Fast Unrounded Scaling: Proof by Ivy

(*Floating Point Formatting, Part 4*)

Russ Cox
January 19, 2026
research.swtch.com/fp-proof

My post "Floating-Point Printing and Parsing Can Be Simple And Fast" depends on fast unrounded scaling, defined as:

$$\begin{aligned} \langle\, x \,\rangle &= \lfloor 2x \rfloor \;||\; (2x \neq \lfloor 2x \rfloor) \\ \text{uscale}(x, e, p) &= \left\langle\, x \cdot 2^e \cdot 10^p \,\right\rangle \end{aligned}$$

The unrounded form of $x \in \mathbb{R}$, $\langle\, x \,\rangle$, is the integer value of $\lfloor x \rfloor$ concatenated with two more bits: first, the "½ bit" from the binary representation of x (the bit representing 2^{-1}; 1 if $x - \lfloor x \rfloor \geq$ ½; or equivalently, $\lfloor 2x \rfloor$ mod 2); and second, a "sticky bit" that is 1 if *any* bits beyond the ½ bit were 1.

These are all equivalent definitions, using the convention that a boolean condition is 1 for true, 0 for false:

$$\begin{aligned} \langle\, x \,\rangle &= \lfloor x \rfloor \;||\; (x - \lfloor x \rfloor \geq ½) \;||\; (x - \lfloor x \rfloor \notin 0, ½) && (\text{'}||\text{' is bit concatenation}) \\ &= \lfloor x \rfloor \;||\; (x - \lfloor x \rfloor \geq ½) \;||\; (2x \neq \lfloor 2x \rfloor) \\ &= \lfloor 2x \rfloor \;||\; (2x \neq \lfloor 2x \rfloor) \\ &= \lfloor 4x \rfloor \;|\; (2x \neq \lfloor 2x \rfloor) && (\text{'}|\text{' is bitwise OR}) \\ &= \lfloor 4x \rfloor \;|\; (4x \neq \lfloor 4x \rfloor) \end{aligned}$$

The uscale operation computes the unrounded form of $x \cdot 2^e \cdot 10^p$, so it needs to compute the integer $\lfloor 2 \cdot x \cdot 2^e \cdot 10^p \rfloor$ and then also whether the floor truncated any bits. One approach would be to compute $2 \cdot x \cdot 2^e \cdot 10^p$ as an exact rational, but we want to avoid arbitrary-precision math. A faster approach is to use a floating-point approximation for 10^p: $10^p \approx pm \cdot 2^{pe}$, where pm is 128 bits. Assuming $x < 2^{64}$, this requires a single 64×128→192-bit multiplication, implemented by two full-width 64×64→128-bit multplications on a 64-bit computer.

The algorithm, which we will call Scale, is given integers x, e, and p subject to certain constraints and operates as follows:

Scale(x, e, p):

1. Let $pe = -127 - \lceil \log_2 10^{-p} \rceil$.
2. Let $pm = \lceil 10^p / 2^{pe} \rceil$, looked up in a table indexed by p.
3. Let $b = \text{bits}(x)$, the number of bits in the binary representation of x.
4. Let $m = e + pe + b - 1$.
5. Let $top = \lfloor x \cdot pm / 2^m / 2^b \rfloor$, $middle = \lfloor x \cdot pm / 2^b \rfloor \bmod 2^m$, $bottom = x \cdot pm \bmod 2^b$.
   Put another way, split $x \cdot pm$ into $top \,||\, middle \,||\, bottom$ where $bottom$ is b bits, $middle$ is m bits, and top is the remaining bits.
6. Return $\left\langle (top \,||\, middle) / 2^{m+1} \right\rangle$, computed as $top \,||\, (middle \neq 0)$ or as $top \,||\, (middle \geq 2)$.

The initial `uscale` implementation in the main post uses ($middle \neq 0$) in its result, but an optimized version uses ($middle \geq 2$).

This post proves both versions of Scale correct for the x, e, and p needed by the three floating-point conversion algorithms in the main post. Those algorithms are:

- `FixedWidth` converts floating-point to decimal. It needs to call Scale with a 53-bit x, $e \in [-1137, 960]$, and $p \in [-307, 341]$, chosen to produce a result $r \in [0, 2 \cdot 10^{18})$, which is at most 61 bits (62-bit output; $b = 53, m \geq 128 - 62 = 66$).

- `Short` also converts floating-point to decimal. It needs to call Scale with a 55-bit x, $e \in [-1137, 960]$, and $p \in [-292, 324]$, chosen to produce a result $r \in [0, 2 \cdot 10^{18})$, still at most 61 bits. (62-bit output; $b = 55, m \geq 128 - 62 = 66$).

- `Parse` converts decimal to floating-point. It needs to call Scale with a 64-bit x and $p \in [-343, 289]$, chosen to produce a result $r \in [0, 2^{54})$, which is at most 54 bits (55-bit output; $b = 64, m \geq 128 - 55 = 73$).

The “output” bit counts include the ½ bit but not the sticky bit. Note that for a given x and p, the maximum result size determines a relatively narrow range of possible e.

To start the proof, consider a hypothetical algorithm Scale$^{\mathbb{R}}$ that is the same as Scale except using exact real numbers. (Technically, only rationals are required, so this *could* be implemented, but it is only a thought experiment.)

Scale$^{\mathbb{R}}(x, e, p)$:

```
1. Let $pe = -127 - \lceil \log_2 10^{-p} \rceil$.
2. Let $pm^{\mathbb{R}} = 10^p / 2^{pe}$.
   (Note: $pm^{\mathbb{R}}$ is an exact value, not a ceiling.)
3. Let $b = \text{bits}(x)$.
4. Let $m = -e - pe - b - 1$.
5. Let $top^{\mathbb{R}} = \lfloor x \cdot pm^{\mathbb{R}} / 2^m / 2^b \rfloor$, $middle^{\mathbb{R}} = \lfloor x \cdot pm / 2^b \rfloor \bmod 2^m$, $bottom^{\mathbb{R}} = x \cdot pm \bmod 2^b$.
   (Note: $top^{\mathbb{R}}$ and $middle^{\mathbb{R}}$ are integers, but $bottom^{\mathbb{R}}$ is an exact value that may not be an integer.)
6. Return $\left\langle (top^{\mathbb{R}} \,||\, middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}}) / 2^{b+m+1} \right\rangle$, computed as $top^{\mathbb{R}} \,||\, (middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} \neq 0)$.
```

Using exact reals makes it straighforward to prove Scale$^{\mathbb{R}}$ correct.

Lemma 1. Scale$^{\mathbb{R}}$ computes uscale(x, e, p).

Proof. Expand the math in the final result:

$$\begin{aligned}
top^{\mathbb{R}} &= \lfloor x \cdot pm^{\mathbb{R}}/2^m/2^b \rfloor && \text{[definition of } top^{\mathbb{R}}\text{]}\\
&= \lfloor x \cdot 10^p/2^{pe}/2^m/2^b \rfloor && \text{[definition of } pm^{\mathbb{R}}\text{]}\\
&= \lfloor x \cdot 10^p/2^{pe}/2^{-e-pe-b-1}/2^b \rfloor && \text{[definition of } m\text{]}\\
&= \lfloor x \cdot 10^p \cdot 2^{-pe+e+pe+b+1-b} \rfloor && \text{[rearranging]}\\
&= \lfloor x \cdot 10^p \cdot 2^{e+1} \rfloor && \text{[simplifying]}\\
&= \lfloor 2 \cdot x \cdot 2^e \cdot 10^p \rfloor && \text{[rearranging]}\\
middle^{\mathbb{R}} \neq 0 \text{ or } bottom^{\mathbb{R}} \neq 0 &= \lfloor x \cdot pm/2^b \rfloor \bmod 2^m \neq 0 \text{ or } x \bmod 2^b \neq 0 && \text{[definition of } middle^{\mathbb{R}}, bottom^{\mathbb{R}}\text{]}\\
&= x \cdot pm^{\mathbb{R}} \bmod 2^{m+b} \neq 0 && \text{[simplifying]}\\
&= \lfloor x \cdot pm^{\mathbb{R}}/2^{m+b} \rfloor \neq x \cdot pm^{\mathbb{R}}/2^{m+b} && \text{[definition of floor and mod]}\\
&= \lfloor 2 \cdot x \cdot 2^e \cdot 10^p \rfloor \neq 2 \cdot x \cdot 2^e \cdot 10^p && \text{[reusing expansion of } x \cdot pm^{\mathbb{R}}/2^m/2^b \text{ above]}\\
\text{Scale}^{\mathbb{R}} &= top^{\mathbb{R}} \,||\, (middle^{\mathbb{R}} \neq 0 \text{ or } bottom^{\mathbb{R}} \neq 0) && \text{[definition of Scale}^{\mathbb{R}}\text{]}\\
&= \lfloor 2 \cdot x \cdot 2^e \cdot 10^p \rfloor \,||\, \lfloor 2 \cdot x \cdot 2^e \cdot 10^p \rfloor \neq 2 \cdot x \cdot 2^e \cdot 10^p && \text{[applying previous two expansions]}\\
&= \langle x \cdot 2^e \cdot 10^p \rangle && \text{[definition of } \langle \dots \rangle\text{]}\\
&= \text{uscale}(x, e, p) && \text{[definition of scale]}
\end{aligned}$$

So $\text{Scale}^{\mathbb{R}}(x, e, p)$ computes $\text{uscale}(x, e, p)$. ∎

Next we can establish basic conditions that make Scale correct.

Lemma 2. If $top = top^{\mathbb{R}}$ and $(middle \neq 0) \equiv (middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} \neq 0)$, then Scale computes $\text{uscale}(x, e, p)$.

Proof. $\text{Scale}^{\mathbb{R}}(x, e, p) = top^{\mathbb{R}} \,||\, (middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} \neq 0)$, while $\text{Scale}(x, e, p) = top \,||\, (middle \neq 0)$. If $top = top^{\mathbb{R}}$ and $(middle \neq 0) \equiv (middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} \neq 0)$, then these expressions are identical. Since $\text{Scale}^{\mathbb{R}}(x, e, p)$ computes $\text{uscale}(x, e, p)$ (by Lemma 1), so does $\text{Scale}(x, e, p)$. ∎

Now we need to show that $top = top^{\mathbb{R}}$ and $(middle \neq 0) \equiv (middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} \neq 0)$ in all cases. We will also show that $middle \neq 1$ to justify using $middle \geq 2$ in place of $middle \neq 0$ when that's convenient.

Note that $pm = \lceil pm^{\mathbb{R}} \rceil = pm^{\mathbb{R}} + \varepsilon_0$ for $\varepsilon_0 \in [0, 1)$, and so:

$$\begin{aligned}
x \cdot pm &= x \cdot (pm^{\mathbb{R}} + \varepsilon_0)\\
&= x \cdot pm^{\mathbb{R}} + \varepsilon_1, \qquad \varepsilon_1 = x \cdot \varepsilon_0 \in [0, 2^b)\\
top \,||\, middle \,||\, bottom &= top^{\mathbb{R}} \,||\, middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} + \varepsilon_1
\end{aligned}$$

The proof analyzes the effect of the addition of ε_1 to the ideal result $top^{\mathbb{R}} \,||\, middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}}$. Since $bottom^{\mathbb{R}}$ is b bits and ε_1 is at most b bits, adding $\varepsilon_1 > 0$ always causes $bottom \neq bottom^{\mathbb{R}}$. (Talking about the low b bits of a real number is unusual; we mean the low b integer bits followed by all the fractional bits: $x \cdot pm^{\mathbb{R}} \bmod 2^b$.)

The question is whether that addition overflows and propagates a carry into $middle$ or even top. There are two main cases: exact results ($middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} = 0$) and inexact results ($middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} \neq 0$).

Exact Results

Exact results have no error, making them match $\text{Scale}^{\mathbb{R}}$ exactly.

Lemma 3. For exact results, Scale computes uscale(x, e, p) and $middle \neq 1$.

Proof. For an exact result, $middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} = 0$, meaning $2 \cdot x \cdot 2^e \cdot 10^p$ is an integer and the sticky bit is 0. Since $bottom^{\mathbb{R}}$ is b zero bits, adding ε_1 affects *bottom* but does not carry into *middle* or *top*. Therefore $top = top^{\mathbb{R}}$ and $middle = middle^{\mathbb{R}} = 0$. The latter, combined with $bottom^{\mathbb{R}} = 0$, makes $(middle \neq 0) \equiv (middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} \neq 0)$ trivially true (both sides are false). By Lemma 2, Scale is correct. And since $middle = 0$, $middle \neq 1$. ∎

Inexact Results

Inexact results are more work. We will reduce the correctness to a few conditions on *middle*.

Lemma 4. For inexact results, if $middle \neq 0$, then Scale(x, e, p) computes uscale(x, e, p).

Proof. For an inexact result, $middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} \neq 0$. The only possible change from $middle^{\mathbb{R}}$ to *middle* is a carry from the error addition $bottom^{\mathbb{R}} + \varepsilon_1$ overflowing *bottom*. That carry is at most 1, so $middle = (middle^{\mathbb{R}} + \varepsilon_2) \bmod 2^m$ for $\varepsilon_2 \in [0, 1]$. An overflow into *top* leaves $middle = 0$. If $middle \neq 0$ then there can be no overflow, so $top = top^{\mathbb{R}}$. By Lemma 2, Scale computes uscale(x, e, p). ∎

For some cases, it will be more convenient to prove the range of $middle^{\mathbb{R}}$ instead of the range of *middle*. For that we can use a variant of Lemma 4.

Lemma 5. For inexact results, if $middle^{\mathbb{R}} \in [1, 2^m - 2]$ then Scale(x, e, p) computes uscale(x, e, p).

Proof. If $middle^{\mathbb{R}} \in [1, 2^m - 2]$, then $middle^{\mathbb{R}} + \varepsilon_2 \in [1, 2^m - 1]$, so the mod in $middle = (middle^{\mathbb{R}} + \varepsilon_2) \bmod 2^m$ does nothing (there is no overflow and wraparound), so $middle = middle^{\mathbb{R}} + \varepsilon_2 \geq 1$. By Lemma 4, Scale$(x, e, p)$ computes uscale(x, e, p). ∎

A related lemma helps with $middle \neq 1$.

Lemma 6. For inexact results, if $middle^{\mathbb{R}} \in [2, 2^m - 2]$, then $middle \geq 2$.

Proof. Again there is no overflow, so $middle \geq middle^{\mathbb{R}} \geq 2$. ∎

Now we need to prove either that $middle^{\mathbb{R}} \in [2, 2^m - 2]$ or that $middle \neq 0$ for all inexact results. We will consider four cases:

- [Small Positive Powers] $p \in [0, 27]$ and $b \leq 64$.
- [Small Negative Powers] $p \in [-27, -1]$ and $b \leq 64$.
- [Large Powers, Printing] $p \in [-400, -28] \cup [28, 400]$, $b \leq 55$, $m \geq 66$.
- [Large Powers, Parsing] $p \in [-400, -28] \cup [28, 400]$, $b \leq 64$, $m \geq 73$.

Small Positive Powers

Lemma 7. For inexact results and $p \in [0, 27]$ and $b \leq 64$, Scale(x, e, p) computes uscale(x, e, p) and $middle \neq 1$.

Proof. $5^p < 2^{63}$, so the non-zero bits of $pm^{\mathbb{R}}$ fits in the high 63 bits. That implies that the b+128-bit product $x \cdot pm^{\mathbb{R}} = top^{\mathbb{R}} \,||\, middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}}$ ends in 65 zero bits. Since $b \leq 64$, that means $bottom^{\mathbb{R}} = 0$ and $middle^{\mathbb{R}}$'s low bit is zero.

Because the result is inexact, $middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} \neq 0$, which implies $middle^{\mathbb{R}} \neq 0$ (since $bottom^{\mathbb{R}} = 0$). Since $middle^{\mathbb{R}}$'s low bit is zero, $middle^{\mathbb{R}} \in [2, 2^m - 2]$. By Lemma 5, Scale$(x, e, p)$ computes uscale(x, e, p). By Lemma 6, $middle \neq 1$. ∎

Small Negative Powers

Lemma 8. For inexact results and $p \in [-27, -1]$ and $b \leq 64$, Scale(x, e, p) computes uscale(x, e, p) and $middle \neq 1$.

Proof. Scaling by 2^e cannot introduce inexactness, since it just adds or subtracts from the exponent. The only inexactness must come from 10^p, specifically the 5^p part. Since $p < 0$ and $1/5$ is not exactly representable in a binary fraction, the result is inexact if and only if $x \bmod 5^{-p} \neq 0$ (remember that $-p$ is positive!).

Since $pm^{\mathbb{R}} \in [2^{127}, 2^{128})$ and $5^{-p} < 2^{-2}$, $pm^{\mathbb{R}} = 5^{-p} \cdot 2^k$ for some $k \geq 130$. Since $m + b \leq 128$, $top^{\mathbb{R}} = \lfloor x \cdot 2^{k-(m+b)} / 5^{-p} \rfloor$ and $middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} = 2^{m+b} \cdot (x \cdot 2^{k-(m+b)} \bmod 5^{-p}) / 5^{-p}$. That is, $middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}}$ encodes some non-zero binary fraction with denominator 5^{-p}. Note also that $x \cdot pm$ is b+128 bits and the output is at most 64 bits we have $m \geq 64$, so $2^m \cdot 2^{-63} \geq 2$.

That implies

$$\begin{aligned} middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} &\in 2^{m+b} \cdot (5^{-27}, 1 - 5^{-27}) \\ &\subset 2^{m+b} \cdot (2^{-63}, 1 - 2^{-63}) \\ middle^{\mathbb{R}} &\in 2^m \cdot (2^{-63}, 1 - 2^{-63}) \\ &\subset (2, 2^{m-2}) \end{aligned}$$

By Lemma 5 and Lemma 6, Scale(x, e, p) computes uscale(x, e, p) and $middle \neq 1$. ∎

Large Powers

That leaves $p \in [-400, -28] \cup [28, 400]$. There are not many pm to check—under a thousand—but there are far too many x to exhaustively test that $middle \geq 2$ for all of them. Instead, we will have to be a bit more clever.

It would be simplest if we could prove that all possible pm and all $x \in [1, 2^{64})$ result in a non-zero middle, but that turns out not to be the case.

For example, using $p = -29$, $x =$ 0x8e151cee6e31e067 is a problem, which we can verify using the Ivy calculator:

```
# hex x is the hex formatting of x (as text)
op hex x = '#x' text x

# spaced adds spaces to s between sections of 16 characters
op spaced s = (count s) <= 18: s; (spaced -16 drop s), ' ', -16 take s

# pe returns the binary exponent for 10**p.
op pe p = -(127+ceil 2 log 10**-p)

# pm returns the 128-bit mantissa for 10**p.
op pm p = ceil (10**p) / 2**pe p

spaced hex (pm -29) * 0x8e151cee6e31e067
```

```
0x7091bfc45568750f 0000000000000000 d81262b60aa6e8b7
```

We might perhaps think the problem is that 10^{-29} is too close to the small negative powers, but positive powers break too:

```
spaced hex (pm 31) * 0x93997b98618e62a1
```

```
0x918b5cd9fd69fdc5 0000000000000000 6d00000000000000
```

We might yet hope that the zeros were not caused by an error carry; then as long as we force the inexact bit to 1, we could still use the high bits. And indeed, for both of the previous examples, the zeros are not caused by an error carry: $\mathit{middle}^{\mathbb{R}}$ is all zeros. But that is not always the case. Here is a middle that is zero due to an error carry that overflowed into the top bits:

```
spaced hex (pm 62) * 0xd5bc71e52b31e483
spaced hex ((10**62) * 0xd5bc71e52b31e483) >> (pe 62)
```

```
0xcfd352e73dc6ddc3 0000000000000000 774bd77b38816199
0xcfd352e73dc6ddc2 ffffffffffffffff e6fdb9b19804952a
```

Instead of proving the completely general case, we will have to pick our battles and focus on the specific cases we need for floating-point conversions.

We don't need to try every possible input width below the maximum b. Looking at Scale, it is clear that the inputs x and $x \cdot 2^k$ have the same top and middle, and also that $\mathit{bottom}(x \cdot 2^k) = \mathit{bottom}(x) \cdot 2^k$. Since the middles are the same, the condition $\mathit{middle} \geq 2$ has the same truth value for both inputs. So we can limit our analysis to maximum-width b-bit inputs in $[2^{b-1}, 2^b)$. Similarly, we can prove that $\mathit{middle} \geq 2$ for $m \geq k$ by proving it for $m = k$: moving more bits from the low end of top to the high end of middle cannot make middle a smaller number.

Proving that $\mathit{middle} \geq 2$ for the cases we listed above means proving:

- [Printing] ($b \leq 55, m \geq 66.$)
  For all large p and all $x \in [2^{54}, 2^{55})$: $x \cdot pm \bmod 2^{55+66=121} \geq 2^{55+1=56}$.
- [Parsing] ($b \leq 64, m \geq 73.$)
  For all large p and all $x \in [2^{63}, 2^{64})$: $x \cdot pm \bmod 2^{64+73=137} \geq 2^{64+1=65}$.

To prove these two conditions, we are going to write an Ivy program to analyze each pm separately, proving that all relevant x satisfy the condition.

Ivy has arbitrary-precision rationals and lightweight syntax, making it a convenient tool for sketching and testing mathematical algorithms, in the spirit of Iverson's Turing Award lecture about APL, "Notation as a Tool of Thought." Like APL, Ivy uses strict right-to-left operator precedence: `1+2*3+4` means `(1+(2*(3+4)))`, and `floor 10 log f` means `floor (10 log f)`. Operators can be prefix unary like `floor` or infix binary like `log`. Each of the Ivy displays in this post is executable: you can edit the code and re-run them by clicking the Play button ("▶"). A full introduction to Ivy is beyond the scope of this post; see the Ivy demo for more examples.

We've already started the proof program above by defining `pm` and `pe`. Let's continue by defining a few more helpers.

First let's define `is`, an assertion for basic testing of other functions:

```
# is asserts that x === y.
op x is y =
    x === y: x=x
    print x '≠' y
    1 / 0

(1+2) is 3
```

```
(2+2) is 5
```

```
4 ≠ 5
```

```
input:1: division by zero
```

If the operands passed to `is` are not equal (the triple `===` does full vaue comparison), then `is` prints them out and divides by zero to halt execution.

Next, we will set Ivy's origin to 0 (instead of the default 1), meaning `iota` starts counting at 0 and array indexes start at 0, and then we will define `seq x y`, which returns the list of integers $[x, y]$.

```
)origin 0

# seq x y = (x x+1 x+2 ... y)
op seq (x y) = x + iota 1+y-x

(seq -2 4) is -2 -1 0 1 2 3 4
```

Now we are ready to start attacking our problem, which is to prove that for a given pm, b, and m, for all x, $middle \,||\, bottom = x \cdot pm \bmod 2^{b+m} \geq 2^{b+1}$, implying $middle \geq 2$, at which point we can use Lemma 4.

We will proceed in two steps, loosely following an approach by Vern Paxson and Tim Peters (the "Related Work" section explains the differences). The first step is to solve the "modular search" problem of finding the minimum $x \geq 0$ (the "first" x) such that $x \cdot c \bmod m \in [lo, hi]$. The second step is to use that solution to solve the "modular minimum" problem of finding an x in a given range that minimizes $x \cdot c \bmod m$.

Modular Search

Given constants c, m, lo, and hi, we want to find the minimum $x \geq 0$ (the "first" x) such that $x \cdot c \bmod m \in [lo, hi]$. This is an old programming contest problem, and I am not sure whether it has a formal name. There are multiple ways to derive a GCD-like efficient solution. The following explanation, based on one by David Wärn, is the simplest I am aware of.

Here is a correct $O(m)$ iterative algorithm:

```
op modfirst (c m lo hi) =
    xr x cx mx = 0 0 1 0
    :while 1
        # (A) xr ≤ hi but perhaps xr < lo.
        :while xr < lo
            xr x = xr x + c cx
        :end
        xr <= hi: x
        # (B) xr - c < lo ≤ hi < xr
        :while xr > hi
            xr x = xr x + (-m) mx
        :end
        lo <= xr: x
        # (C) xr < lo ≤ hi < xr + m
        x >= m: -1
    :end
```

The algorithm walks x forward from 1, maintaining $xr = x \cdot c \bmod m$:

- When xr is too small, it adds c to xr and increments x ($cx = 1$).
- When xr is too large, it subtracts m from xr and leaves x unchanged ($mx = 0$).
- When x reaches m, it gives up: there is no answer.

This loop is easily verified to be correct:

- It starts with $x = 0$ and considers successive x one at a time.
- While doing that, it maintains xr correctly:
    - If xr is too small, we *must* add a c (and increment x).
    - If xr is too large, we *must* subtract an m (and leave x alone).
- If $xr \in [lo, hi]$, it notices and stops.

The only problem with this `modfirst` is that it is unbearably slow, but we can speed it up.

At (A), $xr \le hi$, established by the initial $xr = 0$ or by the end of the previous iteration.

At (B), $xr - c < lo \le hi < xr$. Because $xr - c < lo$, subtracting $m \ge c$ will make xr too small; that will always be followed by at least $\lfloor m/c \rfloor$ additions of c. So we might as well replace m with $-m + c \cdot \lfloor m/c \rfloor$, speeding future trials. We will also have to update mx, to make sure x is maintained correctly.

At (C), $xr \le hi < xr + m$, and by a similar argument, we might as well replace c with $c - m \cdot \lfloor c/m \rfloor$, updating cx as well.

Making both changes to our code, we get:

```
op modfirst (c m lo hi) =
    xr x cx mx = 0 0 1 0
    :while 1
        # (A) xr ≤ hi but perhaps xr < lo.
        :while xr < lo
            xr x = xr x + c cx
        :end
        xr <= hi: x
        # (B) xr - c < lo ≤ hi < xr
        m mx = m mx + (-c) cx * floor m/c
        m == 0: -1
        :while xr > hi
            xr x = xr x + (-m) mx
        :end
        lo <= xr: x
        c cx = c cx + (-m) mx * floor c/m
        c == 0: -1
        # (C) xr < lo ≤ hi < xr+m
    :end
```

Notice that the loop is iterating (among other things) $m = m \bmod c$ and $c = c \bmod m$, the same as Euclid's GCD algorithm, so $O(\log\ c)$ iterations will zero c or m. The old test for $x \ge m$ (made incorrect by modifying m) is replaced by checking for c or m becoming zero.

Finally, we should optimize away the small `while` loops by calculating how many times each will be executed:

```
op modfirst (c m lo hi) =
    xr x cx mx = 0 0 1 0
    :while 1
        # (A) xr ≤ hi but perhaps xr < lo.
        xr x = xr x + c cx * ceil (0 max lo-xr)/c
        xr <= hi: x
        # (B) xr - c < lo ≤ hi < xr
        m mx = m mx + (-c) cx * floor m/c
        m == 0: -1
        xr x = xr x + (-m) mx * ceil (0 max xr-hi)/m
        lo <= xr: x
        c cx = c cx + (-m) mx * floor c/m
        c == 0: -1
        # (C) xr < lo ≤ hi < xr+m
    :end
```

Each iteration of the outer `while` loop is now $O(1)$, and the loop runs at most $O(\log\ c)$ times, giving a total time of $O(\log\ c)$, dramatically better than the old $O(m)$.

We can reformat the code to highlight the regular structure:

```
op modfirst (c m lo hi) =
    xr x cx mx = 0 0 1 0
    :while 1
        xr x  = xr x  +   c  cx * ceil (0 max lo-xr)/c ; xr <= hi : x
        m  mx = m  mx + (-c) cx * floor m/c            ; m == 0   : -1
        xr x  = xr x  + (-m) mx * ceil (0 max xr-hi)/m ; lo <= xr : x
        c  cx = c  cx + (-m) mx * floor c/m            ; c == 0   : -1
    :end

(modfirst 13 256 1 5) is 20   # 20*13 mod 256 = 4 ∈ [1, 5]
(modfirst 14 256 1 1) is -1   # impossible
```

We can also check that `modfirst` finds the exact answer from case 2, namely powers of five zeroing out the middle.

```
(modfirst (pm -3) (2**128) 1 (2**64)) is 125
spaced hex 125 * (pm -3)
```

```
0x40 0000000000000000 0000000000000042
```

Modular Minimization

Now we can solve the problem of finding the $x \in [xmax, xmin]$ that minimizes $x \cdot c \bmod m$.

Define the notation $x_R = x \cdot c \bmod m$ (the "residue" of x). We can construct $x \in [xmin, xmax]$ with minimal x_R with the following greedy algorithm.

1. Start with $x = xmin$.
2. Find the first $y \in [x+1, xmax]$ such that $y_R < x_R$.
3. If no such y exists, return x.
4. Set $x = y$.
5. Go to step 2.

The algorithm finds the right answer, because it starts at $xmin$ and then steps through every succesively better answer along the way to $xmax$. The algorithm terminates because every search is finite and every step moves x forward by at least 1. The only detail remaining is how to implement step 2.

For any x and y, $(x_R - y_R) \bmod m = (x - y)_R$, because multiplication distributes over subtraction. Call that the *subtraction lemma*.

Finding the first $y \in [x+1, xmax]$ with $y_R < x_R$ is equivalent to finding the first $d \in [1, xmax - x]$ with $(x+d)_R < x_R$. By the subtraction lemma, $d_R = ((x+d)_R - x_R) \bmod m$, so we are looking for the first $d \geq 1$ with $d_R \in [m - x_R, m-1]$. That's what `modfirst` does, except it searches $d \geq 0$. But $0_R = 0$ and we will only search for $lo \geq 1$, so `modfirst` can safely start its search at 0.

Note that if $d_R \in [m - (x+d)_R, m-1]$, the next iteration will choose the same d—any better answer would have been an answer to the original search. So after finding d, we should add it to x as many times as we can.

The full algorithm is then:

1. Start with $x = xmin$.
2. Use `modfirst` to find the first $d \geq 0$ such that $d_R \in [m - x_R, m-1]$.
3. If no d exists or $x + d > xmax$, stop and return x. Otherwise continue.
4. Let $s = m - d_R$, the number we are effectively subtracting from x_R.
5. Let n be the smaller of $\lfloor (xmax - x)/d \rfloor$ (the most times we can add d to x before exceeding our limit) and $\lfloor x_R/s \rfloor$ (the most times we can subtract s from x_R before wrapping around).
6. Set $x = x + d \cdot n$.
7. Go to step 2.

In Ivy, that algorithm is:

```
op modmin (xmin xmax c m) =
    x = xmin
    :while 1
        xr = (x*c) mod m
        d = modfirst c m, m - xr 1
        (d < 0) or (x+d) > xmax: x
        s = m - (d*c) mod m
        x = x + d * floor ((xmax-x)/d) min xr/s
    :end

(modmin 10 25 13 255) is 20
```

The running time of `modmin` depends on what limits n. If n is limited by $(xmax - x)/d$ then the next iteration will not find a usable d, since any future d would have to be bigger than the one we just found, and there won't be room to add it. On the other hand, if n is limited by x_R/s, then it means we reduced x_R at least by half. That limits the number of iterations to $\log_2 m$, and since `modfirst` is $O(\log m)$, `modmin` is $O(\log^2 m)$.

The subtraction lemma and `modfirst` let us build other useful operations too. One obvious variant of `modmin` is `modmax`, which finds the $x \in [xmin, xmax]$ that maximizes x_R and also runs in $O(\log^2 m)$.

We can extend `modmin` to minimize $x_R \geq lo$ instead, by stepping to the first $x_R \geq lo$ before looking for improvements:

```
op modminge (xmin xmax c m lo) =
    x = xmin
    :if (xr = (x*c) mod m) < lo
        d = modfirst c m (lo-xr) ((m-1)-xr)
        d < 0: :ret -1
        x = x + d
    :end
    :while 1
        xr = (x*c) mod m
        d = modfirst c m (m-(xr-lo)) (m-1)
        (d < 0) or (x+d) > xmax: x
        s = m - (d*c) mod m
        x = x + d * floor ((xmax-x)/d) min (xr-lo)/s
    :end

op modmin (xmin xmax c m) = modminge xmin xmax c m 0

(modmin 10 25 13 255) is 20
(modminge 10 25 13 255 6) is 21
(modminge 1 20 13 255 6) is 1
(modminge 10 20 255 255 1) is -1
```

We can also invert the search to produce `modmax` and `modmaxle`:

```
op modmaxle (xmin xmax c m hi) =
    x = xmin
    :if (xr = (x*c) mod m) > hi
        d = modfirst c m (m-xr) ((m-xr)+hi)
        d < 0: :ret -1
        x = x + d
    :end
    :while 1
        xr = (x*c) mod m
        d = modfirst c m 1 (hi-xr)
        (d < 0) or (x+d) > xmax: x
        s = (d*c) mod m
        x = x + d * floor ((xmax-x)/d) min (hi-xr)/s
    :end

op modmax (xmin xmax c m) = modmaxle xmin xmax c m (m-1)

(modmax 10 25 13 255) is 19
(modmaxle 10 25 13 255 200) is 15
```

Another variant is `modfind`, which finds the first $x \in [xmin, xmax]$ such that $x_R \in [lo, hi]$. It doesn't need a loop at all:

```
op modfind (xmin xmax c m lo hi) =
    x = xmin
    xr = (x*c) mod m
    (lo <= xr) and xr <= hi: x
    d = modfirst c m, (lo hi - xr) mod m
    (d < 0) or (x+d) > xmax: -1
    x+d

(modfind 21 100 13 256 1 10) is 40
```

We can also build `modfindall`, which finds all the $x \in [xmin, xmax]$ such that $x_R \in [lo, hi]$. Because there might be a very large number, it stops after finding the first 100.

```
op modfindall (xmin xmax c m lo hi) =
    all = ()
    :while 1
        x = modfind xmin xmax c m lo hi
        x < 0: all
        all = all, x
        (count all) >= 100: all
        xmin = x+1
    :end

(modfindall 21 100 13 256 1 10) is 40 79 99
```

Because `modfind` and `modfindall` both call `modfind` $O(1)$ times, they both run in $O(\log m)$ time.

Modular Proof

Now we are ready to analyze individual powers.

For a given pm, b, and m, we want to verify that for all $x \in [2^{b-1}, 2^b)$, we have $\mathit{middle} \geq 2$, or equivalently, $\mathit{middle} \,||\, \mathit{bottom} = x \cdot pm \bmod 2^{b+m} \geq 2^{b+1}$. We can use either `modmin` or `modfind` to do this. Let's use `modmin`, so we can show how close we came to failing.

We'll start with a function `check1` to check a single power, and `show1` to format its result:

```
# (b m) check1 p returns (p pm x middle fail) where pm is (pm p).
# If there is a counterexample to p, x is the first one,
# middle is (x*pm)'s middle bits, and fail is 1.
# If there is no counterexample, x middle fail are 0 0 0.
op (b m) check1 p =
    x = modmin (2**b-1) ((2**b)-1) (pm p) (2**b+m)
    middle = ((x * pm p) mod 2**b+m) >> b
    p (pm p) x middle (middle < 2)

# show1 formats the result of check1.
op show1 (p pm x middle fail) =
    p (hex pm) (hex x) (hex middle) ('.✗'[fail])

show1 64 64 check1 200
```
```
200 0xa738c6bebb12d16cb428f8ac016561dc 0xffe389b3cdb6c3d0 0x34 .
```

For $p = 200$, no 64-bit input produces a 64-bit middle less than 2.

On the other hand, for $p = -1$, we can verify that `check1` finds a multiple of 5 that zeroes the middle:

```
show1 64 64 check1 -1
```
```
-1 0xcccccccccccccccccccccccccccccccd 0x8000000000000002 0x0 ✗
```

Let's check more than one power, gathering the results into a matrix:

```
op (b m) check ps = mix b m check1@ ps
op show table = mix show1@ table

show 64 64 check seq 25 35
```
```
25 0x84595161401484a00000000000000000 0x8000000000000000 0x0 ✗
26 0xa56fa5b99019a5c80000000000000000 0x8000000000000000 0x0 ✗
27 0xcecb8f27f4200f3a0000000000000000 0x8000000000000000 0x0 ✗
28 0x813f3978f89409844000000000000000 0xec03c1a1aa24cc97 0x1 ✗
29 0xa18f07d736b90be55000000000000000 0xe06076f9cb96fe0d 0x5 .
30 0xc9f2c9cd04674edea400000000000000 0xfbd9be9d5bc8934e 0x1 ✗
31 0xfc6f7c40458122964d00000000000000 0x93997b98618e62a1 0x0 ✗
32 0x9dc5ada82b70b59df020000000000000 0xd0808609f474615a 0x2 .
33 0xc5371912364ce3056c28000000000000 0xc97002677c2de03f 0x0 ✗
34 0xf684df56c3e01bc6c732000000000000 0xc97002677c2de03f 0x0 ✗
35 0x9a130b963a6c115c3c7f400000000000 0xfd073be688a7dbaa 0x3 .
```

Now let's write code to show just the start and end of a table, to avoid very long outputs:

```
op short table =
    (count table) <= 15: table
    (5 take table) ,% ((count table[0]) rho box '...') ,% (-5 take table)

short show 64 64 check seq 25 95
```

```
 25 0x84595161401484a00000000000000000 0x8000000000000000 0x0   ❌
 26 0xa56fa5b99019a5c80000000000000000 0x8000000000000000 0x0   ❌
 27 0xcecb8f27f4200f3a0000000000000000 0x8000000000000000 0x0   ❌
 28 0x813f3978f894098440000000000000000 0xec03c1a1aa24cc97 0x1   ❌
 29 0xa18f07d736b90be55000000000000000 0xe06076f9cb96fe0d 0x5   .
...                              ...                ...  ... ...
 91 0x9d174b2dcec0e47b62eb0d64283f9c77 0xde861b1f480d3b9e 0x1   ❌
 92 0xc45d1df942711d9a3ba5d0bd324f8395 0xe621cb57290a897f 0x0   ❌
 93 0xf5746577930d6500ca8f44ec7ee3647a 0xa6bc10ca9dd53eff 0x1   ❌
 94 0x9968bf6abbe85f207e998b13cf4e1ecc 0xd011cce372153a65 0x0   ❌
 95 0xbfc2ef456ae276e89e3fedd8c321a67f 0xa674a3e92810fb84 0x0   ❌
```

We can see that most powers are problematic for $b = 64$, $m = 64$, which is why we're not trying to prove that general case.

Let's make it possible to filter the table down to just the bad powers:

```
op sel x = x sel iota count x
op bad table = table[sel table[;4] != 0]

short show bad 64 64 check seq -400 400
```

```
-400 0x95fe7e07c91efafa3931b850df08e739 0xe4036416c4b21bd6 0x0   ❌
-399 0xbb7e1d89bb66b9b8c77e266516cb2107 0xe4036416c4b21bd6 0x0   ❌
-398 0xea5da4ec2a406826f95daffe5c7de949 0xe4036416c4b21bd6 0x0   ❌
-397 0x927a87139a6841185bda8dfef9ceb1ce 0xfcdbd01bdf2d3eb2 0x0   ❌
-395 0xe4df730ea142e5b60f857dde6652f5d1 0x99535e222a18bc6d 0x0   ❌
 ...                              ...                ...  ... ...
 395 0x8f2bd39f334827e8c5874cc0ec691ba0 0xa462c66df06d90e3 0x0   ❌
 397 0xdfb47aa8c020be5bb4a367ed71643b2a 0x90ae62dc5a2282dd 0x0   ❌
 398 0x8bd0cca9781476f950e620f466dea4fb 0xd0be819cb0f1092e 0x0   ❌
 399 0xaec4ffd3d61994b7a51fa93180964e39 0xa6fece16f3f40758 0x0   ❌
 400 0xda763fc8cb9ff9e58e67937de0bbe1c7 0x8598a4df299005e0 0x0   ❌
```

Now we have everything we need. Let's write a function to try to prove that `scale` is correct for a given b and m.

```
op prove (b m) =
    table = bad b m check (seq -400 -28), (seq 28 400)
    what = 'b=', (text b), ' m=', (text m), ' t=', (text 127-m), '+½'
    (count table) == 0: print '✅ proved    ' what
    print '❌ disproved' what
    print short show table
```

This function builds a table of all the bad powers for b, m. If the table is empty, it prints that the settings have been proved. If not, it prints that the settings are unproven and prints some of the questionable powers.

If we try to prove 64, 64, we get many unproven powers.

```
prove 64 64
```

```
❌ disproved b=64 m=64 t=63+½
-400 0x95fe7e07c91efafa3931b850df08e739 0xe4036416c4b21bd6 0x0   ❌
-399 0xbb7e1d89bb66b9b8c77e266516cb2107 0xe4036416c4b21bd6 0x0   ❌
-398 0xea5da4ec2a406826f95daffe5c7de949 0xe4036416c4b21bd6 0x0   ❌
-397 0x927a87139a6841185bda8dfef9ceb1ce 0xfcdbd01bdf2d3eb2 0x0   ❌
-395 0xe4df730ea142e5b60f857dde6652f5d1 0x99535e222a18bc6d 0x0   ❌
 ...                              ...                ...  ...  ...
 395 0x8f2bd39f334827e8c5874cc0ec691ba0 0xa462c66df06d90e3 0x0   ❌
 397 0xdfb47aa8c020be5bb4a367ed71643b2a 0x90ae62dc5a2282dd 0x0   ❌
 398 0x8bd0cca9781476f950e620f466dea4fb 0xd0be819cb0f1092e 0x0   ❌
 399 0xaec4ffd3d61994b7a51fa93180964e39 0xa6fece16f3f40758 0x0   ❌
 400 0xda763fc8cb9ff9e58e67937de0bbe1c7 0x8598a4df299005e0 0x0   ❌
```

Large Powers, Printing

For printing, we need to prove $b = 55, m = 66$.

```
prove 55 66
```

```
✅ proved    b=55 m=66 t=61+½
```

It works! In fact we can shorten the middle to 64 bits before things get iffy:

```
prove 55 66
prove 55 65
prove 55 64
prove 55 63
prove 55 62
```

```
✅ proved    b=55 m=66 t=61+½
✅ proved    b=55 m=65 t=62+½
✅ proved    b=55 m=64 t=63+½
❌ disproved b=55 m=63 t=64+½
167 0xd910f7ff28069da41b2ba1518094da05 0x7b6e56a6b7fd53 0x0 ❌
❌ disproved b=55 m=62 t=65+½
167 0xd910f7ff28069da41b2ba1518094da05 0x7b6e56a6b7fd53 0x0 ❌
201 0xd106f86e69d785c7e13336d701beba53 0x68224666341b59 0x1 ❌
211 0xf356f7ebf83552fe0583f6b8c4124d44 0x69923a6ce74f07 0x0 ❌
```

Lemma 9. For $p \in [-400, -28] \cup [28, 400]$, $b \leq 55$, and $m \geq 66$, Scale(x, e, p) computes uscale(x, e, p) and *middle* $\neq 1$.

Proof. We calculated above that *middle* ≥ 2. By Lemma 4, Scale(x, e, p) computes uscale(x, e, p). ∎

Large Powers, Parsing

For parsing, we want to prove $b = 64, m = 73$.

```
prove 64 73
```

```
✅ proved    b=64 m=73 t=54+½
```

It also works! But we're right on the edge. Shortening the middle by one bit breaks the proof:

```
prove 64 73
prove 64 72
```

```
✅ proved    b=64 m=73 t=54+½
❌ disproved b=64 m=72 t=55+½
-93 0x857fcae62d8493a56f70a4400c562ddc 0xf324bb0720dbe7fe 0x1 ❌
```

Lemma 10. For $p \in [-400, -28] \cup [28, 400]$, $b \leq 64$, and $m \geq 73$, Scale(x, e, p) computes uscale(x, e, p) and $middle \neq 1$.

Proof. We calculated above that $middle \geq 2$. By Lemma 4, Scale(x, e, p) computes uscale(x, e, p). ∎

Bonus: 64-bit Input, 64-bit Output?

We don't need full 64-bit input and 64-bit output, but if we did, there is a way to make it work at only a minor performance cost. It turns out that for 64-bit input and 64-bit output, for any given p, considering all the inputs x that cause $middle = 0$, either all the middles overflowed or none of them did. So we can use a lookup table to decide how to interpret $middle = 0$.

The implementation steps would be:

1. Note that the proofs remain valid without $middle \neq 1$.
2. Don't make use of the $middle \neq 1$ optimization in the `uscale` implementation.
3. When p is large, force the sticky bit to 1 instead of trying to compute it from *middle*.
4. When $middle = 0$ for a large p, consult a table of hint bits indexed by p to decide whether *middle* has overflowed. If so, decrement *top*.

Here is a proof that this works.

First, define `topdiff` which computes the difference between *top* and $top^{\mathbb{R}}$ for a given b, m, p.

```
# topdiff computes top - topℝ.
op (b m p) topdiff x =
    top = (x * pm p) >> b+m
    topℝ = (floor x * (10**p) / 2**pe p) >> b+m
    top - topℝ
```

Next, define `hint`, which is like `check1` in that it looks for counterexamples. When it finds counterexamples, it computes `topdiff` for each one and reports whether they all match, and if so what their value is.

```
# (b m) hint p returns (p pm x middle fail) where pm is (pm p).
# If there is a counterexample to p, x is the first one,
# middle is (x*pm)'s middle bits, and fail is 1, 2, or 3:
#   1 if all counterexamples have top = topR
#   2 if all counterexamples have have top = topR+1
#   3 if both kinds of counterexamples exist or other counterexamples exist
# If there is no counterexample, x middle fail are 0 0 0.
op (b m) hint p =
    x = modmin (2**b-1) ((2**b)-1) (pm p) (2**b+m)
    middle = ((x * pm p) mod 2**b+m) >> b
    middle >= 1: p (pm p) x middle 0
    all = modfindall (2**b-1) ((2**b)-1) (pm p) (2**b+m) 0 ((2**b)-1)
    diffs = b m p topdiff@ all
    equal = or/ diffs == 0
    carry = or/ diffs == 1
    other = ((count all) >= 100) or or/ (diffs != 0) and (diffs != 1)
    p (pm p) x middle ((1*equal)|(2*carry)|(3*other))
```

Finally, define `hints`, which is like `show check`. It gets the hint results for all large p and prints a summary of how many were in four categories: no hints needed, all hints 0, all hints 1, mixed hints.

```
op hints (b m) =
    table = mix b m hint@ (seq -400 -28), (seq 28 400)
    (box 'b=', (text b), ' m=', (text m)), (+/ mix table[;4] ==@ iota 4)
```

Now we can try $b = 64, m = 64$:

```
hints 64 64
```

```
b=64 m=64 452 184 110 0
```

The output says that 452 powers don't need hints, 184 need a hint that $top^{\mathbb{R}} = top$, and 110 need a hint that $top^{\mathbb{R}} = top - 1$. Crucially, 0 need conflicting hints, so the hinted algorithm works for $b = 64, m = 64$.

Of course, that leaves 64 top bits, and since one bit is the ½ bit, this is technically only a 63-bit result. (If you only needed a truncated 64-bit result instead of a rounded one, you could use $e - 1$ and read the ½ bit as the final bit of truncated result.)

It turns out that hints are not enough to get a full 64 bits plus a ½ bit, which would leave a 63-bit middle. In that case, there turn out to be 63 powers where $middle = 0$ is ambiguous:

```
hints 64 63
```

```
b=64 m=63 241 283 159 63
```

However, if you only have 63 bits of input, then you can have the full 64-bit output:

```
hints 63 64
```

```
b=63 m=64 601 86 59 0
```

Completed Proof

Theorem 1. For the cases used in the printing and parsing algorithms, namely $p \in [-400, 400]$ with (printing) $b \le 55, m \ge 66$ and (parsing) $b \le 64, m \ge 73$, Scale is correct and $\mathit{middle} \ne 1$.

Proof. We proved these five cases:

- Lemma 3. For exact results, Scale computes uscale(x, e, p) and $\mathit{middle} \ne 1$.
- Lemma 7. For inexact results and $p \in [0, 27]$ and $b \le 64$, Scale(x, e, p) computes uscale(x, e, p) and $\mathit{middle} \ne 1$.
- Lemma 8. For inexact results, $p \in [-27, -1]$, and $b \le 64$, Scale(x, e, p) computes uscale(x, e, p) and $\mathit{middle} \ne 1$.
- Lemma 9. For $p \in [-400, -28] \cup [28, 400]$, $b \le 55$, and $m \ge 66$, Scale(x, e, p) computes uscale(x, e, p) and $\mathit{middle} \ne 1$.
- Lemma 10. For $p \in [-400, -28] \cup [28, 400]$, $b \le 64$, and $m \ge 73$, Scale(x, e, p) computes uscale(x, e, p) and $\mathit{middle} \ne 1$.

The result follows directly from these. ∎

A Simpler Proof

The proof we just finished is the most precise analysis we can do. It enables tricks like the hint table for 64-bit input and 64-bit output. However, for printing and parsing, we don't need to be quite that precise. We can reduce the four inexact cases to two by analyzing the exact $pm^{\mathbb{R}}$ values instead of our rounded pm values. We will show that the spacing around the exact integer results is wide enough that all the non-exact integers can't have middles near 0 or 2^m. The idea of proving a minimal spacing around the exact integer results is due to Michel Hack, although the proof is new.

Specifically, we can use the same machinery we just built to prove that $\mathit{middle}^{\mathbb{R}} \in [2, 2^{m+2}]$ for inexact results with $p \in [-400, 400]$, eliminating the need for Lemmas 7 and 8 by generalizing Lemmas 9 and 10. To do that, we analyze the non-zero results of $x \cdot pm^{\mathbb{R}} \bmod 2^{b+m}$. (If that expression is zero, the result is exact, and we are only analyzing the inexact case.)

Let's start by defining `gcd` and `pmR`, which returns `pn pd` such that $pm^{\mathbb{R}} = pn/pd$.

```
op x gcd y =
    not x*y: x+y
    x > y: (x mod y) gcd y
    x <= y: x gcd (y mod x)

(15 gcd 40) is 5
```

```
op pmR p =
    e = pe p
    num = (10**(0 max p)) * (2**-(0 min e))
    denom = (10**-(0 min p)) * (2**(0 max e))
    num denom / num gcd denom

(pmR -5) is (2**139) (5**5)
```

Let's also define a helper `zlog` that is like $\log_2 |x|$ except that `zlog 0` is 0.

```
op zlog x =
    x == 0: 0
    2 log abs x

(zlog 4) is 2
(zlog 0) is 0
```

Now we can write an exact version of `check1`. We want to analyze $x \cdot pm^{\mathbb{R}} \bmod 2^{b+m}$, but to use integers, instead we analyze $x_R = x \cdot pn \bmod pd \cdot 2^{b+m}$. We find the $x \in [xmin, xmax]$ with minimal $x_R > 0$ and the $y \in [xmin, xmax]$ with maximal y_R. Then we convert those to *middle* values by dividing by $pd \cdot 2^b$.

```
op (b m) check1R p =
    pn pd = pmR p
    xmin = 2**b-1
    xmax = (2**b)-1
    M = pd * 2**b+m
    x = modminge xmin xmax pn M 1
    x < 0: p 0 0 0 0
    y = modmax xmin xmax pn M
    xmiddle ymiddle = float ((x y * pn) mod M) / pd * 2**b
    p x y xmiddle ((xmiddle < 2) or (ymiddle > M-2))

op (b m) checkR ps = mix b m check1R@ ps

show1 64 64 check1 200
show1 64 64 check1R 200
```

```
200 0xa738c6bebb12d16cb428f8ac016561dc 0xffe389b3cdb6c3d0 0x34 .
200 0xffe389b3cdb6c3d0 0x8064104249b3c03e 0x1.9c2145p+05 .
```

```
op proveR (b m) =
    table = bad b m checkR (seq -400 400)
    what = 'b=', (text b), ' m=', (text m), ' t=', text ((127-1)-m),'+½'
    (count table) == 0: print '✅ proved    ' what
    print '❌ disproved' what
    print short show table
```

```
proveR 55 66
proveR 55 62
proveR 64 73
proveR 64 72
```

```
✅ proved    b=55 m=66 t=60 + ½
❌ disproved b=55 m=62 t=64 + ½
167 0x7b6e56a6b7fd53 0x463bc17af3f48e 0x1.817b1cp-02 ❌
201 0x68224666341b59 0x588220995c452a 0x1.8e0a91p-02 ❌
211 0x69923a6ce74f07 0x597216983bdc1a 0x1.14fbd3p-03 ❌
221 0x404a552daaaeea 0x50ad765f4fd461 0x1.de3812p+00 ❌
✅ proved    b=64 m=73 t=53 + ½
❌ disproved b=64 m=72 t=54 + ½
-93 0xf324bb0720dbe7fe 0xc743006eaf2d0e4f 0x1.3a8eb6p+00 ❌
```

The failures that `proveR` finds mostly correspond to the failures that `prove` found, except that `proveR` is slightly more conservative: the reported failure for $p = 221$ is a false positive.

```
prove 55 66
prove 55 62
prove 64 73
prove 64 72
```

```
✅ proved    b=55 m=66 t=61+½
❌ disproved b=55 m=62 t=65+½
167 0xd910f7ff28069da41b2ba1518094da05 0x7b6e56a6b7fd53 0x0 ❌
201 0xd106f86e69d785c7e13336d701beba53 0x68224666341b59 0x1 ❌
211 0xf356f7ebf83552fe0583f6b8c4124d44 0x69923a6ce74f07 0x0 ❌
✅ proved    b=64 m=73 t=54+½
❌ disproved b=64 m=72 t=55+½
-93 0x857fcae62d8493a56f70a4400c562ddc 0xf324bb0720dbe7fe 0x1 ❌
```

Lemma 11. For $p \in [-400, 400]$, $b \le 55$, and $m \ge 66$, Scale(x, e, p) computes uscale(x, e, p) and *middle* $\neq 1$.

Proof. Our Ivy code `proveR 55 66` confirmed that $middle^{\mathbb{R}} \in [2, 2^m - 2]$. By Lemma 5 and Lemma 6, Scale(x, e, p) computes uscale(x, e, p) and *middle* $\neq 1$. ∎

Lemma 12. For $p \in [-400, 400]$, $b \le 64$, and $m \ge 73$, Scale(x, e, p) computes uscale(x, e, p) and *middle* $\neq 1$.

Proof. Our Ivy code `proveR 64 73` confirmed that $middle^{\mathbb{R}} \in [2, 2^m - 2]$. By Lemma 5 and Lemma 6, Scale(x, e, p) computes uscale(x, e, p) and *middle* $\neq 1$. ∎

Theorem 2. For the cases used in the printing and parsing algorithms, namely $p \in [-400, 400]$ with (printing) $b \le 55, m \ge 66$ and (parsing) $b \le 64, m \ge 73$, Scale is correct and *middle* $\neq 1$.

Proof. Follows from Lemma 3, Lemma 11, and Lemma 12. ∎

Related Work

Parts of this proof have been put together in different ways for other purposes before, most notably to prove that exact *truncated* scaling can be implemented using 128-bit mantissas in floating-point parsing and printing algorithms. This section traces the history of the ideas as best I have been able to determine it. In these summaries, I am using the terminology and notation of this post—such as top, middle, bottom, x_R and $pm^{\mathbb{R}}$—for consistency and ease of understanding. Those terms and notations do not appear in the actual related work.

This section is concerned with the proof methods in these papers and only touches on the actual algorithms to the extent that they are relevant to what was proved. The main post's related work discusses the algorithms in more detail.

Paxson 1991

→ Vern Paxson, "A Program for Testing IEEE Decimal-Binary Conversion", class paper 1991.

The earliest work that I have found that linked modular minimization to floating-point conversions is Paxson's 1991 paper, already mentioned above and written for one of William Kahan's graduate classes. Paxson credits Tim Peters for the modular minimization algorithms, citing an email discussion on `validgh!numeric-interest@uunet.uu.net` in April 1991:

> In the following section we derive a modular equation which if minimized produces especially difficult conversion inputs; those that lie as close as possible to exactly half way between two representable outputs. We

then develop the theoretical framework for demonstrating the correctness of two algorithms developed by Tim Peters for solving such a modular minimization problem in O(log(N)) time.

I have been unable to find copies of the `numeric-interest` email discussion.

Peters broke down the minimization problem into a two step process, which I followed in this proof. Using this post's notation ($x_R = x \cdot c \bmod m$), the two steps in Paxson's paper (with two algorithms each) are:

- *FirstModBelow*: Find the first $x \geq 0$ with $x_R \leq hi$.
  FirstModAbove: Find the first $x \geq 0$ with $x_R \geq lo$.
- *ModMin*: Find the $x \in [xmin, xmax]$ that maximizes $x_R \leq hi$.
  ModMax: Find the $x \in [xmin, xmax]$ that minimizes $x_R \geq lo$.

(The names *ModMin* and *ModMax* seem inverted from their definitions, but perhaps "Min" refers to finding something below a limit and "Max" to finding something above a limit. They are certainly inverted from this post's usage.)

In contrast, this post's algorithms are;

- `modfirst`: Find the first $x \geq 0$ with $x_R \in [lo, hi]$.
- `modfind`: Find the first $x \in [xmin, xmax]$ with $x_R \in [lo, hi]$.
- `modmin`: Find the $x \in [xmin, xmax]$ that minimizes x_R.
- `modminge`: Find the $x \in [xmin, xmax]$ that minimizes $x_R \geq lo$.
- `modmax`: Find the $x \in [xmin, xmax]$ that maximizes x_R.

It is possible to use `modfirst` to implement Paxson's *FirstModBelow* and *FirstModAbove*, and vice versa, so they are equivalent in power.

In Paxson's paper, the implementation and correctness of *FirstModBelow* and *FirstModAbove* depend on computing the convergents of continued fractions of c/m and proving properties about them. Specifically, the result of *FirstModBelow* must be the denominator of a convergent or semiconvergent in the continued fraction for c/m, so it suffices to find the last even convergent p_{2i}/q_{2i} such that $(q_{2i})_R > hi$ but $(q_{2(i+1)})_R < hi$, and then compute the correct $q_{2i} + k \cdot q_{2i+1}$ by looking at how much $(q_{2i+1})_R$ subtracts from $(q_{2i})_R$ and subtracting it just enough times. I had resigned myself to implementing this approach before I found David Wärn's simpler proof of the direct GCD-like approach in `modfirst`. The intermediate steps in GCD(p, q) are exactly the continued fraction representation of p/q, so it is not surprising that both GCDs and continued fractions can be used for modular search.

No matter how `modfirst` is implemented, the critical insight is Peters's observaton that "find the first" is a good building block for the more sophisticated searches.

Paxson's *ModMin*/*ModMax* are tailored to a slightly different problem than we are solving. Instead of analyzing a particular multiplicative constant (a specific pm or $pm^{\mathbb{R}}$ value), Paxson is looking directly for decimal numbers as close as possible to midpoints between binary floating-point numbers and vice versa. That means finding x_R near $m/2$ modulo m. This post's proof is concerned with those values as well, but also the ones near integers. So we look for x_R near zero modulo $2m$, which is a little simpler. Paxson couldn't use that because it would find numbers near zero modulo m in addition to numbers near $m/2$ modulo m. The former are especially easy to round, so Paxson needs to exclude them. (In contrast, numbers near zero modulo m are a problem for Scale because the caller might want to take their floor or ceiling.)

Hanson 1997

→ Kenton Hanson, "Economical Correctly Rounded Binary Decimal Conversions", published online 1997.

The next analysis of floating-point rounding difficulty that I found is a paper published on the web by Kenton Hanson in 1997, reporting work done earlier at Apple Computer using a Macintosh Quadra, which perhaps dates it to the early 1990s. Hanson's web site is down and email to the address on the paper bounces. The link above is to a copy on the Internet Archive, but it omits the figures, which seem crucial to fully understanding the paper.

Hanson identified patterns that can be exploited to grow short "hard" conversions into longer ones. Then he used those longest hard conversions as the basis for an argument that conversion works correctly for all conversions up to that length: "Once this worst case is determined we have shown how we can guarantee correct conversions using arithmetic that is slightly more than double the precision of the target destinations."

Hanson focused on 113-bit floating-point numbers, using 256-bit mantissas for scaling, and only rounding conversions. I expect that his approach would have worked for proving that 53-bit floating-point numbers can be converted with 128-bit mantissas, but I have not reconstructed it and confirmed that.

Hack 2004

→ Gordon Slishman, "Fast and Perfectly Rounding Decimal/Hexadecimal Conversions", IBM Research Report, April 1990.
→ P.H. Abbott *et al.*, "Architecture and software support in IBM S/390 Parallel Enterprise Servers for IEEE Floating-Point arithmetic", *IBM Journal of Research and Development*, September 1999.
→ Michel Hack, "On Intermediate Precision Required for Correctly-Rounding Decimal-to-Binary Floating-Point Conversion", IBM Technical Paper, 2004.

The next similar discovery appears to be Hack's 2004 work at IBM.

In 1990, Slishman had published a conversion method that used floating-point approximations, like in this post. Slishman used a 16-bit middle and recognized that a non-`0xFFFF` *middle* implied the correctness of the top section. His algorithm fell back to a slow bignum implementation when *middle* was `0xFFFF` and carry error could not be ruled out (approximately $1/2^{16}$ of the time). (Hack defined *pm* to be a floor instead of a ceiling, so the error condition is inverted from ours.)

In 1999, Abbott *et al.* (including Hack) published a comprehensive article about the S/390's new support for IEEE floating-point (as opposed to its IBM hexadecimal floating point). In that article, they observed (like Paxson) that difficult numbers can be generated by using continued fraction expansion of $pm^{\mathbb{R}}$ values. They also observed that bounding the size of the continued fraction expansion would bound the precision required, potentially leading to bignum-free conversions.

Following publication of that article, Alan Stern initiated "a spirited e-mail exchange during the spring of 2000" and "pointed out that the hints at improvement mentioned in that article were still too conservative." As a result of that exchange, Hack launched a renewed investigation of the error behavior, leading to the 2004 technical report.

Hack's report only addresses decimal-to-binary (parsing) with a fixed-length input, not binary-to-decimal (printing), even though the comments in the 1999 article were about both directions and the techniques would apply equally well to binary-to-decimal.

In the terminology of this post, Hack proved that analysis of the continued fraction for a specific $pm^{\mathbb{R}}$ can establish a lower bound L such that $middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} < L$ if and only if $middle^{\mathbb{R}} \,||\, bottom^{\mathbb{R}} = 0$. For an n-digit decimal input, $L = 1/(10^n \cdot (k+2))$ where k is the maximum partial quotient in the continued fraction expansion of $pm^{\mathbb{R}}$ following certain convergents.

Hack summarizes:

> Using Continued Fraction expansions of a set of ratios of powers of two and five we can derive tight bounds on the intermediate precision required to perform correctly-rounding floating-point conversion: it is the sum of

> three components: the number of bits in the target format, the number of bits in the source format, and the number of bits in the largest partial quotient that follows a partial convergent of the "right" size among those Continued Fraction expansions. (This is in addition to the small number of bits needed to cover computational loss, e.g. when multiple truncating or rounding multiplications are performed.)
>
> When both source and target precision are fixed, the set of ratios to be expanded grows linearly with the target exponent range, and is small enough to permit a simple exhaustive search, in the case of the IEEE 754 standard formats: the extra number of bits (3rd component of the sum mentioned above) is 11 for 19-digit Double Precision and 13 for 36-digit Extended Precision.

I admit to discomfort with both Paxson's and Hack's use of continued fraction analysis. The math is subtle, and it seems easy to overlook a relevant case. For example Paxson needs semiconvergents for *FirstModBelow* but Hack does not explicitly mention them. Even though I trust that both Paxson's and Hack's results are correct, I do not trust myself to adapt them to new contexts without making unjustified mathematical assumptions. In contrast, the explicit GCD-like algorithm in `modfirst` and explicit searches based on it seem far less sophisticated and less error-prone to adapt.

Giulietti 2018

→ Raffaello Giulietti, "The Schubfach way to render doubles," published online, 2018, revised 2021.
→ Dmitry Nadhezin, nadezhin/verify-todec GitHub repository, published online, 2018.

Raffaello Giulietti developed the Schubfach algorithm while working on Java bug JDK-4511638, that `Double.toString` sometimes returned non-shortest results, most notably '9.999999999999999e22' for 1e23. Giulietti's original solution contained a fallback to multiprecision arithmetic in certain cases, and he wrote a paper proving the solution's correctness (I have been unable to find that original code, nor the first version of the paper, which was apparently titled "Rendering doubles in Java".)

Dmitry Nadhezin set out to formally check the proof using the ACL2 theorem prover. During that effort, Giulietti and Nadhezin came across Hack's 2004 paper and realized they could remove the multiprecision arithmetic entirely. Nadhezin adapted Hack's analysis and proved Giulietti's entire conversion algorithm correct using the ACL2 theorem prover. As part of that proof, Nadhezin proved (and formally verified) that the spacing around exact integer results that might arise during Schubfach's printing algorithm is at least $\varepsilon = 2^{-64}$ in either direction allowing the use of 126-bit pm values. (Using 126 instead of 128 is necessary because Java has only a signed 64-bit integer type.)

Adams 2018

→ Ulf Adams, "Ryū: Fast Float-to-String Conversion", ACM PLDI 2018.

Independent of Giulietti's work, Ulf Adams developed a different floating-point printing algorithm named Ryū, also based on 128-bit (or in Java, 126-bit) pm values. Adams proved the correctness of a computation for $\lfloor x/10^p \rfloor$ using $\lfloor pm^{\mathbb{R}} \rfloor$ for positive p and $\lceil pm^{\mathbb{R}} \rceil$ for negative p. Doubling x provides the ½ bit, but Ryū does not compute the sticky bit as part of that computation. Instead, Ryū computes an exactness bit (the inverse of the sticky bit) by explicitly testing $x \bmod 2^p = 0$ for $p > 0$ and $x \bmod 5^{-p} = 0$ for $p < 0$. The latter is done iteratively, requring up to 23 64-bit divisions in the worst case. (It is possible to reduce this to a single 64-bit multiplication by a constant obtained from table lookup, but Ryū does not.)

Like any of these proofs, Adams's proof of correctness of the truncated result needs to analyze specific pm or $pm^{\mathbb{R}}$ values. Adams chose to analyze the pm values and defined a function `minmax_euclid`(a, b, M) that returns the minimum and maximum values of $x \cdot a \bmod b$ for $x \in [0, M']$ for some $M' \geq M$ chosen by the algorithm. The paper includes a dense page-long proof of the correctness of `minmax_euclid`, but it must contain a mistake, since `minmax_euclid` turns out not to be correct. As one example, Junekey Jeon has pointed out that `minmax_euclid`(3, 8, 7) returns a minimum of 1 and maximum of 0. We can verify this by implementing `minmax_euclid` in Ivy:

```
op minmax_euclid (a b M) =
    s t u v = 1 0 0 1
    :while 1
        :while b >= a
            b u v = b u v - a s t
            (-u) >= M: :ret a b
        :end
        b == 0: :ret 1 (b-1)
        :while a >= b
            a s t = a s t - b u v
            s >= M: :ret a b
        :end
        a == 0: :ret 1 (b-1)
    :end

minmax_euclid 3 8 7
```

```
1 0
```

Jeon also points out that the trouble begins on the first line of Adams's proof, which claims that $a \le (-a) \bmod b$, but that is false for $a > b/2$. However, the general idea is right, and Adams's Ryū repository contains a more complex and apparently fixed version of the max calculation. Even corrected, the results are loose in two directions: they include x both smaller and larger than the exact range $[2^{b-1}, 2^b - 1)$.

Jeon 2020

→ Junekey Jeon, "Grisu-Exact: A Fast and Exact Floating-Point Printing Algorithm", published online, 2020.

In 2020, Jeon published a paper about Grisu-Exact, an exact variation of the Grisu algorithm without the need for a bignum fallback algorithm. Jeon relied on Adams's general proof approach but pointed out the problems with `minmax_euclid` mentioned in the previous section and supplied a replacement algorithm and proof of its correctness.

```
op minmax_euclid (a b M) =
    modulo = b
    s u = 1 0
    :while 1
        q = (ceil b/a) − 1
        b1 = b − q*a
        u1 = u + q*s
        :if M < u1
            k = floor (M−u) / s
            :ret a ((modulo − b) + k*a)
        :end
        p = (ceil a/b1) − 1
        a1 = a − p*b1
        s1 = s + p*u1
        :if M < s1
            k = floor (M−s) / u1
            :ret (a−k*b1) (modulo − b1)
        :end
        :if (b1 == b) and (a1 == a)
            :if M < s1 + u1
                :ret a1 (modulo − b1)
            :else
                :ret 0 (modulo − b1)
            :end
        :end
        a b s u = a1 b1 s1 u1
    :end

minmax_euclid 3 8 7
```

```
1 7
```

Lemire 2023

→ Daniel Lemire, "Number Parsing at a Gigabyte per Second", *Software—Pratice and Experience*, 2021.
→ Noble Mushtak and Daniel Lemire, "Fast Number Parsing Without Fallback", *Software—Pratice and Experience*, 2023.

In March 2020, Lemire published code for a fast floating-point parser for up to 19-digit decimal inputs using a 128-bit pm, based on an idea by Michael Eisel. Nigel Tao blogged about it in 2020 and Lemire published the algorithm in *Software—Practice and Experience* in 2021.

As published in 2021, Lemire's algorithm uses $pm = \lfloor pm^{\mathbb{R}} \rfloor$ and therefore checks for $middle = 2^{m-1}$ as a sign of possible inexactness. Upon finding that condition, the algorithm falls back to a bignum-based implementation.

In 2023, Mushtak and Lemire published a short but dense followup note proving that $middle = 2^{m-1}$ is impossible, and therefore the fallback check is unnecessary and can be removed. They address only the specific case of a 64-bit input and 73-bit middle, making the usual continued fraction arguments to bound the error for non-exact results.

Empirically, Mushtak and Lemire's computational proof does not generalize to other sizes. I downloaded their Python script and changed it from analyzing $N = m + b = 137$ to analyze other sizes and observed both false positives and false negatives. I believe the false negatives are from omitting semiconvergents (unnecessary for $N = 137$, as proved in their Theorem 2) and the false positives are from the approach not limiting x to the range $[2^{b-1}, 2^b - 1)$.

Jeon 2024

→ Junekey Jeon, "Dragonbox: A New Floating-Point Binary-to-Decimal Conversion Algorithm", published online, 2024.

In 2024, Jeon published Dragonbox, a successor to Grisu-Exact. Jeon changed from using the corrected `minmax_euclid` implementation to using a proof based on continued fractions. Algorithm C.14 ("Finding best rational approximations from below and above") is essentially equivalent to Paxson's algorithms. Like in Ryū and Grisu-Exact, the proof only considers the truncated computation $\lfloor x \cdot 2^e \cdot 10^p \rfloor$ and computes an exactness bit separately.

Conclusion

This post proved that Scale can be implemented correctly using a fast approximation that involves only a few word-sized multiplications and shifts. For printing and parsing of float64 values, computing the top 128 bits of a 64×128-bit multiplication is sufficient.

The fact that float64 conversions require only 128-bit precision has been known since at least Hanson's work at Apple in the mid-1990s, but that work was not widely known and did not include a proof. Paxson used an exact computational worst case analysis of modular multiplications to find difficult conversion cases; he did not bound the precision needed for parsing and printing. In contrast, Hack, Giulietti and Nadhezin, Adams, Mushtak and Lemire, and Jeon all derived ways to bound the precision needed for parsing or printing, but none of them used an exact computational worst case analysis that generalizes to arbitrary floating-point formats, and none recognized the commonality between parsing and printing.

The approach in this post, based on Paxson's general approach and built upon a modular analysis primitive by David Wärn, is the first exact analysis that generalizes to arbitrary formats and handles both parsing and printing.

In this post, I have tried to give credit where credit is due and to represent others' work fairly and accurately. I would be extremely grateful to receive additions, corrections, or suggestions at rsc@swtch.com.