Floating-Point Training
Module 4
Elementary Function Approximation

Paul Zimmermann, Inria Nancy

Plan of the training

- Module 1: The IEEE 754 Standard
- Module 2: Math Fundamentals
- Module 3: Core Algorithms
- **Module 4: Elementary Function Approximation**
- Module 5: Software Tools

Module 4: Elementary Function Approximation

Motivation and State of the art

Hard-to-round cases and the Table Maker's Dilemma

Fast Path, Rounding Test, Accurate Path

Argument reduction and reconstruction

Polynomial or rational approximation

A complete toy example

Asymptotic expansions

Motivation

Maximal known error for `cosh` in double precision, in terms of units in last place:

GNU libc	1.93	Intel ML	0.516
AMD libm	1.85	Newlib	2.67
OpenLibm	1.47	Musl	1.04
Apple	0.523	MSVC	1.91
FreeBSD	1.47	ArmPL	1.93
CUDA	1.40	ROCm	0.563

Accuracy of Mathematical Functions in Single, Double, Extended Double and Quadruple Precision, Gladman et al., 2025.

This mess is due to IEEE 754, which only recommends correct rounding for mathematical functions.

State of the art

Several implementations with correct rounding:

- MathLib/LibUltim
- LIBMCR
- CRLibm
- RLIBM
- LLVM libc
- CORE-MATH
- GNU MPFR

MathLib/LibUltim

Developed by IBM around 1990.

Provides binary64 functions acos, asin, atan, atan2, exp, exp2, log, log2, cos, sin, tan, cot, pow.

Only supports rounding to nearest-even.

Some high-level algorithms described in Fast evaluation of elementary mathematical functions with correctly rounded last bit, ACM Transactions on Math. Software, 1991.

No longer maintained, but was partly integrated in GNU libc 2.27 (2018). Accurate path removed after 2.27 (too slow).

No known bug.

Unofficial copy `https://github.com/dreal-deps/mathlib`.

LIBMCR

Developed by Sun Microsystems until 2004.

Targets only binary64 and rounding to nearest-even.

Provides exp, log, pow, atan, sin, cos, tan.

Algorithms are not detailed.

The power function has issues: for some inputs it does not terminate, for some inputs, it gives a result far from the correct one.

A non-official copy is available from `https://github.com/simonbyrne/libmcr`.

CRLibm

Developed by the team of Jean-Michel Muller until 2006.

Provides binary64 functions: exp, expm1, log, log1p, log2, log10, sin, cos, tan, asin, acos, atan, sinh, cosh, sinpi, cospi, tanpi, atanpi, and (with restrictions) pow.

Each functions has four entry points: `exp_rn`, `exp_rz`, `exp_ru`, `exp_rd`.

Assumes the rounding precision is set to binary64, and the processor rounding function is set to nearest ties-to-even.

No longer maintained, an unofficial copy is available from `https://github.com/taschini/crlibm`.

RLIBM

Developed by the group of Santosh Nagarakatte (Rutgers University).

Provides 16 binary32 functions: acos, asin, atan, cos, cosh, cospi, exp, exp10, exp2, log, log10, log2, sin, sinh, sinpi, tan.

Supports all IEEE rounding modes.

Uses a new approach based on linear programming, to find polynomials that yield correct rounding. Not clear whether that approach scales for binary64.

Code and extensive bibliography available from
`https://people.cs.rutgers.edu/~sn349/rlibm/`.

LLVM libc

The C library that comes with the LLVM compiler, supported by Google.

Its math library provides only correctly rounded functions, for all IEEE rounding modes.

Provides all C99 binary32 functions except erfc, and several binary64 functions: acos, asin, atan, atan2, cbrt, cos, exp, exp2, exp10, expm1, hypot, log, log10, log1p, log2, sin, sincos, tan.

Except for the binary64 hypot function, its efficiency is within a factor of two of the GNU libc and/or the Intel library.

Code available from `https://libc.llvm.org/`.

CORE-MATH

A collection of routines ready to be integrated into mathematical libraries.

Provides all C23 binary32 and binary64 functions, and a few functions in double-extended or quadruple precision.

Supports all four IEEE rounding modes.

Algorithms detailed either as comments in the source code, or as scientific publications.

Code and an extensive bibliography available from `https://core-math.gitlabpages.inria.fr/`.

Also contains large tables of hard-to-round inputs, generated using BaCSeL.

Part of CORE-MATH is already integrated into mathematical libraries: the binary32 tanh function into the AMD library, and 30 binary32 functions into GNU libc.

GNU MPFR

A multiple-precision floating-point library with correct rounding.

Much slower than libraries targetting the IEEE formats.

More useful as a reference implementation.

Provides the floating-point numbers of SageMath.

See more in Module 5.

Objective of this module

Give algorithms to get correct rounding of mathematical functions.

Present various ideas and tricks to speed up these algorithms.

Present ways to prove the correctness of these algorithms.

Concrete software tools will be presented in Module 5.

Hard-to-round and worst cases

A hard-to-round case for a function f is a floating-point number x such that $f(x)$ has many identical bits after the round bit.

Example:

$$\sin(\texttt{0x1.ff822b853ca6p-1}) = 0.\underbrace{110\ldots011}_{53}\underbrace{000\ldots000}_{44}100100\ldots$$

A worst case for a function f in a given floating-point format is a floating-point number x in that format such that $f(x)$ has the largest number of identical bits after the round bit.

Example:

$$\tan(\texttt{0x1.fe6e530194af6p+681}) = \underbrace{101.000\ldots000}_{53}0\underbrace{111\ldots111}_{62}000110\ldots$$

The Table Maker's Dilemma

The Table Maker's Dilemma (TMD) consists in determining, for a given floating-point format, the worst cases, or the hard-to-round cases with at least m identical bits after the round bit.

Solving the TMD is crucial to determine correct rounding.

The Table Maker's Dilemma

Suppose we know the worst cases have at most m identical bits after the round bit.

Then for any number y in precision $p+1$ close to $f(x)$:

$$|f(x) - y| \geq 2^{-m-2}\mathrm{ulp}(y)$$

If we approximate $f(x)$ with z having $p+m+2$ bits:

$$|f(x) - z| \leq \epsilon < \mathrm{ulp}(z) = 2^{-m-2}\mathrm{ulp}(y)$$

Thus the interval $[z-\epsilon, z+\epsilon]$ does not cross a rounding boundary (for a directed rounding).

Thus rounding z yields the correct rounding of $f(x)$.

The Table Maker's Dilemma

The TMD is easy to solve for univariate binary32 functions, since there are at most 2^{32} values to check.

This approach does not scale for binary64 functions or bivariate functions (`pow`, `atan2`, `hypot`).

Efficient algorithms exist (Lefèvre's algorithm, SLZ) with complexity less than $O(2^p)$. They are implemented in BaCSeL (see Module 5).

The Table Maker's Dilemma

Thanks to the work of Muller, Lefèvre, Tisserand and others, the TMD is now fully solved for univariate binary64 functions.

See `https://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm` and the `foo.wc` files distributed within CORE-MATH.

Worst case of $\log(x)$ in binary64 (directed rounding):

$$\log(\texttt{0x1.62a88613629b6p+678}) = \underbrace{111010110.010\ldots001}_{53}\underbrace{000\ldots000}_{65}111001\ldots$$

Worst case of 2^x in binary64 (rounding to nearest):

$$\exp2(\texttt{0x1.e4596526bf94dp-10}) = \underbrace{1.000\ldots011}_{53}0\underbrace{111\ldots111}_{59}010011\ldots$$

A statistical argument

Assume the bits of $f(x)$ after the round bit behave randomly.

The probability to have 2 identical bits after the round bit is $1/2$: patterns 00 and 11 out of 4 bit-patterns.

The probability to have 3 identical bits after the round bit is $1/4$: patterns 000 and 111 out of 8 bit-patterns.

The probability to have k identical bits after the round bit is 2^{1-k}:
patterns $\underbrace{000...000}_{k}$ and $\underbrace{111...111}_{k}$ out of 2^k patterns.

Thus in a given binade of 2^{p-1} values, we expect a maximum of about p identical bits.

And in a format with 2^q values, we can expect a maximum of about q identical bits for a univariate function, and $2q$ identical bits for a bivariate function.

Special inputs

Some functions have special inputs in some regions with huge numbers of identical bits after the round bit.

Example near $x = 0$:

$$\tan x = x + \frac{x^3}{3} + O(x^5)$$

Thus for $x \approx 2^{-k}$, $\tan(x)$ will have about $2k - p$ identical bits after the round bit.

But these cases are easy to round: $\tan(x)$ rounds to x or $\mathrm{nextaway}(x)$.

The TMD in binary64

Maximal number m of identical bits after round bit (excluding special inputs):

function	maximal m
log	64
exp	59
log2	55
exp2	59
sin	68
cos	66
tan	62

This aligns well with the statistical argument.

Thus for $\exp(x)$, a working precision of $53 + 59 + 2 = 114$ is enough.

Ziv's strategy

Assume we know the worst cases of a function $f(x)$, how to implement it with correct rounding? We use Ziv's strategy:

- compute an approximation y_1 in precision $p_1 > p$ and error bounded by ϵ_1:

$$y_1 - \epsilon_1 \leq f(x) \leq y_1 + \epsilon_1$$

  If $y_1 - \epsilon_1$ and $y_1 + \epsilon_1$ round to the same value, return this common value.
- if not, compute an approximation y_2 in precision $p_2 > p_1$ and error bounded by ϵ_2:

$$y_2 - \epsilon_2 \leq f(x) \leq y_2 + \epsilon_2$$

  If $y_2 - \epsilon_2$ and $y_2 + \epsilon_2$ round to the same value, return this common value.
- and so on.

Ziv's strategy

Ziv's strategy can loop indefinitely if $f(x)$ is exact (for directed rounding) or a midpoint (for rounding to nearest).

It is crucial to determine the exact and midpoint values, for example $\mathrm{log10p1}(999)$ or $\mathrm{pow}(11, 7)$ in binary32.

From the knowledge of worst cases, we can bound the maximal precision p_k in Ziv's strategy.

Two-step Ziv's strategy

- fast path: compute an approximation y_1 and error bound ϵ_1:

$$y_1 - \epsilon_1 \leq f(x) \leq y_1 + \epsilon_1$$

- rounding test: If $\circ(y_1 - \epsilon_1) = \circ(y_1 + \epsilon_1)$, return that value
- check for exact and midpoint inputs, if any
- accurate path: compute an approximation y_2 in precision p_2 and return $\circ(y_2)$

Precision p_2 has to be large enough so that for all hard-to-round/worst cases, $\circ(y_2)$ is the correct rounding.

Otherwise, it is possible to hard-code a few exceptional cases in the accurate path.

Why is Ziv's strategy efficient?

For 99.9% of inputs, only the fast path and the rounding test are performed.

Thus the average time is roughly the same as that of the fast path, plus that of the rounding test (a few cycles):

$$t_{\mathrm{Ziv}} = t_{\mathrm{fast}} + t_{\mathrm{rt}} + \frac{1}{1000} t_{\mathrm{acc}}$$

The working precision of the fast path is not much larger than for a routine with 1-ulp accuracy.

Remark: if one disables the rounding test and the accurate path, returning always $\circ(y_1)$, one gets a routine that delivers the correct rounding for 99.9% of inputs.

Argument reduction and reconstruction

Generic algorithm to approximate $f(x)$:

- argument reduction: reduce $f(x)$ to $g(x')$ for a "smaller" value x'
- approximation: approximate $y' \approx g(x')$
- argument reconstruction: deduce from y' an approximation for $f(x)$

Rationale: the approximation phase (usually using polynomials) is easier when x' is small (in absolute value).

This is because Taylor expansions converge better around zero.

However, this works only when there is a functional equation between $f(x)$ and $g(x')$.

Additive argument reduction

When $x' = x - kC$ for some integer k and real C.

$$\sin(x) = \sin(x - 2k\pi)$$

$$\exp2(x) = 2^k \cdot \exp2(x - k)$$

$$\operatorname{tanpi}(x) = \operatorname{tanpi}(x - k)$$

$$\sin(x_h + x_\ell) = \sin(x_h)\cos(x_\ell) + \cos(x_h)\sin(x_\ell)$$

This enables to reduce the argument to $|x'| \leq C/2$.

Multiplicative argument reduction

When $x' = x \cdot C^k$ for some integer k and real $C > 1$.

Examples:

$$\exp(2x) = \exp(x)^2$$

$$\log2(2^k \cdot x) = k + \log2(x)$$

This enables to reduce the argument to $1 \leq x' \leq C$ or $C^{-1/2} \leq x' \leq C^{1/2}$.

Argument reconstruction

Once $g(x')$ is computed, how to recover $f(x)$? Sometimes trivial, sometimes not.

Case $\sin(x) = \sin(x - 2k\pi)$: argument reduction is trivial.

Case $\sin(x) = (-1)^k \sin(x - k\pi)$: trivial too.

Case $\sin(x) = \cos(x - k\pi/2)$: needs to implement both sin and cos.

Case $\mathrm{exp2}(x) = 2^k \cdot \mathrm{exp2}(x - k)$: just update the exponent.

Case $\exp(2x) = \exp(x)^2$: if $x' = x/2^k$, perform k squarings.

Case $\mathrm{log2}(2^k \cdot x) = k + \mathrm{log2}(x)$: add k (warning cancellation if $k = -1$ and $x \approx 2$).

Which approximation to use?

Once we have a reduced argument x, how to approximate $g(x)$ (we now use x instead of x' for simplicity).

We usually use a polynomial or rational approximation.

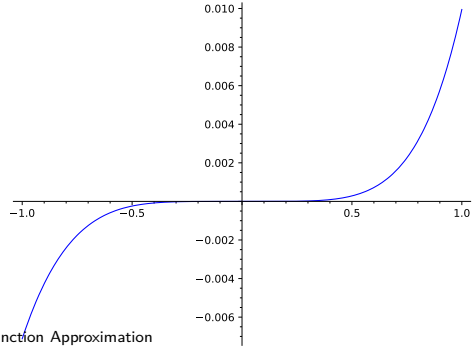
Some functions, for example $\mathrm{atan}(x)$ on $[0, 1]$, are not well approximated by polynomials. Then use a rational approximation. For all other functions, prefer a polynomial approximation to avoid divisions.

Either use one polynomial on the full reduced range, or further cut the reduced range into subranges, and use a different polynomial on each subrange.

Polynomial approximation

Given a function $f(x)$, a range $[a, b]$, and a degree d, what is the best degree-d polynomial approximation of $f(x)$?

This is usually not the Taylor approximation. Take for example $\exp(x)$ on $[-1, 1]$ with degree 4. The Taylor approximation yields the following difference with respect to $\exp(x)$:

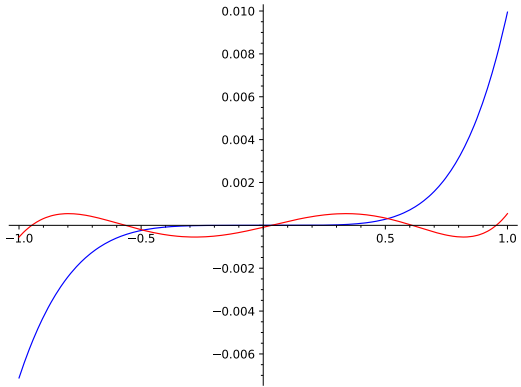


Polynomial approximation

The polynomial

$$p(x) = 1.00009 + 0.997309x + 0.498835x^2 + 0.177345x^3 + 0.0441555x^4$$

yields the red curve:



Minimax polynomial

The polynomial $p(x)$ is computed using Remez algorithm.

The maximal value of $|f(x) - p(x)|$ is attained at $d + 2$ points, with $f(x) - p(x)$ alternatively positive or negative.

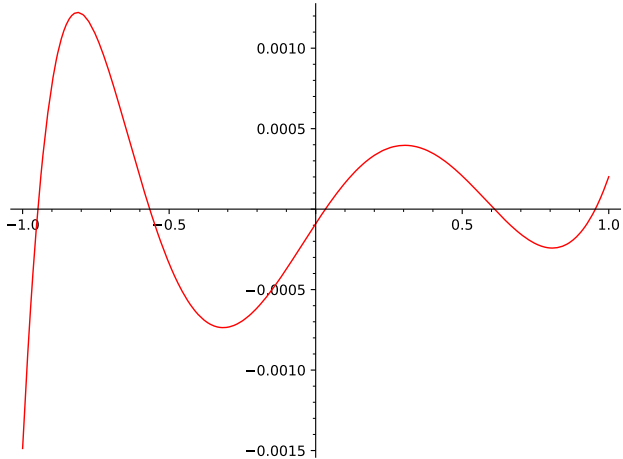
Such a polynomial is called a minimax polynomial:

$$\max_{a \le x \le b} |f(x) - p(x)| = \min_{q(x)} \max_{a \le x \le b} |f(x) - q(x)|$$

It is computed by Sollya `remez` function (cf Module 5).

Absolute and relative approximation

If we want to bound the relative error $p(x)/f(x) - 1$, the "red" polynomial is not optimal:

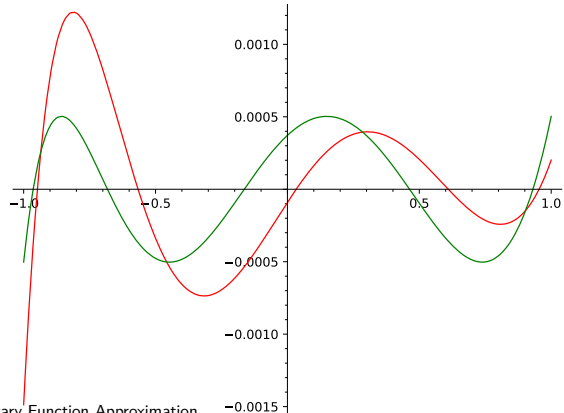


Absolute and relative approximation

We can give the weight $1/f(x)$ to Remez algorithm, to minimize the relative error:

$$p(x) = 0.999628 + 0.997939x + 0.502899x^2 + 0.176486x^3 + 0.0399629x^4$$

This yields the green curve:



Remez polynomials with constraints

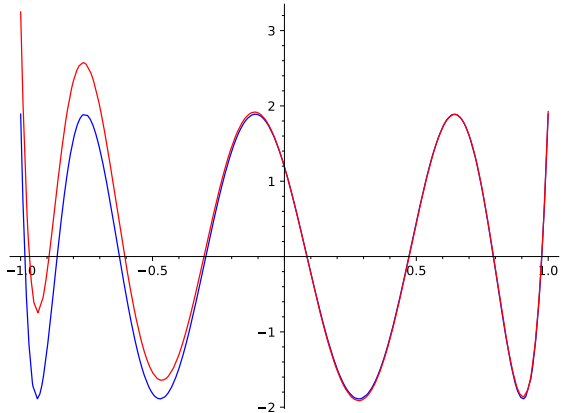
Polynomials output by Remez algorithm are useless in practice.

Indeed, their coefficients are usually not representable in the target precision (binary32, binary64, ...).

If we round each coefficient to the target precision, the accuracy can decrease a lot.

Remez polynomials with constraints

For example, if we compute the degree-7 Remez polynomial for $\exp(x)$ on $[-1, 1]$ minimizing the relative error (in blue), and round its coefficient to binary32, we get the red curve:



Remez polynomials with constraints

Fortunately, Sollya enables to give constraints on coefficients.

We can tell Sollya a given coefficient should be binary32, binary64, double-double, ...

We can tell Sollya to approximate $f(x)$ by an odd polynomial, force the leading term to 1 or x, ...

See more in Module 5.

Adjusting precision of coefficients

Example: we want a minimax polynomial for $\exp(x)$ on $[-0.5, 0.5]$ with absolute error less than 2^{-46}, and coefficients double-floats.

Sollya finds a polynomial of degree 10 with error $< 2^{-46.213}$ and 48-bit coefficients:

$$p = p_0 + p_1 x + \cdots + p_{10} x^{10}$$

where $p_0 = \texttt{0x1.00000000004ap+0}$ and $p_{10} = \texttt{0x1.29ab4374b6acp-22}$.

If we replace p_{10} by its binary32 rounding $\texttt{0x1.29ab44p-22}$, the maximal error is

$$\mathrm{ulp}_{24}(\texttt{0x1.29ab44p-22}) \cdot 0.5^{10} = 2^{-55}$$

This is much less our target error of 2^{-46}, thus no need to use a double-float for p_{10}!

In the end we only need to use double-floats for $p_0, \ldots, p_4$, and we can use floats for $p_5, \ldots, p_{10}$.

Adjusting precision of coefficients

Example: we want a minimax polynomial for $\exp(x)$ on $[-0.5, 0.5]$ with absolute error less than 2^{-46}, and coefficients double-floats.

In the end we only need to use double-floats for $p_0, ..., p_4$, and we can use floats for $p_5, ..., p_{10}$.

This yields two kinds of savings:

- in memory: we have to store only one float for $p_5, ..., p_{10}$ instead of two;
- in speed: within Horner's rule, we don't need to use double-float arithmetic for the evaluation of $p_5 + p_6 x + \cdots + p_{10} x^5$.

Rational approximation

Some functions are not well approximated by polynomials.

For $\sin(x)$ on $[0.1, 1]$, degree 10 and binary32 coefficients, Sollya finds a polynomial with relative error $< 2^{-43.121}$.

For $\mathrm{atan}(x)$ on $[0.1, 1]$, still with degree 10 and binary32 coefficients, Sollya only finds a polynomial with relative error $< 2^{-28.919}$.

In such cases, rational approximations might be better:

$$f(x) \approx \frac{a_0 + a_1 x + \cdots + a_n x^n}{b_0 + b_1 x + \cdots + b_n x^n}$$

Rational approximation

CORE-MATH uses such a rational approximation with numerator and denominator of degree $n = 29$ in the accurate path of the `atan2` function, with 192-bit coefficients, and maximal relative error less than $2^{-193.99}$.

The `rminimax` tool enables to generate such rational approximations. See `doc/rminimax.txt` in the CORE-MATH git.

Using tables

Usually a first argument reduction is not enough.

Consider

$$\exp(k \log(2) + x) = 2^k \cdot \exp(x)$$

It enables to reduce x to $[-\log(2)/2, \log(2)/2]$.

To get at least 67 bits of accuracy (target binary64), we need a polynomial of degree 13 (with binary64 coefficients)!

Even with Estrin's scheme, this will be expensive to evaluate.

Using tables

Instead of using a single polynomial to evaluate $f(x)$ on $[a, b]$, subdivide into n sub-ranges with $a_0 = a$ and $a_n = b$:

$$[a_0, a_1], [a_1, a_2], ..., [a_{n-1}, a_n]$$

On each subrange $[a_i, a_{i+1}]$:

- either use argument reduction if available:

$$\exp(a_i + x) = \exp(a_i) \cdot p(x)$$

- or use a dedicated minimax polynomial:

$$f(x) \approx p_i(x)$$

In the first case, pre-calculate the values $\exp(a_i)$. In the second case, pre-calculate the polynomials $p_i(x)$.

Using tables

Example for $\exp(x)$ on $[-\log(2)/2, \log(2)/2]$, subdivided into 2^k subranges, with relative error $\approx 2^{-67}$.

Each a_i is a double, $\exp(a_i)$ is stored as a double-double, $p(x)$ has double coefficients.

k	$\deg(p)$	total memory (doubles)
0	13	14
1	11	18
2	9	22
3	8	33
4	7	56
6	6	199
7	5	390

Example of fast path for exp

This is simplified for clarity (some steps might need double-double computations).

- reduce x to $[-\log(2)/2, \log(2)/2]$: $x = k\log(2) + x_r$

```
k = round (x * INV_LOG2);
xr = x - k * LOG2;
```

- extract the high bits: $x_r = i/2^7 + x_s$

```
i = round (xr * 0x1p7);
xs = xr - i * 0x1p-7;
```

- extract $\exp(i/2^7)$ from a table and approximate $\exp(x_s)$ by a polynomial:

```
return ldexp (T[i] * poly_eval (xs), k);
```

Example of fast path for exp

It is possible to perform only one call to `round`.

- write $x = j\log(2)/2^7 + x_s$

```
j = round (x * INV_LOG2_scaled);
xs = x - j * LOG2_scaled;
k = j >> 7;
i = j & 0x7f;
```

We now have:

$$x = k\log(2) + i/2^7\log(2) + x_s$$

- extract $2^{i/2^7}$ from a table and approximate $\exp(x_s)$ by a polynomial:

```
return ldexp (U[i] * poly_eval (xs), k);
```

Gal's accurate tables

For $\exp(x)$, after the initial argument reduction, if we further reduce $[-\log(2)/2, \log(2)/2]$ into 2^k subranges, for each subrange $[a_i, a_{i+1}]$, we need to store a_i and $\exp(a_i)$.

a_i is stored as a double and $\exp(a_i)$ as a double-double: we need 3 doubles per subrange.

Gal's accurate table idea is to replace a_i by a binary64 number near a_i such that $\exp(a_i)$ has enough accuracy for the fast path, when stored as a double.

Gal's accurate tables

Example: take $a_i =$ `-0x1.62e42fefa39efp-2` near $-\log(2)/2$.

Then $\exp(a_i) \approx$ `0x1.6a09e667f3bcdp-1` has only about 54 bits of accuracy.

If we take:

$$b_i = a_i + 15003 \cdot \mathrm{ulp}(a_i) = \texttt{-0x1.62e42fef9ff54p-2}$$

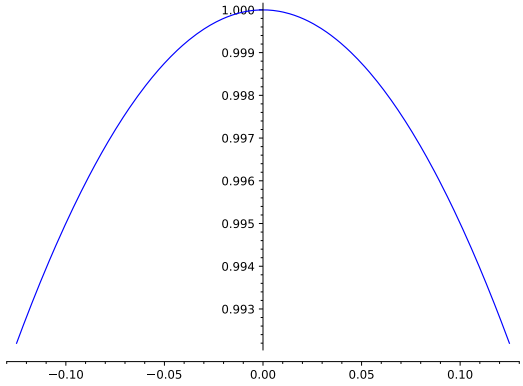
then $\exp(b_i) \approx$ `0x1.6a09e667f5085p-1` has about 70 bits of accuracy.

Instead of storing 3 doubles, we store only 2 (b_i and the 53-bit approximation of $\exp(b_i)$, which is accurate to 70 bits).

Remark: the good values for Gal's method are the bad ones for the TMD.

A complete toy example

Assume we want to get correct rounding of $\cos(x)$ in binary32, using binary64 arithmetic for internal computations, for $|x| \leq 0.125$.



Since $0.992 < \cos(x) \leq 1$, bounding the absolute error is enough.

A complete toy example

We use the degree-6 Taylor expansion:

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + O(x^8)$$

There is an explicit formula for the error term, where $0 \leq \xi \leq x$:

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \cos(\xi)\frac{x^8}{40320},$$

Thus the mathematical error is bounded by $0.125^8/40320 < 2^{-39.2}$.

We replace the Taylor coefficients by their double approximation to nearest.

```
double c2 = -0.5;                    // no error
double c4 = 0x1.5555555555555p-5;    // error < 2^-58.5
double c6 = -0x1.6c16c16c16c17p-10; // error < 2^-64.0
```

A complete toy example

We use Estrin's scheme and neglect lower order error terms:

```
double x = xf;              // exact
double x2 = x * x;
double t0 = c0 + x2 * c2;
double t4 = c4 + x2 * c6;
double x4 = x2 * x2;
double r = t0 + x4 * t4;
```

The error on x_2 is bounded by $\mathrm{ulp}(2^{-7})$ since x_2 lies in the binade $[2^{-7}, 2^{-6}]$ or in a smaller binade:

$$\mathrm{err}(x_2) \leq 2^{-59}$$

The error on $u_0 = \circ(x_2 c_2)$ is bounded by:

$$\mathrm{ulp}(u_0) + \mathrm{err}(x_2) \cdot |c_2| + x_2 \cdot \mathrm{err}(c_2) \leq 2^{-60} + 2^{-59} \cdot \frac{1}{2} + 2^{-6} \cdot 2^{-58.5} \leq 2^{-58.9}$$

A complete toy example

```
double x2 = x * x;
double t0 = c0 + x2 * c2;
double t4 = c4 + x2 * c6;
double x4 = x2 * x2;
double r = t0 + x4 * t4;
```

The error on $t_0 = \circ(c_0 + u_0)$ is bounded by (using the fact that $t_0 \leq 1$):

$$\mathrm{ulp}(t_0) + \mathrm{err}(u_0) \leq 2^{-53} + 2^{-58.9} < 2^{-52.9}$$

The error on $u_4 = \circ(x_2 c_6)$ is bounded by:

$$\mathrm{ulp}(u_4) + \mathrm{err}(x_2) \cdot |c_6| + x_2 \cdot \mathrm{err}(c_6) \leq 2^{-68} + 2^{-59} \cdot 2^{-9.4} + 2^{-6} \cdot 2^{-64} \leq 2^{-66.9}$$

The error on $t_4 = \circ(c_4 + u_4)$ is bounded by:

$$\mathrm{ulp}(t_4) + \mathrm{err}(c_4) + \mathrm{err}(u_4) \leq 2^{-57} + 2^{-58.5} + 2^{-66.9} < 2^{-56.5}$$

A complete toy example

```
double x2 = x * x;
double t0 = c0 + x2 * c2;
double t4 = c4 + x2 * c6;
double x4 = x2 * x2;
double r = t0 + x4 * t4;
```

The error on $x_4 = \circ(x_2 \cdot x_2)$ is bounded by (using the fact that $x_4 \leq 2^{-12}$):

$$\mathrm{ulp}(x_4) + 2\mathrm{err}(x_2) \cdot |x_2| \leq 2^{-65} + 2 \cdot 2^{-59} \cdot 0.125^2 < 2^{-63.4}$$

The error on $v_4 = \circ(x_4 t_4)$ is bounded by:

$$\mathrm{ulp}(v_4) + \mathrm{err}(x_4) \cdot |t_4| + |x_4| \cdot \mathrm{err}(t_4) \leq 2^{-69} + 2^{-63.4} \cdot 2^{-4.5} + 2^{-12} \cdot 2^{-56.5} \leq 2^{-66.8}$$

Finally the error on $r = \circ(t_0 + v_4)$ is bounded by (using the fact that $r \leq 1$):

$$\mathrm{ulp}(r) + \mathrm{err}(t_0) + \mathrm{err}(v_4) \leq 2^{-53} + 2^{-52.9} + 2^{-66.8} < 2^{-51.9}$$

A complete toy example

The mathematical error is bounded by $2^{-39.2}$ and the rounding error by $2^{-51.9}$, thus the total error is bounded by:

$$2^{-39.2} + 2^{-51.9} < 2^{-39}$$

```
float lb = r - 0x1p-39;
float ub = r + 0x1p-39;
if (lb == ub) return ub;
else return accurate_path (x);
```

We will see in Module 5 how to bound the mathematical error with Sollya, and how to bound the rounding error with Gappa.

Asymptotic expansions

In some cases, we have to use asymptotic expansions.

For example, $\tanh(x)$ tends to 1 when $x \to +\infty$.

In binary64, $\tanh(x)$ rounds to 1 to nearest for $x \geq$ `0x1.30fc1931f09cap+4`:

```
sage: a,b = RR(1),RR(2^1024)
sage: while a.nextabove()<b:
sage:   c = (a+b)/2
sage:   if tanh(c)<1:
sage:      a=c
sage:   else:
sage:      b=c
sage:  b.hex()
'0x1.30fc1931f09cap+4'
```

Asymptotic expansions

Another example is the exponential integral:

$$\mathrm{Ei}(x) = \gamma + \log(x) + \sum_{k=1}^{\infty} \frac{x^k}{k \cdot k!}$$

For large x it is better to use the non-converging asymptotic expansion (Abramowitz & Stegun, formula 5.1.51):

$$\mathrm{Ei}(x) \approx e^x \left(\frac{1}{x} + \frac{1}{x^2} + \cdots + \frac{k!}{x^{k+1}} + \cdots \right)$$

The term $k!/x^{k+1}$ decreases when k grows, then increases.

If we truncate the (non-converging) expansion, the absolute error is of the order of the first neglected term.

Asymptotic expansions

The term $k!/x^{k+1}$ is minimal when $k \approx x$, since the ratio to the previous term is k/x.

If we truncate at $k \approx x$, the relative error is thus of order $x!/x^x$.

We know $x! \approx (x/e)^x \sqrt{2\pi x}$.

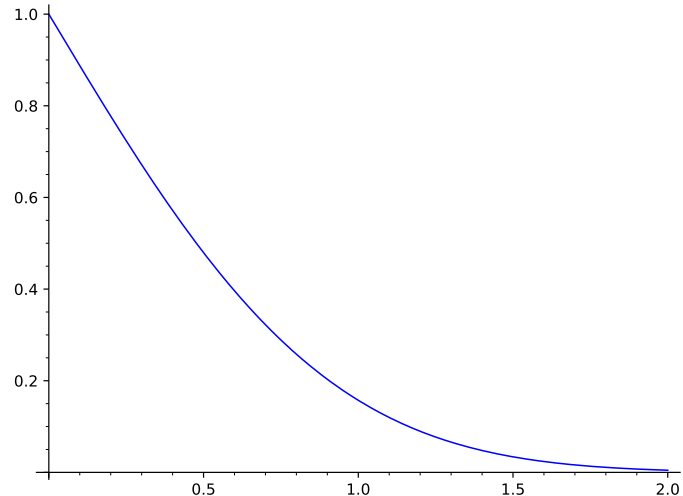
Very roughly, the relative error is in e^{-x}.

Thus the asymptotic expansion can be used only when $e^{-x} < 2^{-p}$ where p is the target precision, i.e., when $x > p \log(2)$.

In binary64, the asymptotic expansion can be used for $x \geq 40$ (for $x = 40$ it needs about 34 terms, whereas the Taylor expansion needs 102 terms).

Asymptotic expansions

Another example: the complementary error function $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$.



Asymptotic expansions

Around $x = 0$ we can use formula 7.1.5 from Abramowitz & Stegun for $\mathrm{erf}(x)$:

$$\mathrm{erf}(x) \approx \frac{2}{\sqrt{\pi}} \sum_{k \geq 0} (-1)^k \frac{x^{2k+1}}{k!\,(2k+1)}$$

For large x we can use the asymptotic expansion (formula 7.1.23):

$$\sqrt{\pi} x\, e^{x^2} \mathrm{erfc}(x) \approx 1 + \sum_{k \geq 1} (-1)^k \frac{1 \cdot 3 \cdots (2k-1)}{(2x^2)^k}$$

In binary64, the asymptotic expansion should be used for $x \geq 7$: for $x = 7$ it needs 20 terms, whereas the Taylor expansion needs 66 terms.

Exercises

Exercise 1: using the same binary compiled with the Intel Math Library, find a mathematical function f and a double-precision input x such that $f(x)$ gives different results on Intel and AMD hardware. Hint: see section 3.2 of Accuracy of Mathematical Functions.

Exercise 2: using GNU MPFR (or SageMath), find a hard-to-round case for $\sin(x)$ in $[3, 4]$ with at least 20 identical bits after the round bit (in binary64).

Exercise 3: using GNU MPFR (SageMath will be too slow), write a program that finds the hardest to round case of $\tan(x)$ for binary32.

Exercise 4: find three midpoints for $\mathrm{pow}(x, y)$ in binary64, i.e., inputs such that x^y is exact in 54 bits, but not in 53 bits. Your three midpoints should have different values of x and y.

Exercises

Exercise 5: what argument reduction can you use for $\sinh(x)$? And for $\log 10(x)$?

Exercise 6: using SageMath, reproduce the graphs for the minimax polynomials of $\exp(x)$ over $[-1, 1]$.

Exercise 7: in the fast path example for `exp`, what is the range of values taken by i (in both variants). Compute the tables `T[i]` and `U[i]`, both in double and double-double.

Exercise 8: following Gal's accurate tables idea, find a binary64 number b_i near $\log(2)/2$ such that $\exp(b_i)$ rounded to nearest has at least 70 bits of accuracy.

Exercise 9: implement the toy example for the fast path of $\cos(x)$ on $[-0.125, 0.125]$, and check it gives the correct rounding (when the rounding test succeeds). What is the probability of success of the rounding test?

Takeover message

• IEEE 754 does not mandate correct rounding for mathematical functions, as a consequence current mathematical libraries yield different results

• as a consequence, computations with mathematical functions are not reproducible between different libraries, sometimes between two versions of the same library, or even with the same library on different hardware!

• however, algorithms and software tools do exist, that enable to obtain correct rounding with very little overhead (see more in Module 5)

References

Handbook of Mathematical Functions, Milton Abramowitz and Irene A. Stegun, Dover, 1973, https://personal.math.ubc.ca/~cbm/aands/

IA-64 and Elementary Functions, Peter Markstein, Hewlett-Packard Professional Books, 2000.

Handbook of Floating-point Arithmetic, Jean-Michel et al., 2nd edition, Birkhäuser, 2018.

Correctly-rounded evaluation of a function: why, how, and at what cost?, Nicolas Brisebarre, Guillaume Hanrot, Jean-Michel Muller, Paul Zimmermann, ACM Computing Surveys, 2025.

Accuracy of Mathematical Functions in Single, Double, Extended Double and Quadruple Precision, Gladman et al., 2025.

The CORE-MATH project, https://core-math.gitlabpages.inria.fr/.

Questions, comments ?

Paul.Zimmermann@inria.fr