Formatting floating-point numbers

Victor Zverovich
The origins

- Floating point arithmetic was "casually" introduced in 1913 paper "Essays on Automatics" by Leonardo Torres y Quevedo, a Spanish civil engineer and mathematician.

- Included in his 1914 electro-mechanical version of Charles Babbage's Analytical Engine.

Portrait of Torres Quevedo by Eulogia Merle (Fundación Española para la Ciencia y la Tecnología / CC BY-SA 4.0)
A bit of history

- 1938 Z1 by Konrad Zuse used 24-bit binary floating point
- 1941 relay-based Z3 had +/- infinity and exceptions (sort of)
- 1954 mass-produced IBM 704 introduced biased exponent

Replica of the Z1 in the German Museum of Technology in Berlin (BLueFiSH.as / CC BY-SA 3.0)
FORTRAN had formatted floating-point I/O in 1950s (same time as comments were invented!):

```
WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
601 FORMAT (4H A= ,I5,5H B= ,I5,5H C= ,I5, &
            8H AREA= ,F10.2, 13H SQUARE UNITS)
```
FP formatting in C


```c
/** print Fahrenheit-Celsius table */
   for f = 0, 20, ..., 300 */
main()
{
   int lower, upper, step;
   float fahr, celsius;

   lower = 0;    /* lower limit of temperature table */
   upper = 300;  /* upper limit */
   step = 20;    /* step size */

   fahr = lower;
   while (fahr <= upper) {
      celsius = (5.0/9.0) * (fahr-32.0);
      printf("%4.0f %6.1f\n", fahr, celsius);
      fahr = fahr + step;
   }
}
```

Still compiles in 2019: https://godbolt.org/z/KsOzjr
Solved problem?

• Floating point has been around for a while

• Programmers have been able to format and output FP numbers since 1950s

• Solved problem

• We all go home now
Solved problem?

- Floating point has been around for a while
- Programmers have been able to format and output FP numbers since 1950s
- Solved problem
- We all go home now
- Not so fast
Solved problem?

References


Solved problem?
Meanwhile in 2019

- Neither `stdio/printf` nor `iostreams` can give you the shortest decimal representation with round-trip guarantees.

- Performance has much to be desired, esp. with `iostreams` (can be 3x slower than `printf`!)

- Relying on global locale leads to subtle bugs, e.g. JSON-related errors reported by French but not English users.
Meanwhile in 2019

- Neither `stdio/printf` nor `iostreams` can give you the shortest decimal representation with round-trip guarantees.

- Performance has much to be desired, esp. with `iostreams` (can be 3x slower than `printf`!)

- Relying on global locale leads to subtle bugs, e.g. JSON-related errors reported by French but not English users :-(

Is floating point math broken?

Consider the following code:

```python
0.1 + 0.2 == 0.3  ->  false
```

```python
0.1 + 0.2  ->  0.30000000000000004
```

Why do these inaccuracies happen?
Is floating point math broken?

Consider the following code:

\[ 0.1 + 0.2 \equiv 0.3 \rightarrow \text{false} \]

\[ 0.1 + 0.2 \rightarrow 0.30000000000000004 \]

Why do these inaccuracies happen?
Floating-point math is not broken, but can be tricky

Formatting defaults are broken or at least suboptimal in C & C++ (loose precision):

```cpp
std::cout << (0.1 + 0.2) << " == " << 0.3 << " is "
  << std::boolalpha << (0.1 + 0.2 == 0.3) << "\n";
```

prints "0.3 == 0.3 is false"

The issue is not specific to C++ but some languages have better defaults: https://0.30000000000000004.com/
Desired properties

Steele & White (1990):

1. No information loss
2. Shortest output
3. Correct rounding
4. Left-to-right generation - irrelevant with buffering
No information loss

Round trip guarantee: parsing the output gives the original value.

Most libraries/functions lack this property unless you explicitly specify big enough precision: C stdio, C++ iostreams & to_string, Python's str.format until version 3, etc.

double a = 1.0 / 3.0;
char buf[20];
sprintf(buf, "%g", a);
double b = atof(buf);
assert(a == b);

// fails:
// a == 0.3333333333333333
// b == 0.333333

double a = 1.0 / 3.0;
auto s = fmt::format("{0}" , a);
double b = atof(s.c_str());
assert(a == b);

// succeeds:
// a == 0.3333333333333333
// b == 0.3333333333333333
Shortest output

The number of digits in the output is as small as possible.

It is easy to satisfy the round-trip property by printing unnecessary "garbage" digits (provided correct rounding):

```c
fprintf("%.17g", 0.1);
prints "0.10000000000000001"
```

```cpp
fmt::print("\{\}", 0.1);
prints "0.1"
```
Correct rounding

• The output is as close to the input as possible.

• Most implementations have this, but MSVC/CRT is buggy as of 2015 (!) and possibly later (both from and to decimal):
  
  

  • Had to disable some floating-point tests on MSVC due to broken rounding in printf and iostreams
C++17 introduced `<charconv>`

- Low-level formatting and parsing primitives:
  `std::to_chars` and `std::from_chars`

- Provides shortest decimal representation with round-trip guarantees and correct rounding!

- Locale-independent!
C++ users after `<charconv>` has been voted into C++17

(public domain)
std::array<char, 20> buf; // What size?
std::to_chars_result result =
    std::to_chars(buf.data(), buf.data() + buf.size(), M_PI);
if (result.ec == std::errc()) {
    std::string_view sv(buf.data(), result.ptr - buf.data());
    // Use sv.
} else {
    // Handle error.
}

- to_chars is great, but

- API is too low-level
  - Manual buffer management, doesn't say how much to allocate
  - Error handling is cumbersome (slightly better with structured bindings)

- Can't portably rely on it any time soon. May be widely available in 5 years or so (YMMV).
ISO C++ standard paper P0645 proposes a high-level formatting facility for C++20 (std::format and friends)

Implemented in the {fmt} library: https://github.com/fmtlib/fmt

The default is the shortest decimal representation with round-trip guarantees and correct rounding (will be enabled in the next release)

Control over locales: locale-independent by default

Example:

```cpp
fmt::print("{} == {} is {}
", 0.1 + 0.2, 0.3, 0.1 + 0.2 == 0.3);
```

prints "0.30000000000000004 == 0.3 is false" (no data loss)
How does it work?
IEEE 754

Binary floating point bit layout:

\[ v = \begin{cases} 
(-1)^{\text{sign}} \times \left(1 + \frac{\text{fraction}}{2^{255}}\right) \times 2^{(\text{exponent} - 127)} & \text{if } 0 < \text{exponent} < 1...1_2 \\
0 & \text{if } \text{exponent} = 0 \\
(-1)^{\text{sign}} \times 2^{(1-bias)} & \text{if } \text{exponent} = 1...1_2, \text{fraction} = 0 \\
(-1)^{\text{sign}} \times \text{Infinity} & \text{if } \text{exponent} = 1...1_2, \text{fraction} \neq 0 \\
\text{NaN} & \text{otherwise} 
\end{cases} \]
IEEE 754

Double-precision binary floating point bit layout:

\[
v = \begin{cases} 
(-1)^{\text{sign}} 1.\text{fraction} \times 2^{(\text{exponent} - \text{bias})} & \text{if } 0 < \text{exponent} < 1\ldots1_2 \\
(-1)^{\text{sign}} 0.\text{fraction} \times 2^{(1 - \text{bias})} & \text{if } \text{exponent} = 0 \\
(-1)^{\text{sign}} \text{Infinity} & \text{if } \text{exponent} = 1\ldots1_2, \text{fraction} = 0 \\
\text{NaN} & \text{if } \text{exponent} = 1\ldots1_2, \text{fraction} \neq 0
\end{cases}
\]

where \( \text{bias} = 1023 \)
Example

π approximation as double (\texttt{M_PI}):
Building up

- Integer
- Fraction

Fixed point

Floating point
Building up

- Integer
- Fraction
- Fixed point
- Floating point

Magic happens
constexpr uint64_t uint64_max_digits =
    std::numeric_limits<uint64_t>::digits10 + 1;

char* format_integer(char* out, uint64_t n) {
    char buf[uint64_max_digits];
    char *p = buf;
    do {
        *p++ = '0' + n % 10;
        n /= 10;
    } while (n != 0);
    do {
        *out++ = *--p;
    } while (p != buf);
    return out;
}
constexpr int max_precision = 17;

// Format a fraction (without "0.") stored in num_bits lower // bits of n.
char* format_fraction(char* out, uint64_t n, int num_bits, int precision = max_precision) {
    auto mask = (uint64_t(1) << num_bits) - 1;
    for (int i = 0; i < precision; ++i) {
        n *= 10;
        *out++ = '0' + (n >> num_bits); // n / pow(2, num_bits)
        n &= mask; // n % pow(2, num_bits)
    }
    return out;
}
Why 17?

- "17 digits ought to be enough for anyone" — some famous person

- In-and-out conversions, David W. Matula (1968):
  Conversions from base $B$ round-trip through base $\nu$ when $B^n < \nu^{m-1}$, where $n$ is the number of base $B$ digits, and $m$ is the number of base $\nu$ digits.

$$\left\lceil \log_{10}(2^{53}) + 1 \right\rceil = 17$$
Fractions

fractional part of \( M_{\Pi} \) (51 bits)

\[
0.001001000011111011101001000100010001011010001100
\]

cchar buf[max_precision + 1];
format_fraction(
    buf,
    0b001'0010'0001'1111'1011'0101'0100'0100'0100'0010'1101'0001'1000,
    51);

buf contains "14159265358979311" - fractional part of \( M_{\Pi} \) (last digit is a bit off, but round trips correctly)
Small exponent

// Formats a number represented as v * pow(2, e).
char* format_small_exp(char* out, uint64_t v, int e) {
    auto p = format_integer(out, v >> -e);
    auto int_digits = p - out;
    *p++ = '.';
    auto fraction_mask = (uint64_t(1) << -e) - 1;
    return format_fraction(p, v & fraction_mask, -e,
                           max_precision - int_digits);
}

auto bits = std::bit_cast<uint64_t>(M_PI);
auto fraction_bits = 52, bias = 1023;
auto implicit_bit = uint64_t(1) << fraction_bits;
auto v = (bits & (implicit_bit - 1)) | implicit_bit;
auto e = ((bits >> fraction_bits) & 0x7ff) - bias - fraction_bits;
char buf[max_precision + 1];
format_small_exp(buf, v, e);

buf contains "3.1415926535897931"
Here be dragons: full exponent, rounding, errors
Exponent

• Full exponent range: $10^{-324} - 10^{308}$

• In general requires multiple precision arithmetic

• glibc pulls in a GNU multiple precision library for printf:

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>57.96%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[. ] __printf_fp</td>
</tr>
<tr>
<td>15.28%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[. ] __mpn_mul_1</td>
</tr>
<tr>
<td>15.19%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[. ] __mpn_divrem</td>
</tr>
<tr>
<td>5.79%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[. ] hack_digit.13638</td>
</tr>
<tr>
<td>5.79%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[. ] vfprintf</td>
</tr>
</tbody>
</table>
Exponent

• Full exponent range: $10^{-324} - 10^{308}$

• In general requires multiple precision arithmetic

• glibc pulls in a GNU multiple precision library for printf:

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>57.96%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[.] __printf_fp</td>
</tr>
<tr>
<td>15.28%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[.] __mpn_mul_1</td>
</tr>
<tr>
<td>15.19%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[.] __mpn_divrem</td>
</tr>
<tr>
<td>5.79%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[.] hack_digit.13638</td>
</tr>
<tr>
<td>5.79%</td>
<td>a.out</td>
<td>libc-2.17.so</td>
<td>[.] vfprintf</td>
</tr>
</tbody>
</table>
• Family of algorithms from paper "Printing Floating-Point Numbers Quickly and Accurately with Integers" by Florian Loitsch (2004)

• DIY floating point: emulates floating point with extra precision (e.g. 64-bit for double giving 11 extra bits) using simple fixed-precision integer operations

• Precomputes powers of 10 and stores as DIY FP numbers

• Finds a power of 10 and multiplies the number by it to bring the exponent in the desired range

• With 11 extra bits Grisu3 produces shortest result in 99.5% of cases and tracks the uncertain region where it cannot guarantee shortness

• Relatively simple: Grisu2 can be implemented in 300 - 400 LOC incl. optimizations
DIY Floating Point

• DIY floating point:

```cpp
struct fp {
    uint64_t f; // fraction (with explicit 1)
    int e; // exponent

    fp(double d) {
        auto bits = std::bit_cast<uint64_t>(d);
        auto fraction_bits = 52, bias = 1023;
        auto implicit_bit = uint64_t(1) << fraction_bits;
        f = (bits & (implicit_bit - 1)) | implicit_bit;
        e = ((bits >> fraction_bits) & 0x7ff) - bias - fraction_bits;
        // Similarly for denormals
    }
};
```

• $x \otimes y$ - rounded multiplication of DIY FP numbers

• Unit in the last place (ulp) - value of the least significant digit if it is 1.
1. **Boundaries**: given FP number $v$, compute $v$'s boundaries $m^-$ and $m^+$. 

2. **Conversion**: convert $v$, $m^-$, and $m^+$ into DIY FPs $w$, $w^-$, and $w^+$ where $w$ and $w^+$ are normalized, $w^-$ and $w^+$ have the same exponent. 

3. **Cached Power**: find the normalized power of 10 $c_{-k}$ such that $\alpha \leq c_{-k}.e + w^+.e + q \leq \gamma$, where $[\alpha, \gamma]$ is a desired exponent range such as $[-60, -32]$. 

4. **Product**: compute $M^- := w^- \otimes c_{-k} - 1 \text{ulp}$, $M^+ := w^+ \otimes c_{-k} + 1 \text{ulp}$, $\Delta := M^+ - M^-$. 

5. **Digit Length**: find the greatest $\kappa$ such that $M^+ \mod 10^\kappa < \Delta$. 

6. **Round**: compute $W := w \otimes c_{-k}$, and let $W^- := W - 1 \text{ulp}$, and $W^+ := W + 1 \text{ulp}$. Set $P_i := \lfloor M^+ / 10^\kappa \rfloor - i$ for $i \geq 0$. Let $m$ be the greatest integer that verifies $P_m \times 10^\kappa > M^-$. Let $u$, $0 \leq u \leq m$ be the smallest integer such that $|P_u \times 10^\kappa - W^+|$ is minimal. Similarly let $d$, $0 \leq d \leq m$ be the largest integer such that $|P_d \times 10^\kappa - W^-|$ is minimal. If $u \neq d$ return failure, else set $P := P_u$. 

7. **Weed**: if not $w^- \otimes c_{-k} + 1 \text{ulp} \leq P \times 10^\kappa \leq w^+ \otimes c_{-k} - 1 \text{ulp}$, return failure. 

8. **Output**: define $V := P \times 10^{k+\kappa}$. 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 $d_i$. 

---

**Grisù3**
Grisù
Neighbors
Grisù

Numbers in \((m^-, m^+)\) round to \(v\)
Grisù

Convert to DIY FP (exact)
Grisù

Convert to DIY FP (exact)

Multiply by $c_k$ (error < 1ulp)
Grisù

Convert to DIY FP (exact)

Multiply by $c_k$ (error < 1ulp)

Numbers outside $(M^-, M^+)$ do not round to $v$
Grisù

Convert to DIY FP (exact)

Extract $\kappa$ most significant digits $P$ from $M^+$ such that $M^+ \mod 10^\kappa < \Delta$

Multiply by $c^{-k}$ (error < 1ulp)
Good stuff: $P \times 10^k$ in this interval rounds correctly, otherwise fallback
Grisù

Convert to DIY FP (exact)

Multiply by $c^{-k}$ (error < 1ulp)

Result: $P \times 10^{k+k}$
Powers of 10

Generating powers of 10 using Python's built-in arbitrary precision arithmetic (can be easily extended to negative powers scaling by $2^N$ for some big $N$):

```python
min_k = 4
max_k = 340
step = 8
for k in range(min_k, max_k + 1, step):
    binary = '{:b}'.format(10 ** k)
    f = (int('{:0<65}'.format(binary[:65], 65), 2) + 1) / 2
    e = len(binary) - 64
    print('fp(0x{:016x}, {})'.format(f, e))
```
Powers of 10

Generating powers of 10 using Python's built-in arbitrary precision arithmetic (can be easily extended to negative powers scaling by $2^N$ for some big $N$):

```python
min_k = 4
max_k = 340
step = 8
for k in range(min_k, max_k + 1, step):
    binary = '{:b}'.format(10 ** k)
    f = (int('{:0<65}'.format(binary[:65], 2)) + 1) / 2
    e = len(binary) - 64
    print('fp(0x{:016x}, {})'.format(f, e))
```

Output:
```
fp(0x9c40000000000000, -50)
fp(0xe8d4a51000000000, -24)
fp(0xad78ebc5ac620000, 3)
fp(0x813f3978f8940984, 30)
...```
Optimizations

- Many integer formatting optimization apply, e.g. reducing the number of integer divisions by computing the number of digits with `__builtin_clz` or faster method when processing integral part of $M^+$

- Use `__builtin_clz` for counting the number of leading zeros when normalizing subnormals

- Use 128-bit integers (e.g. `__uint128_t`) for multiplication by a cached power of 10

- Credit: Milo Yip (GitHub @miloyip)
1. Decode the floating point number

2. Compute the interval of information-preserving outputs similar to $(m^-, m^+)$ from Grisu

3. Convert the interval to a decimal power base using precomputed multipliers $\lfloor 2^k / 5^q \rfloor + 1$ and $\lfloor 5^{-(e-2)-q} / 2^k \rfloor$, where $e$ is the exponent

4. Determine the shortest, correctly-rounded string within this interval by repeated division skipping $q$ initial iterations by choosing appropriate multipliers in step 3

5. Print
Ryū

• Pros:
  • Doesn't need a fallback to guarantee shortness
  • Faster (~30% compared to a somewhat optimized implementation) than Grisù on random numbers; about the same perf on short output (6 digits or less)

• Cons:
  • Fairly complex
  • Larger precomputed tables: 10 KiB vs 870 bytes for double. For comparison all of {fmt} can fit in 30K on some platforms
  • Requires large integer arithmetic e.g. 128-bit for double (can be emulated)
Benchmarks

https://github.com/fmtlib/dtoa-benchmark
(based on miloyip/dtoa-benchmark)
Bechmarks

randomdigit

Time (ns) in log scale

Digit
References


• Grisù implementation: https://github.com/google/double-conversion

• Ryū implementation: https://github.com/ulfjack/ryu
Questions?