The Exponential Function

Deval Deliwala

July 22, 2026

The exponential function e^x is everywhere. This note explains how I learned the algorithm, then designed an accurate, fast, and elegant implementation for `vex`.

I first tried to learn the algorithm by reading the `expf` implementation in ARM optimized-routines. However, in my very humble opinion, the code is unreadable.

```
  /* exp(x) = 2^(k/N) * 2^(r/N) ~= s * (C0*r^3 + C1*r^2 + C2*r + 1) */
  t = T[ki % N];
  t += ki << (52 - EXP2F_TABLE_BITS);
  s = asdouble (t);
  z = C[0] * r + C[1];
  r2 = r * r;
  y = C[2] * r + 1;
  y = z * r2 + y;
  y = y * s;
  return eval_as_float (y);
```

From the comment, they rewrite e^x as $2^{k/N} \times 2^{r/N}$. Computers are good at binary, so this is a natural transformation. Then they use a polynomial to approximate $2^{r/N}$. This is the minimax polynomial from the previous note.

However, the variable names do not help. There is a bit shift with a magic 52. It took me a while to understand the design. Once I did, I saw how clever it is.

I learned a lot from it and designed my own version. This note explains those numerical methods, then walks through my implementation step by step.

Rounding Error

Computing e^x is difficult because its scale changes enormously:

$$e^x \to 0 \ (x \to -\infty) \qquad e^x \to \infty \ (x \to \infty).$$

A tempting implementation is `powf(e, x)`. At $x = 80$:

```
#include <math.h>
#include <stdio.h>

int main(void) {
  const float e = 2.71828182845904523536f;
  float x = 80.0f;

  float bad_result = powf(e, x);
  printf("calculated %f\n", bad_result);

  return 0;
}
```

```
calculated 55406091149243350783493234770313216.000000
```

Compared with the true e^{80}, the absolute and relative errors are

$$|\Delta| = 1.32695 \times 10^{29}, \qquad \frac{|\Delta|}{e^{80}} = 2.39 \times 10^{-6}.$$

Because e^{80} is enormous, the relative error is the more useful measure. Replacing `powf(e, x)` with the dedicated `expf(x)` function reduces that error by roughly a factor of 100. On my machine, it also runs about $2.55 \times$ faster.

A dedicated `expf(x)` function gives room for some clever techniques that improve both accuracy and speed.

Rough Idea

There are three steps to evaluate e^x efficiently and accurately.

1. Transform e^x into a more computationally friendly form.
2. Reduce the range for the minimax polynomial.
3. Evaluate the minimax polynomial.

Step 1

We'll use the identity

$$e^x = 2^{x \log_2(e)}.$$

ARM optimized-routines starts from the equivalent form $e^x = 2^{x/\ln(2)}$, but its implementation introduces magic numbers like 32. It is extremely fast and clever, but the code is incredibly difficult to follow.

If I copied that design directly, there would not be much point in writing `vex` to learn.

First, we'll evaluate $x \times \log_2(e)$. Then we'll split the result into a nearby integer k and a small remainder r.

Hence, the problem becomes

$$e^x = 2^{x \log_2(e)} \quad \to \quad 2^{k+r} = 2^k \times 2^r.$$

Evaluating 2^k, where k is an integer, is straightforward. Evaluating 2^r, where r is a small remainder, involves evaluating a minimax polynomial. This polynomial exists because the function 2^x is continuous (see the previous note).

Then we just multiply the two results and we have our answer.

Step 2

The previous note also showed that smaller approximation intervals produce tighter error bounds.

If we used one polynomial to approximate e^x over $x \in [-80, 80]$, the results would be atrocious. Restricting the interval to $x \in [-0.5, 0.5]$ gives an accurate approximation.

Range reduction transforms a problem into an equivalent one whose polynomial only needs to cover a smaller input interval. This is why, in the previous step, we split the problem into an integer k and a remainder r:

$$e^x \quad \to \quad 2^k \times 2^r.$$

We choose k as the nearest integer to $x \times \log_2(e)$, which keeps r in the range $[-0.5, 0.5]$. We can calculate 2^k directly with bit shifts; only 2^r needs a polynomial approximation. And on this small interval, a degree-5 minimax polynomial is accurate enough for `float32` calculations.

Step 3

Now all that's left is approximating 2^r with the minimax polynomial.

We need to derive it for 2^r, evaluate it efficiently, then multiply the result by 2^k. This should yield a close approximation to e^x.

Implementation

With `-O3` and a warmed-up benchmark, my code is roughly $1.3 \times$ slower than the built-in `expf(x)`.

```
//! expf.c
//!
//! implementation of the `expf` function.

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

/// Computes e^x.
///
/// Reduces the problem to
///
///     e^x = 2^(x log2(e)) = 2^k * 2^r,
///
/// where k is an integer and r is the remaining fraction.
/// 2^k is calculated exactly from the exponent field.
/// 2^r is approximated using a degree-5 minimax polynomial.
float vex_expf(float x) {
    // magic number for rounding-to-nearest
    const float MAGIC_SHIFTER = 12582912.0f;

    // log_2(e)
    const float LOG2E = 1.44269504088896340736f;

    // bounds for a valid normal 2^k
    const float OVERFLOW_BOUND = 88.37626f;
    const float UNDERFLOW_BOUND = -87.33654f;

    // minimax remez coefficients for 2^x on [-0.5, 0.5]
    // constrain c0 = 1.0 so e^0 = 1.0 is exact.
    const float c0 = 1.0f;
    const float c1 = 0.69314724206924438f;
    const float c2 = 0.24022234976291656f;
    const float c3 = 0.055503085255622864f;
    const float c4 = 0.0096718752756714821f;
    const float c5 = 0.0013407259248197079f;

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

    float xlog2e;
    float kf;
    int32_t k;
    float r;
    float two_k;
    float two_r;

    xlog2e = x * LOG2E;

    kf = xlog2e + MAGIC_SHIFTER;
    kf = kf - MAGIC_SHIFTER;
    // integer part of x log_2(e)
    k  = (int32_t) kf;

    // fractional part of x log_2(e) in [-0.5, 0.5]
    r = xlog2e - kf;

    // evaluating 2^k
    two_k = asfloat(((uint32_t)(k + 127) << 23));

    // evaluating 2^r
    // minimax polynomial in Horner's form
    two_r = fmaf(r, c5, c4);
    two_r = fmaf(r, two_r, c3);
    two_r = fmaf(r, two_r, c2);
    two_r = fmaf(r, two_r, c1);
    two_r = fmaf(r, two_r, c0);
    return two_k * two_r;
}
```

Walkthrough

This code is simpler and more readable than the ARM optimized-routines implementation, but it is a bit slower.[1] Four parts still seem like magic:

1. Where `OVERFLOW_BOUND` and `UNDERFLOW_BOUND` come from.

2. What `MAGIC_SHIFTER` is, and how adding and subtracting it rounds $x \log_2(e)$ to the nearest integer.

```
    kf = xlog2e + MAGIC_SHIFTER;
    kf = kf - MAGIC_SHIFTER;
    k  = (int32_t) kf;  // integer part
```

3. How this line constructs 2^k:

```
    two_k = asfloat(((uint32_t)(k + 127) << 23));
```

In particular, where 127 and the 23-bit left shift come from.

4. How the minimax polynomial is evaluated.

```
    two_r = fmaf(r, c5, c4);
    two_r = fmaf(r, two_r, c3);
    two_r = fmaf(r, two_r, c2);
    two_r = fmaf(r, two_r, c1);
    two_r = fmaf(r, two_r, c0);
```

This doesn't look like a polynomial?

I'll explain these four tricks here. The rest follows directly from the ***Rough Idea*** section above.

1. Bounds

Before calculating e^x, we need to know whether the result fits in a `float32`.

The largest finite `float32` is roughly 3.4028235×10^{38}. If e^x is larger, the function must return `INFINITY`.

The theoretical overflow threshold comes from

$$e^x = 3.4028235 \times 10^{38} \quad \to \quad x = \ln(3.4028235 \times 10^{38}).$$

This gives $x \approx 88.72284$. Above this point, the result no longer fits.

However, my code uses the slightly smaller bound `88.37626`, so it returns `INFINITY` a little early. The reason comes from how we construct 2^k, which Section 3 explains.

At the other end, the smallest positive *normal* `float32` is 2^{-126}. I use this as the lower limit, so `UNDERFLOW_BOUND` comes from

$$e^x = 2^{-126} \quad \to \quad x = \ln(2^{-126}) \approx -87.33654.$$

Smaller *subnormal* values are representable, but my method for evaluating 2^k only handles normal `float32` values. I skip that extra range and return `0.0f` instead.

Both cutoffs sacrifice some range to keep the algorithm simple. With that tradeoff understood, the checks are straightforward.

```
    if (isnan(x)) return x;
    if (x == -INFINITY) return 0.0f;
    if (x > OVERFLOW_BOUND) return INFINITY;
    if (x < UNDERFLOW_BOUND) return 0.0f;
```

The first propagates `NaN`. The second follows from $e^{-\infty} = 0$. The last two keep the calculation inside the range this implementation supports.

2. Magic Shifting

This trick blew my mind.

Let

$$y = x \log_2(e).$$

We need to split y into the nearest integer k and the remainder r:

$$k = \lfloor y \rceil, \qquad r = y - k, \qquad y = k + r.$$

Rounding to the nearest integer keeps r between -0.5 and 0.5. For example, if $y = 3.75$, then

$$k = 4, \qquad r = -0.25.$$

We could calculate k with `roundf(y)`.[2] The ARM optimized-routines implementation instead just adds and subtracts a magic `12582912.0f`:

```
  kf = y + 12582912.0f;
  kf = kf - 12582912.0f;
```

Let's follow the bits and use this trick to round `3.75`.

The magic number is

$$12582912 = 1.5 \times 2^{23} = 1.1_2 \times 2^{23}.$$

Written out in binary, the addition is

```
  110000000000000000000000.00
+                        11.11
  ---------------------------
  110000000000000000000011.11
```

The bottom line is the exact mathematical result: `12582915.75`. But is that value representable by a `float32`?

A `float32` stores 23 bits after the leading `1`. Normalizing the exact result gives

$$1.1000000000000000000001111_2 \times 2^{23}.$$

The blue 23 bits fit in the `float32` mantissa. The pink final two bits do not. Those final bits are the `.11` left after the binary point in the addition above:

$$.11_2 = 2^{-1} + 2^{-2} = 0.75.$$

The hardware must therefore round between these two representable values:

```
round down: 110000000000000000000011.00 = 12582915
exact:      110000000000000000000011.11 = 12582915.75
round up:   110000000000000000000100.00 = 12582916
```

The exact result is `0.75` above the lower value and only `0.25` below the upper value. It rounds up to `12582916.0f`. Subtracting the magic number now gives

$$12582916 - 12582912 = 4.$$

That is the nearest integer to `3.75`.

This works throughout our input range because the exponent of the sum stays at 23. After the leading `1`, a `float32` has only 23 stored bits. At this exponent,

those 23 bits end exactly at the ones place. Any bits after the binary point cannot fit, so the addition rounds them away.

In the actual code,

```
    kf = xlog2e + MAGIC_SHIFTER;
    kf = kf - MAGIC_SHIFTER;
    k  = (int32_t) kf;
    r  = xlog2e - kf;
```

The first two lines add and subtract the magic number. This places the rounded value of `xlog2e` in `kf`, which is still a `float32`. The `(int32_t)` cast stores it in the integer `k`, and the final line calculates the leftover r.

3. Constructing 2^k

From the Magic Shifting section above, we now have the nearest integer k to $x \log_2(e)$. The next goal is to calculate 2^k efficiently and exactly.

This implementation does it in one line:

```
    two_k = asfloat(((uint32_t)(k + 127) << 23));
```

Loosely, this line moves $k + 127$ into the exponent field of a `float32`. It then reinterprets those same 32 bits as a `float32`. Because the sign bit is zero and the mantissa bits are all zero, the resulting value is 2^k.

Let's follow the bits for $k = 10$.

First, add the `float32` exponent bias:

$$k + 127 = 10 + 127 = 137.$$

At this point, the result is positive. Casting it to `uint32_t` gives the ordinary unsigned binary representation of 137, padded to 32 bits:

```
00000000000000000000000010001001
```

The final eight bits are `10001001`, which is 137 in binary. The left shift moves these bits 23 places:

```
before: 00000000000000000000000010001001
after:  01000100100000000000000000000000
```

For an integer, shifting left by 23 is the same as multiplying by 2^{23}. At the bit level, it also places `10001001` directly above the 23 trailing zeroes.

Those positions have a second meaning when the same bits are interpreted as a `float32`:

```
sign   exponent          mantissa
 0  |  10001001  |  00000000000000000000000
```

The exponent field is $10001001_2 = 137$. Removing the bias of 127 gives

$$137 - 127 = 10.$$

The sign is positive, and the all-zero mantissa gives a significand of exactly 1. Therefore, these bits exactly represent

$$1 \times 2^{10} = 2^{10}.$$

The `(uint32_t)` cast makes the shift unsigned. This matters because shifting a signed integer into its sign bit can be undefined in C. The cast does not perform the float conversion; it only gives us a well-defined 32-bit unsigned shift.

`asfloat` performs the reinterpretation:

```
static inline float asfloat(uint32_t x) {
    union {
        uint32_t i;
        float f;
    } u = { .i = x };

    return u.f;
}
```

The union first stores the bits as a `uint32_t`, then reads those same bits as a `float`. This is different from a numerical cast. The bits do not change; only their interpretation does.

This also explains why my chosen `OVERFLOW_BOUND` is slightly below the true `float32` limit. The largest exponent field for a finite `float32` is 254. Therefore, the largest value of k this line can construct is

$$k = 254 - 127 = 127.$$

Once $x \log_2(e)$ reaches 127.5, magic shifting rounds it to $k = 128$. Adding the bias would place 255 in the exponent field. An exponent of all `1`s and a mantissa of all `0`s represent infinity, so we need $x \log_2(e) < 127.5$.

Solving for x gives

$$x \log_2(e) = 127.5 \quad \to \quad x = 127.5 \ln(2) \approx 88.37626.$$

That is why my implementation returns `INFINITY` at the earlier bound.

4. Evaluating the Minimax Polynomial

The remaining term is 2^r, where r is between -0.5 and 0.5. I used Sollya to precompute the Remez coefficients for

$$2^r \approx c_0 + c_1 r + c_2 r^2 + c_3 r^3 + c_4 r^4 + c_5 r^5.$$

A direct evaluation would calculate r^2 through r^5, multiply each power by its coefficient, then add the terms. Horner's form factors out r instead:

$$((((c_5 r + c_4) r + c_3) r + c_2) r + c_1) r + c_0.$$

Now the polynomial is just five multiply-add steps. This form is common in numerical code because it evaluates a degree-n polynomial with n multiplications and n additions, without storing any intermediate powers.

The code follows the nested expression from the inside out:

```
    two_r = fmaf(r, c5, c4);
    two_r = fmaf(r, two_r, c3);
    two_r = fmaf(r, two_r, c2);
    two_r = fmaf(r, two_r, c1);
    two_r = fmaf(r, two_r, c0);
```

`fmaf(a, b, c)` calculates $ab + c$ as one fused operation, with one rounding at the end. The first line evaluates $c_5 r + c_4$. Each following line multiplies the current result by r and adds the next coefficient. After the fifth line, `two_r` contains the approximation of 2^r.

Finally,

```
    return two_k * two_r;
```

combines the two parts:

$$2^k \times 2^r.$$

This is the final approximation of e^x.

Benchmarking

In my warmed-up `-O3` benchmark, this implementation takes 3.12 nanoseconds per call. The built-in `expf` takes 2.40 nanoseconds. And yes, I benchmarked well. No constant-folding occured. Many millions of inputs were tested.

Replacing magic shifting with `roundf(xlog2e)` reduces my implementation to 2.88 nanoseconds (see footnote below).

The remaining slowdown comes largely from evaluating the degree-5 minimax polynomial. As discussed in my LAK notes, these five calls form a dependency chain:

```
    two_r = fmaf(r, c5, c4);
    two_r = fmaf(r, two_r, c3);
    two_r = fmaf(r, two_r, c2);
    two_r = fmaf(r, two_r, c1);
    two_r = fmaf(r, two_r, c0);
```

Each `fmaf` needs the previous value of `two_r`, so none of them can run in parallel. Evaluating the polynomial requires a chain of five FMAs.

A degree-5 polynomial gives enough accuracy to approximate 2^r over $r \in [-0.5, 0.5]$ for `float32`. ARM optimized-routines uses a tighter range reduction, so it only needs a degree-3 polynomial and a chain of three FMAs.

Their magic `32` reduces x as

$$x = \frac{k \ln(2)}{32} + \frac{r \ln(2)}{32}, \qquad k \in \mathbb{Z}, \qquad r \in \left[-\frac{1}{2}, \frac{1}{2}\right].$$

The polynomial only approximates $2^{r/32}$, so the effective interval becomes $\approx [-0.01083, 0.01083]$.

That smaller interval needs a lower-degree polynomial. The shorter dependency chain is one reason ARM optimized-routines is faster. The extra range reduction is also one reason it is harder to read.

I still dislike how unreadable high-performance numerical code often is though. Speed and accuracy come first, but readability matters too.

Production numerical code do balance four things at once:

1. Speed
2. Accuracy
3. Edge cases
4. Portability

so things like bit tricks and table lookups may be necessary. The lack of structure and explanation is not.

---

[1] Of course, production routines are also more accurate. My implementation has errors as high as **66 ULP** for very small x. However, replacing the `MAGIC_SHIFTER` rounding method with `double`s and a simple `round` brought it down to 3 ULP. I swept through 2.2 billion `float` encodings of x. Performance barely budged as well.

[2] On modern ARM64, this is actually better. On my Apple M4, `roundf(y)` averaged about 0.24 nanoseconds faster per call and it is more readable. The magic-number trick is too clever not to explain, though.