers are normalized which in base 2 implies with positive numbers.

Therefore its boundaries are \( 9999 \times 10^9 \) and \( 10000 \times 10^9 \). However, size-agnostic and uses \( q \) for the significand’s precision.

### 3. Handmade Floating-Point

Grisu extracts the significand of its \( \text{diy}.f.p \) in an early stage and \( \text{diy}.f.p \) is used only for two operations: subtraction and multiplication. The implementation of the \( \text{diy}.f.p \) type is furthermore simplified by restricting the input and by relaxing the output. For instance, both operations are not required to return normalized results (even if the operands were normalized). Figure 1 shows the C implementation of the two operations.

The operands of the subtraction must have the same exponent and the result of subtracting both significands must fit into the significand-type. Under these conditions the operation clearly does not introduce any imprecision. The result might not be normalized. The multiplication returns a \( \text{diy}.f.p \) \( \hat{r} \) containing the rounded result of multiplying the two given \( \text{diy}.f.p \) \( x \) and \( y \). The result might not be normalized. In order to distinguish this imprecision from the precise multiplication we will use the “rounded” symbol for this operation: \( \hat{r} := x \odot y \).

\[ \text{diy}.f.p; \]

Figure 1: The \( \text{diy}.f.p \) type.

Grisu and its variants only require fixed-size integers, but these integers are used to emulate floating-point numbers. In general reimplementing a floating-point number type is a non-trivial task, but in our context only few operations with severe limitations are needed. In this section we will present our implementation, \( \text{diy}.f.p \), of such a floating-point number type. As can be seen in Figure 1 it consists of a limited precision integer (of higher precision than the input floating-point number), and one integer exponent. For the sake of simplicity we will use the 64 bit long \( \text{uint64}_t \) in the accompanying code samples. The text itself is, however, size-agnostic and uses \( q \) for the significand’s precision.

### 3.1 Operations

Grisu extracts the significand of its \( \text{diy}.f.p \) in an early stage and \( \text{diy}.f.p \) is used only for two operations: subtraction and multiplication. The implementation of the \( \text{diy}.f.p \) type is furthermore simplified by restricting the input and by relaxing the output. For instance, both operations are not required to return normalized results (even if the operands were normalized). Figure 2 shows the C implementation of the two operations.

The operands of the subtraction must have the same exponent and the result of subtracting both significands must fit into the significand-type. Under these conditions the operation clearly does not introduce any imprecision. The result might not be normalized. The multiplication returns a \( \text{diy}.f.p \) \( \hat{r} \) containing the rounded result of multiplying the two given \( \text{diy}.f.p \) \( x \) and \( y \). The result might not be normalized. In order to distinguish this imprecision from the precise multiplication we will use the “rounded” symbol for this operation: \( \hat{r} := x \odot y \).

\[ \text{diy}.f.p; \]

Figure 1: The \( \text{diy}.f.p \) type.
Proof. By definition \( x \otimes y = \left[ \frac{f_i}{2^q} \right] \times 2^{r_i+e_i+q} \). Since \( \frac{f_i}{2^q} \) is either exact or a half-way case we have \( \left[ \frac{f_i}{2^q} \right] \times 2^{r_i+e_i+q} \geq f_i \times 2^{r_i+e_i+q-1} \) and hence \( x \otimes y \geq x \times y \). Also \( |x \times y - x \times y| \leq u_i \times 2^{r_i+e_i+q-1} \) and thus \( x \times y - x \otimes y \leq \frac{u_i}{2} \) ulp.

4. Cached Powers

Similar to White’s approach (see the introduction) Grisu needs a cache of precomputed powers-of-ten. The cache must be precomputed using high-precision integer arithmetic. It consists of normalized \( \text{diy}_f \) values \( c_i := [c_i] \) where \( c_i := 10^q \). Note that, since all \( c_i \) are normalized \( \forall i, 3 \leq e_i - c_i \leq 4 \).

The size of the cache’s \( (k, e) \) range depends on the used algorithm as well as the input’s and \( \text{diy}_f \) precision. We will see in Section 5 how to compute the needed range. For IEEE doubles and 64 bit \( \text{diy}_f \)s a typical cache must hold roughly 635 precomputed values. Without further optimizations the cache thus takes about 8KB of memory. In [Coonen(1984)] Coonen discusses efficient ways to reduce the size of this cache.

The corresponding C procedure has the following signature:

```c
diy_fp cached_power(int k);
```

4.1 \( k \) Computation

Grisu (and its evolutions) need to find an integer \( k \) such that its cached power \( c_k = f_k \times 2^{r_k} = [10^q] \) satisfies \( \alpha \leq c_k + e_k \leq \gamma \) for a given \( e, \alpha \) and \( \gamma \). We impose \( \gamma \geq \alpha + 3 \), since otherwise a solution is not always possible. We now show how to compute the sought \( k \).

All cached powers are normalized and any \( f_k \) thus satisfies \( 2^{e_k} \leq f_k < 2^{e_k+1} \). Hence, \( 2^{e_k+1} \leq c_k < 2^{e_k+2} \).

Suppose that all cached powers are exact (i.e. have no rounding errors). Then \( k \) (and its associated \( c_k \)) can be found by computing the smallest power of ten \( 10^q \) that verifies \( 10^q \geq 2^{e_k+q-1} \).

\[
k := \left\lceil \log_{10} 2^{e_k + q - 1} \right\rceil = \left\lceil (\alpha - e + q - 1) \times \frac{1}{\log_{10}2} \right\rceil
\]

4.2 Computation

The C implementation emulates in a portable way a partial 64-bit multiplication. Since the 64 least significant bits of the multiplication \( f_l \times f_r \) are only used for rounding the procedure does not compute the complete 128-bit result. Note that the rounding can be implemented using a simple addition (line 8).

Since the result is rounded to 64 bits a \( \text{diy}_f \) multiplication introduces some error.

**Lemma 3.3.** Let \( x \) and \( y \) be two \( \text{diy}_f \)s. Then

\[
x \otimes y := \left[ \frac{f_l \times f_r}{2^q} \right] \times 2^{r_l+e_l+q}
\]

The C implementation emulates in a portable way a partial 64-bit multiplication. Since the 64 least significant bits of the multiplication \( f_l \times f_r \) are only used for rounding the procedure does not compute the complete 128-bit result. Note that the rounding can be implemented using a simple addition (line 8).

Since the result is rounded to 64 bits a \( \text{diy}_f \) multiplication introduces some error.

**Lemma 3.4.** Let \( x \) and \( y \) be two \( \text{diy}_f \)s, and \( y \) a real such that \(|y - y| \leq u_i \) ulp. In other words \( y \) is the approximated \( \text{diy}_f \) of \( y \) and has a maximal error of \( u_i \) ulp. Then the errors add up and the result is bounded by \((.5 + u_i) \) ulp.

\[
|xy - xy| \leq u_i \text{ ulp} |x - x| \Rightarrow |xy - xy| < (u_i + .5) \text{ ulp}
\]

**Lemma 3.5.** Let \( x \) be a normalized \( \text{diy}_f \), \( y \) be a \( \text{diy}_f \), and \( y \) a real such that \(|y - y| \leq u_i \) ulp. If \( x = 2^{q-1} \) (the minimal significant) then \( x \otimes y \) underflows by at most \( \frac{u_i}{2} \) ulp compared to \( xx \).

\[
|y - y| \leq u_i \text{ ulp} \land f_l = 2^{q-1} \Rightarrow xy - x \otimes y \leq \frac{u_i}{2} \text{ ulp}
\]
5.1 Idea

Printing a floating-point number is difficult because its significant and exponent cannot be processed independently. Dragon4 and its variants therefore combine the two components and work with high-precision rationals instead. We will now show how one can print floating-point numbers without bignums.

Assume, without loss of generality, that a floating-point number \( v \) has a negative exponent. Then \( v \) can be expressed as \( \frac{t}{10^q} \). The decimal digits of \( v \) can be computed by finding a decimal exponent \( t \) such that \( 1 \leq \frac{t}{10^q} < 10 \).

The first digit is the integer result of this fraction. Subsequent digits are generated by repeatedly taking the remainder of the division. The cached numbers \( \alpha \)'s exponent so that only a small exponent remains. In the following algorithm we cancel out most of powers-of-ten to avoid bignum operations. The cached numbers \( \gamma \)'s exponent is in range \(-1137 \) to +960. Invoking \( k_{\text{comp}} \) from Section 4.1 with these values yields:

\[
\begin{align*}
\alpha &= k_{\text{comp}}(960 + 64) = -289, \\
\alpha &= k_{\text{comp}}(-1137 + 64) = 342.
\end{align*}
\]

In step 3 \( w \) is multiplied with \( c_\alpha \). The goal of this operation is to obtain a \( \text{diy}_\text{fp} \) \( \tilde{D} \) that has an exponent \( e_0 \) such that \( \alpha \leq e_0 \leq \gamma \). Some configurations make the next step (output) easy. Suppose, for instance, that \( e_0 \) becomes zero. Then \( \tilde{D} = f_\gamma \) and the decimal digits of \( \tilde{D} \) can be computed by printing the significand \( f_\gamma \) (a q-bit integer). With an exponent \( e_0 \neq 0 \) the digit-generation becomes slightly more difficult, but since \( e_0 \)’s value is bounded by \( \gamma \) the computation is still straightforward.

Grisu’s result is a string containing \( \tilde{D} \)’s decimal representation followed by the character “e” and \( \tilde{D} \)’s digit. As such it represents the number \( V := \tilde{D} \times 10^p \). We claim that \( V \) yields \( v \) when rounded to floating-point number of precision \( p \).

Theorem 5.1. Grisu’s result \( V \) satisfies the internal identity requirement: \( |V|_p^D = v \).

Proof. In the best case \( V = v \) and the proof is trivial. Now, suppose \( V > v \). This can only happen if \( c_\alpha > c_\gamma \). We will ignore \( V \)’s parity and simply show the stronger strict inequality \( V < m^+ \).

\[
V = w \circ c_\alpha \times 10^4, \quad 10^4 \times c_\gamma = 1, \quad \text{and} \quad m^+ - v = 2^{2^i} \text{ this expands to } w \circ c_\alpha - w \times c_\gamma < 2^{2^i-1} \times c_\gamma.
\]

Since, by hypothesis, \( e_0 \geq e_0 + 2 \) it is hence sufficient to show that \( w \circ c_\alpha - w \times c_\gamma < 2^{2^i+1} \times c_\gamma \).

We have two cases:

\[
\begin{align*}
&\text{1.} \quad f_\gamma > 2^{2^i-1}. \quad \text{By hypothesis } c_\gamma \text{'s error is bounded by } 0.5 \text{ ulp and thus } c_\gamma \geq 2^{(q-1) + \epsilon}. \quad \text{It suffices to show that } w \circ c_\gamma - w \times c_\gamma \text{ is strictly less than } 2^{2^i-1+q+\epsilon} \text{ which is guaranteed by Lemma 3.4.} \\
&\text{2.} \quad f_\gamma = 2^{2^i-1}. \quad \text{Since the next lower } \text{diy}_\text{fp} \text{ is only at distance } 2^{2^i-1} \text{ and } c_\gamma \text{ is rounded to nearest, } c_\gamma \text{'s error is bounded by } 0.5 \text{ ulp.} \quad \text{Clearly } c_\gamma \geq 2^{2^i-1} - \frac{1}{2} \times 2^{2^i-1} \geq 2^{2^i(q-1) + \epsilon} \text{ for any } q \geq 2. \quad \text{The inequality } w \circ c_\gamma - w \times c_\gamma < \frac{1}{2} \times 2^{2^i(q-1) + \epsilon} \text{ is (due to the smaller error of } c_\gamma \text{) guaranteed by Lemma 3.4.}
\end{align*}
\]

We have proved the theorem for \( V > v \). The remaining case \( V < v \) can only happen when \( c_\alpha < c_\gamma \). Now suppose:

\[
\begin{align*}
&\text{1.} \quad f_\gamma > 2^{2^i-1} \text{ and therefore } v - m^+ = 2^{2^i-1}. \quad \text{The proof for this case is similar to the previous cases.}
\end{align*}
\]
• $f_i = 2^{n-1}$ and therefore $v - m^2 = 2^{n-2}$. Since $f_i$ is even we only need to show $m^2 \leq V$. Using similar steps as before it suffices to show that $2^{n-2(v-1)+\alpha} \leq \max(2^{30} - 1, \alpha_e)$ which is guaranteed by Lemma 3.5.

### 5.3 C Implementation

We can now present a C implementation of Grisu. This implementation uses 64 bit integers, but a proof of concept version, using only 55 bits, can be found on the author's homepage.

```c
1: #define TEN7 10000000
2: void cut(diy_fp D, uint32_t parts[3]) {
3:     parts[2] = (D.f % (TEN7) >> D.e) << D.e;
4:     uint64_t temp = D.f / (TEN7 >> D.e);
5:     parts[1] = temp % TEN7;
6:     parts[0] = temp / TEN7;
7: }
8: void grisu(double v, char* buffer) {
9:     diy_fp w; uint32_t ps[3];
10:    int q = 64, alpha = 0, gamma = 3;
11:    w = normalize(diy_fp(doublesize(diy_fp(v)));
12:    int mk = kcomp(w.e + q, alpha, gamma);
13:    diy_fp c_pkm = cached_power(mk);
14:    diy_fp D = multiply(w, c_pkm);
15:    cut(D, ps);
16:    sprintf(buffer, "%s%.0f%.0f\n", ps[0], ps[1], ps[2], -mk);
17: }
18: }
```

Figure 4: C implementation of Grisu with $\alpha, \gamma = 0.3$.

In Figure 4 line 8 we show the core grisu procedure specialized for $\alpha = 0$ and $\gamma = 3$. It accepts a non-special positive double and fills the given buffer with its decimal representation. Up to line 15 the code is a direct translation from the pseudo-algorithm to C. In this line starts step 4 (output).

By construction $D.e$ is in the range $0 - 3$. With a sufficiently big data-type one could simply shift $D.f$, the significant, and dump its decimal digits into the given buffer. Lacking such a type (we assume that uint64_t is the biggest native type), Grisu cuts $D$ into three smaller parts (stored in the array $ps$) such that the concatenation of their decimal digits gives D's decimal digits (line 15).

Note that $2^{30} = 147573952589676412928$ has 21 digits. Three 7-digit integers will therefore always be sufficient to hold all decimal digits of $D$.

In line 16 ps’s digits and the decimal exponent are dumped into the buffer. For simplicity we have used the stdlib’s sprintf procedure. A specialized procedure would be significantly faster, but would unnecessarily complicate the code.

Another benefit of cutting D’s significand into smaller pieces is that the used data-type (uint32_t) can be processed much more efficiently. In our specialized printing procedure (replacing the call to sprintf) we have noticed tremendous speed improvements due to this choice. Indeed, current processors are much faster when dividing uint32_t’s than uint64_t’s. Furthermore, the digits for each part can be computed independently which removes pipeline stalls.

### 5.4 Interesting target exponents

We will now discuss some interesting choices for $\alpha$ and $\gamma$. The most obvious choice $\alpha, \gamma := 0.3$ has already been presented in the previous section. Its digit-generation technique (cutting $D$ into three parts of 7 digits each) can be easily extended to work for target exponents in the range $\alpha := 0$ to $\gamma := 9$. One simply has to cut $D$ into three uint32_t’s of 9 decimal digits each. As a consequence $D$’s decimal representation might need up to 27 digits.

On the one hand the bigger $\gamma$ increases the output size (without increasing its precision), but on the other hand the extended range provides more room to find a suitable cached power-of-ten. The increased clearance can, for instance, be used to reduce the number of cached powers-of-ten. It is possible to remove two thirds of the cache while still being able to find the required $c_k$ of step 2. Indeed, two cached powers-of-ten $c_i$ and $c_k$ will always satisfy $c_i.e - c_k.e \leq 10$.

Another technique uses the increased liberty to choose the “best” cached power-of-ten among all that satisfy the requirement. For example, a heuristic could prefer exact cached numbers over inexact ones. Without additional changes to the core algorithm there is however little benefit in using such a heuristic.

Despite the added optimization opportunities the basic digit-generation technique still stays the same, though. We therefore move on to the next interesting exponent range: $\alpha, \gamma := -63, -60$.

```c
1: 1. int digit_gen_no_div(diy_fp D, char* buffer) {
2:     int i = 0, q = 64; diy_fp one;
3:     one.f = (uint64_t) 1 << (D.e); one.e = D.e;
4:     buffer[i++] = 0; (D.f >> -one.e); //division
5:     uint64_t f = D.f & (one.f - 1); // modulo
6:     buffer[i++] = f;
7:     while (-one.e > q - 5) {
8:         uint64_t tmp = f < (C2) & (one.f - 1);
9:         int d = f >> (-one.e - 3);
10:        d += 6; f = f + tmp; d += f >> (-one.e - 1);
11:        buffer[i++] = d;  // d is a 7-digit number
12:        one.e++; one.f >>= 1;
13:        f &= one.f - 1;
14:    }
15:    while (i < 19) {
16:        f += 10;
17:        buffer[i++] = f; //f > -one.e;
18:    }
19:    return i;
20: }
21: }
```

Figure 5: Digit generation for $\alpha = -63$ and $\gamma = -60$.

The beauty of this exponent range lies in the fact that the normalized diy_fp one, representing the number 1, is composed of $f_{one} = 2^6$ and $e_{one} = -63$. Usually expensive operations, such as division and modulo, can be implemented very efficiently for this significant. The C implementation in Figure 5 dispenses of division and modulo operators entirely and uses only inexpensive operations such as shifts and additions. With the exception of the exponent (which has at most 3 digits) Grisu manages to produce a decimal representation of an input IEEE floating-point number without any division at all. The price for this feat is the complicated code of Figure 5. Its complexity is necessary to avoid overflows. For simplicity we will start by describing the algorithm without caring for data-type sizes.

**Algorithm digit-gen-no-div**

**Input:** a diy_fp $D$ with exponent $-63 \leq e_p \leq -60$.

**Output:** a decimal representation of $D$.

**Procedure:**

1. **One:** determine the diy_fp one with $f_{one} = 2^{-e_p}$ and $e_{one} = e_p$.
2. **Digit0:** compute $d_0 := \left\lfloor \frac{d_i}{\text{one}} \right\rfloor$ and $D_1 := D \mod \text{one}$
3. **Ten:** If $d_0 \geq 10$ emit the digit “1” followed by the character representing $d_0 - 10$. Otherwise emit the character representing the digit $d_0$.
4. **Digits:** generate and emit digits $d_i$ as follows
   
   \[ d_i := \left\lfloor 10 x D_i \right\rfloor \]
\begin{itemize}
\item emit the character representing the digit \( d_i \)
\item \( D_{i+1} := 10 \times D_i \text{ mod one} \)
\end{itemize}

5. \textbf{Stop:} stop at the smallest positive integer \( n \) such that \( D_n = 0 \).

We will now show that the algorithm computes a decimal representation of \( D \). Let \( R_i \) be the number that is obtained by reading the emitted characters up to and including \( d_i \).

In step 2b \( d_i \) is printed. Since \( d_i \) consists of at most 4 binary digits it cannot exceed 15, and therefore (after this step) \( R_i \) evaluates to \( d_i \). We declare the following invariant for the loop of step 4:

\[ D = R_i + \frac{d_i}{10^i} \]

Clearly the invariant holds for \( i = 0 \), and the invariant is still valid after the execution of the loop-body. We can hence conclude that \( D = R_{i+1} \).

The C implementation of this algorithm is more involved as it has to deal with overflows. When multiplying \( D_i \) by ten (step 4) the result might not fit into a uint64_t. The code has therefore been split into two parts, one that deals with potential overflows, and another where the product safely fits in the data-type. The test in line 7 checks if the result fits into a uint64_t. Indeed, \( D_i < \text{one} \) for any \( 1 \leq i \leq n \) and with 4 additional bits the multiplication will not overflow. The easy, fast case is then handled in line 15. This loop corresponds to the loop of step 4. Note that digit_gen_no_div produces at most 18 digits. We will discuss this choice shortly.

Should \( 10 \times D_i \) not fit into a uint64_t the more complicated loop of line 7 is used. As to avoid overflows the code combines the multiplication by ten with the division/modulo by one. By construction \( e_p = e_{\text{one}} \) and \( f_{\text{one}} = 2^{e_p} \). The division by one can thus be written as

\[ D_x \times 10^{32} \over \text{one} = 2^{e_p} \times 10^{32} \over \text{one} \]

From this equation it is then only a small step to the implementation in Figure 5. In order to escape from this slow case digit_gen_no_div introduces an implicit common denominator. In line 12 one is divided by this denominator. This way one’s exponent decreases at each iteration and after at most 5 iterations the procedure switches to the lightweight loop.

Our implementation takes some shortcuts compared to the described algorithm: it skips step 2b and prints at most 18 digits. The first shortcut is only possible when Grisu uses the smallest cached power-of-ten that satisfies the range requirement, since in that case \( d_0 < 10 \). The 18 digit shortcut relies on the high precision (64 bits) used in the implementation. An implementation featuring only two extra-bits (55 bits for IEEE doubles) is forced to continue iterating until \( D_i = 0 \). Since each iteration clears only one bit one could end up with 55 decimal digits.

In this section we will present evolutions of Grisu: Grisu2 and Grisu3. Both algorithms are designed to produce shorter outputs. Grisu may be fast, but its output is clearly suboptimal. For example, the number 1.0 is printed as \( 10000000000000000000e-19 \). The optimal solution (printed by Grisu2 and Grisu3) avoids the trailing ‘0’ digits.

Grisu2 and Grisu3 use the extra capacity of the used integer type to shorten the produced output. That is, if the diy fp integer type has more than two extra bits, then these bits can be used to create shorter results. The more bits are available the more often the produced result will be optimal. For 64-bit integers and IEEE doubles (with a 53-bit significand) more than 99% of all input-numbers can be converted to their shortest decimal representation.

Grisu2 and Grisu3 differ in the way they handle the non-optimal solutions. Grisu2 simply generates the best solution that is possible with the given integer type, whereas Grisu3 rejects numbers for which it cannot prove that the computed solution is optimal. For demonstration purposes we include rounding as an optimality requirement for Grisu3. It is simple to adapt Grisu2 so it rounds its outputs, too.

Finally we render Grisu2 and Grisu3 more flexible compared to Grisu. There are different ways to format a floating-point number. For instance the number 1.23 could be formatted as 1.23, 123e-2, or 0.123e1. For genericity it is best to leave the formatting to a specialized procedure. Contrary to Grisu, Grisu2 and

![Figure 6: Digit generation for \( \alpha = -59 \) and \( \gamma = -32 \).](image-url)
Grisu3 do not produce a complete decimal representation but simply produce its digits (“123”) and the corresponding exponent (-2). The formatting procedure then needs to combine this data to produce a representation in the required format.

6.1 Idea

We will first present the general idea of Grisu2 and Grisu3, and then discuss each algorithm separately. Both algorithms try to produce optimal output (with respect to shortness) for a given input number \( v \).

The optimal output of input \( v \) represents a number \( V \) with the smallest leading length that still satisfies the internal identity requirement for \( v \). The “leading length” of \( V \) is its digit length once it has been stripped of any unnecessary leading and trailing “0” digits.

**Definition 6.1.** Let \( v \) be a positive real number and \( n, l \) and \( s \) be integers, such that \( l \geq 1 \), \( 10^{l-1} \leq s < 10^l \), \( v = s \times 10^{l-1} \) and \( l \) as small as possible. Then the \( l \) decimal digits of \( s \) are \( v \)'s leading digits and \( l \) is \( v \)'s leading length.

In the following we demonstrate how the optimal \( V \) can be computed. Let \( v \) be a floating-point number with precision \( p \) and let \( m^v \), \( m^s \) be its boundaries (as described in Section 2.2). Assume, without loss of generality, that its significand \( f_v \) is even. The optimal output consists of a number \( V \) such that \( m^- < V \leq m^+ \) and such that \( V \)'s significant length is minimal.

The current state of art [Burger and Dybvig (1996)] computes \( V \) by generating the input number \( v \)'s digits from left to right and by stopping once the produced decimal representation would evaluate to \( v \) when read again. Basically the algorithm tests for two termination conditions \( tc1 \) and \( tc2 \) after each generated digit \( d_v \):

- \( tc1 \) is true when the produced number (consisting of digits \( d_0, \ldots, d_i \)) is greater than \( m^- \), and
- \( tc2 \) is true when the rounded up number (consisting of digits \( d_0, \ldots, (d_i + 1) \)) is less than \( m^+ \).

In the first case a rounded down number (of \( v \)) would be returned, whereas in the second case the result would be rounded up.

Since these two tests are slow and cumbersome to write we have developed another technique that needs only one. The basic approach is similar: one produces the decimal digits from left to right, but instead of using \( v \) to compute the digits the faster approach generates the digits of \( m^s \). By construction any rounded up number of the generated digits will be greater than \( m^- \) and thus not satisfy the internal identity requirement anymore. Therefore the second termination condition will always fail and can hence be discarded.

We can show that this technique generates the shortest possible number.

**Theorem 6.2.** Let \( x \) and \( y \) two real numbers, such that \( x \leq y \). Let \( \kappa \) be the greatest integer, such that \( y \mod 10^\kappa \leq y - x \). Then \( V := \left[ \frac{y - x}{10^\kappa} \right] \times 10^\kappa \) satisfies \( x \leq V \leq y \). Furthermore \( V \)'s leading length \( l \) is the the smallest of all possible numbers in this interval: any number \( V' \) such that \( x \leq V' \leq y \) has a leading length \( l' \geq l \).

**Proof.** We start by showing \( x \leq V \leq y \): since \( v = V + y \mod 10^\kappa \) we know that \( V \leq y \). Also \( y \mod 10^\kappa \leq y - x \) and therefore \( V \geq x \).

For the sake of contradiction assume that there exists a \( V' \) with leading length \( l' \), such that \( x \leq V' \leq y \) and \( l' < l \).

The number \( V' \) has a leading length of \( l' \) and by definition there exists hence an \( s' \) and \( n' \) such that \( 10^{l'-1} \leq s' < 10^{l'} \) and \( V' = s' \times 10^{l'-1} \).

There are three cases to consider:

1. \( s' > \left\lfloor \frac{10^n}{10^p} \right\rfloor \): impossible since this implies \( V > y \).
2. \( s' = \left\lfloor \frac{10^n}{10^p} \right\rfloor \): contradiction, since this implies \( V = V' \).
3. \( s' < \left\lfloor \frac{10^n}{10^p} \right\rfloor \): we first prove the case for \( n' > k \).

By hypothesis \( k \) is maximal and hence \( y \mod 10^\kappa > y - x \). Given that \( y \mod 10^\kappa = y - \left\lfloor \frac{y}{10^\kappa} \right\rfloor \times 10^\kappa \) we can conclude that \( y - x \times 10^\kappa > y - x \) and thus \( V' < x \). Contradiction.

Suppose now that \( n' \leq k \). By definition of “leading length” we know that \( V \geq 10^{1-k} \times 10^\kappa - 1 \) and \( V' < 10^{l'-1} \times 10^\kappa \). Since \( l' < l \) we have \( V' < 10^{l'-1-k} \times 10^\kappa \). Also \( x \leq V \) and \( y \leq V' \) and therefore \( x < 10^{l'-1-k} \times y \). Clearly \( y - 10^{l'-1-k} < y - x \) and thus, using the same equality as before, \( y \mod 10^{l'-1-k} \leq y - x \) which contradicts the minimality property of \( k \).

\[ \square \]

6.2 Grisu2

In this section we will present Grisu2. As described above it will use extra bits to produce shorter outputs. As an evolution of Grisu, Grisu2 will not work with exact numbers (requiring bignum representations) either, but compute approximations of \( m^- \) and \( m^+ \), instead. In order to avoid wrong results (outputs that do not satisfy the internal identity requirement) we add a safety margin around the approximated boundaries. As a consequence Grisu2 sometimes fails to return the shortest optimal representation which could lie outside the conservative approximation. Also this safety-margin requires us to change the precondition. Indeed, using only 2 extra bits, the computation is so imprecise that Grisu2 could end up with an empty interval. In that case Grisu2 could simply fall back to Grisu, but this would unnecessarily complicate the following algorithm.

We thus opted to give Grisu2 an extra bit: \( q \geq p + 3 \).

**Algorithm Grisu2**

Input: same as for Grisu.

**Preconditions:** \( \text{diy}_{\text{fp}} \)'s precision \( q \) satisfies \( q \geq p + 3 \), and the powers-of-ten cache contains precomputed rounded normalized \( \text{diy}_{\text{fp}} \) values \( c_k = \left[ 10^k \right] \).

**Output:** decimal digits \( d_i \) for \( i \leq 0 \leq n \) and an integer \( K \) such that the real \( V := d_0 \ldots d_n \times 10^k \) verifies \( \left\lfloor V \right\rfloor_p = v \).

**Procedure:**

1. **Boundaries:** compute \( v \)'s boundaries \( m^- \) and \( m^+ \).
2. **Conversion:** determine the normalized \( \text{diy}_{\text{fp}} \) \( w^+ \) such that \( w^+ = m^+ \). Determine the \( \text{diy}_{\text{fp}} \) \( w^- \) such that \( w^- = m^- \) and that \( e_w = e_w^+ \).
3. **Cached Power:** find the normalized \( c_k = f_k \) such that \( \alpha \leq e_c + e_w^+ + q < \gamma \) (with \( \alpha \) and \( \gamma \) as discussed for Grisu).
4. **Product:** compute \( M^- := w^- \otimes c_k, M^+ := w^+ \otimes c_k, \) and let \( M^+ := M^+ + 1 \uparrow_{\alpha} p, \) \( M^- := M^- - 1 \uparrow_{\alpha} p, \) \( \delta := M^+ - M^- \).
5. **Digit Length:** find the greatest \( \kappa \) such that \( M^+_\kappa \mod 10^\kappa \leq \delta \) and define \( P := \left\lfloor \frac{M^+}{M^+_{\kappa}} \right\rfloor \).
6. **Output:** define \( V := P \times 10^{q+k} \). The decimal digits \( d_i \) and \( n \) are obtained by producing the decimal representation of \( P \) (an integer). Set \( K := k + n \), and return it with the \( n \) digits \( d_i \).

We will show efficient implementations combining step 5 and 6 later, but first, we prove Grisu2 correct. As a preparation we start by showing that \( M^+_\kappa \leq M^-_{\kappa} \).

\[ \square \]
Lemma 6.3. The variables $M_r^-$ and $M_r^+$ as described in step 4 verify $M_r^- \leq M_r^+$.

Proof. By definition

$M_r^- = M_r^- + 1 \ulp$

$= w^- \odot C + 1 \ulp$

$s = \left(\frac{f_x - f_i}{2^t} + 1\right) \times 2^{x+r+q}$

and similarly $M_r^+ = \left(\frac{f_x + f_i}{2^t} - 1.5\right) \times 2^{x+r+q}$. Since $f_x \geq f_i + 2^{t-1}$ and $q - p \geq 3$ it suffices to show $s \leq \frac{2^{t-1} \times (2^{t} + 2)}{2^t}$.

Theorem 6.4. Grisu2's result $V = d_0 \ldots d_n \times 10^k$ satisfies the internal identity requirement: $|V|_p = v$.

Proof. We will show that $m^- \leq M_r^- \leq V \leq M_r^+ \leq M_r^+ \leq m^+$ (with $m^-$ and $m^+$ $V$'s boundaries).

The inner inequality, $M_r^- \leq V \leq M_r^+$, is a consequence of Theorem 6.2. Remains to show that $m^- \leq M_r^- \leq M_r^+ \leq m^+$. By Lemma 3.4 $M_r^-$ and $M_r^+$ have an error of strictly less than 1ulp, and therefore $m^- \leq M_r^- \leq M_r^+ \leq M_r^+ \leq m^+$. As a consequence $m^- \leq V \leq m^+$.

Grisu2 does not give any guarantees on the shortness of its result. Its result is the shortest possible number in the interval $M_r^- \leq V \leq M_r^+$. The variables $M_r^- \leq V \leq M_r^+$ are dependent on $\text{diy}\_fp$'s precision $q$. The higher $q$, the closer $M_r^- \leq V \leq M_r^+$ are to the actual boundaries $m^- \leq m^+$. For $q = 64$ and $p = 53$ (as in our code samples) Grisu2 produces the shortest number for approximately 99.9% of its input.

The C implementation of Grisu2 is again cut into two parts: a core routine, independent of the chosen $\alpha, \gamma$, and a digit-generation procedure that needs to be tuned for the chosen target exponents. The core procedure is straightforward and we will therefore omit its C implementation.

In Figure 7, we present a version of the digit-generation routine tuned for $\alpha, \gamma = -59, -32$. The input variables $M_p$, and $delta$ correspond to $M_r^- \leq V \leq M_r^+$ and $\Delta$ respectively. The $\text{len}$ and $K$ are used as return values (with obvious meanings). We assume that $K$ has been initialized with $k$. We hence only need to add the $\alpha, \gamma$.

The proposed implementation combines step 5 and 6. While trying all possible $\alpha, \gamma$ (starting from the “top”) it generates the digits of $M_r^+ \leq V \leq M_r^-$. There are two digit-generation loops. One for the most-significant digits $\left[m_\alpha / 10^\alpha\right]$, stored in $p1$, and one for the least-significant digits $\text{mod} \; \text{one}$, stored in $p2$. Let $R$ be the number that is obtained by reading the generated digits ($R := 0$ if no digit has been generated yet). Then the following invariants holds for both loops (line 9 and line 17): $R = \left[m_\alpha / 10^\alpha\right]$. For the first loop we can show that $p1 \times \text{one} + p2 = M_p \mod 10^{\text{app.p}}$. The equation in line 13 thus tests if $M_r^- \mod 10^\alpha \leq \delta$.

The following invariant holds for the second loop (line 17): $p2 = \frac{M_r^+ - 1}{10^{\text{app.p}}} \mod \text{one}$.

6.3 Grisu3

Given enough extra precision, Grisu2 computes the best result (still with respect to shortness) for a significant percentage of its input. However there are some numbers where the optimal result lies outside the conservative approximated boundaries. In this section we present Grisu3, an alternative to Grisu2. It will not be able to produce optimal results for these numbers either, but it reports failure when it detects that a shorter number lies in the uncertain region. We denote with “uncertain region” the interval around the approximated boundaries that might, or might not be inside the boundaries. That is, it represents the error introduced by Grisu3’s imprecision.

Until now, optimality was defined with respect to the leading length (and of course accuracy) of the generated number $V$. For Grisu3 we add “closeness” as additional desired property: whenever there are several different numbers that are optimal with respect to shortness, Grisu3 should chose the one that is closest to $V$.

Instead of generating a valid number and then verifying if it is the shortest possible, Grisu3 will produce the shortest number inside the enlarged interval and verify if it is valid. Whereas Grisu2 used a conservative approximation of $m^-$ and $m^+$, Grisu3 uses a liberal approximation and then, at the end, verifies if its result lies in the conservative interval, too.

Algorithm Grisu3

Input and preconditions: same as for Grisu2.

Output: failure, or decimal digits $d_i$ for $i \leq 0 \leq n$ and an integer $K$ such that the integer value $V = d_0 \ldots d_n \times 10^k$ verifies $\sqrt{v} = v$. $V$ has the shortest leading length of all numbers verifying this property. If more than one number has the shortest leading length, then $V$ is the closest to $v$.

Procedure:

1-2. same as for Grisu2.

2b. Conversion: determine the normalized diy_fp w such that $w = v$.

3-4. same as for Grisu2.

4b. Product2: let $M_r^- := M_r^- - 1 \ulp$, $M_r^+ := M_r^+ + 1 \ulp$, and $\Delta := M_r^- - M_r^+$.
Digit Length: find the greatest $\kappa$ such that $M_i^+ \mod 10^\kappa < \Delta$.

6. Round: compute $\tilde{W} := w_0 + \delta$, and let $W := \tilde{W} - 1 \ulp$, and $W_1 := \tilde{W} + 1 \ulp$. Set $P_i := \lfloor M_i^+ / wp \rfloor - i$ for $i \geq 0$. Let $m$ be the greatest integer that verifies $P_m \times 10^m > M_i^+$. Let $u, 0 \leq u \leq m$ the smallest integer such that $|P_u \times 10^u - W_1|$ is minimal. Similarly let $d, 0 \leq d \leq m$ the largest integer such that $|P_d \times 10^d - W| = \min$. If $u \neq d$ return failure, else set $P := P_u$.

7. Weed: if not $M_i^+ \leq P \times 10^m \leq M_i^+$ return failure.

8. Output: define $V := P \times 10^{k+n}$. The decimal digits $d_i$ and $n$ are obtained by producing the decimal representation of $P$ (an integer). Set $K := k + \kappa$, and return it with the $n$ digits $d_i$.

Grisu3 uses the liberal boundary approximations ($M_i^-$ and $M_i^+$) instead of the conservative ones ($M_i$ and $M_i^+$). These values are guaranteed to lie outside the actual interval $m^{-}$ to $m^{+}$. The test in step 5 therefore features a strict inequality. This way $M_i^-$ is excluded from consideration. There is however no mechanism to exclude $M_i^+$. In the rare event when $M_i^+ \mod 10^m = 0$ then Grisu3 will, at this point of the algorithm, wrongly assume that $M_i^+$ is a potentially valid representation of $v$. Since $M_i^+$ lies outside the conservative region Grisu3 would return the failure in step 7. This case is rare and counter-measures are expensive, so we decided to accept this flaw.

Rounding is performed in step 6. Grisu3 simply tries all possible numbers with the same leading length and picks the one that is closest to $v$. At this stage Grisu3 works with approximated values and $\tilde{W}$, the approximation of $v$, may have an error of up to $1 \ulp$. The closest representation $P \times 10^m$ must not only be the closest to $\tilde{W}$ but to all possible values in $v$'s margin of error. Grisu3 first finds the closest $P_u$ to the upper boundary, and $P_d$ for the lower boundary. If they are not the same, then the precision is not enough to determine the optimal rounding and Grisu3 aborts.

Finally, just before outputting the computed representation, Grisu3 verifies if the best pick is within the conservative boundaries. If it is not, then the optimal solution lies in the uncertain region and Grisu3 returns failure.

![Figure 8: Grisu3's round-and-weed procedure.](image)

The digit-generation routine of Grisu2 has to be modified to take the larger liberal boundary interval into account, but these changes are minor and obvious. The interesting difference can be summarized in the `round_weed` procedure shown in Figure 8 which combines step 6 and 7. The function is invoked with the parameters set to the following values: $\text{buffer} := \text{do...do}_m$, where $\text{do}_m$ are the decimal digits of $\left\lfloor \frac{w_0}{\text{wp}} \right\rfloor$, $\text{wp} := \tilde{W} - \tilde{W} \Delta$, $\text{rest} := W_1^+ \mod 10^m$, $\text{ten}_\kappa := 10^m$, and $\ulp$ the value of $1 \ulp$ relative to all passed DIY functions.

Let $T := \text{do}_m \times 10^m$. By construction $T$ lies within the unsafe interval: $W_1^- < T < W_1^+$. Furthermore, among all possible values in this interval, it has the shortest leading length $len$. If there are other possible values with the same leading length and in the same interval, then they are smaller than $T$.

The loop in line 7 iterates all possible alternatives to find the closest to $W_t$. The first test, $\text{rest} < \text{wp} \ulp$, ensures that $T$ is not already less than $W_1$ (in which case the current $T$ must be the closest). Then follows a verification that the next lower number with the same leading length is still in the interval $W_1^- \leq T \leq W_1^+$.

If all tests succeed, then the number $\text{do}_m \times 10^m$ is guaranteed to lie inside the interval $W_1^- \leq T \leq W_1^+$ and is furthermore closer to $\tilde{W}_t$ than the current $T$. The body of the loop thus replaces $T$ (physically modifying the buffer) with its smaller alternative.

The if in line 14 then verifies the chosen $T$ is also closest to $W_t$. If this check fails then there are at least two candidates that could be the closest to $\tilde{W}$ and Grisu3 returns failure.

Now that the buffer has been correctly rounded a final weeding test in line 19 verifies that $W_1^- \leq T \leq W_1^+$. That is, that the chosen $T$ is inside the safe conservative interval.

7. Benchmarks

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>R</th>
<th>R/PP</th>
<th>S</th>
<th>S/PP</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sprintf %c</code></td>
<td>2.60</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>sprintf %g</code></td>
<td>22.83</td>
<td>-</td>
<td>24.17</td>
<td>-</td>
</tr>
<tr>
<td><code>sprintf %17e</code></td>
<td>36.03</td>
<td>-</td>
<td>36.17</td>
<td>-</td>
</tr>
<tr>
<td><code>grisu Fig[4]</code></td>
<td>61.53</td>
<td>61.49</td>
<td>28.73</td>
<td>28.66</td>
</tr>
<tr>
<td><code>grisu 0.3</code></td>
<td>9.17</td>
<td>-</td>
<td>9.98</td>
<td>-</td>
</tr>
<tr>
<td><code>grisu -63,-59</code></td>
<td>2.85</td>
<td>-</td>
<td>3.26</td>
<td>-</td>
</tr>
<tr>
<td><code>grisu -35,-32</code></td>
<td>3.36</td>
<td>-</td>
<td>3.77</td>
<td>-</td>
</tr>
<tr>
<td><code>grisu -59,-56</code></td>
<td>2.66</td>
<td>-</td>
<td>3.04</td>
<td>-</td>
</tr>
<tr>
<td><code>grisu2 -59,-56</code></td>
<td>2.50</td>
<td>-</td>
<td>2.96</td>
<td>-</td>
</tr>
<tr>
<td><code>grisu2b -59,-56</code></td>
<td>3.80</td>
<td>4.88</td>
<td>3.07</td>
<td>4.04</td>
</tr>
<tr>
<td><code>grisu3 -59,-56</code></td>
<td>5.38</td>
<td>6.42</td>
<td>4.40</td>
<td>5.48</td>
</tr>
<tr>
<td><code>grisu3b -59,-56</code></td>
<td>4.47</td>
<td>5.61</td>
<td>3.49</td>
<td>4.55</td>
</tr>
</tbody>
</table>

Figure 9: Speed of `sprintf`, Burger/Dybvig and Grisu.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>optimal</th>
<th>shortest</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>grisu2 -59,-56</code></td>
<td>0</td>
<td>99.92</td>
</tr>
<tr>
<td><code>grisu2b -59,-56</code></td>
<td>99.85</td>
<td>99.92</td>
</tr>
<tr>
<td><code>grisu3 -59,-56</code></td>
<td>99.49</td>
<td>99.49</td>
</tr>
</tbody>
</table>

Figure 10: Optimality of Grisu2 and Grisu3.
9. Conclusion

We have presented three new algorithms to print floating-point numbers: Grisu, Grisu2 and Grisu3. Given an integer type with at least two more bits than the input’s significand, then Grisu prints its input correctly. Grisu2 and Grisu3 are designed to benefit from integer types that have more than just two extra bits.

Grisu2 may be used where an optimal decimal representation is desired but not required. Given a 64 bit unsigned integer Grisu2 computes the optimal decimal representation 99.8% of the times for IEEE doubles.

Grisu3 should be used when the optimal representation is required. In this case Grisu3 will efficiently produce the optimal result for 99.5% of its input (with doubles and 64-bit integers), and reject the remaining numbers. The rejected numbers must then be printed by a more accurate (but slower) algorithm. Since Grisu3 is about 4 times faster than these alternative algorithms the average speed-up is still significant.

Grisu (and its evolutions) are furthermore straightforward to implement and do not feature many special cases. This is in stark contrast to efficient printing-algorithms that are based on bignums. Indeed, the major contributions to Dragon4 (one of the first published bignum-algorithms) have been the identification of special cases that could be handled more efficiently. We hope that Grisu3 renders this special cases uneconomic, thus simplifying the complete development of floating-point printing algorithms.

References


