Writing logf: From 2 Million ULPs to 4

Deval Deliwala

July 26, 2026

You know I first learned about catastrophic cancellation in a numerical analysis course during undergrad.

This note documents the first time I encountered it in my own code and the frustration it caused.

The Logarithm Function

After writing readable, accurate, and fast scalar and vectorized implementations of `expf(x)`, the next step is to do the same for `logf(x)`, which evaluates $\ln(x)$.

I learned a lot from reading ARM optimized-routines for the previous note, but I decided to attack this problem myself. This caused many of the problems that follow.

The basic steps are the same though:

1. Transform the problem into a more computationally friendly form.
2. Reduce the range.
3. Evaluate a minimax polynomial.

I'll start with my first implementation. It passed every accuracy test I had written. However, after comparing it against the standard-library `logf` over tens of millions of sampled inputs, I found a worst-case error of **2097155 ULPs**. One ULP is one step between adjacent representable `float32` values. An error of n ULPs means the result is n such steps from the reference result. This is atrocious. For comparison, my scalar implementation of `expf(x)` has a worst-case error of 3 ULPs.

Specifically:

```
maximum ULP: 2097155
from input: 1.000001
vex measurement: 7.152557373e-07
reference measurement: 5.960462772e-07
```

At $x \approx 1$, my result was only 1.19×10^{-7} away from the reference result. That absolute error looks small, but the relative error is **0.2**, or 20%. My result is 2097155 ULPs away.

My First Implementation

My idea was to decompose a positive x as

$$x = 2^{\varepsilon} f.$$

Here, ε is the true exponent of x and f is its significand. This decomposition follows directly from IEEE 754.

A normal `float32` stores a sign bit s, a biased exponent E, and 23 mantissa bits. Let M_2 be the fractional value of those mantissa bits. The represented value is

$$x = (-1)^s 2^{E-127}(1 + M_2).$$

Since x is positive, $s = 0$. I define

$$\varepsilon = E - 127, \qquad f = 1 + M_2 = 1.M_2.$$

Therefore, ε is the true, unbiased exponent and f is the significand. This gives my decomposition $x = 2^{\varepsilon} f$.

Using $\ln(ab) = \ln(a) + \ln(b)$,

$$\ln(x) = \ln(2^{\varepsilon} f) = \varepsilon \ln(2) + \ln(f).$$

I can calculate ε directly by reading the exponent field E and subtracting 127. In principle, I could calculate f with

$$f = \frac{x}{2^{\varepsilon}},$$

but division is expensive. I used a pretty slick bit trick instead.

This decomposition also gives some range reduction for free. Since ε is the true exponent of x,

$$2^{\varepsilon} \le x < 2^{\varepsilon+1}.$$

Dividing through by 2^{ε} gives

$$1 \le \frac{x}{2^{\varepsilon}} < 2,$$

so $f \in [1, 2)$. This interval has width 1, the same as the reduced interval $[-0.5, 0.5]$ in my `expf(x)` implementation.

I calculate $\varepsilon \ln(2)$ directly and approximate $\ln(f)$ with a minimax polynomial $P(f)$. The final result is

$$\ln(x) \approx \varepsilon \ln(2) + P(f).$$

My first version used this decomposition directly, but approximating $\ln(f)$ over $[1, 2)$ still required a very high-degree minimax polynomial. I reduced the range again by redefining

$$f = \frac{x}{2^{\varepsilon+1}}.$$

Now

$$x = 2^{\varepsilon+1} f,$$

so

$$\ln(x) = (\varepsilon + 1)\ln(2) + \ln(f).$$

Compared with the original decomposition, this new f is half as large. Since the original was in $[1, 2)$, the new f is in $[0.5, 1)$. I then set

$$r = f - 1,$$

which places r in $[-0.5, 0)$. The polynomial now approximates $\ln(f) = \ln(1 + r)$ over this smaller value interval.

I made this shift mainly so the polynomial could return $\ln(1) = 0$ exactly. If $f = 1$, then $r = 0$. Constraining the constant coefficient to $c_0 = 0$ forces the polynomial to return 0 at that point.

Moving the input interval closer to 0 may also help because `float32` values have smaller absolute spacing there. I have not tested whether that makes a meaningful difference, though. The exact value at $\ln(1)$ is the main reason.

```c
//! scalar/logf.c
//!
//! scalar implementation of `logf(x)`.


#include "vex.h"
#include "helpers.h"
#include <float.h>
#include <math.h>
#include <stdint.h>


/// Overwrites `x`'s exponent bits with 01111110
/// and returns the new float value.
///
/// If `x` = 2^e (1 + m), this is equivalent to
/// returning (1 + m) / 2.
float set_exponent_to_126(float x) {
    uint32_t bits = asuint(x);

    // overwrite exponent field with 126
    bits = (bits & 0x007fffffu) | (126u << 23);
    return asfloat(bits);
}


float vex_logf(float x) {
    const double LN2 = 0.69314718055994530941723212145817656 8;

    const float OVERFLOW_BOUND = FLT_MAX;
    const float UNDERFLOW_BOUND = FLT_MIN;

    // minimax remez coefficients for ln(1+r) on [-0.5, 0]
    // constrain c0 = 0.0 so ln(1) = ln(1 + 0) = 0;
    const double c1 =  0.99999985242729561;
    const double c2 = -0.50003654322887447;
    const double c3 =  0.33186261984516702;
    const double c4 = -0.27225053856081005;
    const double c5 =  0.038018061198611182;
    const double c6 = -0.78543102733181391;
    const double c7 = -1.0815401875157433;
    const double c8 = -1.1849314839059311;

    // checks
    if (isnan(x)) return x;
    if (x < 0.0f) return NAN;
    if (x < UNDERFLOW_BOUND) return -INFINITY;
    if (x > OVERFLOW_BOUND) return INFINITY;

    uint32_t bits = asuint(x);
    uint32_t biased_e = (bits >> 23) & 0xffu;

    // true exponent
    int32_t e = (int32_t) biased_e - 127;

    // f = 1.M_2 / 2
    // making the exponent 126 yields 2^-1 * 1.M_2
    float f = set_exponent_to_126(x);

    // r in [-0.5, 0]
    float r = f - 1.0;

    // evaluate minimax polynomial of ln(1 + r)
    float q = fmaf(r, c8, c7);
    q = fmaf(r, q, c6);
    q = fmaf(r, q, c5);
    q = fmaf(r, q, c4);
    q = fmaf(r, q, c3);
    q = fmaf(r, q, c2);
    q = fmaf(r, q, c1);
    float lnf = r * q;

    // ln(x) = ln(2^(e + 1)) + ln(f)
    return (float)((double) (e + 1) * LN2) + lnf;
}
```

A degree-8 minimax polynomial was still needed over this smaller interval.

The bit trick for calculating

$$f = \frac{x}{2^{\varepsilon+1}}$$

is `set_exponent_to_126(float x)`. It does exactly what the name says: it reads the raw bits of `x`, replaces the 8-bit exponent field with 126, or `01111110`, then reinterprets the result as a `float32`.

To see why this produces f, recall that

$$x = 2^{\varepsilon}(1 + M_2) \quad \to \quad \frac{x}{2^{\varepsilon}} = (1 + M_2).$$

Therefore, the value we want is

$$f = \frac{x}{2^{\varepsilon+1}} = \frac{1 + M_2}{2}.$$

The stored exponent 126 represents the true exponent

$$126 - 127 = -1.$$

Changing only the exponent field preserves the mantissa bits M_2. Therefore, replacing the exponent field with 126 makes the new float

$$2^{-1}(1 + M_2) = \frac{1 + M_2}{2} = f.$$

Testing ULPs

My quick test file literally just passes these values to both my implementation and the standard-library `logf(x)`, then compares the results:

```c
float inputs[] = {
    -INFINITY, -10.0f, -1.0f, 0.0f, 0.5f,
    1.0f, 2.0f, 10.0f, 80.0f, INFINITY
};
```

With my standard floating-point comparison, these tests only passed when I used a degree-8 minimax polynomial.

Clearly, these tests are not exhaustive. After they passed, I wrote a more rigorous test that uniformly samples millions of `float32` values and reports a histogram of the measured ULP errors.

The result is not forgiving:

```
scalar logf
maximum ULP: 2097155
from input: 1.000001
vex measurement: 7.152557373e-07
reference measurement: 5.960462772e-07
ulp distribution
0       75513458     75.513458%
1       24211489     24.211487%
2       164121       0.164121%
3       41349        0.041349%
...     ...          ...
253     4            0.000004%
254     3            0.000003%
255     1            0.000001%
256+    1052         0.001052%
total samples: 100000000
```

Again, the maximum error is over 2 million ULPs. My result was more than 2 million representable `float32` steps from the reference result. The worst input gives us a clue:

```
from input: 1.000001
```

When $x \approx 1$, my implementation is incredibly inaccurate.

Catastrophic Cancellation

Let's walk through the algorithm for $x = 1.01$.

The true exponent is $\varepsilon = 0$, so

$$f = \frac{x}{2^{\varepsilon+1}} = 0.505.$$

Hence,

$$\begin{aligned} \ln(x) &= (\varepsilon + 1)\ln(2) + \ln(f) \\ \ln(1.01) &= \ln(2) + \ln(0.505) \\ \ln(1.01) &\approx 0.693 - 0.683 = 0.01. \end{aligned}$$

Mathematically, this is clean. Numerically, though, this is **catastrophic cancellation**. We are subtracting two numbers that are very close in magnitude. Their leading digits cancel, so the result depends on the few trailing digits that remain. This causes a severe loss of precision.

The input that produced the 2-million-ULP error was $x = 1.000001$. For this input, the algorithm evaluated

$$0.69314718056 - 0.69314618056.$$

The two values differ by only 1.0×10^{-6}. No wonder the ULP error blew up.

This is a fundamental problem with my range reduction. Mathematically, it is clean and computationally friendly. Near 1, it is incredibly imprecise.

My New Implementation

The failed implementation always calculated

$$\ln(x) = (\varepsilon + 1)\ln(2) + \ln(f).$$

Near $x = 1$ from above, $\varepsilon = 0$ and $f \approx 0.5$. The two terms become $\ln(2)$ and approximately $-\ln(2)$, so they almost completely cancel.

The fix is to avoid this decomposition when $\varepsilon = 0$. Let

$$m = 1 + M_2$$

be the original significand, where $m \in [1, 2)$. We have two mathematically equivalent decompositions:

$$\begin{aligned} \ln(x) &= \varepsilon \ln(2) + \ln(m) \\ \ln(x) &= (\varepsilon + 1)\ln(2) + \ln\left(\frac{m}{2}\right). \end{aligned}$$

For $x = 1.01$, the first decomposition gives

$$\ln(1.01) = 0\ln(2) + \ln(1.01).$$

There is nothing to cancel. I therefore use the first decomposition when $\varepsilon = 0$ and the second everywhere else. Just below 1, $\varepsilon = -1$, so the second decomposition already makes $(\varepsilon + 1)\ln(2) = 0$. This avoids cancellation on both sides of 1.

The downside is that the code now needs two minimax polynomials. In the first branch, $f = m$ and

$$r = f - 1 \in [0, 1).$$

In the second branch, $f = \frac{m}{2}$ and

$$r = f - 1 \in [-0.5, 0).$$

Different intervals require different coefficients, so the code is a bit messier.

```c
//! scalar/logf.c
//!
//! scalar implementation of `logf(x)`.


#include "vex.h"
#include "helpers.h"
#include <assert.h>
#include <float.h>
#include <math.h>
#include <stdint.h>


float set_exponent_to(float x, uint32_t exponent) {
    assert(exponent < 128 && "exponent must be less than 128");
    uint32_t bits = asuint(x);

    bits = (bits & 0x007fffffu) | (exponent << 23);

    return asfloat(bits);
}


/// Minimax evaluation for f when x's true exponent is not 0.
float negative_poly(float r) {
    // minimax remez coefficients for ln(1 + r) on [-0.5, 0)
    const double c1 =  0.99999985242729561;
    const double c2 = -0.50003654322887447;
    const double c3 =  0.33186261984516702;
    const double c4 = -0.27225053856081005;
    const double c5 =  0.038018061198611182;
    const double c6 = -0.78543102733181391;
    const double c7 = -1.0815401875157433;
    const double c8 = -1.1849314839059311;

    float q = fmaf(r, c8, c7);
    q = fmaf(r, q, c6);
    q = fmaf(r, q, c5);
    q = fmaf(r, q, c4);
    q = fmaf(r, q, c3);
    q = fmaf(r, q, c2);
    q = fmaf(r, q, c1);
    float lnf = r * q;

    return lnf;
}

/// Minimax evaluation for f when x's true exponent is 0.
float positive_poly(float r) {
    // minimax remez coefficients for ln(1 + r) on [0, 1)
    const double c1  =  0.99999999528721917;
    const double c2  = -0.49999902422775899;
    const double c3  =  0.33329962416588654;
    const double c4  = -0.24954133655537836;
    const double c5  =  0.19675939255501274;
    const double c6  = -0.15305592566389509;
    const double c7  =  0.10603654877058005;
    const double c8  = -0.056959534201723228;
    const double c9  =  0.019851462335085494;
    const double c10 = -0.0032440251717337546;

    float q = fmaf(r, c10, c9);
    q = fmaf(r, q, c8);
    q = fmaf(r, q, c7);
    q = fmaf(r, q, c6);
    q = fmaf(r, q, c5);
    q = fmaf(r, q, c4);
    q = fmaf(r, q, c3);
    q = fmaf(r, q, c2);
    q = fmaf(r, q, c1);
    float lnf = r * q;

    return lnf;
}


float vex_logf(float x) {
    const double LN2 = 0.69314718055994530941723212145817656 8;

    const float OVERFLOW_BOUND = FLT_MAX;
    const float UNDERFLOW_BOUND = FLT_MIN;

    // checks
    if (isnan(x)) return x;
    if (x < 0.0f) return NAN;

    // no subnormals
    if (x < UNDERFLOW_BOUND) return -INFINITY;
    if (x > OVERFLOW_BOUND) return INFINITY;

    uint32_t bits = asuint(x);
    uint32_t biased_e = (bits >> 23) & 0xffu;

    // true exponent
    int32_t e = (int32_t) biased_e - 127;

    float f;
    double k;
    float r;
    float lnf;
    if (e == 0) {
        // f = 1.M_2
        f = set_exponent_to(x, 127u);
        k = (double) e * LN2;
        r = f - 1.0f;
        lnf = positive_poly(r);

    } else {
        // f = 1.M_2 / 2
        f = set_exponent_to(x, 126u);
        k = (double) (e + 1) * LN2;
        r = f - 1.0f;
        lnf = negative_poly(r);
    }

    return k + lnf;
}
```

Running the ULP test again,

```
scalar logf
maximum ULP: 4
from input: 0.633905
vex measurement: -4.558562934e-01
reference measurement: -4.558561742e-01
ulp distribution
0       97599500     97.599510%
1       2249902      2.249902%
2       129899       0.129899%
3       20441        0.020441%
4       258          0.000258%
total samples: 100000000
```

Success! My idea worked. I reduced the worst-case error from over 2 million ULPs to 4. Of the 100000000 inputs sampled between `FLT_MIN` and `FLT_MAX` in bit-space, 97.6% exactly match the standard-library `logf`.

FreeBSD msun

Afterwards, I looked at the `logf` implementation in FreeBSD `msun`. Its range reduction is incredibly close to mine. Both choose between m and $\frac{m}{2}$. My implementation chooses using the true exponent:

$$f = \begin{cases} m & \varepsilon = 0 \\ \frac{m}{2} & \varepsilon \neq 0. \end{cases}$$

The `msun` implementation instead chooses using the significand:

$$f = \begin{cases} m & m < \sqrt{2} \\ \frac{m}{2} & m \ge \sqrt{2}. \end{cases}$$

Since $m \in [1, 2)$, the first branch gives

$$1 \le f < \sqrt{2} \quad \to \quad 0 \le r < \sqrt{2} - 1.$$

The second branch gives

$$\frac{1}{\sqrt{2}} \le f < 1 \quad \to \quad \frac{1}{\sqrt{2}} - 1 \le r < 0.$$

Combining both branches,

$$r \in \left[\frac{1}{\sqrt{2}} - 1, \sqrt{2} - 1\right) \approx [-0.292893, 0.414214).$$

Both branches therefore produce r in the same small interval, so `msun` only needs **one minimax polynomial**. This interval is wider than the $[-0.5, 0)$ interval used by my `negative_poly`, but narrower than the $[0, 1)$ interval used by my `positive_poly`.

My approach uses a exponent-based branch and also avoids catastrophic cancellation near 1. The tradeoff is that I need **two minimax polynomials** as the combined range $r \in [-0.5, 1)$, is too large.

Separating the two polynomial evaluations into different functions still keeps the code relatively elegant, so I am content with my implementation. It's $\sim 1.5 \times$ slower though.